Maîtrise d'informatique — Examen

Programmation des Architectures Parallèles

Durée : 3h — Notes de cours autorisées

1 MPI (échauffement)

Question 1 Quelles sont les différences entre une opération Send, Isend et Ssend ? (on pourra s'aider de croquis)

Question 2 Du point de vue du déploiement et de la gestion des processus, quel est le principal défaut (manque) de MPI ?

2 Réseaux à capacité d'adressage

Question 1 Rappelez le principe de fonctionnement d'un réseau à capacité d'adressage tel que SCI. Expliquez par quel mécanisme les accès à distance peuvent être rendus transparents pour le processeur.

Question 2 Cette transparence a-t-elle un impact sur les performances des transmissions de données ? Expliquez pourquoi et, le cas échéant, indiquez comment garantir l'obtention de bonnes performances (en terme de débit) sur ce type de réseau.

Question 3 Deux processus P_1 et P_2 (situés sur des machines distinctes) peuvent utiliser un réseau à capacité d'adressage pour communiquer. Le processus P_1 souhaite copier dans son espace d'adressage des données se trouvant dans la mémoire de P_2 , alors que P_2 est en train d'effectuer un traitement quelconque. On suppose que P_2 autorise n'importe quel processus à projeter à distance cette zone mémoire. On suppose également que la bibliothèque de communication fournit une primitive permettant de déclencher un signal à distance (via le réseau) au sein d'un processus "victime" pour le forcer à exécuter du code situé dans un traitant de signal.

Donnez deux façons possibles pour P_1 d'effectuer cette copie mémoire. Pour chacune, discutez de l'efficacité de la transmission, ainsi que de l'occupation processeur occasionnée par P_1 et P_2 .

3 Implantation de l'envoi de messages

On s'intéresse à l'implantation d'une bibliothèque d'envoi de messages (telle que MPI) au-dessus d'une interface de communication de bas niveau n'offrant que des fonctionnalités d'accès mémoire à distance à l'aide de deux primitives :

```
void remote_read(void *local_addr, unsigned proc, void *remote_addr, unsigned len);  
et
```

```
void remote_write(void *local_addr, unsigned proc, void *remote_addr, unsigned len);
```

La primitive `remote_read` permet de réaliser une copie mémoire de `len` octets depuis le processus distant numéro `proc` (à l'adresse `remote_addr`) vers le processus courant à l'adresse `local_addr`.

La primitive `remote_write` permet de réaliser une copie mémoire de `len` octets depuis le processus courant (à l'adresse `local_addr`) vers le processus distant numéro `proc` à l'adresse `remote_addr`.

On suppose que lorsque le paramètre `len` est inférieur ou égal à `sizeof(int)`, alors ces opérations sont atomiques du point de vue de la transaction mémoire effectuée sur la machine `proc`.

On s'intéresse dans un premier temps à la transmissions de messages de taille bornée (inférieure ou égale à `MAX_SMALL`). L'idée est alors d'utiliser, sur chaque processus, des zones de mémoire particulières (de taille `SEGMENT_SIZE`, que l'on peut supposer très supérieure à `MAX_SMALL`) qui serviront de "boîtes aux lettres" dans lesquelles les messages lui étant destinés seront recopiés.

Pour simplifier, on considère que chaque paire de processus dans le système (P_i, P_j) utilise deux zones mémoires privées (c'est à dire non utilisées par un autre couple) pour les communications entre P_i et P_j . L'une de ces zones appartient à l'espace d'adressage de P_i et contiendra les messages déposés par P_j , et réciproquement pour l'autre zone.

Chaque processus possède des tableaux de pointeurs lui permettant de connaître les adresses des zones qui lui sont utiles. Au sein d'un processus P_i , le pointeur `local[j]` désigne l'adresse de la zone de mémoire locale dans laquelle P_j déposera des messages. De même, le pointeur `remote[k]` désigne l'adresse de la zone de mémoire distante (au sein du processus P_k) dans laquelle il faudra recopier les messages à destination de P_k .

Question 1 On souhaite implanter les fonctions suivantes en utilisant les primitives `remote_read` et `remote_write`.

```
void send_small(unsigned to, void *data, unsigned len);
void recv_small(unsigned from, void *data, unsigned *len);
```

La primitive `send_small` dépose un message de taille `len` ($\leq \text{MAX_SMALL}$) dans la boîte aux lettres du processus `to`. L'opération est terminée lorsque le message est intégralement recopié (on n'attend pas que le récepteur le consomme).

La primitive `recv_small` attend un message de la part du processus `from` (on ne demande pas d'éviter l'attente active). Lorsque le message est disponible, il doit être recopié à l'emplacement pointé par `data`. La taille du message est un paramètre de sortie : elle est rangée dans `*len`.

Pour cette question, on suppose qu'un processus n'enverra jamais deux messages vers un même destinataire avant que le premier n'ait été consommé. Autrement dit, les boîtes aux lettres ne contiendront toujours qu'un message au maximum.

Donnez le code des primitives `send_small` et `recv_small` et expliquez (à l'aide d'un petit dessin) comment vous avez décidé de structurer et de gérer les zones de mémoire faisant office de boîtes aux lettres.

NB : Attention à bien éviter que l'attente dans la fonction `recv_small` n'occasionne pas trop de trafic sur le réseau.

Question 2 On se place maintenant dans le cas plus réaliste où un processus peut envoyer des messages "en rafale" vers un autre processus sans que celui-ci les consomme au fur et à mesure. En supposant que la constante `SEGMENT_SIZE` est si grande qu'aucun risque de débordement des boîtes aux lettres n'est à craindre, donnez une nouvelle version des primitives `send_small` et `recv_small` et expliquez la nouvelle organisation des zones mémoire.

Question 3 Dans la vraie vie, il faut bien sûr assurer un contrôle de flux pour éviter la saturation des boîtes aux lettres. On décide donc que la primitive `send_small` est bloquante lorsqu'il ne reste pas assez de place dans la boîte aux lettres de destination pour contenir l'intégralité du message à déposer.

Proposez une solution pour implanter ce contrôle de flux en prenant soin d'éviter un fort trafic réseau lorsqu'une opération `send_small` est bloquée.

Donnez la nouvelle version des primitives `send_small` et `recv_small`.

Question 4 On souhaite maintenant pouvoir envoyer des messages de grande taille ($> \text{MAX_SMALL}$) en évitant leur recopie dans la boîte aux lettres du récepteur. Pour cela, on accepte que l'opération `send_big` (mêmes paramètres que `send_small`) soit bloquante tant que l'opération `recv_big` correspondante n'est pas exécutée. Ainsi, il est possible de placer le message directement au bon emplacement mémoire dans le processus.

Rappelez pour quelles raisons il est préférable/nécessaire d'éviter les recopies pour les gros messages ? Pourquoi ne décide-t-on pas de basculer sur ce mode de fonctionnement uniquement pour les messages de taille supérieure à `SEGMENT_SIZE` ?

Question 5 Donnez le code des primitives `send_big` et `recv_big`. Si possible, appuyez-vous sur les primitives `send_small` et `recv_small`.

Question 6 Serait-ce envisageable de n'utiliser qu'une "boîte aux lettres" par récepteur (utilisées par tous les émetteurs) ? Expliquez précisément quelle serait la principale difficulté à résoudre et proposez une solution (on ne demande pas le code).

4 Équilibrage de charge sur grappes de PC

Nous avons vu avec l'application PovRay/PM2 qu'une façon d'équilibrer la charge de manière dynamique sur une grappe de PC était de répartir uniformément toutes les tâches (sous forme de threads) au départ, puis d'utiliser la migration de threads pour corriger les déséquilibres qui apparaissent au fil du temps.

Une autre méthode serait de gérer un *pool* de tâches en attente sur le processus maître et de les distribuer au fur et à mesure aux processus lorsqu'ils n'ont rien à faire.

Question 1 Discutez des avantages/inconvénients des deux méthodes. Par exemple si par miracle aucune migration ne s'avère finalement nécessaire au cours de l'exécution d'un programme, qu'a-t-on gagné en utilisant la seconde méthode ?

Question 2 En supposant que l'opération de migration soit très coûteuse, on pourrait penser que la première méthode n'est jamais avantageuse. Cherchez un exemple "fictif" (quelques processeurs et quelques tâches en précisant leur durée) qui montre pourtant que la solution avec migration peut parfois être meilleure... (pensez au fait que le système est préemptif, et donc que les exécutions des tâches sont entrelacées dans la première solution.)