Parallel and Distributed Algorithms and Programs
TP n°2 - The Broadcast on the Rings

One Message to find them all. One Routine to send them,
One Routine to receive them all and in the buffers, bind them.

Oguz Kaya
oguz.kaya@ens-lyon.fr

Pierre Pradic
pierre.pradic@ens-lyon.fr

07/11/2016

Dans cette session, on va se focaliser sur l'implementation et l'optimisation des algorithmes de broadcast sur des machines qui sont connectées par un réseau d'anneau. On va transmettre la donné disponible dans le processus de racine (avec le rang 0) à tous les autres processus. Pour evaluer les algorithms avec de differents paramètres de réseau, on va utiliser le logiciel Simgrid pour faire des simulations sur ces réseaux.

Parmi les fichiers que l'on vous a fournis, vous allez trouver un code squelette `bcast_skeleton.c` qui gére l'initialisation du MPI, de la message à transmettre, et d'autres variables dont on aura besoin. Vous n'avez pas à modifier ce fichier, cependant, n'hésitez pas à le regarder si vous le sentez! Ouvrez le code `bcast_solution.c` qui contient des variables (en commentaire) qui sont déjà définies dans le code squelette. Gardez-les en commentaire car elle sont déjà définies donc vous pouvez directement les utiliser. Voici une description de ces variables:

- `num_procs`, `rank`: Le nombre de processus disponibles, et le rang du processus.
- `bcast_implementation_name`: Le nom de l'algorithme de broadcast à faire tourner. Ces nom seront définis dans la suite.
- `chunk_size`: La taille de chaque morceau dans les algorithmes de pipeline. On va y revenir dans la suite.
- `buffer`: Un tableau des chars (octets) qui contient la message à transmettre/à recevoir. Ce tableau est déjà alloué dans tous les processus.
- `NUM_BYTES`: Le nombre d'octets à transmettre dans `buffer`.

On va utiliser SMPI pour simuler une topologie d'anneu ayant 32 machines. SMPI s'utilise de la meme façon que MPI, sauf pour compiler et exécuter le code il faudrait les routines du SMPI `smpicc` et `smpirun`. Pour générer le programme `bcast`, on peut compiler le code à l'aide de la commande suivante:

```
smpicc bcast_skeleton.c -o bcast
```

Afin de pouvoir "simuler" l'exécution d'un programme sur une machine parallèle, il faudrait une description de cette machine à fournir au simulateur. Cette description contient une liste des *hôtes*, cet à dire des machines, qui sont fournis dans un fichier de *hostfile*. On a un deuxième fichier de *platforme* qui détaille les connexions entre les machines. Pour créer ces fichiers automatiquemment, on fournit le script `smpi-generate-ring.py` qui est capable de générer une topologie d'anneau avec un certain nombre de processus donné. Pour générer une topologie d'anneau ayan 32 processeurs avec une latence de $1\mu s$ et une bande-passante de $100Gops$, il faudrait taper ce qui suit:

```
python smpi-generate-ring.py 32 100Gf 100Gf 100Gbps 1us
```

Ceci devrait crée le fichier hostfile `ring-32-hostfile.txt` ainsi que le fichier de platforme `ring-32-platform.xml`.

Le deuxième et le troisiéme parametres correspondent à la puissance de calcul de votre machine/laptop et la machine que l'on veut simuler. On va les garder à 100Gfps (giga-flops par second)

Maintenant, vous pouvez faire tourner le code avec la commande suivante en utilisant la machine que l'on vient de créer:

```
smpirun -hostfile ring-32-hostfile.txt -platform -ring-32-platform.xml ./bcast
                        [bcast_implementation_name]
```

On a déjà une implementation de broadcast appelée `default_bcast` dans le code squelette, donc vous pouvez lancer le code avec cet algorithme pour tester

Part 1

Broadcast naive!

Dans cet exercice, on va implanter un broadcast. Le processus racine (ayant le rang 0) contient une message qui est à distribuer à tous les autres processus. Le processus racine envoi la message directement à chaque processus à l'aide `MPI_Send`. On va donner le nom `naive_bcast` à cette implementation (ce que l'on va fournir dans la variable `bcast_implementation_name` du code squelette).

Question 1

a) Implantez un algorithme de broadcast qui envoi la message dans le processus 0 à tous les autre processus à l'aide du `MPI_Send` and `MPI_Recv`.

b) Tournez le code avec une topologie ayant une bande passante de $100Gpbs$ et une latence de $1\mu s$. Notez le temps d'exécution.

c) Essayez d'augmenter la latence à $10\mu s$ puis à $100\mu s$. Qu'est-ce qu'il se passe au niveau du temps d'exécution? Est-ce que c'est attendu?

d) Trouvez le coût d'exécution de cet algorithme sur une topologie d'anneau avec le modèle $\alpha - \beta$ model, supposant que M est la taille message, P est le nombre de processus, α est la latence pour chaque lien et β est la bande passante de chaque lien. Convient-il le résultat que l'on a obtenu avec la latence qui différs

Part 2

(Don't) Use the ring!

You might have realized that sending all the messages from the root process $\mathcal{P}_0$ at each step could be inefficient, and that the naive algorithm exploits nothing related to the topology being ring. However, the ring may bestow unprecedented powers to thoso who know how to wield it! For example, once a process $\mathcal{P}_k$ receives the message on a ring, what is preventing it from simply passing it to its neighbor $\mathcal{P}_{k+1}$?

Question 2

a) Implement the described ring algorithm with the name `ring_bcast`. The process $\mathcal{P}_k$ should always expect and receive the message from its left neighbor $\mathcal{P}_{k-1}$, then store it in its local buffer, and finally pass it over to its right neighbor $\mathcal{P}_{k+1}$.

b) Run your implementation on the same topology with the latency $1\mu s$. How much of an improvement did you get? Are you happy? If not, try increasing the latency to $10\mu s$, and $100\mu s$, just as before. How does the performance compare to the naive broadcast this time?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 3

One bite at a time...

One major problem with the previous ring broadcast algorithm is that it involves $P - 1$ send/receive rounds, in each of which only one of the links in the network is active. In order to make use of the bandwidth available in all other links at the same time, the idea of pipelining comes into play. Instead of sending M bytes of data in one chunk, we can divide it into chunks of size C, and send it in $\lceil M/C \rceil$ rounds that can be pipelined. For instance, once $\mathcal{P}_1$ receives the first chunk from $\mathcal{P}_0$, it can pass it to $\mathcal{P}_2$ and start receiving the second chunk from $\mathcal{P}_0$, and so on. If we continue in this manner, after $P - 1$ steps all links in the network will start to be actively used.

Question 3

a) Implement the described algorithm with the name `pipelined_ring_bcast`. In the executable, use the option `-c [chunk_size]` to fill in the varible `chunk_size`.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Try to find the best chunk size for each latency, and note the broadcast times. How does the performance compare to `ring_bcast`?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 4

Wait for it, wait for it!

So far we have only used `MPI_Send` and `MPI_Recv` in communications. One peculiarity about these routines is that they are "blocking" calls, meaning that once the process calls the function, it will "wait" and stall until this blocking send or receive operation entirely finishes. This is bad in terms of efficiency, because in this case the receive link, or the send link of each node in the ring will stay idle (depending on whether the node is performing a send, or a receive at that time, respectively). To prevent this, we will employ asynchronous communication routines that allow us to hold two melons with one hand!

Question 4

a) Implement the described algorithm with the name `asynchronous_pipelined_ring_bcast`. You need to properly replace the `MPI_Send` calls with `MPI_Isend` routines, and use `MPI_Wait` on the communication handlers to make sure that the communications take place and finish correctly. You can keep the blocking `MPI_Recv` calls as they are, as establishing asynchronicity in one end should suffice to enable overlapping sends and receives.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Is the algorithm always faster? If not, or what are the cases in which it runs slower? How would you explain this in any case?

Part 5

To recurse, or not to recurse, which one is the curse?

You managed to finish all the exercises and get this far? No-freaking-way! If you are procrastinating here not having finished the other tasks, go back to where you were and do some productive sh.t! Otherwise, that is wonderful, and here is your last exercise that serves as a marvelous closing scene to this TP session.

One brilliant and simple idea to perform a broadcast on a ring of size $P = 2^k$ for some integer $k > 1$ goes as follows. First, the process $\mathcal{P}_0$ sends the message to its "pair" $\mathcal{P}_{P/2}$, and make it responsible for broadcasting the message to the second half of the ring, i.e., to processes $\mathcal{P}_{P/2+1} \dots \mathcal{P}_{P-1}$. Afterwards, we can recursively apply this same idea to the first and the second halves of the ring (which are also rings of size $P/2$) to broadcast the data to the rest of the processes.

Question 5

a) Implement the described algorithm with the name `asynchronous_pipelined_bintree_bcast`. In the executable, use the option `-c [chunk_size]` to fill in the varible `chunk_size`. Try to make it pipelined, as the name suggests, if you have enough time.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Try to find the best chunk size for each latency, and note the broadcast times. How does the performance compare to `asynchronous_pipelined_ring_bcast`? Does pipelining seem to help? Why, or why not?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 6

Make sure to get a copy of your files as it might serve a good reference in the future!