Dans les exercices qui suivent, nous utiliserons le fichier `bsmlAnalyser.ml` qui contient un module BsmlAnalyser. Ce module contient des versions instrumentées des primitives BSML ainsi qu'une fonction bsp_test : ('a→ 'b)→ 'a→ 'b qui prend en paramètre une fonction et une valeur et affiche le coût BSP de l'application de la fonction à la valeur.

La fonction bytes_in: 'a → int donne la taille en octets d'une valeur OCaml. Cette fonction est utilisée par bsp_test pour calculer le coût h de l'échange de données.

Dans la suite vous utiliserez les fonctions BSML déjà vues ou implémentées ainsi que les fonctions **proj** et bsp_test.

Exercice 1. Test des fonctions précédentes.
Reprenez les 6 fonctions parallèles de l'exercice 5 du TD précédent et leurs 6 arguments, les vecteurs parallèles de l'exercice 3. Utilisez la fonction d' *instrumentation* bsp_test pour appliquer chaque fonction à son argument en vérifiant que le coût en communication et en synchronisation est nul.

Exercice 2. Du local au global.
Utiliser la fonction **proj** pour définir une fonction first_element: 'a **par** → 'a telle que first_element v est la valeur du vecteur parallèle v au processeur 0. De même définir last_element qui rend la valeur du vecteur parallèle v au processeur le plus à droite.

Exercice 3. Coût de communication

a. Observez avec bsp_test que l'application de first_element ou last_element a un coût en communication et en synchronisation non-nul.

b. Donnez la formule théorique de coût pour first_element et vérifiez avec bytes_in qu'elle est bien confirmée sur des valeurs de taille croissante. Prenez par exemple des listes de taille 0, 1, 2, ... comme éléments du vecteur donné en entrée et observez l'évolution de h en fonction de la valeur bytes_in l où l est une liste de même taille que celles contenues dans le vecteur parallèle.

c. Même question avec un vecteur déséquilibré (Taille de liste différente d'un processeur à l'autre, par exemple le vecteur from_0_to_pid).

Exercice 4. Tri séquentiel.
Utilisez **proj** et Sort.list pour écrire une fonction de tri sort_seq: 'a **par** → 'a list, telle que sort_seq $\ll v_0$, ..., $v_{(p-1)} \gg$ soit le résultat du tri de la liste [v_0 ; ...; $v_{(p-1)}$].

Exercice 5. Tri séquentiel d'une liste distribuée.

a. Donnez une fonction sort_seq_lists : 'a list **par** → 'a list, implémentée à l'aide de **proj** et de List.sort telle que sort_seq_lists $\ll l_0$, ..., $l_{(p-1)} \gg$ donne le même résultat que tri de (l_0 @ l_1 @ ... @ $l_{(p-1)}$).

b. Donnez une fonction équivalente sort_seq_lists_2 : 'a list **par** → 'a list, implémentée à l'aide de **proj**, sort_par et List.merge.

Exercice 6. Coût du tri.
Donnez le coût BSP de ces deux fonctions. Testez votre hypothèse à l'aide de bsp_test.

Exercice 7. Itération parallèle
Ecrivez une fonction div2: float → float qui divise tout nombre >= 0 par 2 et qui rend 0 pour tout nombre négatif. Donnez ensuite un vecteur x : float **par** contenant des valeurs aléatoires. Ecrivez aussi une fonction div2par : float **par** → float **par** qui applique partout div2. Enfin définissez une fonction div_while: float → float **par** → float **par** telle que (div_while epsilon v) applique div2par à v tant que son argument v contient une valeur au moins aussi grande qu'espilon.

Exercice 8. BONUS
Quel est le coût BSP de div_while ? Pouvez-vous faire une meilleur implémentation ?