Introduction à la Programmation Parallèle: BSPlib

Frédéric Gava et Gaétan Hains

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours du M2 SSI option PSSR

Plan

Plan

Plan

Déroulement du cours

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

The PUB

Kasako ?

- Une bibliothèque C dédiée au calcul BSP
- Proche du standard BSPLIB (`http://www.bsp-worldwide.org/`, possible d'écrire en standard) et de nouvelles fonctionnalités
- Un nombre très limité de fonctions (une petite vingtaine) par rapport à MPI (plus de 100) ; mais très expressives et très efficaces
- librement téléchargeable à `http://www.uni-paderborn.de/~bsp/`
- on trouve aussi 2 autres implantations (moins récentes) de BSPLib :
  1. `http://www.bsp-worldwide.org/implmnts/oxtool/`
  2. `http://bsponmpi.sourceforge.net/`

Sur quels types d'architectures

En fait, presque tous : grappes de PC (mode TCP/IP ou MPI) avec réseaux spéciaux (SCI), machines parallèles commerciales (Cray T3E, SUN *etc.*), mémoire partagées (mode SHMEM) et implantation MPI... La liste est disponible dans l'aide.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Comment ?

Installation

Actuellement problématique. La dernière version stable ne compile plus sur les Linux récents et la version en cours (beta-version) ne fonctionne pas bien...Solution : il faut prendre les .c et .h de la "vieille" et les mettre dans la beta-version (me demander de l'aide en cas de pb chez vous).

Les nouvelles fonctionnalités non-BSP pure ?

1. sous-synchronisation (barrière non globals mais par groupe de processus)
2. synchronisation légère (n'attend pas tout les processeurs mais juste un certain nombre de messages)
3. processus virtuelles et migration de processus légers

On ne vera que la synchronisation légère. Les autres sont trop polémiques.

Pour commencer

Première utilisation

- On compile avec `pubcc` comme un programme C
- on exécute avec la commande `pubrun options program arguments` :
  - `--procs=n` pour avoir `n` processus
  - `-n hostlist` exemple `-p 3 -n node1 node2 node3`
  - `-m hostfile` comme MPI

Premier programme

- Comme en MPI, on programme en mode SPMD
- On initialise les arguments du programme (si besoin) puis toujours la machine parallèle
- Comme en MPI, on peut obtenir ensuite le nombre de processus et le "pid" du processus
- Un programme BSPLib doit toujours de-initialiser la machine parallèle

Pour commencer

Première utilisation

- On compile avec `pubcc` comme un programme C
- on exécute avec la commande `pubrun options program arguments` :
  - `--procs=n` pour avoir `n` processus
  - `-n hostlist` exemple `-p 3 -n node1 node2 node3`
  - `-m hostfile` comme MPI

Premier programme

- Comme en MPI, on programme en mode SPMD
- On initialise les arguments du programme (si besoin) puis toujours la machine parallèle
- Comme en MPI, on peut obtenir ensuite le nombre de processus et le "pid" du processus
- Un programme BSPLib doit toujours de-initialiser la machine parallèle

Pour commencer

Première utilisation

- On compile avec `pubcc` comme un programme C
- on exécute avec la commande `pubrun options program arguments` :
  - `--procs=n` pour avoir `n` processus
  - `-n hostlist` exemple `-p 3 -n node1 node2 node3`
  - `-m hostfile` comme MPI

Premier programme

- Comme en MPI, on programme en mode SPMD
- On initialise les arguments du programme (si besoin) puis toujours la machine parallèle
- Comme en MPI, on peut obtenir ensuite le nombre de processus et le "pid" du processus
- Un programme BSPLib doit toujours de-initialiser la machine parallèle

Pour commencer

Première utilisation

- On compile avec `pubcc` comme un programme C
- on exécute avec la commande `pubrun options program arguments` :
  - `--procs=n` pour avoir `n` processus
  - `-n hostlist` exemple `-p 3 -n node1 node2 node3`
  - `-m hostfile` comme MPI

Premier programme

- Comme en MPI, on programme en mode SPMD
- On initialise les arguments du programme (si besoin) puis toujours la machine parallèle
- Comme en MPI, on peut obtenir ensuite le nombre de processus et le "pid" du processus
- Un programme BSPLib doit toujours de-initialiser la machine parallèle

Pour commencer

Première utilisation

- On compile avec `pubcc` comme un programme C
- on exécute avec la commande `pubrun options program arguments` :
  - `--procs=n` pour avoir `n` processus
  - `-n hostlist` exemple `-p 3 -n node1 node2 node3`
  - `-m hostfile` comme MPI

Premier programme

- Comme en MPI, on programme en mode SPMD
- On initialise les arguments du programme (si besoin) puis toujours la machine parallèle
- Comme en MPI, on peut obtenir ensuite le nombre de processus et le "pid" du processus
- Un programme BSPLib doit toujours de-initialiser la machine parallèle

Pour commencer

Exemple

```
#include <pub.h>
int main(int argc, char* argv[] ) {
 t_bsp bsp;
 int pid, nprocs;
 bsplib_saveargs(\&argc,\&argv);
 bsplib_init(BSPLIB_STDPARAMS,\&bsp);
 nprocs = bsp_nprocs(\&bsp);
 pid = bsp_pid(\&bsp);
 printf( "Hello world from processor \%d of \%d!\n",pid,nprocs);
 bsplib_done();
 return 0 }
```

Déroulement du cours

2 modes de communications

Mode par passage de messages

C'est le mode le plus simple et qui ressemble le plus à MPI : on "send" des messages et on "getmsg" après la barrière de synchronisation.

Mode par partage de mémoire

Dans ce mode, chaque processus peut rendre accessible aux autres processus une partie de sa mémoire : une lecture distante n'est cohérente qu'après une barrière de synchronisation.

Que choisir ?

En fait, les 2 modes sont utiles et cela dépend de votre algorithme. Pour envoyer des données récemment calculées à un autre processus, il est souvent plus simple d'utiliser le mode par passage de messages. A contraiori, pour lire des données calculées par un autre processus, mieux vaut que celui-ci mettent ses données dans un zone partagée.

2 modes de communications

Mode par passage de messages

C'est le mode le plus simple et qui ressemble le plus à MPI : on "send" des messages et on "getmsg" après la barrière de synchronisation.

Mode par partage de mémoire

Dans ce mode, chaque processus peut rendre accessible aux autres processus une partie de sa mémoire : une lecture distante n'est cohérente qu'après une barrière de synchronisation.

Que choisir ?

En fait, les 2 modes sont utiles et cela dépend de votre algorithme. Pour envoyer des données récemment calculées à un autre processus, il est souvent plus simple d'utiliser le mode par passage de messages. A contraiori, pour lire des données calculées par un autre processus, mieux vaut que celui-ci mettent ses données dans un zone partagée.

2 modes de communications

Mode par passage de messages

C'est le mode le plus simple et qui ressemble le plus à MPI : on "send" des messages et on "getmsg" après la barrière de synchronisation.

Mode par partage de mémoire

Dans ce mode, chaque processus peut rendre accessible aux autres processus une partie de sa mémoire : une lecture distante n'est cohérente qu'après une barrière de synchronisation.

Que choisir ?

En fait, les 2 modes sont utiles et cela dépend de votre algorithme. Pour envoyer des données récemment calculées à un autre processus, il est souvent plus simple d'utiliser le mode par passage de messages. A contraiori, pour lire des données calculées par un autre processus, mieux vaut que celui-ci mettent ses données dans un zone partagée.

Programmation BSP

Paramètres BSP

- **int** bsp_nprocs (t_bsp* bsp) : nombre de processeurs
- **int** bsp_pid (t_bsp* bsp) : pid du processeur
- **double** bsp_g (t_bsp* bsp) (idem pour L et s)
- t_bsp est le groupe BSP=tout les processeurs car pas de sous-groupes.

Synchronisation

- Synchronisation global void bsp_sync (t_bsp* bsp) : tout les processus sont synchronisés et après cet appel, tout les messages ont été envoyés.
- Synchronisation légère void bsp_oblsync (t_bsp* bsp, int nmsgs) : idem mais on précise le nombre de message à attendre (c'est donc plus efficace mais il faut connaître ce nombre). Chaque send, put, get et chaque opération collective ajoute un message à attendre.
- Il est possible de suivre l'état des communications.

Programmation BSP

Paramètres BSP

- **int** bsp_nprocs (t_bsp* bsp) : nombre de processeurs
- **int** bsp_pid (t_bsp* bsp) : pid du processeur
- **double** bsp_g (t_bsp* bsp) (idem pour L et s)
- t_bsp est le groupe BSP=tout les processeurs car pas de sous-groupes.

Synchronisation

- Synchronisation global **void** bsp_sync (t_bsp* bsp) : tout les processus sont synchronisés et après cet appel, tout les messages ont été envoyés.
- Synchronisation légère **void** bsp_oblsync (t_bsp* bsp, **int** nmsgs) : idem mais on précise le nombre de message à attendre (c'est donc plus efficace mais il faut connaître ce nombre). Chaque send, put, get et chaque opération collective ajoute un message à attendre.
- Il est interdit de mixer les 2 dans une même super-étape

Programmation BSP

Paramètres BSP

- **int** bsp_nprocs (t_bsp* bsp) : nombre de processeurs
- **int** bsp_pid (t_bsp* bsp) : pid du processeur
- **double** bsp_g (t_bsp* bsp) (idem pour L et s)
- t_bsp est le groupe BSP=tout les processeurs car pas de sous-groupes.

Synchronisation

- Synchronisation global **void** bsp_sync (t_bsp* bsp) : tout les processus sont synchronisés et après cet appel, tout les messages ont été envoyés.
- Synchronisation légère **void** bsp_oblsync (t_bsp* bsp, **int** nmsgs) : idem mais on précise le nombre de message à attendre (c'est donc plus efficace mais il faut connaître ce nombre). Chaque send, put, get et chaque opération collective ajoute un message à attendre.
- Il est interdit de mixer les 2 dans une même super-étape

Programmation BSP

Paramètres BSP

- **int** bsp_nprocs (t_bsp* bsp) : nombre de processeurs
- **int** bsp_pid (t_bsp* bsp) : pid du processeur
- **double** bsp_g (t_bsp* bsp) (idem pour L et s)
- t_bsp est le groupe BSP=tout les processeurs car pas de sous-groupes.

Synchronisation

- Synchronisation global **void** bsp_sync (t_bsp* bsp) : tout les processus sont synchronisés et après cet appel, tout les messages ont été envoyés.
- Synchronisation légère **void** bsp_oblsync (t_bsp* bsp, **int** nmsgs) : idem mais on précise le nombre de message à attendre (c'est donc plus efficace mais il faut connaître ce nombre). Chaque send, put, get et chaque opération collective ajoute un message à attendre.
- Il est interdit de mixer les 2 dans une même super-étape

Programmation BSP

Paramètres BSP

- **int** bsp_nprocs (t_bsp* bsp) : nombre de processeurs
- **int** bsp_pid (t_bsp* bsp) : pid du processeur
- **double** bsp_g (t_bsp* bsp) (idem pour L et s)
- t_bsp est le groupe BSP=tout les processeurs car pas de sous-groupes.

Synchronisation

- Synchronisation global **void** bsp_sync (t_bsp* bsp) : tout les processus sont synchronisés et après cet appel, tout les messages ont été envoyés.
- Synchronisation légère **void** bsp_oblsync (t_bsp* bsp, **int** nmsgs) : idem mais on précise le nombre de message à attendre (c'est donc plus efficace mais il faut connaître ce nombre). Chaque send, put, get et chaque opération collective ajoute un message à attendre.
- Il est interdit de mixer les 2 dans une même super-étape

Comment envoyer

Envoie

Il y a 3 méthodes :

- **void** bsp_send (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) envoi simple d'un buffer de taille size pour le processus dest ; le buffer peut être modifier après l'appel de la fonction
- **void** bsp_sendmsg (t_bsp* bsp, **int** dest, t_bspmsg* msg, **int** size) envoie un message msg. Un message est construit par la fonction t_bspmsg* bsp_createmsg (t_bsp* bsp, **int** size) et l'on obtient un pointeur sur la donnée avec **void*** bspmsg_data (t_bspmsg* msg). Il ne faut plus toucher à la zone pointé par le message.
- **void** bsp_hpsend (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) identique que bsp_send mais l'envoie est "unbuffered" c'est-à-dire qu'il ne faut pas modifier le buffer car aucune copie n'est effectuée.

Comment envoyer

Envoie

Il y a 3 méthodes :

- **void** bsp_send (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) envoi simple d'un buffer de taille size pour le processus dest ; le buffer peut être modifier après l'appel de la fonction
- **void** bsp_sendmsg (t_bsp* bsp, **int** dest, t_bspmsg* msg, **int** size) envoie un message msg. Un message est construit par la fonction t_bspmsg* bsp_createmsg (t_bsp* bsp, **int** size) et l'on obtient un pointeur sur la donnée avec **void*** bspmsg_data (t_bspmsg* msg). Il ne faut plus toucher à la zone pointé par le message.
- **void** bsp_hpsend (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) identique que bsp_send mais l'envoie est "unbuffered" c'est-à-dire qu'il ne faut pas modifier le buffer car aucune copie n'est effectuée.

Comment envoyer

Envoie

Il y a 3 méthodes :

- **void** bsp_send (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) envoi simple d'un buffer de taille size pour le processus dest ; le buffer peut être modifier après l'appel de la fonction
- **void** bsp_sendmsg (t_bsp* bsp, **int** dest, t_bspmsg* msg, **int** size) envoie un message msg. Un message est construit par la fonction t_bspmsg* bsp_createmsg (t_bsp* bsp, **int** size) et l'on obtient un pointeur sur la donnée avec **void*** bspmsg_data (t_bspmsg* msg). Il ne faut plus toucher à la zone pointé par le message.
- **void** bsp_hpsend (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) identique que bsp_send mais l'envoie est "unbuffered" c'est-à-dire qu'il ne faut pas modifier le buffer car aucune copie n'est effectuée.

Comment envoyer

Envoie

Il y a 3 méthodes :

- **void** bsp_send (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) envoi simple d'un buffer de taille size pour le processus dest ; le buffer peut être modifier après l'appel de la fonction
- **void** bsp_sendmsg (t_bsp* bsp, **int** dest, t_bspmsg* msg, **int** size) envoie un message msg. Un message est construit par la fonction t_bspmsg* bsp_createmsg (t_bsp* bsp, **int** size) et l'on obtient un pointeur sur la donnée avec **void*** bspmsg_data (t_bspmsg* msg). Il ne faut plus toucher à la zone pointé par le message.
- **void** bsp_hpsend (t_bsp* bsp, **int** dest, **void*** buffer, **int** size) identique que bsp_send mais l'envoie est "unbuffered" c'est-à-dire qu'il ne faut pas modifier le buffer car aucune copie n'est effectuée.

Comment envoyer

Que choisir ?

Encore une fois cela dépend de votre algorithme :

- s'il y a très peu de communications : bsp_send
- s'il y a beaucoup de données à envoyer à un processeur mais de manière non-contigue en mémoire : bsp_sendmsg
- si l'on n'a pas besoin de modifier le buffer avant la prochaine super-étape : bsp_hpsend (hight-performance)

Exemple

```
int i, destination, data=1;
destination = (pid+1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
data = 2;
destination = (pid-1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
bsp_sync(&bsp);
```

Comment envoyer

Que choisir ?

Encore une fois cela dépend de votre algorithme :

- s'il y a très peu de communications : bsp_send
- s'il y a beaucoup de données à envoyer à un processeur mais de manière non-contigue en mémoire : bsp_sendmsg
- si l'on n'a pas besoin de modifier le buffer avant la prochaine super-étape : bsp_hpsend (hight-performance)

Exemple

```
int i, destination, data=1;
destination = (pid+1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
data = 2;
destination = (pid-1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
bsp_sync(&bsp);
```

Comment envoyer

Que choisir ?

Encore une fois cela dépend de votre algorithme :

- s'il y a très peu de communications : bsp_send
- s'il y a beaucoup de données à envoyer à un processeur mais de manière non-contigue en mémoire : bsp_sendmsg
- si l'on n'a pas besoin de modifier le buffer avant la prochaine super-étape : bsp_hpsend (hight-performance)

Exemple

```
int i, destination, data=1;
destination = (pid+1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
data = 2;
destination = (pid-1) % nprocs;
bsp send(&bsp, destination, &data, sizeof(int));
bsp_sync(&bsp);
```

Comment envoyer

Que choisir ?

Encore une fois cela dépend de votre algorithme :

- s'il y a très peu de communications : bsp_send
- s'il y a beaucoup de données à envoyer à un processeur mais de manière non-contigue en mémoire : bsp_sendmsg
- si l'on n'a pas besoin de modifier le buffer avant la prochaine super-étape : bsp_hpsend (hight-performance)

Exemple

```
int i, destination, data=1;
destination = (pid+1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
data = 2;
destination = (pid−1) % nprocs;
bsp send(&bsp, destination, &data, sizeof(int));
bsp_sync(&bsp);
```

Comment envoyer

Que choisir ?

Encore une fois cela dépend de votre algorithme :

- s'il y a très peu de communications : bsp_send
- s'il y a beaucoup de données à envoyer à un processeur mais de manière non-contigue en mémoire : bsp_sendmsg
- si l'on n'a pas besoin de modifier le buffer avant la prochaine super-étape : bsp_hpsend (hight-performance)

Exemple

```
int i, destination, data=1;
destination = (pid+1) % nprocs;
bsp_send(&bsp, destination, &data, sizeof(int));
data = 2;
destination = (pid−1) % nprocs;
bsp send(&bsp, destination, &data, sizeof(int));
bsp_sync(&bsp);
```

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t_bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t_bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Reception des données

Après la synchronisation, les données de la précèdente super-étape sont physiquement accessibles. Pour cela, on utilise :

- **int** bsp_nmsgs (t_bsp* bsp) retourne le nombre de message reçus
- t_bspmsg* bsp_getmsg (t_bsp* bsp, **int** index) retourne le indexième message reçu (pas d'ordre spécifié)
- t_bspmsg* bsp_findmsg (t_bsp* bsp, **int** id, **int** index) retourne le indexième message provenant du processus id. Comme pour bsp_getmsg si index n'est pas un index valide, un pointeur NULL est retourné.
- **int** bspmsg_size (t_bspmsg* msg) retourne la taille du message
- **int** bspmsg_src (t bspmsg* msg) retourne la provenance du message (pid du processus source)

Recevoir des données

Exemple 1

Lire tout les messages recus :

```
t_bspmsg *msg;
int i, *data;
for (i=0; i<bsp_nmsgs(&bsp); i++) {
  msg = bsp_getmsg(&bsp, i);
  data = (int*) bspmsg_data(msg);
  /* utiliser data */ }
```

Exemple 2

Lire tout les messages recus d'un processeur :

```
t_bspmsg *msg;
int i=0; /* itérer tant que NULL n'est pas retourné */
while (msg=bsp_findmsg(&bsp, sender, i)) {
 /* utiliser msg */
 i++; }
```

Recevoir des données

Exemple 1

Lire tout les messages recus :

```
t_bspmsg *msg;
int i, *data;
for (i=0; i<bsp_nmsgs(&bsp); i++) {
   msg = bsp_getmsg(&bsp, i);
   data = (int*) bspmsg_data(msg);
   /* utiliser data */ }
```

Exemple 2

Lire tout les messages recus d'un processeur :

```
t_bspmsg *msg;
int i=0; /* itérer tant que NULL n'est pas retourné */
while (msg=bsp_findmsg(&bsp, sender, i)) {
 /* utiliser msg */
 i++; }
```

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

1. Faite attention à ce que les zones de mémoire partagées ne se chevauchent pas. Cela rend les programmes rapidement incohérents.
2. On ne peut pas dynamiquement modifier la taille d'une zone partagée, donc bien prévoir la taille.

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

1. Faite attention à ce que les zones de mémoire partagées ne se chevauchent pas. Cela rend les programmes rapidement incohérents.
2. On ne peut pas dynamiquement modifier la taille d'une zone partagée donc bien prévoir la taille
3. Dans la PUB, on n'est pas obligé de faire une sycnhronisation après les déclarations. En BSPlib standard oui. Faîtes alors attention.

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

1. Faite attention à ce que les zones de mémoire partagées ne se chevauchent pas. Cela rend les programmes rapidement incohérents.
2. On ne peut pas dynamiquement modifier la taille d'une zone partagée donc bien prévoir la taille
3. Dans la PUB, on n'est pas obligé de faire une sycnhronisation après les déclarations. En BSPlib standard oui. Faîtes alors attention.

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

1. Faite attention à ce que les zones de mémoire partagées ne se chevauchent pas. Cela rend les programmes rapidement incohérents.
2. On ne peut pas dynamiquement modifier la taille d'une zone partagée donc bien prévoir la taille
3. Dans la PUB, on n'est pas obligé de faire une sycnhronisation après les déclarations. En BSPlib standard oui. Faîtes alors attention.

Enregistrer une zone partagée

Déclaration

- **void** bsp_push_reg (t_bsp* bsp, **void*** ident, **int** size) permet de rendre partagé (on parle d'enregistrer pour un accès global) la zone mémoire pointé par ident et de taille size. Cette taille peut être différente en chaque processus. Par contre, il faut que l'ordre d'appel des enregistrement soit identique en chaque processus.
- **void** bsp_pop_reg (t_bsp* bsp, **void*** ident) permet de supprimer comme "partagée" la zone mémoire pointé par ident

Conseil

1. Faite attention à ce que les zones de mémoire partagées ne se chevauchent pas. Cela rend les programmes rapidement incohérents.
2. On ne peut pas dynamiquement modifier la taille d'une zone partagée donc bien prévoir la taille
3. Dans la PUB, on n'est pas obligé de faire une sycnhronisation après les déclarations. En BSPlib standard oui. Faîtes alors attention.

Lire et écrire dans une zone partagée

Ecrire

Cela se fait par l'intermédiaire d'une des fonction suivante :

- **void** bsp_put(t_bsp* bsp, **int** destPID, **void*** src, **void*** dest, **int** offset, **int** nbytes)
- **void** bsp_hpput(t_bsp* bsp, **int** destPID, **void*** src, **void*** dest, **int** offset, **int** nbytes)

Ces fonctions permettent de copier nbytes depuis la mémoire locale src sur la mémoire globale (partagée) dest à l'adresse offset du processeur destPID. La copie est sûre d'être complètement effectué qu'après un appel à bsp_sync ou bsp_oblsync et que tout les bsp_get aient été effectués. bsp_hpput est "unbuffered" : il ne faut donc pas modifier le buffer tant que la synchronisation n'a pas été effectuée.

Lire et écrire dans une zone partagée

Lire

Cela se fait par l'intermédiaire d'une des fonction suivante :

- **void** bsp_get(t_bsp* bsp, **int** srcPID, **void*** src, **int** offset, **void*** dest, **int** nbytes)
- **void** bsp_hpget(t_bsp* bsp, **int** srcPID, **void*** src, **int** offset, **void*** dest, **int** nbytes)

Ces fonctions permettent de copier nbytes depuis la mémoire globale src à l'adresse offset du processeur srcPID sur la mémoire locale dest. Comme précèdement, la copie n'est valide qu'après la synchronisation et bsp_hpget est "unbuffered".

Lire et écrire dans une zone partagée

Exemple

```
int array[100], privateArray[100];
src=0;
dest=0;
offset=0;
/* Enregistrer la zone "array" */
bsp_push_reg(&bsp, array, sizeof(array));
/* Calcul sur les tableaux */
...
/* chaque processeur lit le tableau "array" du processeur 0 */
bsp_get(&bsp, src, array, offset, privateArray, sizeof(array));
/* chaque processeur écrit depuis le tableau privateArray
                                        sur le tableau "array" du processeur 0 */
bsp_put(&bsp, dest, &privateArray[1], array, bsp_pid(bsp)*sizeof(int),sizeof(int));
/* Synchroniser */
bsp_sync(&bsp);
/* De-enregistrer la zone "array" */
bsp_pop_reg(&bsp, array);
```

Opérations collective

Opérations

La PUB contient 2 types d'opérations collectives :

1. Celle pour la diffusion d'une valeur d'un processeur à un intervalle ou une liste de processeur
2. calcul des préfixes (scan et reduce)

Pour le calcul des préfixes, on peut définir une fonction comme opérateur pour la réduction parallèle. Comme précèdement, on ne peut être sûr que tout les calculs (et communications) ont bien été effectués qu'après une barrière de synchronisation. Nous nous référons aux manuels pour plus de détails.

Migration de processus virtuelles et sous-groupes

Ces deux fonctionnalités non BSP ne sont pas portables (limitations dans le manuel). En plus, elle ne sont pas des plus efficace, rendent le code source des programmes plus compliqués et font perdre la prévision des performances...

Opérations collective

Opérations

La PUB contient 2 types d'opérations collectives :

1. Celle pour la diffusion d'une valeur d'un processeur à un intervalle ou une liste de processeur
2. calcul des préfixes (scan et reduce)

Pour le calcul des préfixes, on peut définir une fonction comme opérateur pour la réduction parallèle. Comme précèdement, on ne peut être sûr que tout les calculs (et communications) ont bien été effectués qu'après une barrière de synchronisation. Nous nous référons aux manuels pour plus de détails.

Migration de processus virtuelles et sous-groupes

Ces deux fonctionnalités non BSP ne sont pas portables (limitations dans le manuel). En plus, elle ne sont pas des plus efficace, rendent le code source des programmes plus compliqués et font perdre la prévision des performances...

Déroulement du cours

Programmes simples

Schémas de communications

Programmer (et définir les types) en C+PUB les fonctions suivantes :

1. diffusion simple d'un buffer
2. diffusion en 2 super-étapes d'un buffer
3. échange total de buffers

Donnez une version par passage de message et une autre par mémoire partagée. Quelle sont les versions les plus simples et les plus efficaces ?

Calcul scientifique

Programmer (avec les 2 méthodes) le calcul de pi (avec des **double**).

Programmes simples

Schémas de communications

Programmer (et définir les types) en C+PUB les fonctions suivantes :

1. diffusion simple d'un buffer
2. diffusion en 2 super-étapes d'un buffer
3. échange total de buffers

Donnez une version par passage de message et une autre par mémoire partagée. Quelle sont les versions les plus simples et les plus efficaces ?

Calcul scientifique

Programmer (avec les 2 méthodes) le calcul de pi (avec des **double**).

Tri parallèle

Le tri par échantillonage

Trier des données est un problème classique en algorithmes parallèles. Beaucoup de tris parallèles ont été proposé avec différentes complexités. Dans cet exercice, nous nous intéressons à un tri par "échantillonage" dans sa version BSP (due a Alexandre Tiskin en 1998).

Initialisation

Prenons un ensemble d'éléments $\mathcal{X}$ (par exemple les entiers ou les chaines de caractères). Nous assumons que tous les éléments de $\mathcal{X}$ sont différents. Nous notons $\langle a, b\rangle$ un interval ouvert, c'est à dire l'ensemble de tous les éléments $c \in \mathcal{X}$ tels que $a < c < b$. Nous supposons que le tableau initial x a été partitioné en p sous-tableaux $x^1, \ldots, x^p$ de taille n/p, un sous-tableau par processeur. L'algorithme par échantillonage procède comme suit.

Tri parallèle

Le tri par échantillonage

Trier des données est un problème classique en algorithmes parallèles. Beaucoup de tris parallèles ont été proposé avec différentes complexités. Dans cet exercice, nous nous intéressons à un tri par "échantillonage" dans sa version BSP (due a Alexandre Tiskin en 1998).

Initialisation

Prenons un ensemble d'éléments $\mathcal{X}$ (par exemple les entiers ou les chaines de caractères). Nous assumons que tous les éléments de $\mathcal{X}$ sont différents. Nous notons $\langle a, b \rangle$ un interval ouvert, c'est à dire l'ensemble de tous les éléments $c \in \mathcal{X}$ tels que $a < c < b$. Nous supposons que le tableau initial x a été partitioné en p sous-tableaux $x^1, \ldots, x^p$ de taille n/p, un sous-tableau par processeur. L'algorithme par échantillonage procède comme suit.

Tri parallèle

Première super-étape

Tous les sous-tableaux x^q sont triés indépendamment en chaque processeur q. Le problème consiste maintenant à réunir ces sous-tableaux pour avoir un tableau trié. En chaque processeur, $p+1$ éléments régulièrement espacés du sous-tableau sont sélectionnés et constituent l'échantillon primaire (le premier et le dernier éléments du sous-tableau sont inclus dans cet échantillon). Nous notons cet échantillon primaire du sous-tableaux x^q (au processeur q) par $\bar{x}_0^q, \ldots, \bar{x}_p^q$. Cet échantillon primaire découpe le sous-tableau x^q en p bloques primaires (de tailles n/p^2) que nous noterons $[\bar{x}_0^q, \bar{x}_1^q], \ldots, [\bar{x}_{p-1}^q, \bar{x}_p^q]$.

Première communication

Un échange total des p échantillons primaires est effectué (donc $p \times (p+1)$ éléments échangés).

Tri parallèle

Première super-étape

Tous les sous-tableaux x^q sont triés indépendament en chaque processeur q. Le problème consiste maintenant à réunir ces sous-tableaux pour avoir un tableau trié. En chaque processeur, $p+1$ éléments régulièrement espacés du sous-tableau sont sélectionnés et constituent l'échantillon primaire (le premier et le dernier éléments du sous-tableau sont inclus dans cet échantillon). Nous notons cet échantillon primaire du sous-tableaux x^q (au processeur q) par $\bar{x}_0^q, \ldots, \bar{x}_p^q$. Cet échantillon primaire découpe le sous-tableau x^q en p bloques primaires (de tailles n/p^2) que nous noterons $[\bar{x}_0^q, \bar{x}_1^q], \ldots, [\bar{x}_{p-1}^q, \bar{x}_p^q]$.

Première communication

Un échange total des p échantillons primaires est effectué (donc $p \times (p+1)$ éléments échangés).

Tri parallèle

Seconde super-étape

Chaque processeur trie l'ensemble des éléments des échantillons primaires. Nous noterons ses sous-tableaux triés (de l'ensemble des échantillons primaires) y^q. De nouveau en chaque processeur, $p+1$ éléments de y^q sont selectionés de la même manière (le premier et le dernier élements sont de nouveaux inclus) et constituent l'échantillon secondaire. Nous notons cet échantillons secondaire $\bar{\bar{x}}_0, \ldots, \bar{\bar{x}}_p$ (remarquez que cet échantillon est le même sur tous les processeurs). L'échantillon secondaire partitione les éléments de x (et pas de x^q) en p bloques secondaires qui correspondent aux intervals ouverts $\langle \bar{\bar{x}}_0, \bar{\bar{x}}_1 \rangle, \ldots, \langle \bar{\bar{x}}_{p-1}, \bar{\bar{x}}_p \rangle$. Les bloques secondaires sont distribués sur les processeurs.

Troisième super-étape

La troisième et dernière super-étape consiste à ce que chaque processeur q récupère les éléments (des autres processeurs) de l'interval ouvert $\langle \bar{\bar{x}}_q, \bar{\bar{x}}_{q+1} \rangle$ (avec $0 \leq q < p$).

Tri parallèle

Seconde super-étape

Chaque processeur trie l'ensemble des éléments des échantillons primaires. Nous noterons ses sous-tableaux triés (de l'ensemble des échantillons primaires) y^q. De nouveau en chaque processeur, $p+1$ éléments de y^q sont selectionés de la même manière (le premier et le dernier élements sont de nouveaux inclus) et constituent l'échantillon secondaire. Nous notons cet échantillons secondaire $\bar{\bar{x}}_0, \ldots, \bar{\bar{x}}_p$ (remarquez que cet échantillon est le même sur tous les processeurs). L'échantillon secondaire partitione les éléments de x (et pas de x^q) en p bloques secondaires qui correspondent aux intervals ouverts $\langle \bar{\bar{x}}_0, \bar{\bar{x}}_1 \rangle, \ldots, \langle \bar{\bar{x}}_{p-1}, \bar{\bar{x}}_p \rangle$. Les bloques secondaires sont distribués sur les processeurs.

Troisième super-étape

La troisième et dernière super-étape consiste à ce que chaque processeur q récupère les éléments (des autres processeurs) de l'interval ouvert $\langle \bar{\bar{x}}_q, \bar{\bar{x}}_{q+1} \rangle$ (avec $0 \leq q < p$).

Tri parallèle

Fonctionnement

Expliquer comment procède la dernière super-étape (où et comment récupérer les bons éléments). Justifier alors le bon fonctionement de l'algorithme (une preuve formelle n'est pas demandée). Essayer sur un exemple pour bien comprendre le fonctionnement de l'algorithme.

Implantation

Donnez une implantation C+PUB du tri d'un tableau de **float**.

Formule de coût

Vous supposerez que la complexité d'un tri d'un tableau de n élements est de $O(n \log(n))$ et celle de la séléction de n éléments est $O(n)$. Vous supposerez aussi que le tableau initial (à trier) comporte $n \geq p^3$ éléments.

Combien d'éléments (au plus) comporte un bloque secondaire. Justifier votre réponse. En déduire la formule de coût associé à l'algorithme.

Tri parallèle

Fonctionnement

Expliquer comment procède la dernière super-étape (où et comment récupérer les bons éléments). Justifier alors le bon fonctionement de l'algorithme (une preuve formelle n'est pas demandée). Essayer sur un exemple pour bien comprendre le fonctionnement de l'algorithme.

Implantation

Donnez une implantation C+PUB du tri d'un tableau de **float**.

Formule de coût

Vous supposerez que la complexité d'un tri d'un tableau de n élements est de $O(n \log(n))$ et celle de la séléction de n éléments est $O(n)$. Vous supposerez aussi que le tableau initial (à trier) comporte $n \geq p^3$ éléments.

Combien d'éléments (au plus) comporte un bloque secondaire. Justifier votre réponse. En déduire la formule de coût associé à l'algorithme.

Tri parallèle

Fonctionnement

Expliquer comment procède la dernière super-étape (où et comment récupérer les bons éléments). Justifier alors le bon fonctionement de l'algorithme (une preuve formelle n'est pas demandée). Essayer sur un exemple pour bien comprendre le fonctionnement de l'algorithme.

Implantation

Donnez une implantation C+PUB du tri d'un tableau de **float**.

Formule de coût

Vous supposerez que la complexité d'un tri d'un tableau de n élements est de $O(n \log(n))$ et celle de la séléction de n éléments est $O(n)$. Vous supposerez aussi que le tableau initial (à trier) comporte $n \geq p^3$ éléments.

Combien d'éléments (au plus) comporte un bloque secondaire. Justifier votre réponse. En déduire la formule de coût associé à l'algorithme.

Et BSML alors ?

On ne l'oublie pas ;-)

Refaire tout les précèdents exercices mais en BSML ! Mais cette fois-ci avec des données quelconques (fonctions polymorphes !)

A la semaine prochaine