Systèmes et supports d'exécution pour le calcul parallèle et distribué

Université de Bordeaux 1 — Master 2 SDRP, 2005 – 2006

Corrigé du partiel

Alexandre DENIS

23 novembre 2005

Durée : 2 h. **Documents interdits.**
Les exercices sont indépendants les uns des autres. Le barème est donné à titre indicatif. Les parties de code demandées doivent être rédigées en langage C.

1 Questions de cours : multi-threading et SMP (6 points)

1. Rappelez brièvement la définition et les propriétés essentielles de :
   (a) fibre/filament

   On appelle filaments *les différents fils d'exécution qu'une application crée et gère elle-même (changements de contexte explicites), par exemple à l'aide de* `longjmp` *ou de* `swapcontext`*. Les filaments peuvent être mis en œuvre par une bibliothèque en esapce utilisateur mais n'ont aucun support dans le noyau. Ils introduisent un* entrelacemenet *des fils d'exécution, mais* pas de concurrence *(pas de SMP).*

   (b) ordonnanceur de *threads* coopératif

   On appelle ordonnanceur coopératif *un ordonnanceur de threads qui n'eefectue des changements de contexte que lors d'appels à la bibliothèque de threads potentiellement bloquants (*`mutex_lock`*, par exemple). Ceci permet de minimiser le nombre de changement de contextes donc d'optimiser l'usage du processeur. En contrepartie, l'ordonnancement peut être inéquitable, un thread pouvant même empêcher tous les autres d'avoir la main.*

   (c) ordonnanceur de *threads* préemptif

   Par opposition à ordonnanceur coopératif, on appelle ordonnanceur préemptif *un ordonnanceur qui alloue aux threads une certaine tranche de temps et provoque un changement de contexte une fois qu'elle est écoulée, de façon préemptive (c'est-à-dire de façon « autoritaire »). Les changements de contexte sont alors rythmés par une horloge. L'équité d'un tel ordonnanceur est bien meilleure qu'un ordonnanceur coopératif, mais il est plus complexe à implémenter.*

   (d) ordonnanceurs de niveau noyau, utilisateur, hybride

Un ordonnanceur de niveau noyau *est implémenté à l'intérieur du système d'exploitation, en espace noyau, en relation étroite avec l'ordonnanceur de processus (parfois, ordonnanceur de processus et de threads sont confondus). Un ordonnanceur de niveau* utilisateur *est implémenté en espace utilisateur, sous la forme d'une bibliothèque, à l'intérieur d'un processus. Les ordonnanceurs noyau permettent l'exploitation de plusieurs processeurs mais ont un coût assez élevé ; à l'inverse, les ordonnanceurs utilisateur permettent d'exploiter un seul processeur mais sont plus performants.*
Un ordonnanceur hybride est en fait constitué d'un ordonnanceur utilisateur exploitant plusieurs threads de niveau noyau. Il permet de combiner la performance d'un ordonnanceur utilisateur avec la capacité d'utiliser plusieurs processeurs.

2. Qu'est-ce qu'un processeur *dual core* (ou *multi core*) ? Quel type d'ordonnanceur de *threads* sont adaptés à leur programmation ?

Un processeur multi-core *est un assemblage de plusieurs cœurs de processeurs sur une seule puce. On appelle* dual core *le cas particulier où il y en a deux. Du point de vue logiciel, un système multi-core apparaît comme une machine SMP classique et se programme comme tel. On les exploite donc à l'aide d'un ordonnanceur de threads de niveau noyau ou hybride.*

3. Quel est le problème avec le placement des données sur une machine NUMA ? Est-il possible que le même programme soit plus lent sur une NUMA que sur une machine monoprocesseur ?

Une NUMA est une machine à accès mémoire non-uniforme. Elle est constituée de plusieurs blocs regroupant de la mémoire et un ou plusieurs processeurs. Les processeurs peuvent accéder à la mémoire des autres processeurs, mais ces accès sont plus coûteux.
*Il en résulte qu'un mauvais placement des données en mémoire (*i.e. *mémoire et processeurs dans deux blocs différents) peut causer ce surcoût pour 100 % des accès mémoire. Dans ce cas, un programme séquentiel sera nettement plus lent sur une NUMA que sur une machine monoprocesseur.* Note : *ce problème disparaît bien entendu si données et fil d'exécution sont correctement placés.*

2 Architecture des intergiciels (6 points)

1. Qu'appelle-t-on une construction d'intergiciel par abstraction ?

Une construction d'intergiciel par abstraction est basée sur une architecture générale en couches logicielles, chaque couche apportant un niveau d'abstraction plus élevé que la couche qui lui est immédiatement inférieure.
Pour assurer la portabilité sur différentes variantes de la ressource considérée, la mise en œuvre peut utiliser plusieurs implémentations de la couche basse fournissant la même interface ; ces différentes implémentations au-dessus de différentes ressources sont appelées des pilotes*.*
Pour adapter l'intergiciel aux besoins des logiciels l'utilisant, il peut également y avoir plusieurs implémentations de la couche la plus haute ; on parle alors de personnalités*.*

2. Qu'appelle-t-on une construction d'intergiciel par composants logiciels ?

Un intergiciel construit par composants logiciels est composés de « briques » de base assemblables selon plusieurs configurations pour s'adapter aux conditions. Chaque « brique » appellée composant logiciel, est une entité totalement encaspulée, communicant avec l'extérieur au travers d'interfaces bien spécifiées. L'assemblage n'est pas fait par les composants eux-mêmes mais par une tierce partie.
Dans un intergiciel construit par composants logiciels, il n'y a pas de notion de couche logicielle, car la taille de la pile logicielle varie selon les assemblages.

3. Quels sont les avantages et les inconvénients de chacune de ces deux approches ?

L'approche par abstraction est relativement simple et permet de s'adapter à de nombreux cas ; pour gérer une nouvelle ressource, il suffit la plupart du temps d'écrire un nouveau pilote. Cependant, l'architecture générale de l'intergiciel ainsi que les API sont figées : un pilote ne peut fournir que les services attendus par la couche supérieure. Cette approche n'est donc pas adaptée quand l'intergiciel vise une portabilité vers des ressources extrêmement variées pouvant difficilement être programmées par la même API.
À l'inverse, l'approche par composants logiciels est plus complexe à mettre en œuvre, en particulier le cadre logiciel qui gère l'assemblage des composants. Cependant, ce type d'architecture permet une flexibilité maximale pour gérer des ressources très différentes, nécessitant des services (nombre de couches) différenciés, et permet même d'ajouter facilement par la suite des composants dotés d'API qui n'étaient pas connues au moment de la conception de l'intergiciel.

4. *Globus XIO* est la bibliothèque de communication de l'intergiciel pour grilles *Globus*. *Globus XIO* est bâti autour d'une interface de programmation simple composée de quatre fonctions (`xio_open`, `xio_close`, `xio_read`, et `xio_write`) et repose sur deux types d'entités :
   – des *pilotes*, qui implémentent l'API *Globus XIO* sur diverses ressources physiques ;
   – des *filtres*, qui offrent l'API *Globus XIO* et utilisent cette même API *Globus XIO*.
   Les entrées/sorties (fichiers, réseaux) réalisées avec *Globus XIO* se font donc au travers d'une pile XIO composée d'un nombre variable de *filtres* (éventuellement zéro) superposés au-dessus d'un pilote (toujours exactement un et un seul pilote, toujours en bas de pile). Les applications assemblent elles-mêmes la pile qu'elles désirent utiliser.

   (a) Dans quelle catégorie d'intergiciel classeriez-vous *Globus XIO* ? Justifiez.

   De part son approche basée sur des entités encapsulées et composables, Globus XIO est essentiellement un intergiciel construit par composants logiciels. Cependant, le cadre est un peu restreint en n'autorisant pas tout type de composant et tout type d'assemblage. Certaines simplifications ont été faites dans Globus XIO par rapport aux approches « classiques » de composants logiciels.

   (b) Quels sont les avantages de cette approche ?

   Cette approche par composants restreints *permet de combiner en grande partie la flexibilité d'une approche par composants (nombre de couches variable) avec la simplicité de la plate-forme (une seule API !) et une simplicité d'écriture des pilote similaire à ce qui existe dans une approche par abstraction.*

   (c) Quelles sont les limitations de cette approche ? Donnez au moins un exemple de scénario difficile à faire rentrer dans l'architecture prévue par *Globus XIO*.

   Cette approche présente deux limitations majeures :
   – *toutes les opérations et ressources doivent être exprimables dans l'API unique Globus XIO ;*
   – *tous les assemblages doivent être* linéaires*, e.g. un filtre ne peut pas utiliser deux pilotes.*
   On peut citer en exemple ne rentrant pas dans le cadre de Globus XIO :
   – *le* channel bonding *(ou* trunking*) qui consiste à répartir les communications sur deux réseaux quand deux routes sont possibles (2 pilotes) ;*
   – *les* passerelles *entre deux réseaux (2 pilotes) ;*
   – *l'*épissure TCP *(`connect` simultané) qui a besoin d'un canal de communication auxiliaire (assemblage pas linéaire) ;*
   – *les schémas à base de* socket factory *(2 pilotes : un pour la connexion, un pour les communications) ;*
   – *les composants d'optimisation à la volée ou de sélection dynamique d'assemblage (assemblage pas linéaire) ;*
   – *l'exploitation efficace de réseaux haute performance (zéro copie difficile, l'API ne permet pas d'exprimer les opérations collextives).*

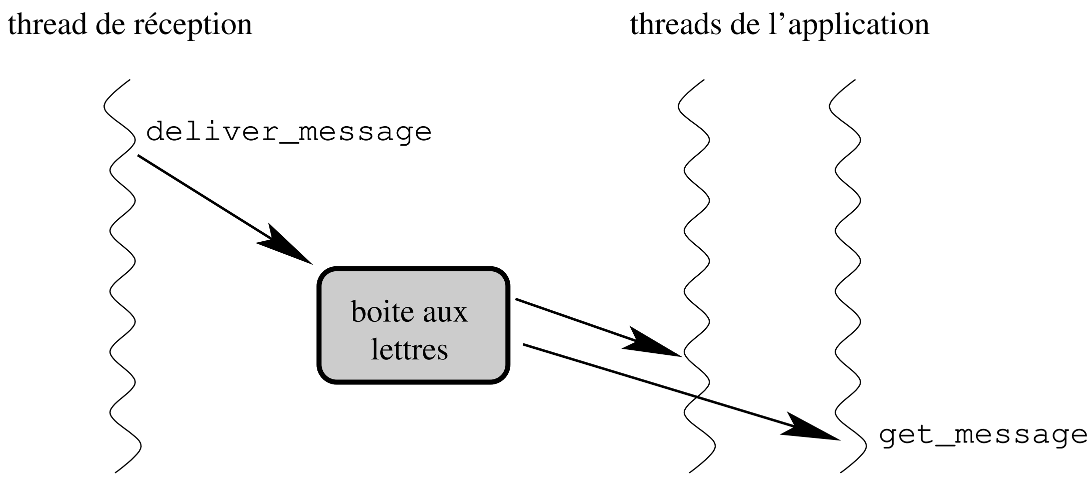
(d) Comment *Globus XIO* peut-il réussir à programmer des réseaux avec pour seul primitive d'ouverture `xio_open` ? Essayez de deviner le prototype de cette fonction et les implications sur l'implémentation.

Pour gérer les connexions réseaux au travers d'une interface conçue essentiellement pour les fichiers, l'établissement d'une connexion se fait en passant une URL *à* `xio_open`*. Les paramètres de connexion (addresse, port, etc.) sont encodés sous forme de chaine de caractère dans cette* URL.

3 Problème : synchronisation — boîte aux lettres (8 points)

Nous nous plaçons dans le cadre d'un intergiciel qui, pour certaines raisons techniques, réalise toutes ses réceptions réseau dans le même *thread* interne à l'intergiciel ; l'application qui utilise cet intergiciel dispose de ses propres *threads*. L'objectif de cette partie est d'étudier le mécanisme de *boîte aux lettres* utilisé pour faire passer les données du *thread* de réception aux *threads* de l'application.

En pratique, le *thread* de réception est une boucle qui effectue la réception réseau puis appelle la fonction :
`deliver_message``(`*`int`*` size, `*`void`*`*buffer);`
en passant en paramètre la taille du message reçu et un pointeur vers les données reçues. Les *threads* applicatifs récupèrent les données dans la boîte aux lettres en appelant la fonction :
`get_message``(`*`int`*`*size, `*`void`*`**buffer);`



3.1 Boîte aux lettres simple

Dans un premier temps, nous supposons que la boîte aux lettres ne peut contenir qu'un seul message et qu'elle est toujours vide quand un message arrive depuis le réseau (*i.e.* quand `deliver_message` est appelée).

1. Écrire la structure de données représentant une boîte aux lettres.

```
struct bal_s
{
  int size;
  void*buffer;
  pthread_mutex_t m; /* verrou d'accès à la boîte aux lettres */
  pthread_cond_t c;  /* attente quand la boîte est vide */
} bal;
```

2. Écrire les fonctions `deliver_message` et `get_message`.

```
void deliver_message(int size, void*buffer)
{
  pthread_mutex_lock(&bal.m);
  bal.size   = size;
  bal.buffer = buffer;
  pthread_cond_signal(&bal.c);
  pthread_mutex_unlock(&bal.m);
}
```

```
void get_message(int*size, void**buffer)
{
  pthread_mutex_lock(&bal.m);
  while(bal.buffer == NULL)
    pthread_cond_wait(&bal.c, &bal.m);
  *size   = bal.size;
  *buffer = bal.buffer;
  pthread_mutex_unlock(&bal.m);
}
```

3. Que se passe-t-il si plusieurs *threads* appellent `get_message` en même temps ?

Le mutex *va sérialiser les différents appels. Si la boîte est vide, chaque thread va se mettre en attente sur la condition. Quand un message arrive, l'un des threads sera réveillé, mais on n'a aucune garantie que ce sera le premier arrivé qui sera le premier servi. Au contraire, si la boîte aux lettre est pleine, le premier thread qui obtient le mutex aura le message.*

3.2 Boîte aux lettres simple avec attente

Nous supposons maintenant qu'un message peut arriver sur le réseau alors que la boîte aux lettres n'est pas vide (*i.e.* `deliver_message` est appelé alors qu'aucun `get_message` n'a encore vidé la boîte aux lettres).

1. Comment gère-t-on ce cas ? Quel est l'impact éventuel sur les performances ?

Quand est message arrive est que la boîte aux lettres est pleine, il faudra provoquer une attente dans `deliver_message`*, ce qui provoque une attente dans le thread réseau. Selon le protocole réseau, des paquets peuvent éventuellement être perdus. De plus, des messages qui auraient pu être reçus immédiatement sont différés, ce qui fait que les performances sont moins bonnes que si on avait pu recevoir et stocker tous les messages au fur et à mesure. En contrepartie, on n'a pas besoin de gérer le difficile problème du contrôle de flux.*

2. Proposer une nouvelle version de votre code qui gère ce cas.

```
void deliver_message(int size, void*buffer)
{
  pthread_mutex_lock(&bal.m);
  while(bal.buffer != NULL)
    pthread_cond_wait(&bal.c, &bal.m);
  bal.size   = size;
  bal.buffer = buffer;
  pthread_cond_signal(&bal.c);
  pthread_mutex_unlock(&bal.m);
}
```

```
void get_message(int*size, void**buffer)
{
  pthread_mutex_lock(&bal.m);
  while(bal.buffer == NULL)
    pthread_cond_wait(&bal.c, &bal.m);
  *size   = bal.size;
  *buffer = bal.buffer;
  pthread_cond_broadcast(&bal.c);
  pthread_mutex_unlock(&bal.m);
}
```

3.3 Boîte aux lettres complète

Nous nous souhaitons maintenant pouvoir placer plusieurs messages en attente dans la boîte aux lettres.

1. Quel est l'intérêt d'avoir une boîte aux lettre pouvant contenir plusieurs messages ?

Avec une boîte aux lettres pouvant contenir plusieurs messages, le thread de réception peut recevoir les données au fur et à mesure qu'elles arrivent du réseau, même si aucun thread applicatif ne les attend encore. De cette façon, quand les threads applicatifs reçoivent les données, elles sont déjà là ce qui améliore la performance. Si la boîte aux lettres peut contenir plusieurs messages, alors les données peuvent être reçues en avance pour plusieurs threads ou pour plusieurs appels successifs à `gget_message` *d'un thread.*

2. Donner une idée des modifications à apporter à la version précédente (sans tout ré-écrire).

En utilisant le `list.h` *donné en TP :*

```
LIST_TYPE(msg,
          int size;
          void*buffer;
         ); /* définit msg_t et msg_list_t */
struct bal_s
{
  msg_list_t list;
  pthread_mutex_t m;
  pthread_cond_t c;
} bal;
```

```
void deliver_message(int size, void*buffer)
{
  pthread_mutex_lock(&bal.m);
  msg_t msg = msg_new();
  msg->size   = size;
  msg->buffer = buffer;
  msg_list_push_back(bal.list, msg);
  pthread_cond_signal(&bal.c);
  pthread_mutex_unlock(&bal.m);
}
```

```
void get_message(int*size, void**buffer)
{
  pthread_mutex_lock(&bal.m);
  while(msg_list_empty(bal.list))
    pthread_cond_wait(&bal.c, &bal.m);
  msg_t msg = msg_list_pop_front(bal.list);
  *size   = msg->size;
  *buffer = msg->buffer;
  msg_delete(msg);
  pthread_mutex_unlock(&bal.m);
}
```

Cette solution est incomplète car elle ne met pas en œuvre de contrôle de flux. *En effet, l'algorithme proposé reçoit les données en provenance du réseau et les stocke en mémoire, sans borner la taille des données en attente. En cas de mauvaise programmation ou d'utilisateur malveillant (ou les deux !) il y a un risque de consommer toute la RAM pour ce qui n'aurait dû être qu'un* cache.

3.4 Zéro copie (bonus)

Les schémas précédents introduisent une copie en mémoire des données reçues. Dans la réalité, ces copies sont très couteuses. Il est possible de les éviter dans les cas où le `get_message` a lieu avant le `deliver_message`.

1. Quel sont les principaux problèmes pour pouvoir recevoir les données en zéro copie ?

   Pour recevoir en zéro copie, il faut que le thread de réception connaisse directement l'adresse finale des données. Il faut donc renverser le sens de la boîte aux lettres : les threads applicatifs postent la taille des données attendues et le pointeur où ranger les données ; le thread de réception les reçoit directement en place.
   Un autre problème important provient du fait que pour recevoir directement en place, il faut que la taille des données qui arrivent sur le réseau soit égale à la taille des données attendues.

2. Quelles modifications faut-il apporter au code de gestion des boîtes aux lettres et au *thread* de réception ?

   Il faut à peu pprès tout réécrire, y compris en dehors *de la fonction* `deliver_message` *dans le thread de réception...*