



ING5 SE

PROGRAMMATION PARALLÈLE

TP3 - SYNCHRONISATION

David OLIVARES | Emma PALFI

Q1 (0pts) : Combien votre machine a-t-elle de coeurs physiques ? de coeurs logiques ? de RAM ? Quel système utilisez-vous ? (e.g. Linux natif, WSL/Windows10, Mac, Machine virtuelle/Windows 8, etc.)

La machine possède **4 coeurs physiques (cores per socket)**.

La technologie "Hyperthreading" permet à un cœur d'exécution de traiter plusieurs programmes en parallèle. Sur cette machine, on peut voir qu'il y a 2 threads par cœur. Ainsi, la machine possède **8 coeurs logiques (threads per core = 2, CPU(s) = 2 x 4 = 8)**.

Le système d'exploitation utilisé est Ubuntu 18.04 en **Linux natif**.

La machine possède **8Go de RAM**.

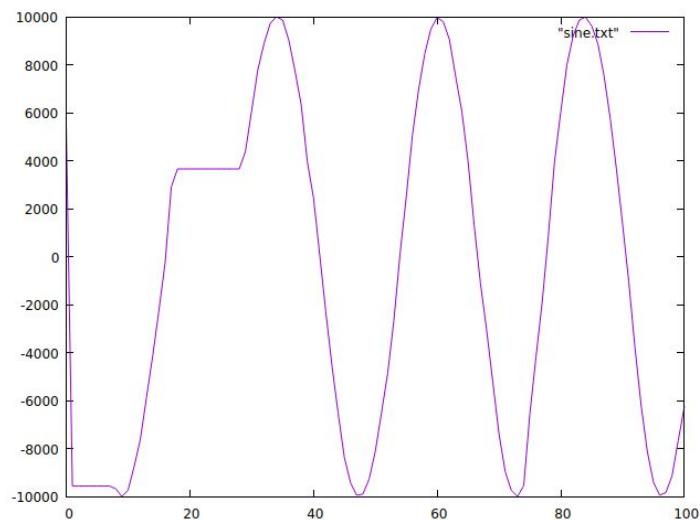
```
david@lpulu:~/prog_parallel/seance1$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:  0-7
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 142
Model name:            Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:               11
CPU MHz:               800.116
CPU max MHz:           4600,0000
CPU min MHz:           400,0000
BogoMIPS:              3999.93
Virtualization:        VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               256K
L3 cache:               8192K
NUMA node0 CPU(s):     0-7
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mttr
                        rch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
                        nt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowpref
                        st bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt int
                        arch_capabilities
david@lpulu:~/prog_parallel/seance1$ lsmem
RANGE                      SIZE STATE REMOVABLE BLOCK
0x0000000000000000-0x000000006fffffff 1,8G online      yes 0-13
0x0000000100000000-0x0000000287fffffff 6,1G online      yes 32-80

Memory block size:          128M
Total online memory:        7,9G
Total offline memory:       0B
david@lpulu:~/prog_parallel/seance1$
```

Exécution de la commande lscpu.

Q2 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

```
david ~ > Downloads > TP3 > 1-sine_sync > time -p ./sine_sync
real 1,29
user 1,53
sys 0,08
david ~ > Downloads > TP3 > 1-sine_sync >
```

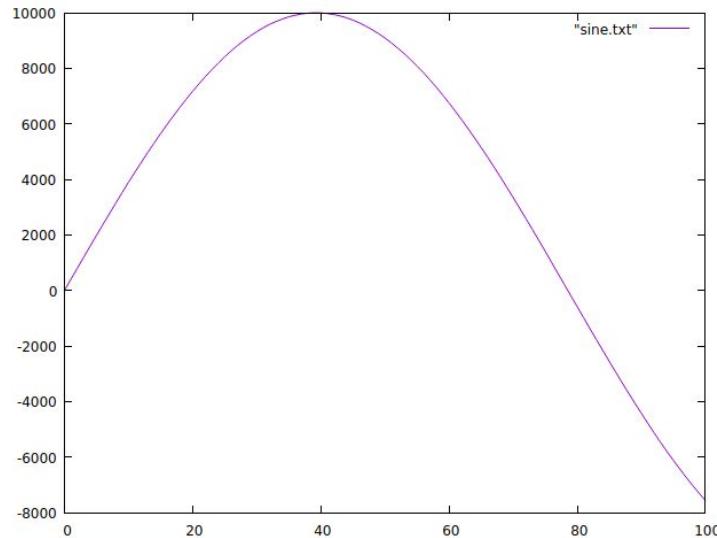


Q3 (0,5pt) : Quel est le problème avec ce programme ?

Deux threads sont utilisés, un pour produire et un pour écrire et utilisent la même variable en simultané (sine_value). Cependant, ils ne sont pas synchronisés, donc le producer peut produire plusieurs fois de suite sans que le writer n'écrive (par exemple sur la courbe: décalage trop important entre deux valeurs qui se suivent) et inversement, le writer peut écrire plusieurs fois de suite sans que le producer ne produise (par exemple sur la courbe: plat).

Q4 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

En utilisant des mutex pour synchroniser les deux threads, on obtient:



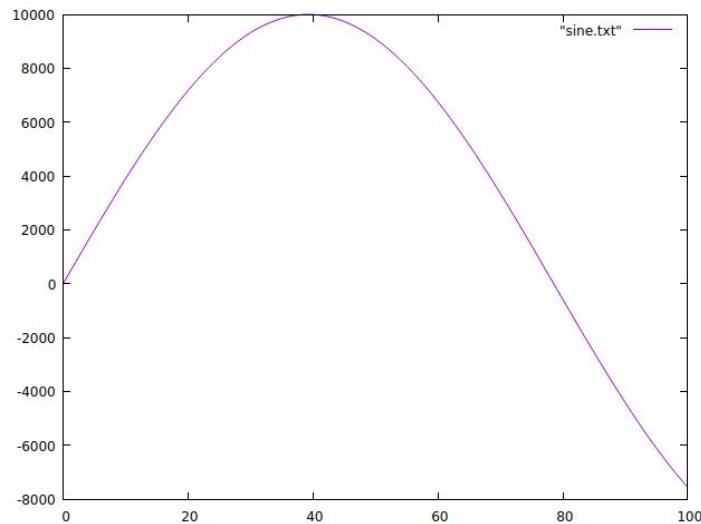
Pour synchroniser, nous avons utilisé deux mutex. Ils permettent de contrôler l'accès à des ressources partagées. Nous avons identifié deux sections critiques dans chaque fonction des threads qui sont la modification de la valeur de la variable `sine_value` et l'écriture dans le fichier de cette même variable.

Principe:

- Les deux mutex 1 et 2 sont lock.
- Le producer calcule la valeur de `sine_value`.
- Pendant ce temps, le writer est bloqué car il essaie de locker le mutex 1 qui est déjà lock.
- Une fois que le producer a terminé, le mutex 1 est unlock et le producer peut écrire dans le fichier. Pendant ce temps, le producer est bloqué car il essaie de lock le mutex 2 qui est déjà lock.
- Une fois que le writer a fini, le mutex 2 est unlock et le producer calcule de nouveau et ainsi de suite.

Q5 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.

En parallélisant avec openmp, on obtient:



Pour paralléliser le code, nous avons utilisé les *pragma sections* d'openmp. Elles permettent de paralléliser un morceau de code contenu dans chaque section.

Pour synchroniser les deux sections et donc les deux threads, nous avons utilisé des *omp_lock_t* qui vont gérer l'accès à la variable *sine_value* de la même manière que pour les mutex.

Q6 (0,5pt) : Mesurer les latences de vos programmes modifiés.

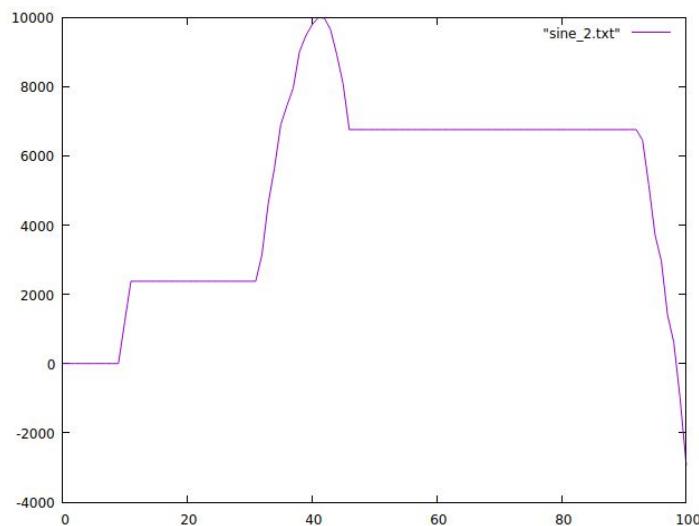
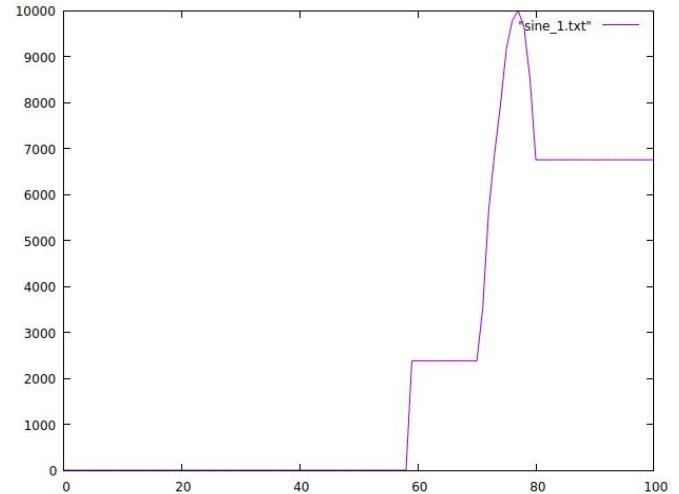
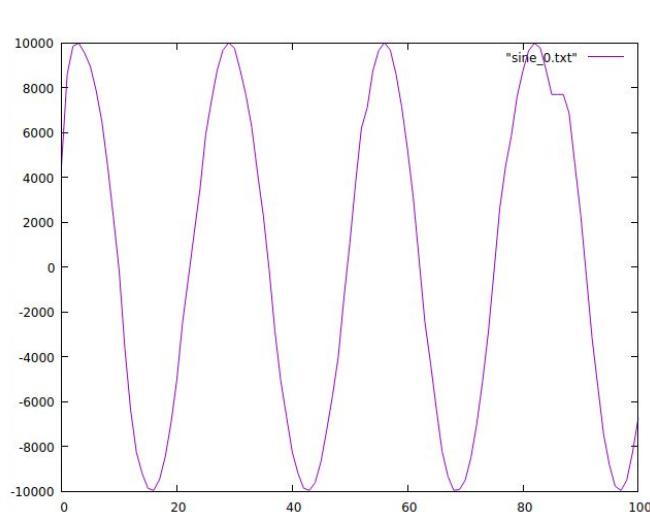
```
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 1-sine_sync_pthreads ➤ time -p ./sine_sync_pthreads
real 57,50
user 20,97
sys 56,42

david ➤ ⚡ david ➤ ... > seance3 > TP3 > 1-sine_sync_openmp ➤ time -p ./sine_sync_openmp
real 4,76
user 9,29
sys 0,28
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 1-sine_sync_openmp ➤ |
```

Le temps real est inférieur à la somme du temps user et du temps sys ce qui montre que le code est bien parallélisé.

Q7 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

```
david ~ > Downloads > TP3 > 2-sine_multi_writers_sync > time -p ./sine_multi_sync
real 1,53
user 4,67
sys 0,30
david ~ > Downloads > TP3 > 2-sine_multi_writers_sync >
```

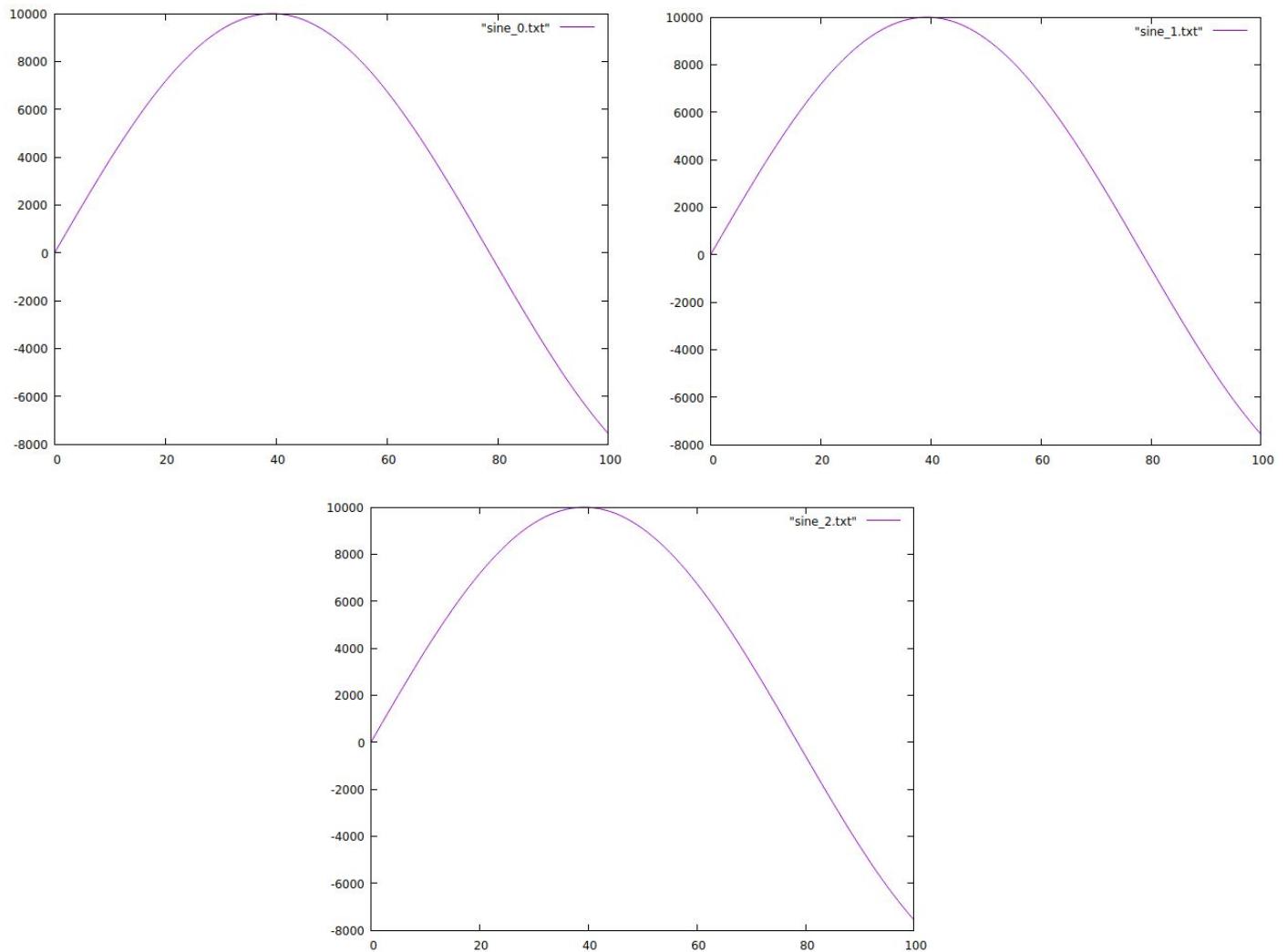


Q8 (0,5pt) : Quel est le problème avec ce programme ?

Ici, nous avons un producer pour 3 writers chacun exécuté par un thread. Chaque writer va écrire dans un fichier différent la valeur du *sine_value* produite par le producer. Lorsque les threads ne sont pas synchronisés, le producer peut produire plusieurs fois de suite ou inversement le writer peut écrire plusieurs fois de suite. Il

peut arriver qu'un ou deux writers écrive la bonne valeur mais que le producer produise la valeur suivante sans attendre le ou les derniers writers.

Q9 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

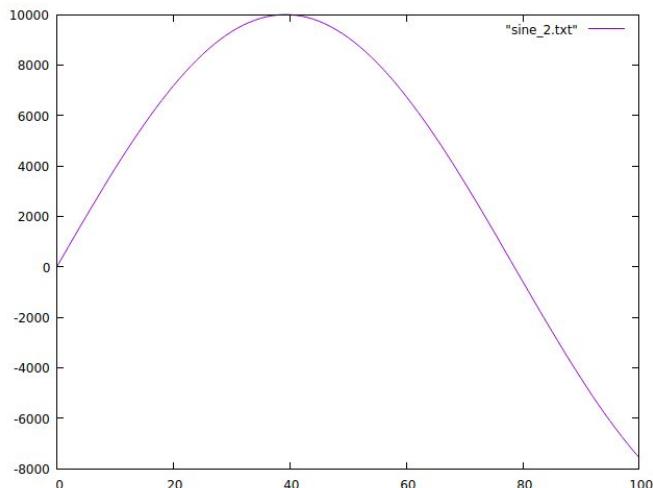
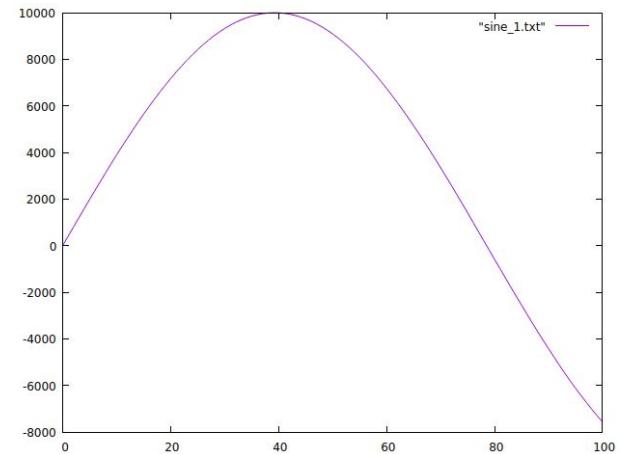
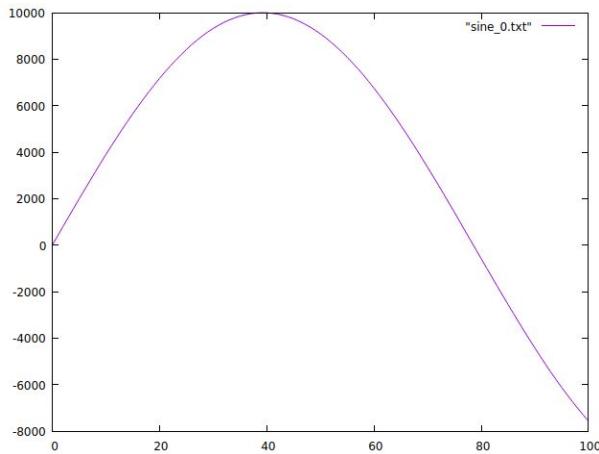


Pour corriger ce code, nous avons utilisé 6 sémaphores. Grâce à ces derniers, nous pouvons faire en sorte que le producer produise une fois puis que les 3 writers écrivent parallèlement la même valeur puisque comme les mutex, ils vont limiter l'accès à la ressource partagée.

Principe:

- Les 6 sémaphores sont initialisés à 0 jetons.
- Le producer commence par calculer la première valeur de *sine_value*. Pendant ce temps là, les 3 threads writer sont bloqués chacun par la commande `sem_wait()` des sémaphores 0 à 2 (un sem par thread) car le nombre de jetons des sémaphores 0 à 2 étant à 0, la fonction `sem_wait()` ne peut pas décrémenter le nombre de jetons de chacun d'entre eux.
- Une fois que le producer a modifié la variable *sine_value*, il va utiliser la fonction `sem_post()` sur les sémaphores 0 à 2 ce qui va incrémenter de 1 le nombre de jetons de chaque sémaphore. Les fonctions `sem_wait()` appelées par les 3 writers peuvent alors décrémenter le nombre de jetons de chaque sémaphore et les writers peuvent écrire dans le fichier la valeur de *sine_value* en parallèle. Pendant ce temps là, le producer appelle la fonction `sem_wait()` pour les sémaphores 3 à 5, mais leur nombre de jetons étant à 0, la fonction ne peut pas les décrémenter et le producer attend.
- Une fois que chaque writer a fini d'écrire dans son fichier (pas forcément en même temps), ils vont chacun appeler la fonction `sem_post()` et donc incrémenter de 1 le nombre de jetons d'un sémaphore 3, 4 ou 5 selon le writer. Tant que tous les writers n'ont pas terminé d'écrire et donc tant que les 3 sémaphores 3, 4 et 5 ne sont pas décrémentables, le producer attend avant de pouvoir calculer la prochaine valeur de *sine_value*.
- Et ainsi de suite.

Q10 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.



La logique suivie est la même que la précédente. Ici, nous utilisons un *pragma parallel for* qui va lancer 4 threads, 1 pour le producer et 3 pour les writers. A la place des sémaphores, nous avons utilisé des *omp_lock_t*.

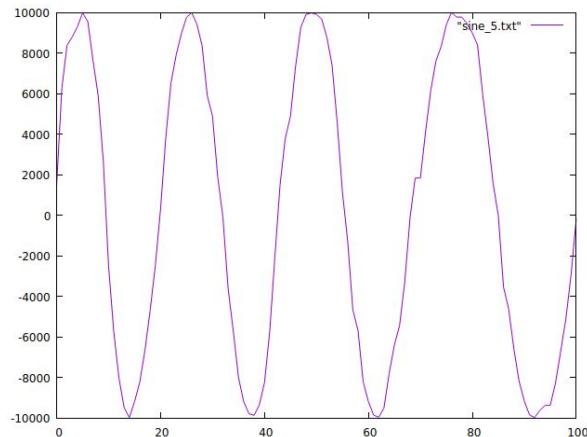
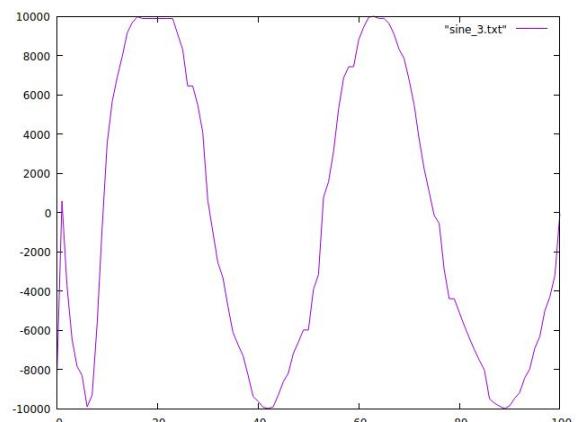
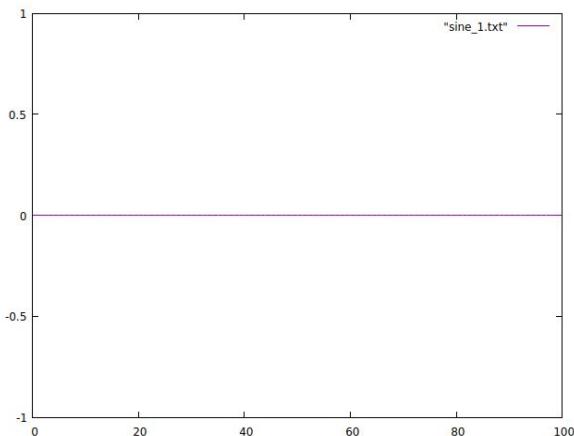
Q11 (0,5pt) : Mesurer les latences de vos programmes modifiés.

```
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 2-sine_multi_writers_sync ➤ time -p ./sine_multi_sync
real 74,98
user 48,27
sys 111,59
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 2-sine_multi_writers_sync ➤ |
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 2-sine_multi_sync_openmp ➤ time -p ./sine_multi_sync_openmp
real 6,11
user 24,23
sys 0,27
david ➤ ⚡ david ➤ ... > seance3 > TP3 > 2-sine_multi_sync_openmp ➤ |
```

Le temps real est inférieur à la somme du temps user et du temps sys ce qui montre que le code est bien parallélisé.

Q12 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

```
david ~ > Downloads > TP3 > 3-sine_full_sync > time -p ./sine_full_sync
real 1,65
user 6,48
sys 0,25
david ~ > Downloads > TP3 > 3-sine_full_sync >
```



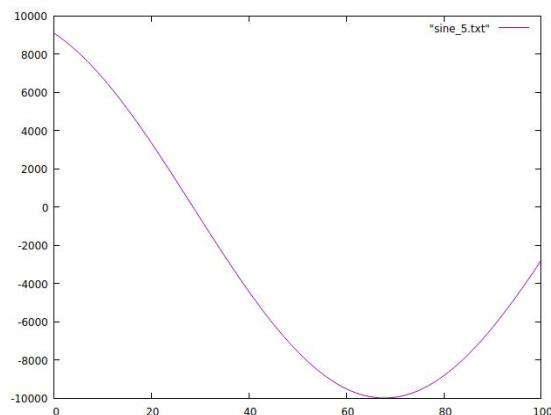
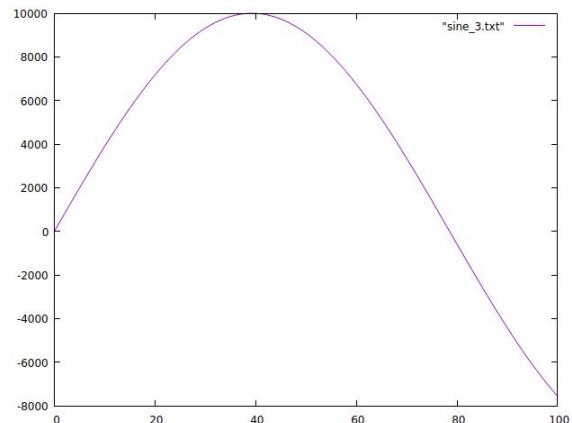
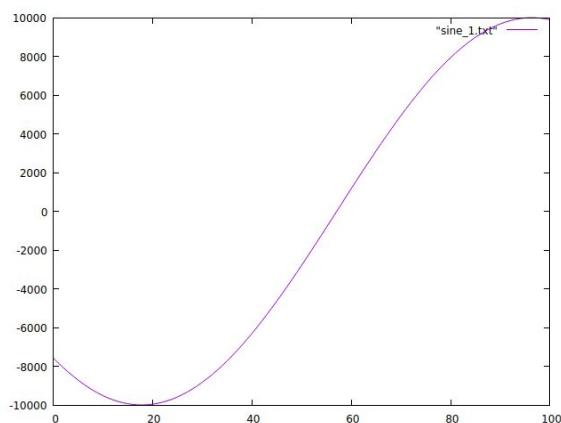
Q13 (0,5pt) : Quel est le problème avec ce programme ?

Ici, nous avons 3 producers et 3 writers. Chaque producer va produire pour un writer différent grâce à un tableau de 3 cases. Les producers ne sont donc pas dépendants entre eux et les writers non plus. Chaque paire de producer/writer peut donc aller à son rythme. Le problème revient donc au même que celui du programme *sine_sync* mais multiplié par 3. Il suffit donc de synchroniser chaque producer avec son writer.

Les couples sont ainsi:

- ID producer: 0 avec ID writer: 3
- ID producer: 2 avec ID writer: 5
- ID producer: 4 avec ID writer: 1

Q14 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

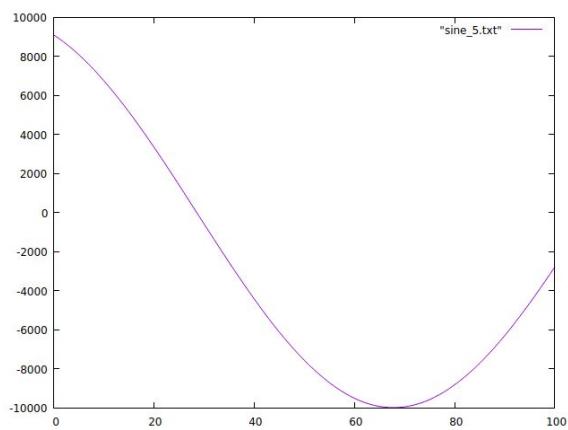
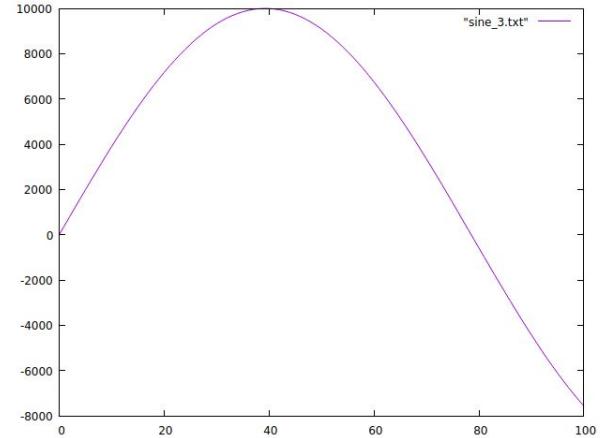
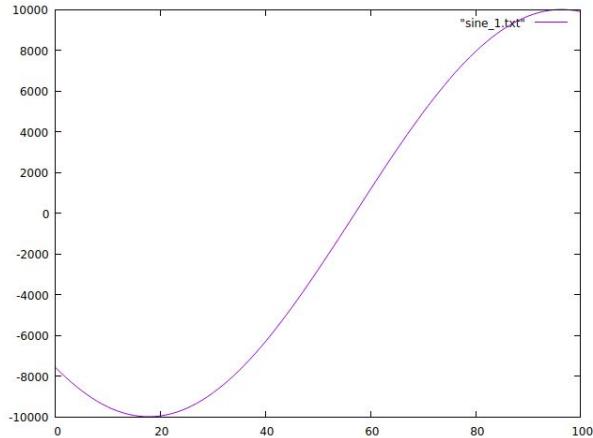


On peut remarquer que les courbes obtenues sont différentes pour chaque writer et donc pour chaque producer. Le décalage de la fonction sin est dû au fait que la formule permettant de calculer le x , utilisé dans le $\sin(x)$, n'est pas la même pour tous les threads puisque le numéro de l'id du thread producer qui calcule le x est ajouté au calcul du sin de base.

Pour synchroniser les threads, nous avons utilisé 6 sémaphores. Le principe d'utilisation des sémaphores est le même que celui des mutex de l'exercice 1.

Principe pour le couple 0/3 : même principe que l'exercice 1.

Q15 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.



Pour paralléliser avec openmp, nous avons utilisé un *pragma omp parallel for* qui va paralléliser le code de la boucle for en un nombre de thread défini dans le code (ici 6).

A la place des sémaphores nous avons utilisé, en respectant le même principe, des *omp_lock_t*.

Q16 (0,5pt) : Mesurer les latences de vos programmes modifiés.

```
david ➤ ⌂ david > ... > seance3 > TP3 > 3-sine_full_sync ➤ time -p ./sine_full_sync
real 84,13
user 88,33
sys 231,10
david ➤ ⌂ david > ... > seance3 > TP3 > 3-sine_full_sync ➤ |
```



```
david ➤ ⌂ david > ... > seance3 > TP3 > 3-sine_full_sync_openmp ➤ time -p ./sine_full_sync_openmp
real 8,00
user 43,61
sys 0,79
david ➤ ⌂ david > ... > seance3 > TP3 > 3-sine_full_sync_openmp ➤ |
```

Le temps real est inférieur à la somme du temps user et du temps sys ce qui montre que le code est bien parallélisé.

Q17 (1pt) : Maintenant que vous avez mesuré de nombreuses fois les latences de tous ces programmes, avez-vous une remarque particulière ? Comment expliquer ce phénomène ?

On remarque que les programmes parallélisés et synchronisés avec openMP sont plus performants (moins de latency) que ceux parallélisés et synchronisés avec pthread. Cela est dû à la manière dont openMP gère les context switch entre les threads et à l'utilisation des tâches.

On remarque que pour chaque programme utilisant openmp, le temps user est divisé par 2 par rapport au temps user avec pthreads et le temps sys est quasiment nul alors qu'il était très élevé avec pthreads. Openmp ne semble donc pas utiliser de tâches système contrairement à pthreads.

Il est inutile de paralléliser avec pthreads puisque les latences sont énormes et que le but de la parallélisation de ce TP est de réduire les latences. Il vaut mieux utiliser openmp pour des parallélisations simples.

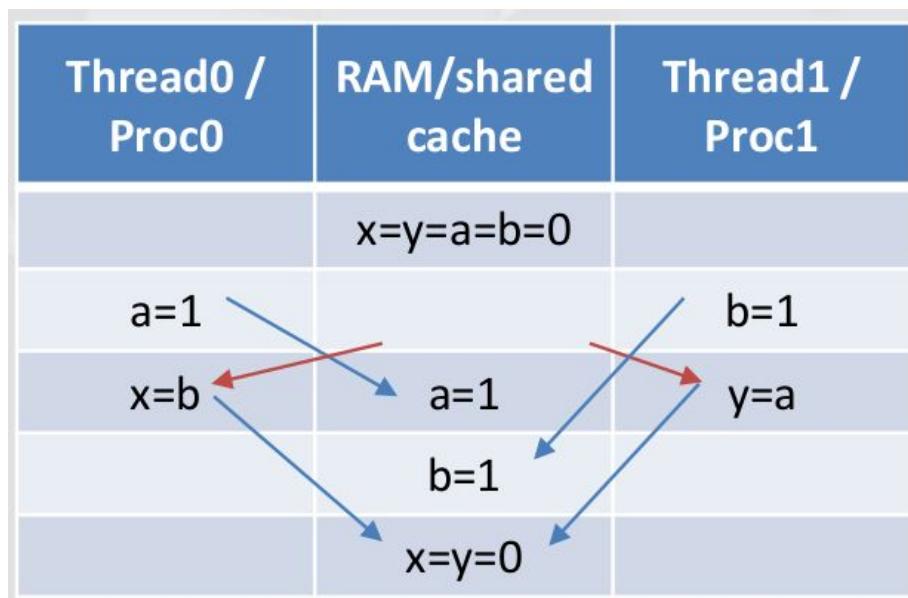
Q18 (0,5pt) : Compilez et exécutez le programme. Quelle remarque faites-vous ?

```
david ➤ ⌂ david > ... > seance3 > TP3 > 4-consume_sync ➤ ./consume_sync
(x = 0, y = 0) :      1185 times (0.012%)
(x = 0, y = 1) :  9975778 times (99.758%)
(x = 1, y = 0) :      21127 times (0.211%)
(x = 1, y = 1) :      1910 times (0.019%)
david ➤ ⌂ david > ... > seance3 > TP3 > 4-consume_sync ➤ |
```

On obtient le couple de coordonnée $x = 0$ et $y = 0$ alors que ce n'est pas censé être possible.

Cela s'explique par **l'inconsistance mémoire**. En effet, lorsque deux threads partagent la même mémoire, si l'un écrit dans une variable avant que l'autre ne la lise, il se peut que même si les deux instructions ont été appelées dans cet ordre, que ce dernier soit changé lors de leur exécution. En effet, l'écriture d'une variable demande un accès à l'hardware, pendant ce temps là, le CPU exécute la lecture. Cela crée une inconsistance mémoire car le deuxième thread lira une ancienne valeur.

Dans ce programme, le temps que le CPU attribue la valeur 1 à $a[0]$, il va exécuter la commande $y[0] = a[0]$, de ce fait $y[0]$ vaudra 0 car $a[0]$ n'aura pas changé. De même pour $b[0]$ et $x[0]$. On peut donc obtenir le couple $x[0] = 0$ et $y[0] = 0$.



L'inconsistance mémoire se produit lorsque plusieurs threads n'ont pas la même vue sur la même donnée de la mémoire.

Q19 (1pt) : Grâce aux memory barriers, vous devez corriger ce programme afin que le cas 00 ne se produise jamais.

La commande `asm volatile ("mfence" ::: "memory")` est une barrière au niveau du compilateur et du CPU. Elle empêche de modifier l'ordonnancement des instructions. Le mot `volatile`, quant à lui, empêche le compilateur de supprimer la commande `asm`.

La commande fait en sorte que tous les accès Read/Write avant la barrière doivent terminer avant d'exécuter un accès Read/Write après la barrière.

```
david ➤ ⚡ david ... > seance3 > TP3 > 4-consume_sync ➤ ./consume_sync
(x = 0, y = 0) :      0 times (0.000%)
(x = 0, y = 1) :  8187386 times (81.874%)
(x = 1, y = 0) :    7814 times (0.078%)
(x = 1, y = 1) : 1804800 times (18.048%)
david ➤ ⚡ david ... > seance3 > TP3 > 4-consume_sync ➤ |
```

Principe :

- Après l'initialisation des variables a, b, x et y à 0, le *#pragma omp parallel* crée une équipe de threads (au nombre de 2), chaque section s'exécute en parallèle.
- Les deux *asm volatile* sont placés juste après la modification de la variable a[0] = 1 et b[0] = 1. Cela signifie que x[0] ne peut pas prendre la valeur de b[0] si a[0] ne vaut pas 1 et y[0] ne peut pas prendre la valeur de a[0] si b[0] ne vaut pas 1.
- Imaginons maintenant les 3 scénarios possibles :
 - Les instructions du thread 1 sont exécutées avant celles du thread 2. a[0] prend la valeur 1 puis x[0] prend la valeur de b[0] or b[0] vaut 0 car l'instruction a a lieu avant l'instruction b[0] = 1. Ensuite, b[0] prend la valeur 1 et y[0] prend la valeur de a[0]. x[0] vaut 0 et y[0] vaut 1.
 - Les instructions du thread 2 sont exécutées avant celles du thread 1. b[0] prend la valeur 1 puis y[0] prend la valeur de a[0] or a[0] vaut 0 car l'instruction a a lieu avant l'instruction a[0] = 1. Ensuite, a[0] prend la valeur 1 et x[0] prend la valeur de b[0]. x[0] vaut 1 et y[0] vaut 0.
 - Les instructions du thread 1 et du thread 2 se croisent. a[0] prend la valeur 1. b[0] prend la valeur de 1. x[0] prend la valeur de b[0] et y[0] prend la valeur de a[0]. x[0] vaut 1 et y [0] vaut 1.