En d'autres mots l'efficacité parallèle tend vers $\frac{p}{2}/p = 50\%$ sans plus dépendre de la machine BSP, une fois que les blocs de données sont assez volumineux par rapport au nombre de processeurs pour masquer le surcoût en communication et synchronisation qui est de $gp + l + p$.

2.4 Un algorithme important : le tri de Tiskin-McColl

Le tri est une opération fondamentale en informatique. Les algorithmes de tri sont utilisés dans les bases de données, les moteurs de recherche et font partie de toutes les bibliothèques de programmation. Nous présentons ici un ingénieux algorithme BSP qui réalise le tri d'un tableau de n données en deux super-étapes, sans que la distribution des données d'entrée n'ait d'effet sur le temps de calcul. Il a été inventé par A. Tiskin et W.F. McColl [7] comme l'adaptation au modèle BSP d'un algorithme PRAM de *tri par échantillonnage parallèle* dû à Shi et Shaeffer [6].

Le problème de tri, version BSP lorsque $n > p$ est défini ainsi :
TRI-BSP-BLOCS
ENTREE : Un vecteur de n scalaires $X = [x_1, x_2, \ldots, x_n]$, distribué en p blocs $\langle X_0, X_1, \ldots, X_{p-1}\rangle_p$ c'est-à-dire que $X = X_0X_1 \ldots X_{p-1} = [x_1, x_2, \ldots, x_n]$ [5] et X_i est de taille n/p (pour simplifier on suppose que p divise n). On suppose aussi que **les valeurs x_i sont toutes distinctes**.
Une relation d'ordre total $x < y$ est donnée sur les scalaires.
SORTIE : $\langle Y_0, Y_1, \ldots, Y_{p-1}\rangle_p$ où Y_i est de taille n/p et

$$Y = Y_0Y_1 \ldots Y_{p-1} = [y_1, y_2, \ldots y_{n-1}, y_n]$$

de sorte que les données de sortie sont les mêmes que les données d'entrée

$$\{y_1, y_2, \ldots y_{n-1}, y_n\} = \{x_1, x_2, \ldots x_{n-1}, x_n\}$$

et qu'elles sont en ordre croissant

$$y_1 < y_2 < \ldots < y_{n-1} < y_n.$$

On spécifie ainsi le problème de tri comme une opérations qui consiste à prendre les données uniformément réparties en blocs locaux X_i sur les processeurs, et rendre en sortie les données en autant de blocs Y_i uniformément répartis de telle sorte que la lecture globale des données des blocs Y_i égale à la liste des données d'entrée triée en ordre croissant.

Nous allons expliquer par un exemple l'algorithme de Tiskin et McColl puis le présenter en détail et analyser son coût. Enfin on montrera comment cet algorithme peut aussi servir à trier des données dont certaines valeurs

5. On rappelle que la notation $X_0X_1 \ldots X_{p-1}$ représente la concaténation des p blocs (vecteurs locaux).

sont répétées, ce qu'on a exclu de la définition théorique du problème pour en simplifier l'explication. Si par exemple $p = 4, n = 12$ alors les données d'entrée sont réparties ainsi

$$\langle [x_0, x_1, x_2], [x_3, x_4, x_5], [x_6, x_7, x_8][x_9, x_{10}, x_{11}] \rangle.$$

Supposons qu'elle valent

$$\langle [5, 11, 6], \quad [3, 2, 14], \quad [0, 7, 9], \quad [8, 1, 4] \rangle.$$

et qu'alors TRI-BSP-BLOCS consiste à trier le vecteur global

$$X = [5, 11, 6, 3, 2, 14, 0, 7, 9, 8, 1, 4]$$

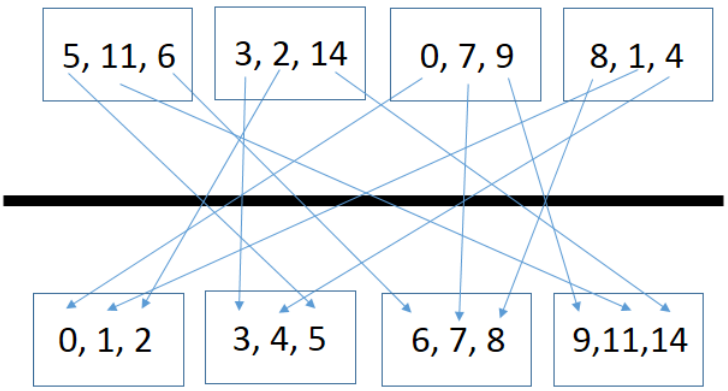
pour obtenir un vecteur global de sortie

$$Y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 14]$$

réparti sur les mémoires locales ainsi

$$\langle [0, 1, 2], \quad [3, 4, 5], \quad [6, 7, 8], \quad [9, 11, 14] \rangle.$$

Une méthode simple mais inefficace consisterait à rassembler toutes les données sur un seul processeur, disons le processeur 0, puis y trier les n données par un algorithme séquentiel, et enfin redistribuer les donnéees vers les autres processeurs dans le bon ordre. Il est bien évident que cette approche ne peut produire d'accélération parallèle puisque l'essentiel du travail y est réalisé par un seul processeur. La méthode qu'utilisent les algorithmes dits de tri par échantillonnage est de récolter et analyser un échantillon des données qui est suffisant pour indiquer comment les données peuvent être routées directement de leur bloc local X_i au bon bloc local Y_j de sorte que l'ordre final soit le bon. Sur notre exemple cela revient à déterminer les bons messages à échanger pour que le routage des données soit :



On voit sur cet exemple que l'information à récolter doit permettre au processeur 0 de savoir que la données 5 est destinée au processeur 1, que la donnée 11 est destinée au processeur 3 et que la donnée 6 est destinée au processeur 2. Elle doit aussi permettre au processeur 1 de savoir que la donnée 3 est destinée à lui-même, que la donnée 2 est destinée au processeur 0 et de même pour toutes les $n = 12$ données.

Nous montrons maintenant comment l'algorithme de Tiskin et McColl permet de récolter cette information de routage, avant d'effectuer le mouvement final des données. Comme la plupart des algorithmes parallèles, celui-ci a été conçu pour des données de grande taille. Sa définition détaillée repose sur l'hypothèse que $n \geq p^2$ ou $\frac{n}{p} \geq p$ c'est-à-dire que chaque bloc local X_i contient au moins autant de données qu'il n'y a de processeurs. Nous allons expliquer l'algorithme sur la trace d'un tout petit ensemble de données mais qui respecte cette contrainte et qui est suffisant pour l'explicaion. Supposons donc que $p = 3$ et $n = 15$ donc que les blocs sont de taille 5, et que les données avec leur distribution parallèle initiale sont les suivantes :

$$\langle [27, 18, 0, 4, 11], \quad [21, 19, 30, 16, 28], \quad [8, 5, 22, 17, 9] \rangle.$$

Comme première étape chaque processeur applique un algorithme de tri séquentiel à son bloc local de données. Le résultat est donc d'avoir des données en ordre local croissant, même si l'odre global n'est pas encore le bon :

$$\langle [0, 4, 11, 18, 27], \quad [16, 19, 21, 28, 30], \quad [5, 8, 9, 17, 22] \rangle.$$

Chaque processeur dispose alors d'une liste (bloc) locale dont les valeurs sont en ordre croissant. Alors chacun sélectionne $p + 1$ valeurs parmi elles qui sont un ensemble de *séparateurs locaux*. Cela signifie que ces $p + 1$ valeurs sont uniformément espacées dans la liste locale, qu'elles y définissent p sous-blocs de longueurs égales à ± 1 près. Les séparateurs ne sont pas des limites d'itervalles de *valeurs* mais bien la limites de blocs de valeurs de tailles (localement) égales. Si par exemple le bloc local était

$$[1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500]$$

et que $p = 3$ alors les séparateurs pourraient être $1, 5, 50, 500$ de sorte que les p sous-blocs définis soient $[1, 2, 3, 4, 5]$, $[10, 20, 30, 40, 50]$ et $[100, 200, 300, 400, 500]$. Dans notre très petit exemple parallèle ci-dessus chaque processeur sélectionne une liste de 4 séparateurs qui sont par exemple les valeurs soulignées ici :

$$\langle [\underline{0}, 4, \underline{11}, \underline{18}, \underline{27}], \quad [\underline{16}, 19, \underline{21}, \underline{28}, \underline{30}], \quad [\underline{5}, 8, \underline{9}, \underline{17}, \underline{22}] \rangle.$$

L'utilité des séparateurs locaux est indirecte : ils contiennent de l'information utile sur les valeurs mais cette information ne concerne que le bloc local. Pour

globaliser cette information l'algorithme réalise un échange global des séparateurs locaux ce qui fait que chaque processeur aura à sa disposition tous les $(p+1)p$ séparateurs locaux. Cela se réalise par une phase de communication-synchronisation dont voici le résultat :

$$\langle [[0,11,18,27],[16,21,28,30],[5,9,17,22]]@i\rangle_3.$$

Puisque les trois listes de séparateurs reçues sont déjà triées, il est alors facile de les fusionner en une liste triée de $p(p+1)$ séparateurs, et ceci simultanément sur chaque processeur $i=0,1,2$. Le résultat de cette fusion est alors

$$\langle [0,5,9,11,16,17,18,21,22,27,28,30]@i\rangle_3.$$

L'étape suivante de l'algorithme consiste à faire un échantillonage des échantillons locaux. Il s'agit de sélectionner $p+1$ séparateurs, non plus parmi les données locales, mais parmi l'ensemble global des séparateurs locaux qui est maintenant à disposition de chaque processeur. Ces nouvelles valeurs sont appelées les *séparateurs globaux* et son ici soulignées parmi l'ensemble global des séparateurs locaux :

$$\langle [\underline{0},5,9,11,\underline{16},17,18,21,\underline{22},27,28,\underline{30}]@i\rangle_3.$$

On élimine alors tous les séparateurs locaux qui ne sont pas globaux, et il n'en reste plus que $p+1$, disponibles sur chaque processeur :

$$\langle [0,16,22,30]@i\rangle_3$$

soit

$$\langle [0,16,22,30],\quad [0,16,22,30],\quad [0,16,22,30]\rangle$$

Les séparateurs globaux définissent l'information nécessaire et suffisante pour le routage de toutes les données à trier. En général, le nombre total de données n est illimité par rapport à p et les intervalles définis par les séparateurs sont ainsi peuplés par un nombre illimité de valeurs lorsque $n\to\infty$. Sur notre très petit exemple les séparateurs sont plus nombreux que les autres valeurs mais ce n'est qu'un effet nécessaire de la typographie. La dernière étape de l'algorithme utilise les séparateurs secondaires comme une définition des intervalles de valeurs parmi lesquels celui de rang $j=0,1,\ldots,p-1$ détermine les données devant être routées vers le processeur j. Sur notre exemple cela signifie que les valeurs x telles que $0\leq x<16$ doivent être routées vers le processeur 0, etc. Ainsi $I_0=[0,16)$, $I_1=[16,22)$, $I_2=[22,30]$ et chaque processeur doit envoyer les valeurs de son bloc local vers ces p "casiers" que sont les intervalles I_j, où j est le numéro du processeur où se trouve I_j. Sur notre exemple, les blocs locaux de données n'ont pas bougé pendant le calcul des séparateurs, ils sont toujours :

$$\langle [0,4,11,18,27],\quad [16,19,21,28,30],\quad [5,8,9,17,22]\rangle.$$

On prépare alors les messages à envoyer en parcourant localement le bloc de données et en attribuant à chacune un intervalle I_j de destination. Les messages à envoyer sont notés $x \to j$ pour signifier que $x \in I_j$ et doit donc être routée vers le processeur j. Les voici :
$\langle [0 \to 0, 4 \to 0, 11 \to 0, 18 \to 1, 27 \to 2],$
$[16 \to 0, 19 \to 1, 21 \to 1, 28 \to 2, 30 \to 2],$
$[5 \to 0, 8 \to 0, 9 \to 0, 17 \to 1, 22 \to 2]\rangle$. La suite de l'algorithme consiste simplement en une communication-synchronisation réalisant ce routage, ce qui fait que les valeurs reçues sont les suivantes :

$$\langle [[0, 4, 11], [16], [5, 8, 9]], \quad [[18], [19, 21], [17]], \quad [[27], [28, 30], [22]]\rangle$$

c'est-à-dire que chaque processeur a reçu $p = 3$ listes de valeurs, une de chaque processeur. Il peut ensuite fusionner ces listes localement pour obtenir la liste croissante correspondant à son intervalle I_j :

$$\langle [0, 4, 5, 8, 9, 11, 16], \quad [17, 18, 19, 21], \quad [22, 27, 28, 30, 22]\rangle$$

Cela produit le résultat final attendu de l'algorithme, la liste triée de toutes les valeurs, répartie de manière équilibrée en p blocs locaux. Puisque les séparateurs sont en ordre croissant, ne sont pas inférieurs à la plus petite valeur, ni supérieurs à la plus grande valeur, il est clair que le routage des valeurs suivi des fusions locales produira un tri global des valeurs. Mais cette remarque est aussi valable *pour plusieurs autres suites croissantes de $p+1$ qu'on aurait choisi comme séparateurs*. Sur notre exemple si les séparateurs utilisés pour le routage avaient été disons $[0, 4, 5, 22]$ alors les valeurs auraient été réparties ainsi :

$$\langle [0], \quad [4], \quad [5, 8, 9, 11, 16, 17, 18, 19, 21, 22, 27, 28, 30, 22]\rangle$$

ce qui est effectivement une répartition triée mais, étant fortement déséquilibrée elle aurait eu un effet néfaste sur les performances puisque le processeur 2 aurait alors reçu la quasi totalité des messages. L'équilibre des blocs locaux de sortie ne fait pas partie de la spécification de TRI-BSP-BLOCS, mais l'analyse des coûts d'exécution montre qu'il faut à tout prix minimiser les déséquilibres et surtout ne pas produire des blocs comme ceux ci-dessus où un seul bloc contient presque toutes les n données. Or l'algorithme de Tiskin et McColl produit des blocs de sortie dont les tailles sont relativement égales. La démonstration mathématique en a été publiée par les auteurs [7] mais on peut s'en convaincre assez facilement ainsi.

1. Par définition de la première étape de l'algorithme, l'intervalle entre deux séparateurs locaux contient au moins $\frac{n}{p}/p = \frac{n}{p^2} \pm 1$ valeurs, soient celles de son bloc local qui s'y trouvent certainement.
2. Par definition des séparateurs globaux, l'intervalle I_j entre deux d'entre eux contient au moins $(\frac{n}{p^2} \pm 1)p = \frac{n}{p} \pm p$ valeurs.

3. Le pire cas de déséquilibre entre les I_j serait donc celui où $p-1$ des intervalles contiennent le minimum de valeurs possibles soient $\frac{n}{p}-p$. Ensemble ils contiendraient alors $(\frac{n}{p}-p)(p-1) \approx (\frac{n}{p}-p)p = n-p^2$.
4. Le dernier intervalle contiendrait alors au plus environ $n-(n-p^2) = p^2$ valeurs ce qui ne pose aucun problème de déséquilibre sur de grandes données c'est-à-dire lorsque n est beaucoup plus grand que p.

A toutes fins utiles on peut donc estimer que la taille des blocs de données résultant de l'algorithme de Tiskin-McColl est égale $\frac{n}{p} \pm p$ ou plus approximativement $\frac{n}{p}$ sur chaque processeur.

Nous allons maintenant répéter le cas général des étapes expliquées ci-dessus sur notre exemple, ce qui constitue la définition de l'algorithme de Tiskin et McColl pour TRI-BSP-BLOCS :

```
// tri Tiskin-McColl, BSP n>p

// première phase, tri du bloc local et sélection des séparateurs locaux
< X'_i:= sort(X_i);
  L_i := [X'_i[0], X'_i[(n/p^2)-1], ... , X'_i[(n/p)-1]] // séparateurs locaux
       @ i >_p ;
// deuxième phase, sélection des séparateurs globaux
< (send L_i to [0..p-1]) @ i >_p ;
< LL_i:= merge(L_0, L_1, ..., L_(p-1)) ;      // longueur p(p+1)
  G := [ LL[0], LL[p], ..., LL[p(p+1)-1] ] ; // p+1 séparateurs globaux
    @ i >_p ;
// troisième phase, routage des valeurs
< for k=0 to (n/p)- 1 do
    if X'_i[k] = G[p(p+1)-1] then  // X'_[k]: I_(p-1)
  send X'_[k] to p-1
else
      j:= {pid | (G[pid] <= X'_i[k] < G[pid+1])} // X'_[k]: I_j
    send X'_i[k] to j ;
  done;
@ i >_p ;
// dernière phase, fusion locale des données reçues
< Z_j_i[0..n_j] = { X'_i[k] | reçu du processeur i} // n_i ~ (n/p)
  Y_j:= merge(Z_j_0, Z_j_1, ..., Z_j_(p-1)) ;
@ j >_p ;
// Fin: les blocs Y_j constituent un tri global des données.
```

La figure 2.8 illustre notre algorithme en masquant les détails du calcul des séparateurs car les paramètres de taille $p = 4, n = 12$ sont trop petits y être visibles.

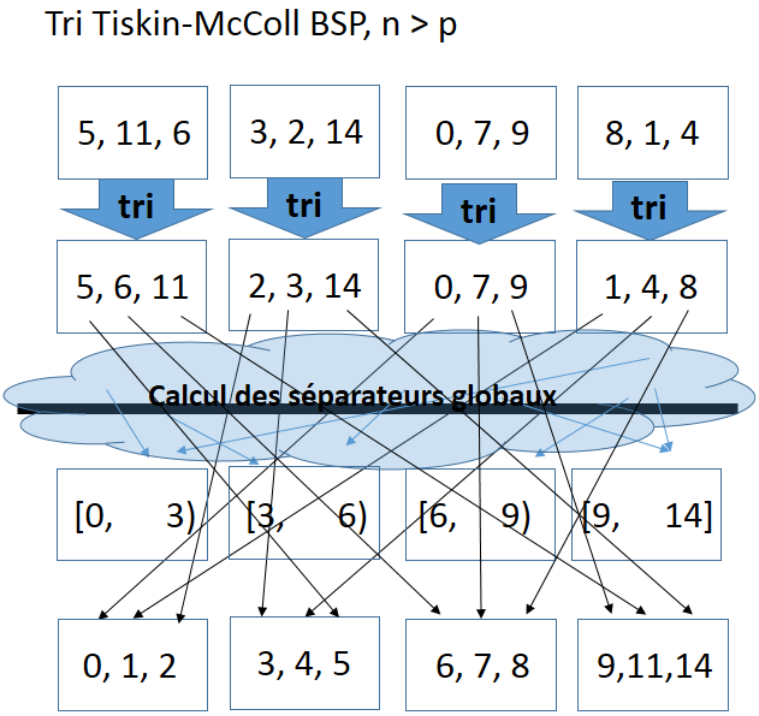


FIGURE 2.8 – L'algorithme de tri BSP de Tiskin et McColl ($p < n$).

Analysons maintenant le coût BSP de l'algorithme soit notre modélisation de son temps de calcul selon les principes expliqués dans ce chapitre.

1. Des algorithmes séquentiels existent pour trier un tableau ou une liste de m valeurs en temps $O(m \log m)$ [1]. La première opération `sort` nécessite donc environ $m \log m$ opérations où m est la longueur du bloc soit $\frac{n}{p}$. Le coût du tri est donc de
$$\frac{n}{p}(\log \frac{n}{p}) = \frac{n}{p}(\log n - \log p) \leq \frac{n}{p}(\log n) = \frac{n \log n}{p}$$
et on peut donc l'approximer à
$$\frac{n \log n}{p}.$$
Ensuite la sélection des séparateurs locaux `L_i` parmi ce bloc trié peut être réalisée en temps proportionnel à la longueur du bloc soit
$$p + 1$$
(on suppose que le bloc est un tableau et qu'on y accède directement aux valeurs des bons rangs pour les copier dans un tableau des séparateurs).
2. Puis chaque processeur envoie ses séparateurs locaux à tous les autres. C'est là une phase de communication-synchronisation où les paramètres BSP sont $h^+ = p(p+1)$ et $h^- = p(p+1)$ car chaque processeur envoie p fois son bloc de $p+1$ séparateurs locaux et de même reçoit p blocs de séparateurs locaux. Donc $h = \max(h^+, h^-) = p(p+1)$ et le coût BSP de cette phase est
$$gh + L = gp(p+1) + L.$$
A la deuxième phase, chaque processeur a reçu p suites de $(p+1)$ séparateurs locaux qu'il fusionne (`merge`) en temps linéaire [1] donc en temps $p(p+1)$ pour produire tableau de cette taille. De là les séparateurs globaux peuvent être sélectionnés en temps proportionnel au nombre de séparateurs soit
$$p + 1.$$
Cela produit le tableau G des $p+1$ séparateurs globaux.
3. La troisième phase est une communication-sychronisation où chaque processeur envoie chacune des valeurs de son bloc vers la bonne destination. Ainsi $h^+ = \frac{n}{p}$ puisque chacun envoie cette quantité de scalaires vers d'autres processeurs selon les casiers prévus. De même chaque processeur reçoit autant de valeurs que n'en contient son casier. Or on a démontré que leur taille est approximativement la même que celle des blocs de départ. Donc $h^- \approx \frac{n}{p}$ et on peut approximer $h = \frac{n}{p}$. La phase coûte donc
$$g\frac{n}{p} + L.$$

On additionne les coûts des trois phases ainsi :

$$\frac{n \log n}{p} + (p+1) + gp(p+1) + L + (p+1) + g\frac{n}{p} + L$$

soit

$$T_{\text{par}} = [\frac{n \log n}{p} + 2(p+1)] + g[\frac{n}{p} + p(p+1)] + 2L$$

où on a d'abord le terme additif du temps de calcul séquentiel (asynchrone sur chaque processeur) puis le temps de communication et enfin le coût en synchronisations globales.

Comparons maintenant ce temps de calcul estimé avec celui d'un algorithme de tri séquentiel. Comme on l'a rappelé, un bon algorithme de tri séquentiel peut traiter toutes les n données en temps estimé à

$$T_{\text{seq}} = n \log n.$$

Le facteur d'accélération est donc

$$\frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{n \log n}{[\frac{n \log n}{p} + 2(p+1)] + g[\frac{n}{p} + p(p+1)] + 2L}$$

et puisque

$$\frac{T_{\text{seq}}}{T_{\text{par}}} = \{[\frac{1}{p} + \frac{2(p+1)}{n \log n}] + g[\frac{1}{p \log n} + \frac{p(p+1)}{n \log n}] + \frac{2L}{n \log n}\}^{-1}$$

on obtient

$$\lim_{n \to \infty} \frac{T_{\text{seq}}}{T_{\text{par}}} = \{[\frac{1}{p}] + g[0] + 0L\}^{-1} = p$$

ce qui signifie que l'algorithme tend vers une accélération parfaite sur de très grandes données. Le facteur d'efficacité parallèle, soit l'accélération divisée par p tend ainsi vers 100% lorsque $n \to \infty$.

2.5 Complexité, coût BSP et temps de calcul

Pour mieux comprendre ce que signifie le modèle BSP et ses formules de coût (temps d'exécution) parallèle, revenons encore brièvement sur la notion de complexité d'un algorithme.
Soit par exemple un algorithme séquentiel qui transforme un tableau de n données en un tableau de même taille et dont le temps de calcul est $O(n)$. Cela signifie qu'il existe une constante positive a pour laquelle le nombre d'opérations séquentielles exécutée par A sur une entrée de taille n est estimé par

$$T(n) \approx an.$$