TP 6 : Programmation `verbs` sur réseau *Infiniband*
Protocoles bas niveau pour *Infiniband*

Alexandre DENIS
denis@labri.fr

Systèmes et supports d'exécution — 2008–2009

L'objectif de ce TP est de découvrir le réseau haute performance *Infiniband* à travers son interface de programmation bas niveau *verbs*. Vous implémenterez un mécanisme de contrôle de flux et de *multiplexage* des communications réseau permettant à plusieurs *threads* de communiquer simultanément sur un seul canal partagé.

1 Présentation du réseau *Infiniband*

Le réseau *Infiniband* est un réseau haute performance ayant pour but une extrême flexibilité. Son objectif est de couvrir un spectre d'échelles très large, de la liaison entre une machine et son disque jusqu'aux liaisons internationales. *Infiniband* est un standard, c'est-à-dire qu'il existe des cartes *Infiniband* de plusieurs constructeurs et qu'elles sont interopérables.

1.1 Infrastructures

Les machines du CREMI équipées de cartes *Infiniband* sont les machines `infini1` à `infini4` ainsi que `hagrid`. Les cartes *Infiniband* du CREMI sont des cartes PCI-e Silverstorm basées sur le chipset Mellanox InfiniHost III Ex (MT25208). Les performances nominales sont un débit de 20 Gbit/s (sur le fil ; $\approx$ 12 Gbit/s exploitables en pratique) et une latence de 4 μs.

Les pilotes *Infiniband* sont installés dans `/usr/local/` qui doit être théoriquement déjà présent dans votre `PATH`. Pour vérifier que tout fonctionne, lancez la commande `ibv_devices` qui doit renvoyer une sortie du genre :

```
infini1% ibv_devices
    device                 node GUID
    ------              ----------------
    mthca0              0002c9020021704c
```

1.2 Programmation des réseaux *Infiniband*

Les cartes *Infiniband* s'utilisent au travers d'une pile logicielle appelée *OFED* (OpenFabrics Enterprise Distribution, anciennement *OpenIB*) qui comprend un module noyau et une bibliothèques de niveau utilisateur. Plusieurs interfaces d'accès sont offertes par *OFED* :

verbs — les commandes élémentaires envoyées à la carte *Infiniband* s'appellent des *verbs* (voir la norme – 1700 pages ! – donnée sur la page web du cours). En espace utilisateur, on y accède par la bibliothèque `ibverbs`.

DAPL — DAPL (Direct Access Programming Library) est une interface de plus haut niveau, censée être portable sur d'autres réseaux que *Infiniband*. Il existe *kDAPL* en espace noyau et *uDAPL* en espace utilisateur.

SDP — SDP (Socket Direct Protocol) est une émulation *socket* sur *Infiniband*.

SRP — SRP (SCSI RDMA Protocol) est un protocole d'accès à des disques SCSI distants par *Infiniband*.

IPoIB — IPoIB est une encapsulation de paquets IP sur un réseau *Infiniband*.

Tous ces protocoles de plus haut nieau, appelés ULP (Upper Level Protocols) sont construits au-dessus des *verbs*. Dans ce TP, dont l'objectif est de manipuler les opérations RDMA à bas niveau, nous utiliserons directement les *verbs*.

1.3 Lexique *Infiniband*

Le réseau *Infiniband* n'étant pas construit autour des standards Ethernet et IP, le vocabulairee utilisé est légèrement différent de ce que l'on connaît habituellement et utilise un grand nombre d'acronymes.

Général

HCA — Host Channel Adapter. Une carte *Infiniband* ou une puce *Infiniband* intégrée à une carte mère, au chipset, à un disque dur.

Port — Un connecteur *Infiniband*. Les cartes *Infiniband* du CREMI sont *dual port*, c'est-à-dire qu'elles ont deux connecteurs. Un seul (le premier) est connecté.

PD — Protection Domain. Connexion entre un HCA *Infiniband* et les mécanismes de mémoire virtuelle de l'OS.

MR — Memory Region. Une zone de mémoire enregistrée dans un PD.

Connexion

CQ — Completion Queue. Une file de requêtes asynchrones (émission ou réception). Par extension, une connexion unidirectionnelle.

QP — Queue Pair. Une paire de CQ (une en émission, une en réception). Par extension, une connexion bidirectionnelle.

QPN — Queue Pair Number (32 bits). Le numéro qui identifie une QP (à la façon d'un numéro de port TCP) sur un port donné.

PSN — Packet Sequence Number (32 bits). Numéro de séquence d'un paquet voyageant via une QP. De la même façon que TCP, le paquet initial d'une connexion n'a pas le PSN 0 mais un nombre aléatoire.

RC, UC, UD — Reliable Connection, Unreliable Connection, Unreliable Datagram. Modes de connexion (et donc : protocole) d'une QP.

Adressage

subnet — Un sous-réseau. L'ensemble des HCA ayant une connexion directe (par un *switch*) et dépendant du même domaine d'administration.

LID — Local ID (16 bits). Numéro identifiant un port dans un subnet.

GUID — Globally Unique ID (64 bits). Numéro identifiant de façon unique un HCA (à la façon des adresses MAC en Ethernet) attribué par le constructeur. C'est ce numéro qui est affiché par la commande `ibv_devices`.

GID — Group ID (64 bits). Numéro unique identifiant un subnet. Souvent : le GUID du switch.

Communications

RDMA — Remote Direct Memory Access. Accès direct à la mémoire distante (voir ci-dessous).

WR — Work Request. Une requête d'opération soumise à la carte *Infiniband*.

WC — Work Completion. Un élément d'une CQ, indiquant la terminaison (ou non) d'une WR.

Dans ce TP, nous ne manipulerons ni GUID ni GID ; toutes les communications restant sur le réseau *Infiniband* du CREMI, le LID suffit. Le code d'initialisation correspondant à la construction de PD, MR, CQ et QP vous sera donné. Le mode de connexion sera RC.

1.4 Utilisation pas à pas du réseau *Infiniband*

Notions de RDMA

Le réseau *Infiniband* utilise la notion de RDMA (*Remote Direct Memory Access*) ce qui signifie que les cartes accèdent à la mémoire des noeuds distants. Les accès en lecture ou en écriture sont possibles. Pour des raisons de performance, on préfèrera les accès en écriture (envoi simple, alors que la lecture nécessite un aller-retour). Le PD représente l'ensemble des MR, les zones accessibles à distance. Les accès en dehors des MR ne sont pas possibles et provoquent une erreur. Pour effectuer une écriture à distance, aussi bien les données sources que la zone de destination doivent être dans une MR.

Les adesses mémoire manipulées sont les adresses virtuelles telles qu'elles sont vues depuis l'espace utilisateur, converties en entier 64 bits. Ceci signifie que l'émetteur doit connaître l'adresse de destination dans l'espace d'adressage du processus destinataire. Il est donc nécessaire d'échanger des pointeurs ! En pratique, on échangera un unique pointeur à la connexion ; toutes les adresses distantes seront ensuite *calculées* relativement à ce pointeur de base.

Le réseau *Infiniband* est, techniquement parlant, capable d'effectuer également des opérations en mode envoi/réception et pas uniquement en RDMA. Cependant, le mode RDMA est le plus performant sur ce type de matériel, et nous n'utiliserons donc que le RDMA.

Initialisation

L'initialisation comporte les étapes suivantes :

initialisation de la carte — ouverture du pilote, allocation de PD, enregistrement d'une MR.

ouverture des QP — création des CQ d'émission, de réception, création de la QP, initialisation de la QP. Les CQ ont une taille bornée définie à l'avance.

échange des adresses — échange de LID, QPN et PSN (adresse proprement dite), ainsi que du pointeur de base de la MR et de la clef de la MR.

établissement de la connexion — initialisation de la réception (état RTR), puis de l'émission (RTS).

Le code d'initialisation vous est donné sur le site web.

Envoi en RDMA

L'envoi en RDMA se fait en mode asynchrone. Dans un premier temps, l'émetteur poste une requête (WR – *Work Request*) de façon non-bloquante. Dans un deuxième temps, il peut scruter la terminaison de la requête (WC – *Work Completion*). L'utilisation de ces deux primitives est explicitée par un exemple dans le fichier source donné en exemple.

Du côté du récepteur, la réception est pilotée par la carte *Infiniband*, de façon concurrente à l'exécution du programme sur le CPU. Du point de vue du CPU, la carte *Infiniband* modifie de la mémoire de façon *non-atomique* (i.e. les paquets n'arrivent pas d'un coup — on peut voir un paquet partiel), sans exclusion mutuelle possible. Cependant, les modifications en mémoire apparaissent *dans l'ordre*.

2 Travail à faire

2.1 Envoi simple

Ecrire un programme récepteur et un programme émetteur envoyant le message « *Hello world !* ». Comment détecte-t-on la présence d'un message en réception ? Comment sait-on que l'intégralité du message à été reçue ?

Étendre le programme précédent pour réaliser un ping-pong. À l'aide du `timing.h`, mesurez le temps de transmission.

2.2 Taille variable bornée

Modifiez votre code de façon à ce que des messages de tailles différentes puissent être échangés. Vous pourrez vous limiter à des messages de moins de 64 ko pour l'instant. Comment doit-on disposer la charge utile du message et les en-têtes en mémoire dans la MR ? Pourquoi ?

Écrivez un benchmark qui envoie des messages de taille croissante (les puissances de 2 par exemple), et mesurez le temps de transfert pour chacune des tailles. Tracez ensuite les courbes de latence et de débit en fonction de la taille des messages à l'aide de `gnuplot` (ou de tout autre outil de tracé de courbes). Pour la courbe de débit, choisissez une échelle logarithmique en x, pour la courbe de temps une échelle logarithmique en x et en y. N'hésitez pas à zoomer uniquement sur les tailles de message qui vous semblent pertinentes.

2.3 Taille non-bornée

Ajoutez à votre code la prise en charge de messages de taille non-bornée, qui peuvent nécessiter une fragmentation en plusieurs envois sur le réseau. Pourquoi ne peut-on pas les envoyer directement sans fragmentation ? Expérimentez. Recommencez votre benchmark avec des messages jusque 16 Mo. Conclusion ?

Indication : il vous sera nécessaire d'implémenter un contrôle de flux, c'est-à-dire que le destinataire doit prévenir l'émetteur qu'il a reçu et « consommé » le paquet avant que l'émetteur puisse envoyer le paquet suivant. Dans le cas contraire, on risquerait d'écraser un paquet que le récepteur n'aurait pas encore consommé.

2.4 Double buffering

1. Le système de fragmentation précédent présente des performance assez faibles. Pourquoi ?

   Une optimisation classique consiste en un *double buffering*. Il s'agit de disposer de deux buffers d'émission : la carte *Infiniband* en envoie un pendant que le CPU prépare le suivant, en *pipeline*.
2. Implémentez un système de double buffering en émission et évaluez son impact sur les performances.
3. Implémentez le même système du côté récepteur, sans oubliez d'adapter le mécanisme de contrôle de flux en conséquence.
4. Conclusion ?

2.5 Multiplexage et interactions *threads*–réseau

On désire maintenant faire communiquer plusieurs *threads* (`pthread`) en même temps. Si l'on n'y prend pas garde, les *threads* peuvent recevoir un message destiné à un autre *thread*.

La solution est appelée *multiplexage* : on ajoute une étiquette (*tag*) ou encore « clef de multiplexage » à chaque message émis pour réaliser des canaux logiques.

1. Ajoutez un *tag* (entier) à chaque message émis.
2. Ajoutez la prise en charge des *threads* (verrouillage) et la possibilité de filtrer les tags en réception. Attention : un *thread* ne sait pas *a priori* si un message lui est destiné avant sa réception ! Quel *thread* effectue la scrutation de la boîte aux lettres ?
3. Transformer votre benchmark pour que deux *threads* effectuent un ping-pong en parallèle, chacun avec son propre *tag*.
4. Comparer les performances de votre code avec multiplexage et sans multiplexage dans le cas d'un seul *thread*. Quel est l'impact de votre implémentation du multiplexage sur les performances ? (copies éventuelles, en-têtes, ...)
5. Mesurer et commentez les performances de votre code quand deux *threads* exécutent le benchmark. Donnez (sans l'implémenter) les grandes lignes d'un système qui garantirait une meilleure performance et une meilleure équité.

2.6 Optimisations

Nous cherchons maintenant à améliorer la latence de notre protocole.

1. Quel est le nombre de paquets émis sur le réseau pour chaque message ?
2. Revoyez la disposition en mémoire dans la MR pour pouvoir effectuer du *piggy backing* et réduire le nombre de paquet émis à un seul par message quand la fragmentation n'est pas nécessaire.
3. Au final, quel est le surcoût du multiplexage ?

2.7 Plus loin... *(bonus)*

Malgré tous nos efforts, nous n'atteignons pas encore les performances nominales du réseau *Infiniband*. Il reste certaines optimisations à effectuer qui dépendent de certains paramètres des cartes *Infiniband*.

Latence

La latence est influencée par la MTU, la taille des trames.

– Localisez dans le code d'initialisation le choix de MTU est essayez plusieurs valeurs (512, 1024, 2048, 4096).

La latence pour les très petits paquets est également pénalisée par le fait qu'en réalité deux trames réseau circulent sur le fil : une pour l'en-tête RDMA, une pour les données. Le *piggy backing* est possible mais doit être demandé explicitement et dépend des caractéristiques de la carte.

– Localisez dans le code d'initialisation la propriété `max_inline_data` et observez sa valeur *après* le `ibv_create_qp` en fonction de sa valeur avant et de la valeur de la MTU.
– Pour des paquets plus petits que la valeur après négociation de `max_inline_data`, ajoutez le flag `IBV_SEND_INLINE` au champ `send_flags` dans le code d'envoi.

Débit

Pour des tailles suffisamment grandes de paquets, il peut être plus « rentable » de créer à la volée une MR autour des données plutôt que de les recopier dans une MR pré-existante (même avec le double buffering).

– Implémentez et testez cette fonctionnalité du côté de l'émetteur.
– Comment procéder côté récepteur ?
– Combien coûte réellement la création d'une MR ? À partir de quelle taille cette méthode est-elle rentable ?