Consignes

Ce projet sera à réaliser en une semaine. Vous débuterez lors de la séance de travaux dirigés du 28 mars, et vous rendrez votre code à la fin de la séance du 4 avril.

Les instructions suivantes devront impérativement être suivies : **toute personne ou groupe ne respectant pas ces consignes se verra attribuer pour note 0.**

- Un groupe de 2 à 4 personnes devra être constitué au plus tard lors de la séance du 28 mars. Les noms seront donnés à l'enseignant, un numéro de groupe vous sera alors attribué.
- Le code devra être rendu lors de la séance du 4 avril. Aucun code ne sera accepté après la séance.
- Le code rendu devra passer la phase de compilation sans erreur.
- Chaque groupe doit travailler indépendemment des autres groupes, le partage de code entre groupes n'est pas permis.

Si et **seulement si** toutes les instructions précédentes sont validées, les éléments évalués seront les suivants (triés par ordre décroissant d'importance) : pour chaque fonction implantée,

- le résultat du calcul est correct, quelque soit le nombre de processeurs ;
- le coût BSP de l'implantation est donné dans un commentaire précédant le code ;
- l'implantation est optimisée (calculs bien répartis sur les différents processeurs, coût de communication minimal, nombre de synchronisations minimal)
- le code est précédé d'un commentaire expliquant son fonctionnement.

1 Sujet

Le but de votre projet est de proposer une implantation parallèle d'une sous partie de la bibliothèque standard de programmation Array de OCaml. L'idée est de concevoir une bibliothèque BSML offrant à l'utilisateur des fonctions pour créer et manipuler des tableaux distribués sans que l'utilisateur n'ai à écrire lui-même de code parallèle. Il faudra donc choisir une structure de donnée et donner une implantation parallèle des fonctions listées en annexe.

Le fichier `arrayPar.mli` disponible sur le site http://tesson.julien.free.fr/teaching/bsml/ fournit les signatures que devront vérifier vos implantations. Le fichier `arrayPar.ml` vous est proposé comme point de départ pour votre travail, il contient quelques indications. Pour compiler votre fichier en forçant la vérification des signatures fournies dans le fichier mli, compilez votre programme avec la commande `bsmlopt.mpi arrayPar.mli arrayPar.ml`

Recommandations

Réflechissez à l'implantation des fonctions avant de choisir votre structure de donnée.

Les sections de l'annexe sont données par ordre approximativement croissant de difficulté, il est recommandé de les travailler à peu près dans l'ordre (sauf si votre choix d'implantation dépend des fonctions suivantes évidemment).

Pour chaques fonctions, commencez par implantater une version simple (souvent peu efficace) puis essayez de l'optimiser. **Attention : une seul implantation sera évaluée, celle dont le nom correspond à la signature fournie dans ce sujet.**

Ne faites pas l'impasse sur le calcul du coût de vos implantation, les points affectés à cette partie ne pourront pas être compensés par les autres parties du projet. À titre indicatif, le barême pourrait être :

- fonctions correctes, efficacité de l'implémentation 13pt ;
- coûts donnés et correspondant au code 5pt ;
- commentaires expliquant le code pour toutes les fonctions 2pt

Lisez la documentation de la bibliothèque de programmation Array :
http://caml.inria.fr/pub/docs/manual-ocaml-312/libref/Array.html

A Module `ArrayPar` : Parallel implementation of arrays

A.1 Type des tableaux parallèles

`type 'a parallel_array`
Type of parallel arrays

A.2 Constructeurs de tableaux parallèles

`val make : int -> 'a -> 'a parallel_array`
`make n x` returns a fresh parallel array of length `n`, initialized with copy of `x`.
Raise `Invalid_argument` if `n < 0`.

`val init : int -> (int -> 'a) -> 'a parallel_array`
`init n f` returns a fresh parallel array of length `n`, with element number `i` initialized to the result of `f i`.
Raise `Invalid_argument` if `n < 0`.

`val copy : 'a parallel_array -> 'a parallel_array`
`copy a` returns a copy of `a`, that is, a fresh parallel array containing the same elements as `a`.

`val of_array : 'a array -> 'a parallel_array`
`of_array a` returns a fresh parallel array containing the elements of `a`.

`val to_array : 'a parallel_array -> 'a array`
`to_array a` returns a fresh array containing the elements of `a`.

A.3 Operations basiques sur les tableaux parallèles.

`val distribution : 'a parallel_array -> int array`
return an array `a` of size `bsp_p` where the `i`th element of `a` (noté `a.(i)`) is the number of elements stored at processor `i` in the given parallel array.

`val length : 'a parallel_array -> int`
Return the length (number of elements) of the given parallel array.

`val get : 'a parallel_array -> int -> 'a`
`get a n` returns the element number `n` of parallel array `a`. The first element has number 0. The last element has number `length a - 1`.
Raise `Invalid_argument "index out of bounds"` if `n` is outside the range 0 to `(length a - 1)`.

`val set : 'a parallel_array -> int -> 'a -> unit`
`set a n x` modifies parallel array `a` in place, replacing element number `n` with `x`.
Raise `Invalid_argument "index out of bounds"` if `n` is outside the range 0 to `length a - 1`.

`val fill : 'a parallel_array -> int -> int -> 'a -> unit`
`fill a ofs len x` modifies the array `a` in place, storing `x` in elements number `ofs` to `ofs + len - 1`.
Raise `Invalid_argument "fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

A.4 map/reduce sur les tableaux parallèles.

`val map : ('a -> 'b) -> 'a parallel_array -> 'b parallel_array`

`map f a` applies function `f` to all the elements of `a`, and builds a parallel array with the results returned by `f` : [f (get a 0); f (get a 1); ...; f (get a (length a − 1))].

`val mapi : (int -> 'a -> 'b) -> 'a parallel_array -> 'b parallel_array`

Same as `map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

`val reduce : ('a -> 'a -> 'a) -> 'a -> 'a parallel_array -> 'a`

`reduce f e a` computes `f (... (f (f (get a 0) (get a 1)) (get a 2)) ...) (get a (n-1))`, where `n` is the length of the parallel array `a`. The function `f` is associative. The element `e` is neutral for operation `f` : `f e a = f a e = a`

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b parallel_array -> 'a`

`fold_left f x a` computes `f (... (f (f x (get a 0)) (get a 1)) ...) (get a (n-1))`, where `n` is the length of the parallel array `a`. The function `f` has to be associative.

A.5 Tri

`val sort : ('a -> 'a -> int) -> 'a parallel_array -> 'a parallel_array`

Sort a parallel array in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see below for a complete specification). For example, `Pervasives.compare` is a suitable comparison function, provided there are no floating-point NaN values in the data. `sort` returns an array sorted in increasing order.

Specification of the comparison function : Let `a` be the array and `cmp` the comparison function. The following must be true for all x, y, z in a :

— `cmp x y` > 0 if and only if `cmp y x` < 0

— if `cmp x y` ≥ 0 and `cmp y z` ≥ 0 then `cmp x z` ≥ 0

The result contains the same elements as the given parallel array, reordered in such a way that for all i and j valid indices of `a` :

— `cmp a.(i) a.(j)` ≥ 0 if and only if i $\geq$ j

The result should satifies the following equality : `arrayPar.to_array (arrayPar.sort cmp ap) = Array.sort cmp (arrayPar.to_array ap)`

A.6 Functions avec schéma de communication non-trivial

`val balance : 'a parallel_array -> 'a parallel_array`

`balance v` returns a fresh parallel array containing the same values as the array `v` but where data are evenly distributed among processors. (i.e. each processor has n/p or 1 + n/p elements)

`val sub : 'a parallel_array -> int -> int -> 'a parallel_array`

`sub a start len` returns a fresh array of length `len`, containing the elements number `start` to `start + len - 1` of array `a`.

Raise `Invalid_argument "sub"` if `start` and `len` do not designate a valid subarray of `a`; that is, if `start < 0`, or `len < 0`, or `start + len > length a`.

```
val blit :
  'a parallel_array ->
  int -> 'a parallel_array -> int -> int -> unit
```

`blit v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap.

Raise `Invalid_argument "Array.blit"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.