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

Université de Bordeaux 1 — Master 2 SDRP

Examen 2004–2005

Alexandre DENIS

26 janvier 2005

Durée : 1 h 30. Documents autorisés.
Les exercices sont indépendants les uns des autres. Le barème est donné à titre indicatif. Vous rédigerez le code en langage C.

1 Réseaux haute performance et multi-threading (5 points)

1. Quelles sont les différences de conception entre les réseaux haute performance et les réseaux locaux (de type *Ethernet*) ?
2. Quelles sont les différences de conception entre les bibliothèques de *threads* pour usage général et pour le calcul parallèle ?
3. Rappelez brièvement l'architecture du réseau *Myrinet*. Expliquez, éventuellement en vous aidant d'un schéma, ce qui se passe lors de l'émission et de la réception d'un message sur *Myrinet*.
4. Quelle est la différence entre latence et réactivité ?
5. Quelles sont les solutions mises en œuvre pour améliorer la réactivité ?

2 Communications en milieu hétérogène (5 points)

Différentes architectures de machines ont différentes représentations des données. Nous considérons dans cet exercice le seul critère de l'ordre des octets : *big endian* (poids fort en premier, poids faible en dernier) et *little endian* (ordre inverse). Nous étudions l'impact des conversions sur les performances. Il existe trois approches utilisées couramment :

XDR — on convertit toujours les données en *big endian* avant de les envoyer. En pratique, seules les machines *little endian* font une conversion avant d'émettre ou après avoir reçu les données ; pour les autres, une simple copie est faite. C'est l'approche utilisée par les RPC de *Sun* et les RMI Java.

CDR — on envoie les données telles quelles avec un flag disant dans quel ordre elles sont. Le récepteur convertit seulement si nécessaire, et reçoit directement en place (et ne fait pas de copie) sinon. C'est l'approche retenue par CORBA.

XML — les données sont converties en XML (i.e. les entiers sont représentés sous leur format ASCII avec `printf`) puis parsées par le récepteur. C'est l'approche retenue par SOAP.

1. Calculez le débit de transmission apparent sur une grappe de PC équipée du réseau *Quadrics QsNet-II*, pour chacun des encodages suivants : codage XDR, codage CDR, et codage XML, dans le cas où émetteur et récepteur ont une représentation *little endian*.
2. Même question dans le cas où l'émetteur est *big endian* (exemple : *Apple G5*) et le récepteur *little endian* (exemple : *Intel Pentium-4*).
3. Discutez des avantages et inconvénients relatifs des différents encodages dans le cadre d'une utilisation sur un réseau haute performance.
4. Calculez le débit de transmission apparent sur un réseau longue distance pour chacun des encodages suivants : codage XDR, codage CDR, et codage XML, dans le cas où émetteur et récepteur ont une représentation *little endian*.
5. Même question dans le cas où l'émetteur est *big endian* et le récepteur *little endian*.
6. Discutez des avantages et inconvénients relatifs des différents encodages dans le cadre d'une utilisation sur réseau longue distance.
7. Quel vous paraît le meilleur compromis pour une utilisation sur une grille de calcul ?

Données numériques

- copie en mémoire : débit 2.4 Go/s.
- copie en intervertissant l'ordre des octets : débit 1.2 Go/s.
- après encodage en XML, les données sont en moyenne 8 fois plus volumineuses.
- l'encodage XML s'effectue à 20 Mo/s (par rapport à la taille d'origine).
- le parseur XML, utilisé pour le décodage XML, parse les données à 4 Mo/s (par rapport à la taille encodée).
- réseau *Quadrics QsNet-II* : débit nominal 1.1 Go/s, latence 850 ns.
- réseau longue distance : débit 20 Mbit/s, latence 20 ms.

Attention aux unités !

3 Problème : *Active Messages* (10 points)

La bibliothèque de communication « *Active Messages* » a été développée à l'université de Berkeley dans les années 90. Elle a été portée sur un grand nombre de réseaux et largement utilisée. Comme son nom l'indique, elle est basée sur la notion de *message actif*, c'est-à-dire que la réception est implicite, réalisée par un *thread* interne de *Active Messages*. L'utilisateur enregistre son traitant (« *handler* » en anglais) ; *Active Messages* appelle le traitant pour chacun des messages reçus. Pour *Active Messages*, les messages sont constitués d'un en-tête d'exactement 32 octets, suivi d'un corps de message de taille variable. Un guide d'utilisation d'*Active Messages* est donné en annexe A.

Tout au long du problème, pour les courbes, on supposera que l'on utilise un réseau *Myrinet*, de débit nominal 250 Mo/s et de latence 4 μs.

3.1 Passage de messages sur *Active Messages*

1. Écrivez une fonction `my_send` qui réalise l'envoi d'un bloc de données. Elle aura le prototype suivant :
   `my_send(int channel, int dest, void*ptr, int len);`
   où :

   `channel` est le canal à utiliser ;

   `dest` est le numéro du nœud destinataire ;

   `ptr` est un pointeur vers les données à envoyer ;

   `len` est la longueur des données à envoyer.
2. Écrivez une fonction `my_recv` qui réalise la réception explicite d'un bloc de données. Elle aura le prototype suivant :
   `my_send(int channel, void*ptr, int len);`
   où :

   `channel` est le canal à utiliser ;

   `ptr` est un pointeur vers une zone mémoire où recevoir les données ;

   `len` est la longueur du message à recevoir.

   Remarque : quand un message arrive alors qu'il n'était pas attendu (pas de `my_recv` correspondant), il suffit d'attendre le `my_recv`.
3. Tracer l'allure de la courbe de débit (débit de transmission en fonction de la taille des messages) obtenue par votre implémentation ainsi que la courbe de temps (temps de transmission en fonction de la taille des messages) pour de petits messages.
4. Quelle optimisation pouvez-vous proposer ? Donner une idée du principe et des modifications du code, sans écrire intégralement le code correspondant.
5. Tracer l'allure des courbes de débit et temps de transmission pour la version optimisée. Qu'a-t-on gagné ?

3.2 Messages de taille arbitraire

En réalité, *Active Messages* permet d'envoyer des messages de taille maximum 8 ko (8192 octets). Pour envoyer des messages plus volumineux, c'est à vous de fragmenter les messages en segments de taille inférieure ou égale à 8 ko et de les réassembler.

1. Quel est la durée réelle d'envoi d'un message de 8 ko sur *Myrinet* ? À quel débit cela correspond-il ?
2. Y a-t-il possibilité de *pipeline* ? Justifiez à l'aide d'un schéma commenté.
3. Écrivez les fonctions
   `my_send_any(int channel, int dest, void*ptr, int len);`
   et `my_send_any(int channel, void*ptr, int len);`
   capables d'envoyer et de recevoir des messages de longueur arbitraire. Vos fonctions doivent notamment fragmenter puis réassembler les messages, en respectant si possible le schéma donné à la question précédente.
4. Tracer l'allure des courbes de débit et temps de transmission en fonction de la taille des messages. Justifiez.

A Guide de référence d'*Active Messages* 2.0

`int AM_Init();`
Initialise la bibliothèque *Active Messages*. Cette fonction doit être appelée sur tous les nœuds. La valeur de retour vaut 0 en cas de succès, -1 en cas d'erreur.

`int AM_MaxChannels(void);`
Retourne le nombre de canaux (selon la même signification que les canaux *Madeleine*) utilisables. Vaut habituellement 4 sur *Myrinet*.

`typedef void (*AM_Function)(int channel, int source, void*h, void*token);`
Type des fonctions définies par l'utilisateur et appelées par *Active Messages* lors de la réception d'un message. Les arguments d'un telle fonction sont :

`channel` — numéro du canal où le message est arrivé ;

`source` — numéro du nœud source qui a envoyé le message ;

`h` — pointeur vers l'en-tête reçu, de taille 32 octets ;

`token` — jeton à donner à `AM_Get` pour recevoir le corps de message.

`AM_SetHandler(int channel, AM_Function function);`
Déclare une fonction traitant les message sur un canal donné.

`channel` — numéro de canal pour lequel le traitant est enregistré ;

`function` — pointeur sur la fonction à appeler pour gérer la réception des messages.

Suite à un appel à `AM_SetHanlder`, à chaque message reçu sur le canal, la fonction enregistrée sera appelée par *Active Messages*. **Attention** : *Active Messages* a son propre *thread* réseau. Pensez à protéger l'accès aux variables partagées entre les traitants et le code utilisateur. Pour un canal donné, *Active Messages* n'appelle qu'un seul traitant à la fois (les messages ne se doublent pas).

`AM_Get(void*token, void*ptr, int len);`
Fonction à utiliser à l'intérieur d'un traitant pour récupérer les données du corps du message. Il est invalide d'utiliser `AM_Get` en dehors d'un traitant.

`token` — le jeton donné au traitant par *Active Messages*. C'est un objet opaque ; il suffit de le donner tel qu'il a été reçu par le traitant.

`ptr` — pointeur vers la zone mémoire où recevoir les données ;

`len` — longueur (en octets) des données à recevoir.

`AM_Request(int channel, int dest, void*h, void*ptr, int len);`
Envoie un message composé d'un en-tête et d'un corps de message.

`channel` — numéro du canal où envoyer le message ;

`dest` — numéro de nœud du destinataire ;

`h` — poiteur vers l'en-tête à envoyer, de taille fixe égale à 32 octets ;

`ptr` — pointeur vers le corps de message ;

`len` — longeur des données à envoyer (données pointées par `ptr`, sans compter l'en-tête).

Note : il est possible d'envoyer seulement un en-tête avec `ptr=NULL` et `len=0`.

`AM_Terminate(void);`
Termine un programme *Active Messages*. Cette fonction réalise une barrière de synchronisation pour s'assurer qu'aucun message n'a été perdu.