# TP5: Partage du bus dans les architectures multi-processeurs

***B. Architecture matérielle***

**Question B1: En consultant l'en-tête du fichier pibus_frame_buffer.h, précisez la signification des arguments du constructeur du composant PibusFrameBuffer.**

```
    PibusFrameBuffer (sc_core::sc_module_name       name,           // instance name
uint32_t                tgtid,          // target index
soclib::common::PibusSegmentTable   &segtab,        // segment table
uint32_t                latency,        // access latency
uint32_t                width,          // frame width
uint32_t                height,         // frame height
int                 subsampling = 420);     // pixel format
```

On peut voir que les signaux dans le constructeur de PibusFrameBuffer: 1. name: Le nom de instance pour créer le structure. 2. tgtid: Le id de cible composant. 3. &segtab: Le tableau de PibusSegment d'espace adresse. 4. latency: L'initialisation de cycles d'attend pour access dans le mémoire. 5. width: Largeur de la segment. 6. height: Hauteur de la segment. 7. subsampling: Le type du pixel format, ici on a format 4:2:0.

**Question B2: quelle est la longueur du segment associé à ce composant ?**

Consultation après le definition: FB_NPIXEL = 256, FB_NLINE = 256 Donc la taille de segment = 256 * 256 = 65536

***C. Compilation de l'application logicielle***

**Question C1 : Pourquoi faut-il utiliser un appel système pour accéder (en lecture comme en écriture) au contrôleur de frame-buffer ? Que se passe-t-il si un programme utilisateur essaie de lire ou d'écrire directement dans le Frame Buffer, sans passer par un appel système ?**

```
#define SEG_FBF_BASE    0x96000000
```

On sait que l'adresse de SEG_FBF_BASE est l'adresse de noyau, il faut utiliser un appel système dans cette adresse, sinon l'utilisateur est interdit à acceder dans cette adresse.

**Question C2 : Pourquoi préfère-t-on construire une ligne complête dans un tableau intermédiaire de 256 pixels plutôt que d'écrire directement l'image - pixel par pixel - dans le frame buffer?**

C'est parce que le technique d'écriture pixel par pixel est trôp lente à écrire, mais on peut utiliser le vector pour écrire dircectement, et plus, utiliser un tableau

peut économiser les cycles pour access dans le mémoire.

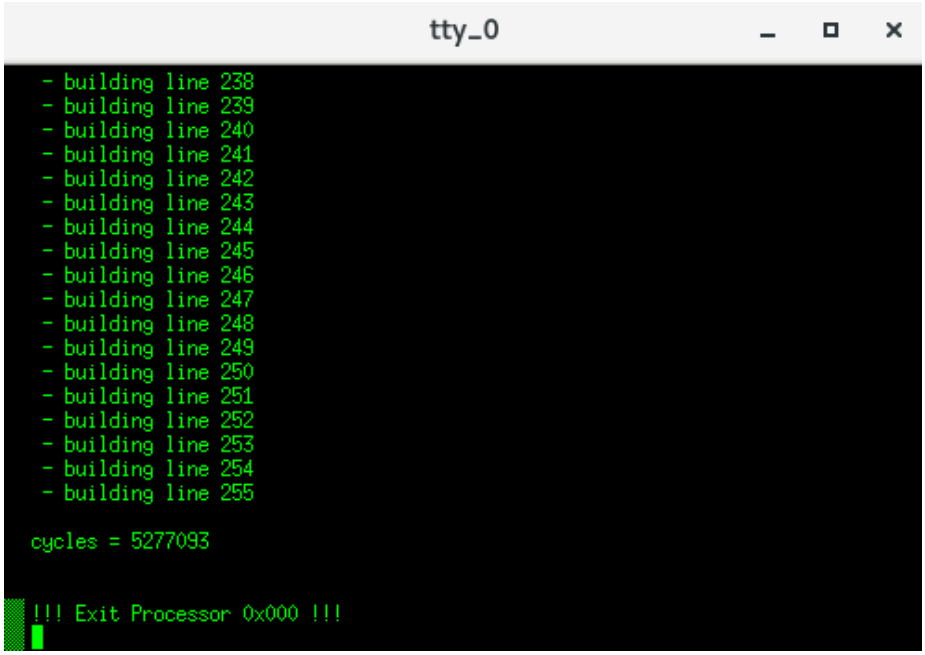
**Question C3 : Consultez le fichier stdio.c, et expliquez la signification des trois arguments de l'appel système fb_sync_write(). Complétez le fichier main.c en conséquence.**

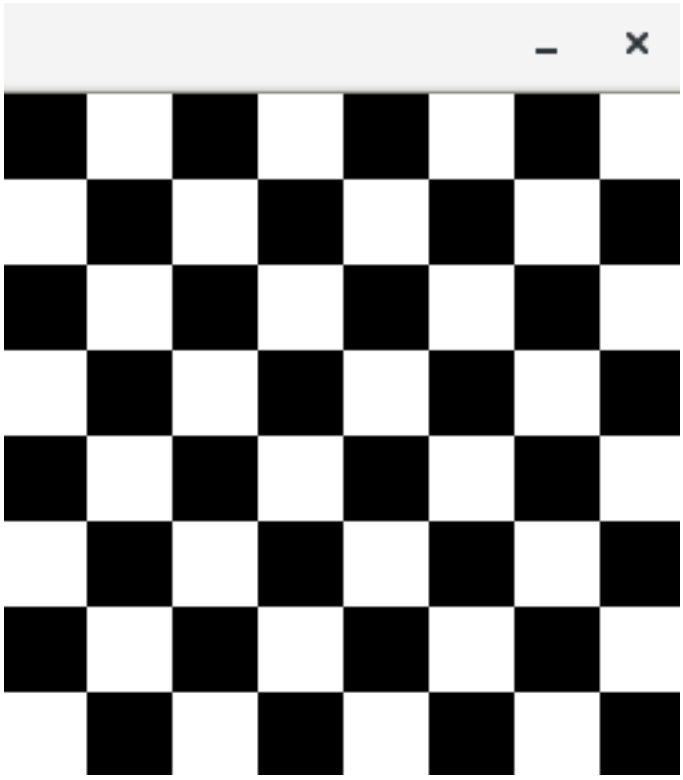
```
unsigned int fb_sync_write(unsigned int offset, void *buffer, unsigned int length)
```

- – offset : offset (in bytes) in the frame buffer
- – buffer : base address of the memory buffer
- – length : number of bytes to be transfered

***D. Caractérisation de l'application logicielle***

**Question D1 : Quel est le nombre de cycles nécessaires pour afficher l'image avec un seul processeur. Combien d'instructions ont été exécutées pour afficher l'image? Quel est le nombre moyen de cycles par instruction (CPI) ?**





On peut voir que il coût 5277093 cycles pour affichier le graphe complet.

```
*** proc[0] at cycle 5277092
- INSTRUCTIONS       = 3785738
- CPI                = 1.39394
- CACHED READ RATE   = 0.255661
- UNCACHED READ RATE = 0.00139101
- WRITE RATE         = 0.133484
- IMISS RATE         = 0.00974394
- DMISS RATE         = 0.0106006
- IMISS COST         = 15.9162
- DMISS COST         = 14.7426
- UNC COST           = 6
- WRITE COST         = 0
```

Dans le dernier cycle, il calcule le CPI: 1.39394 Cycles/instruction

**Question D2 : Quel est le pourcentage d'écritures sur l'ensemble des instructions exécutées ? Quel est le pourcentage de lectures de données ? Quels sont les taux de miss sur le cache instruction et sur le cache de données ? Quel est le coüt d'un miss sur le cache instructions ? Quel est le coût d'un miss sur le cache de données ? Quel est le coût d'une écriture pour le processeur ? On rappelle que les couts se mesurent en nombre moyen de cycles de gel du processeur. Comment expliquez-vous que ces coûts ont des valeurs non entières?**

Dans le dernier cycle, on peut voir que le pourcentage d'écriture est 0.133484, le pourcentage de lecture est séparer dans deux parties, pourcentage de lecture cached: 0.255661 et pourcentage de lecture uncached: 0.00139101. Le taux de Miss sur le cache est aussi séparer dans deux parties, taux miss pour icache est 0.00974394 et taux miss pour dcache est 0.0106006. Le coût de un miss sur icache est 15.9162 cycles, le coût de un miss sur dcache est 14.7426 cycles. Le coût d'une écriture est 0 grâce à le depth de buffer est plus grand, depth de buffer = 8.

```
std::cout << "- IMISS COST         = " << (float)c_imiss_frz/c_imiss_count << std::endl;
std::cout << "- DMISS COST         = " << (float)c_dmiss_frz/c_dmiss_count << std::endl;
```

Pour calculer les coût de miss, on peut voir que le programme il fait les cycles freezed diviser par les cycles de miss, donc chaque fois les miss ne sont pas le même chooses, donc le coût de miss n'est pas entier.

**Question D3 : Evaluez le nombre de transactions de chaque type pour cette application. Que remarquez-vous ?**

Nombre de lecture cached = nombre de instructions * cached read rate = 967865.5628 = 967866

Nombre de lecture uncached = nombre de instructions * uncached read rate = 5265.999415 = 5266

Nombre d'écriture = nombre de instructions * write rate = 505335.4512 = 505336

***E. Exécution sur architecture multi-processeurs***

**Question E1 : Modifiez la boucle principale dans le fichier main.c pour partager le travail entre les différents processeurs de l'architecture.**

Main.c

```c
__attribute__ ((constructor)) void main() {
    unsigned char buf[NPIXEL];
    int n = procid();
    int nprocs = NB_PROCS;
    int line;
    int pixel;

    for (line = 0; line < NLINE; line += 1) {
        if (n == line % nprocs){
            for (pixel = 0; pixel < NPIXEL; pixel += 1) {
                buf[pixel] = build(pixel, line, 5);
            }
            if (fb_sync_write( line*NPIXEL , buf , NPIXEL )) {
                tty_printf(" !!! wrong transfer to frame buffer for line %d\n", line);
            }
            else {
                tty_printf(" - building line %d\n", line);
            }
        }
    }

    tty_printf("\ncycles = %d\n", proctime());
    exit();
} // end main
```

Reset.s

```
# initializes stack pointer
    mfc0 $27, $15, 1
    addiu $27, $27, 1
    sll $27, $27, 16
    la  $29,    seg_stack_base
    addu    $29,    $29,    $27#0x4000      # stack size = 16 Kbytes
```

config.h

```
#define NB_PROCS    8
#define NB_MAXTASKS 1
```

**Question E2 : Pourquoi les piles d'exécution des N programmes s'exécutant sur les N processeurs doivent-elles être strictement disjointes ? Modifiez le fichier reset.s, pour initialiser le pointeur de pile à une valeur dépendant du numéro du processeur, de telle sorte que chaque tâche possède une pile de 64 Koctets, quelque soit le nombre de processeurs (compris entre 1 et 8).**

Les piles d'exécution sont des zones de mémoire qui stockent les variables locales et les informations de contexte pour chaque fonction exécutée dans un programme. Chaque processus ou programme nécessite une pile d'exécution pour fonctionner correctement. Si deux processus partagent une même pile d'exécution, cela peut causer des erreurs d'exécution et des comportements imprévisibles, car les variables locales et les informations de contexte pourraient être écrasées ou modifiées par l'autre processus.

Ainsi, pour garantir que chaque processus s'exécute correctement et de manière indépendante des autres, les piles d'exécution doivent être strictement disjointes. Cela signifie que chaque processus doit avoir sa propre pile d'exécution, qui ne doit pas chevaucher ou partager de mémoire avec les piles d'autres processus.

Dans le fichier reset.s, on peut modifier chaque stack est stocké en 16k octets, donc chaque fois on va change le stack pour compiler.

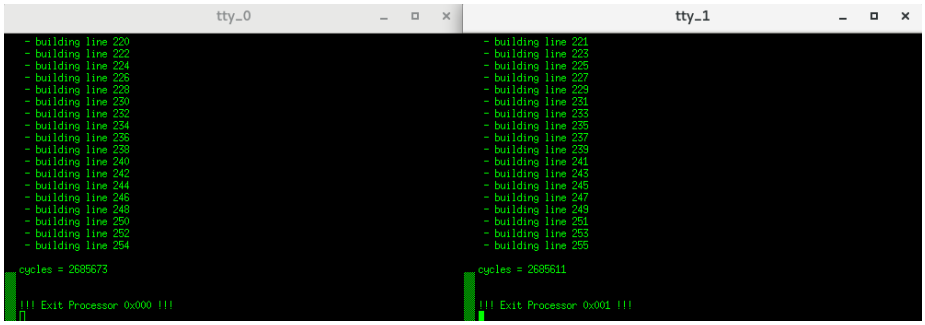
**Question E3 : Donnez deux raisons pour lesquelles le code binaire doit être recompilé chaque fois qu'on change le nombre de processeurs.**

1. Allocation de la mémoire : le nombre de processeurs affecte l'allocation de la mémoire et donc les adresses de mémoire utilisées par le programme. Si le nombre de processeurs change, les adresses de mémoire affectées à chaque processeur vont également changer, ce qui nécessite une recompilation pour s'assurer que les adresses de mémoire sont correctement allouées.

2. Utilisation des instructions de synchronisation : lorsque plusieurs processeurs travaillent ensemble, il est nécessaire d'utiliser des instructions de synchronisation pour s'assurer que les différents processeurs se coordonnent correctement. Les instructions de synchronisation dépendent de l'architecture du processeur et peuvent varier en fonction du nombre de processeurs. Si le nombre de processeurs change, les instructions de synchronisation nécessaires peuvent également changer, ce qui nécessite une recompilation pour garantir que le programme utilise les bonnes instructions pour coordonner les différents processeurs.

**Question E4 Remplissez le tableau ci-dessous, et représentez graphiquement speedup(N) en fonction de N.**

`speedup(N) = Temps de calcul sur 1 processeur / Temps de calcul sur N processeurs`

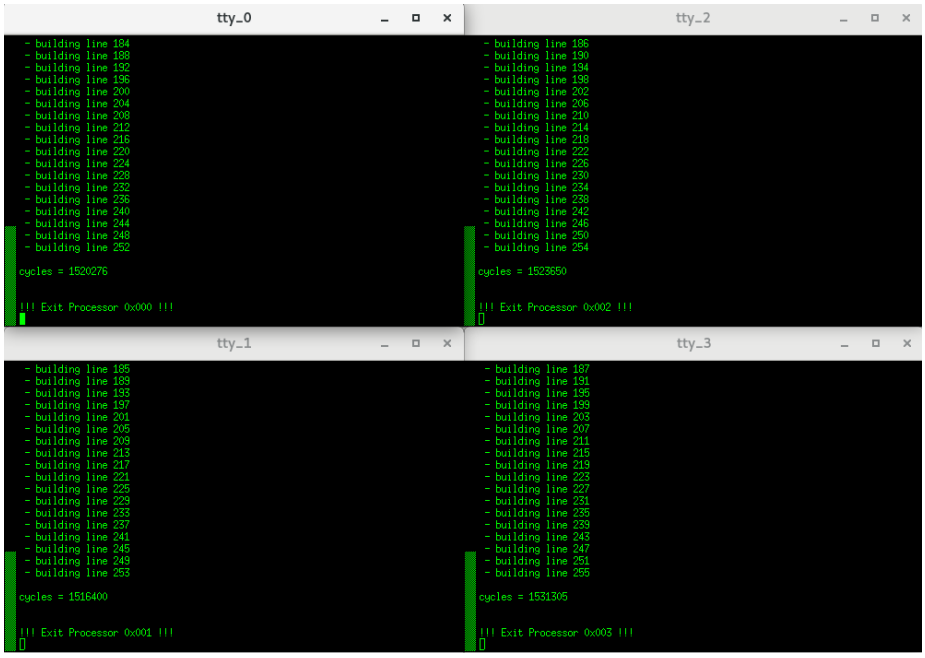



PROCS = 2, cycles tty_0 = 2685673, cycles tty_1 = 2685611 Cycles = 2685673

speedup(N) = 1.9649

```
proc[0] at cycle 2685672
- INSTRUCTIONS       = 1896651
- CPI                = 1.41601
- CACHED READ RATE   = 0.255432
```

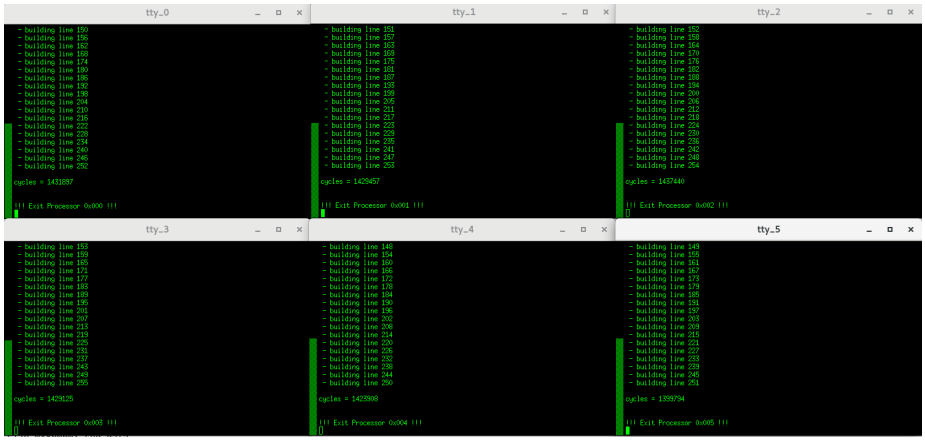
```
- UNCACHED READ RATE = 0.00138824
- WRITE RATE         = 0.133293
- IMISS RATE         = 0.00946194
- DMISS RATE         = 0.00955072
- IMISS COST         = 18.2133
- DMISS COST         = 17.5987
- UNC COST           = 6.32131
- WRITE COST         = 0
```




Procs = 4, cycles tty_0 = 1520276, cycles tty_1 = 1516400, cycles tty_2 = 1523650, cycles tty_3 = 1531305 Cycles = 1531305

speedup(N) = 3.44614

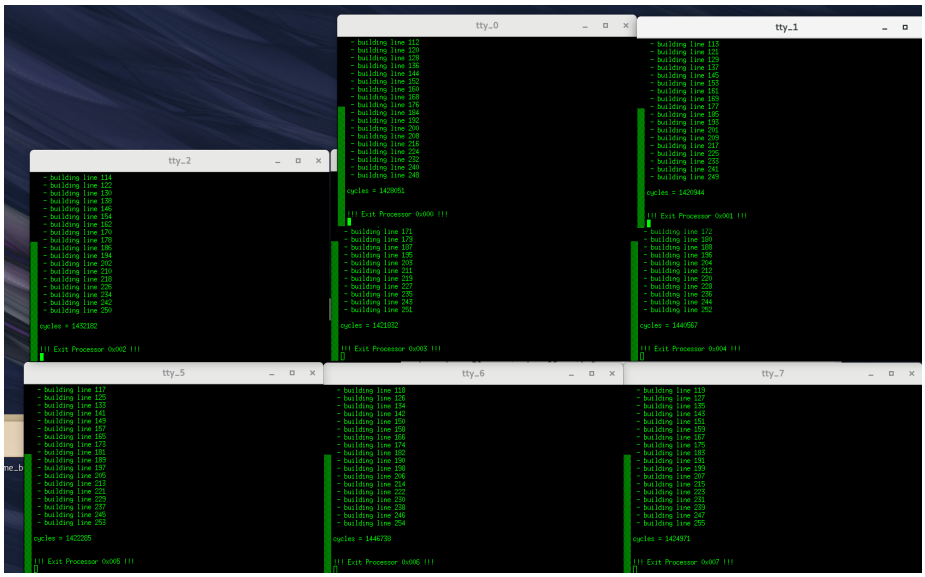
```
proc[0] at cycle 1531304
- INSTRUCTIONS       = 953863
- CPI                = 1.60537
- CACHED READ RATE   = 0.255573
- UNCACHED READ RATE = 0.00143102
- WRITE RATE         = 0.133021
- IMISS RATE         = 0.00956427
- DMISS RATE         = 0.00989409
- IMISS COST         = 27.755
- DMISS COST         = 26.6206
- UNC COST           = 17.3846
- WRITE COST         = 0.394037
```




Procs = 6, cycles tty_0 = 1431897, cycles tty_1 = 1429457, cycles tty_2 = 1437440, cycles tty_3 = 1429125, cycles tty_4 = 1423908, cycles tty_5 = 1399794 Cycles = 1437440

speedup(N) = 3.67117

```
proc[0] at cycle 1437439
- INSTRUCTIONS       = 640480
- CPI                = 2.24432
- CACHED READ RATE   = 0.255775
- UNCACHED READ RATE = 0.00139583
- WRITE RATE         = 0.133055
- IMISS RATE         = 0.00952411
- DMISS RATE         = 0.00991948
- IMISS COST         = 41.443
- DMISS COST         = 38.1809
- UNC COST           = 40.1197
- WRITE COST         = 3.74678
```



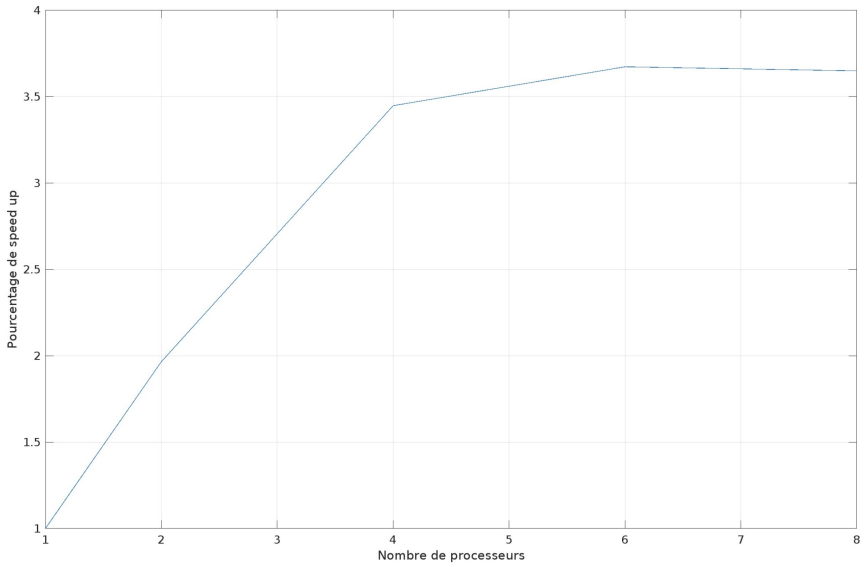
Procs = 8, cycles tty_0 = 1428051, cycles tty_1 = 1420944, cycles tty_2 = 1432182, cycles tty_3 = 1421832, cycles tty_4 = 1440567, cycles tty_5 = 1422285, cycles tty_6 = 1446738, cycles tty_7 = 1424971 Cycles = 1446738

speedup(N) = 3.64758

```
proc[0] at cycle 1446737
- INSTRUCTIONS       = 479618
- CPI                = 3.01644
- CACHED READ RATE   = 0.256264
- UNCACHED READ RATE = 0.00145532
- WRITE RATE         = 0.132879
- IMISS RATE         = 0.00967228
- DMISS RATE         = 0.0102515
- IMISS COST         = 60.8362
- DMISS COST         = 54.3833
- UNC COST           = 58.298
- WRITE COST         = 7.52444
```

| | 1 proc | 2 proc | 4 proc | 6 proc | 8 proc |
|---|---|---|---|---|---|
| cycles | 5277093 | 2685673 | 1531305 | 1437440 | 1446738 |
| speedup | 1 | 1.9649 | 3.44614 | 3.67117 | 3.64758 |

**Comment expliquez-vous que le speedup ne soit pas linéaire?**




Les caractéristiques du programme : certains programmes nécessitent une communication fréquente avec d'autres processeurs, par exemple pour les opérations de verrouillage, de synchronisation, de partage de mémoire, etc. Dans ce cas, l'ajout de processeurs augmente les coûts de communication et entraîne une baisse des performances.

Les caractéristiques des données : dans certains problèmes, la taille et la complexité des données sont fixes. Lorsque le nombre de processeurs augmente, la quantité de données à traiter par chaque processeur diminue, ce qui augmente la proportion de communication entre les processeurs et entraîne une baisse des performances.

***F. Evaluation des temps d'accès au bus***

**Question F1 Pourquoi faut-il absolument effectuer la mesure au moment précis où l'application se termine?**

Si la mesure est effectuée avant la fin de l'application, elle ne prendra pas en compte les opérations effectuées après la demande d'accès au bus. Cela peut inclure des opérations de gestion de la mémoire, des interruptions de service ou des tâches de fond qui peuvent affecter le temps d'accès au bus. Si la mesure est effectuée après la fin de l'application, elle peut inclure des opérations qui ne sont pas liées à l'accès au bus, telles que la libération des ressources utilisées par l'application, qui peuvent fausser les résultats.

**Question F2 Remplissez le tableau ci-dessous, en exécutant successivement le programme sur différentes architectures:**

| NPROCS | 1 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| IMISS COST | 15.9162 | 18.2133 | 27.755 | 41.443 | 60.8362 |
| DMISS COST | 14.7426 | 17.5987 | 26.6206 | 38.1809 | 54.3833 |
| WRITE COST | 0 | 0 | 0.394037 | 3.74678 | 7.52444 |
| ACCESS_TIME | 5277093 | 2685673 | 1531305 | 1437440 | 1446738 |
| CPI | 1.39394 | 1.41601 | 1.60537 | 2.24432 | 3.01644 |

**Question F3 : Interprétez ces résultats. La dégradation de la valeur du CPI quand on augmente le nombre de processeurs est-elle principalement dûe à l'augmentation du coût des MISS, ou à l'augmentation du coût des écritures?**

Par la tableau, nous pouvons voir qu'à mesure que le nombre de processeur augmente, le coût du miss de icache et dcache augmente et le coût d'écriture augmente également. On suppose que si le nombre de processeur augmente, plus de tâches sont appaître en même temps, donc le cache il faut traîter un par un, donc le plus des cycles pour le miss est demandé. Même pour l'écriture, la taille du buffer est fixé, donc le plus des processeurs, plus des tâches d'écriture sont occupés dans le buffer, le plus des cycles est demandé pour l'écriture.

### *G. Modélisation du comportement du bus*

**Question G1 : En exploitant votre compréhension du protocole PIBUS, calculez le nombre de cycles d'occupation du bus pour chacun des 4 types de transaction. Attention : il ne faut pas compter le temps d'attente pour accéder au bus, car la phase REQ->GNT n'utilise pas le bus et a lieu pendant la fin de la transaction précédente. Par contre, il faut compter un cycle supplémentaire correspondant au "cycle mort" entre deux transactions.**

```
Transaction IMISS (Instruction miss) :
La transaction IMISS est utilisée pour récupérer une instruction manquante dans le cache de

Cycle 1 : Le maître envoie une demande IMISS (REQ) sur le bus.
Cycle 2 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 3 : Le maître envoie l'adresse de la ligne de cache à récupérer sur le bus.
Cycle 4 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 5 : Le maître reçoit les données de la ligne de cache du bus.
Cycle 6 : Le maître relâche le bus.

Transaction DMISS (Data miss) :
La transaction DMISS est utilisée pour récupérer une donnée manquante dans le cache de donné
```

```
Cycle 1 : Le maître envoie une demande DMISS (REQ) sur le bus.
Cycle 2 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 3 : Le maître envoie l'adresse de la ligne de cache à récupérer sur le bus.
Cycle 4 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 5 : Le maître envoie les données à écrire dans la ligne de cache sur le bus.
Cycle 6 : Le maître relâche le bus.

Transaction UNCACHABLE :
La transaction UNCACHABLE est utilisée pour accéder à une adresse non mise en cache. Le nomb

Cycle 1 : Le maître envoie une demande UNCACHABLE (REQ) sur le bus.
Cycle 2 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 3 : Le maître envoie l'adresse de l'emplacement de mémoire à accéder sur le bus.
Cycle 4 : Le maître relâche le bus.

Transaction WRITE :
La transaction WRITE est utilisée pour écrire des données dans le cache. Le nombre de cycles

Cycle 1 : Le maître envoie une demande WRITE (REQ) sur le bus.
Cycle 2 : Le contrôleur de bus arbitre la demande et envoie un signal GNT au maître.
Cycle 3 : Le maître envoie les données à écrire dans le cache sur le bus.
Cycle 4 : Le maître relâche le bus.
```

Il y a toujours un cycle mort entre deux transactions pour permettre au bus de se reposer. Ce cycle mort est inclus dans les nombres de cycles d'occupation du bus pour chaque transaction

**Question G2 : En exploitant les résultats de la partie D (pourcentage de lectures cachées, pourcentage d' écritures, taux de MISS sur les caches, valeur du CPI), calculez la fréquence de chacun des 4 types de transaction et complétez le tableau ci-dessous. Puisque notre unité de temps est le cycle, ces fréquences seront exprimées en nombre d'événements par cycle.**

Proc = 1 | | Temps_Occupation | Fréquence | | ——— | ——— | ——— | | IMISS | 48360.93 | 36873.28 | | DMISS | 52313.49 | 40156.03 | | UNC | 6150.25 | 5267.09 | | WRITE | 696237.63 | 505338.61 |

Proc = 2 | | Temps_Occupation | Fréquence | | ——— | ——— | ——— | | IMISS | 34551477.1783 | 0.5049 | | DMISS | 33387208.4737 | 0.5049 | | UNC | 16974371.8103 | 3724.41 | | WRITE | 0 | 357752.29 |

Proc = 4 | | Temps_Occupation | Fréquence | | ——— | ——— | ——— | | IMISS | 26469579.765 | 0.3722 | | DMISS | 25392366.5178 | 0.3722 | | UNC | 26592381.0384 | 2190.25 | | WRITE | 603615.384568 | 203677.58 |

Proc = 6 | | Temps_Occupation | Fréquence | | ——— | ——— | ——— | | IMISS | 26562077.44 | 0.1986 | | DMISS | 24438362.312 | 0.1986 | | UNC | 57646325.2683 | 2009.82 | | WRITE | 5390533.89902 | 191223.49 |

Proc = 8 | | Temps_Occupation | Fréquence | | ——— | ——— | ——— | | IMISS | 29193181.6956 | 0.1158 | | DMISS | 26084021.7814 | 0.0973 | | UNC | 84285656.126 | 2104.36 | | WRITE | 10893811.7713 | 192216.47 |

**Question G3 : En utilisant les résultats des deux questions précédentes, calculez le pourcentage de la bande passante du bus utilisé par un seul processeur. En déduire le nombre de processeurs au delà duquel le bus commence à saturer.**

Pourcentage = 0.255661 + 0.00139101 + 0.133484 = 0.39053601 Si le vitesse du bus est 2.5 Go/s 2.5 / 0.39053601 = 6.404 Nous en concluons qu'au delà de 6 processeurs, le bus commence à saturer.