Oui

Absolument. Voici les commandes et les commentaires pour la section A.3, qui explore la gestion de l'affinité des IRQs pour optimiser les performances, notamment réseau.

🇫🇷 A.3 Affinité et isolement (optionnel si sudo)

Cette section nécessite généralement les droits sudo pour désactiver des services ou modifier des paramètres noyau.

Tester irqbalance activé/désactivé

1. Rôle d'irqbalance

Le service **irqbalance** est un démon qui surveille l'utilisation des IRQs sur un système SMP (multiprocesseur symétrique). Son objectif est d'**équilibrer dynamiquement** la charge d'interruption entre tous les cœurs CPU disponibles, en essayant de respecter l'affinité de cache et la topologie NUMA pour maximiser le débit et la réactivité du système.

2. Commandes et Scénarios

Action	Commande	Commentaire
Vérifier l'état	systemctl status irqbalance	Indique si le service est actif ou inactif.
Désactiver le service	sudo systemctl stop irqbalance	Permet d'observer la gestion par défaut du noyau (souvent centralisée sur CPU0).
Réactiver le service	sudo systemctl start irqbalance	Restaure l'équilibrage dynamique.
Observer l'effet	watch -n 0.5 'grep eth0	Comparez la répartition des compteurs IRQ (e.g., eth0 ou votre IRQ réseau) : ils devraient être fixes après désactivation, et se répartir entre les cœurs avec irqbalance activé et sous charge.

istribuées entre les CPUs, évitant un
rant le débit réseau global.
nt statique ou simple. Souvent, la
sur un seul CPU (souvent le premier,
(CPU *hot spot*), limitant le débit

forte charge car ce CPU est

Pinner un processus réseau sur un CPU

Ce test vise à démontrer le bénéfice de la **co-localisation** : s'assurer que le *thread* applicatif qui consomme les données réseau s'exécute sur le **même CPU** que celui qui gère les interruptions et les SoftIRQs associées à la réception des données.

1. Préparation (Isolement de l'IRQ)

Pour un contrôle optimal, l'IRQ réseau utilisée par le test doit être isolée sur un CPU spécifique (e.g., CPU 2).

- **Identifier l'IRQ réseau** : Supposons l'IRQ **24** (comme dans l'exemple A.2) gère la réception.
- **Déterminer le masque d'affinité** : Pour affecter l'IRQ **uniquement au CPU 2**, le masque binaire est 0100 (bit 2 est à 1), ce qui correspond à la valeur hexadécimale **4**.

Bash

```
# Assurez-vous que irqbalance est arrêté pour éviter qu'il ne modifie votre réglage.
sudo systemctl stop irqbalance

# Isoler l'IRQ 24 sur le CPU 2 (masque hexadécimal 4)
echo 4 | sudo tee /proc/irq/24/smp_affinity
```

2. Exécution des tests Taskset + iperf3

Test A : Co-localisation (IRQ et Thread sur CPU 2)

Bash

```
# 1. Le serveur iperf3 est forcé de s'exécuter sur le CPU 2 (taskset -c 2).
# C'est le CPU qui gère l'IRQ réseau (selon l'étape de préparation).
taskset -c 2 iperf3 -s &

# 2. Le client iperf3 (local) s'exécute sur le CPU 3 (taskset -c 3) pour éviter d'influencer le serveur.
# L'interface loopback (127.0.0.1) est utilisée pour un test purement CPU.
taskset -c 3 iperf3 -c 127.0.0.1 -t 10
```

Test B : Sans Taskset (Répartition par défaut)

Bash

```
# 1. Le serveur s'exécute sur n'importe quel CPU disponible (selon le scheduler).
iperf3 -s &

# 2. Le client s'exécute sur n'importe quel CPU.
iperf3 -c 127.0.0.1 -t 10
```

3. Analyse de la Co-localisation

Scénario	Débits (Taskset)	Latence / Jitter	Explication
Co-localisation (Test A)	**Élevés (meilleurs)**	**Faibles (meilleurs)**	Le *thread* iperf3 -s s'exécute sur **CPU 2**. Les paquets arrivent (IRQ 24) et sont traités par les SoftIRQs **sur CPU 2**. Les données sont déjà présentes dans le **cache L1/L2 de CPU 2**. Le thread peut y accéder sans délai de migration inter-cœur.
Sans Taskset (Test B)	Inférieurs	Plus élevés	Le *thread* iperf3 -s peut être planifié sur **CPU 1** (par exemple), tandis que l'IRQ est traitée par **CPU 2**. Chaque fois que le thread doit lire les données du paquet, il doit faire face à une **faute de cache** (*cache miss*), nécessitant une coûteuse **migration de cache** entre les cœurs.

Discussion :
L'expérience démontre que l'**alignement des IRQs et des *threads* de traitement sur le même cœur (co-localisation)** est une technique d'optimisation cruciale pour les applications à haute performance (HPC, trading, virtualisation).

1. **Réduction des "Cache Misses"** : En maintenant le traitement de l'IRQ, la SoftIRQ, et le *thread* utilisateur sur le même CPU, on maximise l'**affinité de cache**. Le CPU gérant l'interruption charge les données du paquet dans son cache, et le processus utilisateur peut y accéder immédiatement.
2. **Minimisation de la Latence** : L'élimination des transferts de données coûteux entre les caches des CPUs réduit considérablement la latence de traitement des paquets, ce qui se traduit par un meilleur débit et un *jitter* (variation de latence) plus faible.