SARROCHE Nicolas
SI4 G1

Programmation Concurrente

Simulation d'un plaque chauffée

Introduction

Ce projet de programmation concurrente a pour but de simuler la diffusion de chaleur sur une plaque chauffée à une certaine température en son centre. Il permettra de mettre en avant les avantages et inconvénients de la programmation parallèle à l'aide de threads, barrière, sémaphores, etc … L'avancement de ce projet se fait en cinq étapes successives : de manière itérative (un processeur de calcul, un thread), avec des threads utilisant des barrières Posix (un processeur de calcul, plusieurs threads), avec des threads utilisant des barrières "faites maison" (un processeur de calcul, plusieurs threads), etc …

Version Itérative

Cette version a pour but de simuler la plaque qui chauffe en son centre, et ce, en n'utilisant qu'un seul coeur de calcul.

Algorithme principal

- récupération des paramètres
- pour chaque exercice spécifié :
    - pour chaque taille S de matrice spécifiée :
    - création de la matrice de taille 2^(S+4)
        - réinitilisation de la matrice
        - exécution de l'exercice* et récupération des stats
        - affichage des stats si demandé

SARROCHE Nicolas
SI4 G1

Dans notre cas on exécute la fonction “runIterative“

- Exécution de l’exercice :
- démarrage des chronos
- réinitialisation de la matrice
- **pour** 0 <= i < exec_number
- diffusion_2d en X // diffusion latérale
- diffusion_2d en Y // diffusion verticale
- mise à jour de la température centrale
- **finpour**
- fin des chronos

SARROCHE Nicolas
SI4 G1

Version Thread Posix

Cette version diffère de la version itérative car il faut maintenant paralléliser l'exécution sur plusieurs thread. Le nombre de thread variant de 1 à 1024 et pouvant être passé en paramètre. Pour synchroniser les diférents threads, on utilisera une barrière Posix.

Chaque Thread manipulera une structure "matrix_chunk" qui contient :

```
int size; // la taille du chunk
int idX, idY; // coordonnées du chunk
matrix_2d * m; // référence vers la matrice de base
int execution_number; // nombre d'éxécution (i.g. 10000)
```

Algorithme principal

- récupération des paramètres
- pour chaque exercice spécifié :
    - pour chaque taille S de matrice spécifiée :
    - création de la matrice de taille 2^(S+4)
        - pour chaque nombre t de thread spécifié :
            - réinitilisation de la matrice
            - exécution de l'exercice* et récupération des stats
            - affichage des stats si demandé

Dans notre cas on exécute la fonction "runThreadPosix"

- Exécution de l'exercice :
- démarrage des chronos
- réinitialisation de la matrice
- calcul du nombre de thread, du nombre de thread par ligne, de la leur taille
- initialisation de barrières **Posix** globales
- initalisation des chunks
- initialisation et lancement des threads*
- jointure des threads
- fin des chronos

SARROCHE Nicolas
SI4 G1

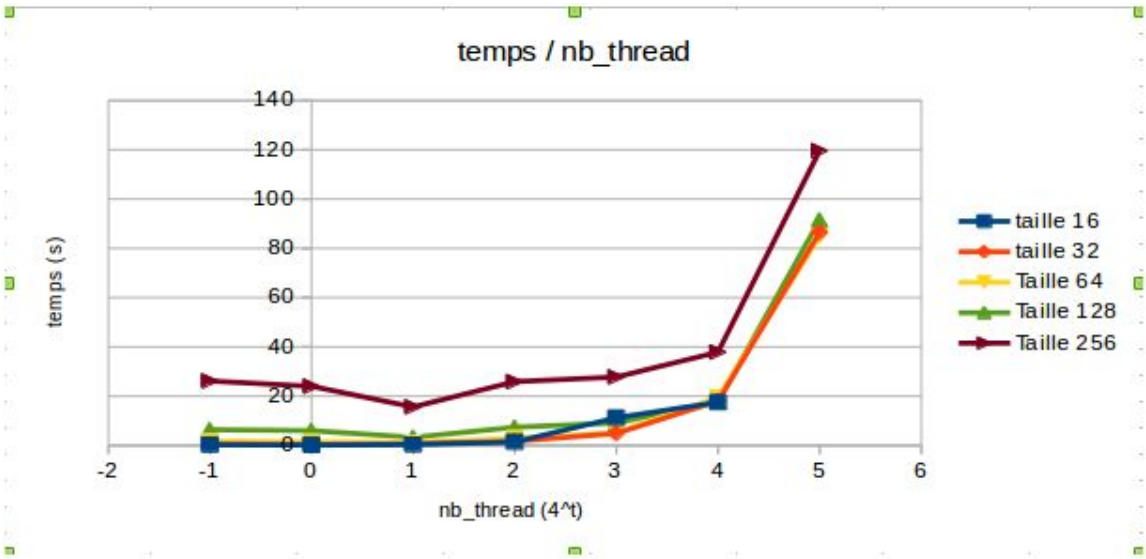
L'initialisation des threads se fait en passant en paramètre la fonction "main_posix_thread" avec le chunk associé :

- Initialisation des threads :
- récupération du chunk associé // passé en paramètre
- réinitialisation du buffer associé a la matrice
- barrière A // on attend que tous les threads aient récupéré leur chunk
- pour 0 <= i < execution_number :
    - on lance l'exécution horizontale
    - barrière A // on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - barrière A // on attend que l'échange soit fait
    - on lance l'exécution verticale
    - barrière A // on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - on réchauffe le centre

L'échange de pointeur est indispensable pour ne pas écraser les anciennes données et ainsi permettre d'avoir un système équivalent à avoir une matrice "écriture" et une matrice "lecture".

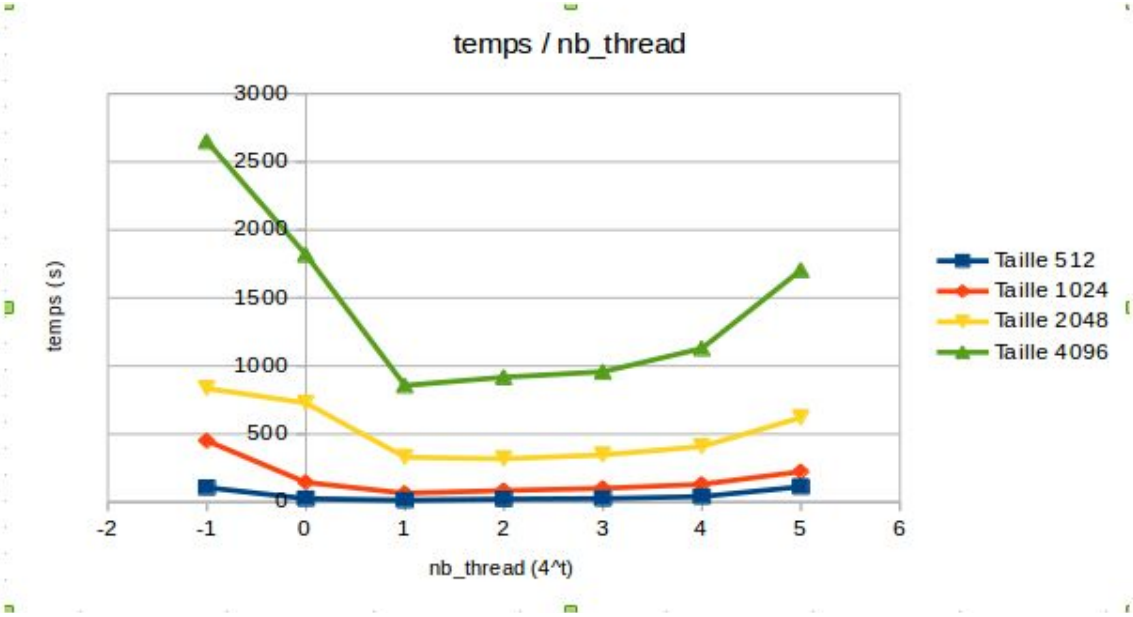
Résultats

Pour une matrice de taille NxN, N<=256, le temps d'exécution utilisateur augmente de manière exponentielle avec le nombre de thread
Taille 16x16: (pas de donnée pour 1024 threads)
(En abscisse "-1", représente la version itérative)



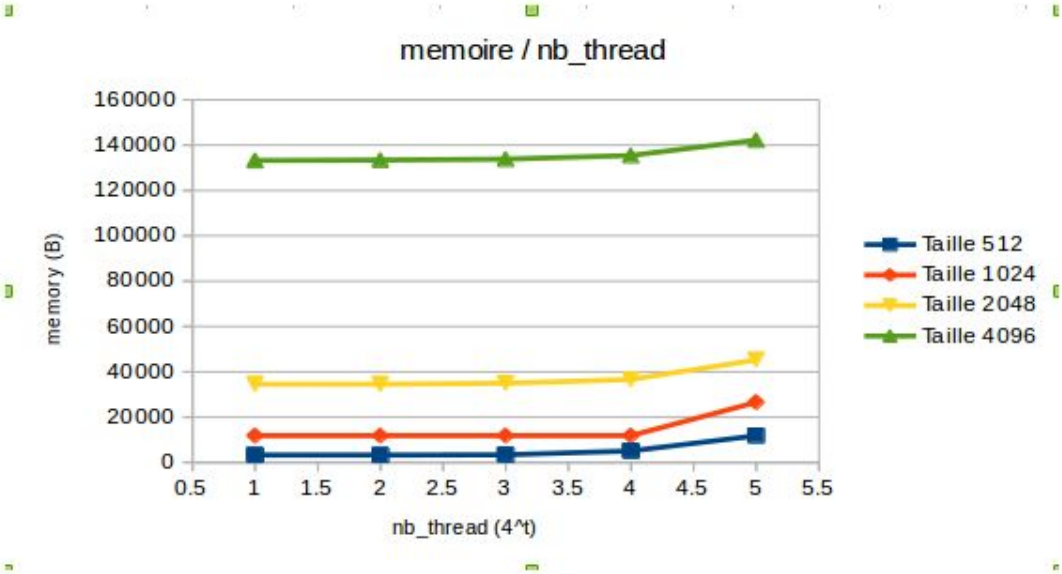
SARROCHE Nicolas
SI4 G1

Pour une matrice de taille NxN, N > 256, on se rend compte que le temps d'exécution itératif est plus long que le temps avec 4 threads.
(En abscisse "-1", représente la version itérative)



Mémoire

On constate que pour 1024 threads, l'espace mémoire est plus important que pour 256, 64 et 16 threads quelque soit la taille de la matrice.



SARROCHE Nicolas
SI4 G1

Version Thread variable condition

Cette version diffère de la précédente en un seul sens : les barrières utilisées ici sont une implémentation personnelle mais similaire à la barrière Posix en son utilisation.

Cette barrière est définie comme une structure “Barrière_impl” qui contient :

int count; // le nombre de thread attendant actuellement
int max_thread; // le nombre maximal de thread à attendre
pthread_cond_t cond_t; // variable de condition de la barrière
pthread_mutex_t mutex_t; // mutex de la barrière

Algorithme principal

L’algorithme est très similaire, puisque la seule chose qui diffère est l’utilisation de cette nouvelle barrière.

- récupération des paramètres
- pour chaque exercice spécifié :
    - pour chaque taille S de matrice spécifiée :
    - création de la matrice de taille 2^(S+4)
        - pour chaque nombre t de thread spécifié :
            - réinitilisation de la matrice
            - exécution de l’exercice* et récupération des stats
            - affichage des stats si demandé

Dans notre cas on exécute la fonction “runThreadCustom“

- Exécution de l’exercice :
- démarrage des chronos
- réinitialisation de la matrice
- calcul du nombre de thread, du nombre de thread par ligne, de la leur taille
- initialisation de barrières **Custom** globales
- initalisation des chunks
- initialisation et lancement des threads*
- jointure des threads
- fin des chronos

L'initialisation des threads se fait en passant en paramètre la fonction "main_custom_thread" avec le chunk associé :

- Initialisation des threads :
- récupération du chunk associé // passé en paramètre
- réinitialisation du buffer associé a la matrice
- barrière A // on attend que tous les threads aient récupéré leur chunk
- pour 0 <= i < execution_number :
    - on lance l'exécution horizontale
    - barrière A // on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - barrière A // on attend que l'échange soit fait
    - on lance l'exécution verticale
    - barrière A // on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - on réchauffe le centre

L'échange de pointeur est indispensable pour ne pas écraser les anciennes données et ainsi permettre d'avoir un système équivalent à avoir une matrice "écriture" et une matrice "lecture".

Barrière

Pour initialiser la barrière avec N thread :

- initialisation de counter à 0
- initialisation de max_thread à N
- initialisation de la variable de condition
- initialisation du mutex

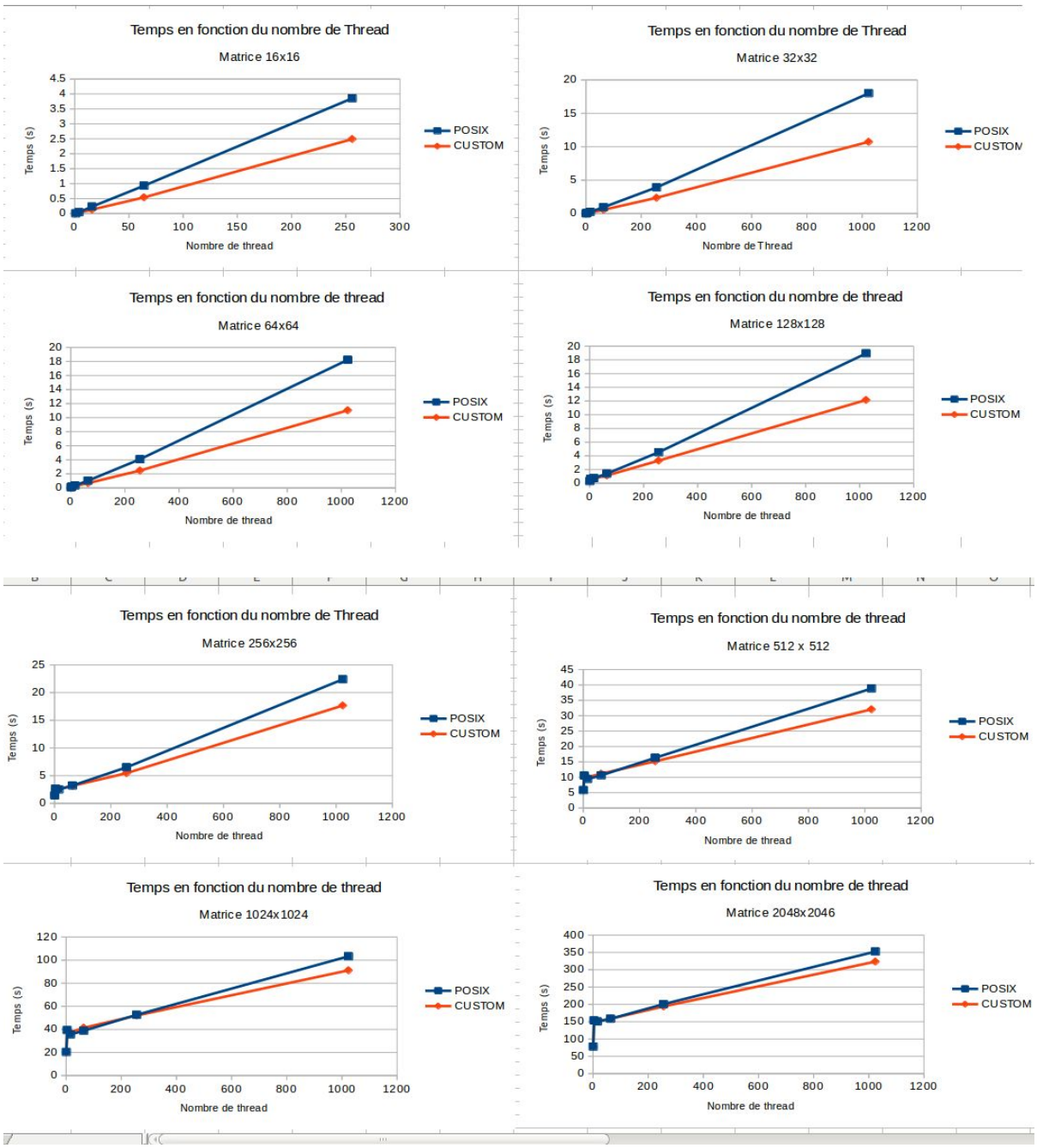
Chaque fois qu'un thread appel la fonction "wait" :

- incrémentation de la variable counter
- si counter = max_thread // tous les threads on atteint la barrière
    - remise à 0 de counter
    - envoie d'un signal de réveil à tous les threads
- sinon
    - mise en attente du thread

SARROCHE Nicolas
SI4 G1

Résultats

On observe que la barrière Posix est sensiblement plus lente que la barrière implémentée. Ceci peut être dû à des différences de contraintes et de vérifications qui ne sont pas effectuées dans notre version par exemple.



Cependant, et même si le graphe est manquant, l'exécution devient plus lente pour une matrice de taille 4096x4096, et ce, quelque soit le nombre de thread.

SARROCHE Nicolas
SI4 G1

Version Thread et Sémaphore

Dans cette version, le barrières utilisent des sémaphores embarqués dans une structure. L'utilisation reste sensiblement la même comparée aux deux versions précédentes.

Cette structure "Semaphore_impl" contient :

```
int count; // le nombre de thread attendant actuellement
int max_thread; // le nombre maximal de thread à attendre
sem_t * attente_t; // semaphore de comptage de thread
sem_t * mutex_t; // mutex de la barrière
```

Algorithme principal

L'algorithme est très similaire, puisque la seule chose qui diffère est l'utilisation de cette nouvelle barrière.

- récupération des paramètres
- pour chaque exercice spécifié :
    - pour chaque taille S de matrice spécifiée :
    - création de la matrice de taille 2^(S+4)
        - pour chaque nombre t de thread spécifié :
            - réinitilisation de la matrice
            - exécution de l'exercice* et récupération des stats
            - affichage des stats si demandé

Dans notre cas on exécute la fonction "runThreadSemaphore"

- Exécution de l'exercice :
- démarrage des chronos
- réinitialisation de la matrice
- calcul du nombre de thread, du nombre de thread par ligne, de la leur taille
- initialisation de la **structure Semaphore_impl SE** globales
- initalisation des chunks
- initialisation et lancement des threads*
- jointure des threads
- fin des chronos

L'initialisation des threads se fait en passant en paramètre la fonction "main_semaphore_thread" avec le chunk associé :

- Initialisation des threads :
- récupération du chunk associé // passé en paramètre
- réinitialisation du buffer associé a la matrice
- wait_on_semaphore SE // on attend que tous les threads aient récupéré leur chunk
- pour 0 <= i < execution_number :
    - on lance l'exécution horizontale
    - wait_on_semaphore SE//on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - wait_on_semaphore SE // on attend que l'échange soit fait
    - on lance l'exécution verticale
    - wait_on_semaphore SE//on attend que tous les threads aient fini leurs calculs
    - on échange les pointeurs de la matrice et du buffer
    - on réchauffe le centre
    - wait_on_semaphore SE // on attend que le coeur de la plaque soit chaud

L'échange de pointeur est toujours indispensable pour ne pas écraser les anciennes données et ainsi permettre d'avoir un système équivalent à avoir une matrice "écriture" et une matrice "lecture".

Sémaphore

Pour initialiser la structure sémaphore avec N thread :

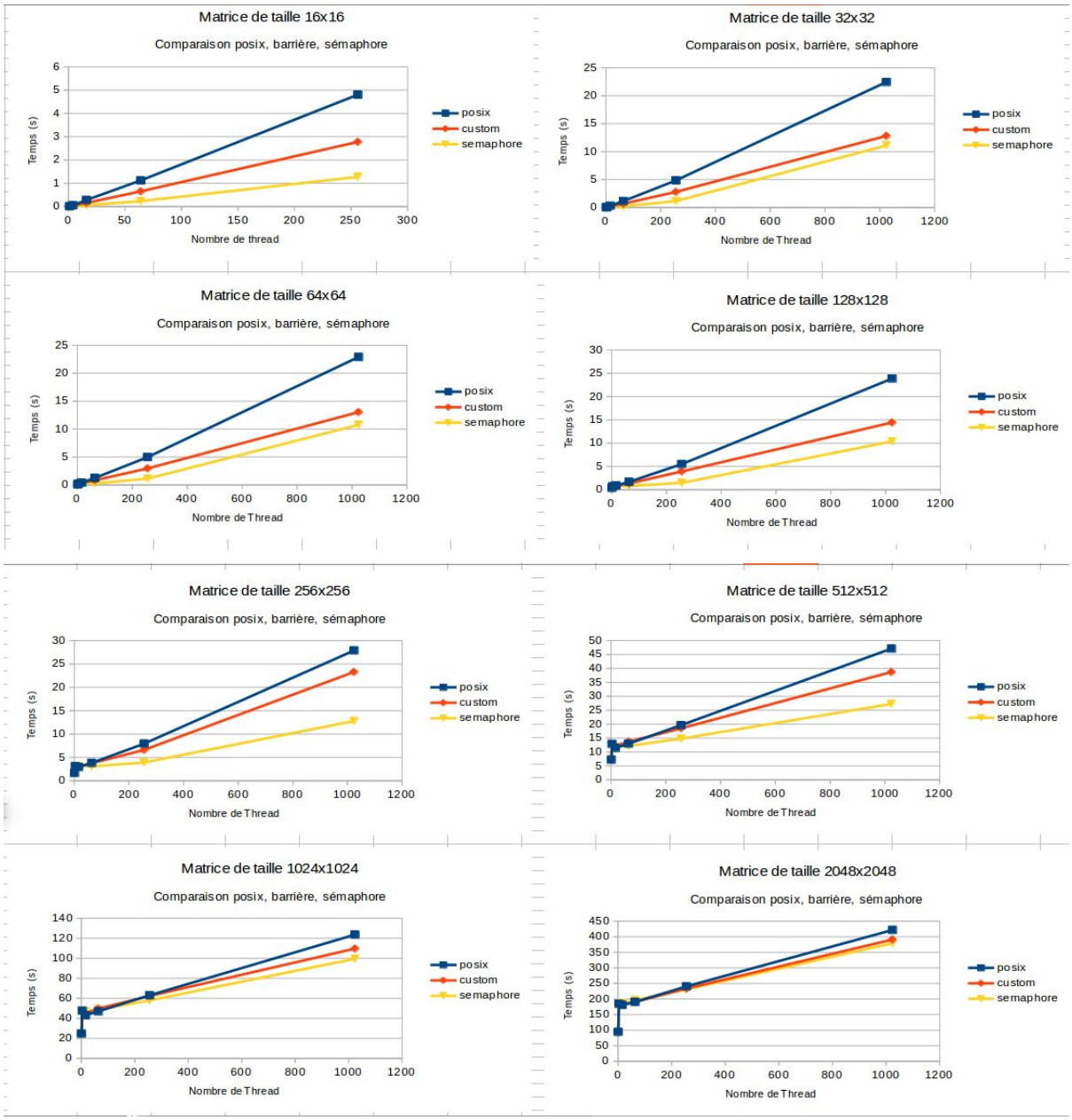
- initialisation de counter à 0
- initialisation de max_thread à N
- initialisation du semaphore "attente_t"
- initialisation du semaphore "mutex_t"

Chaque fois qu'un thread appel la fonction "wait_on_semaphore" :

- prendre le sémaphore "mutex_t"
- incrémenter la variable counter
- si counter = max_thread // tous les threads on atteint la barrière
    - pour 0 < i < max_thread
        - libérer les thread en attente sur "attente_t"
    - remise à 0 de counter
    - libérer le mutex
- sinon
    - rendre le sémaphore "mutex_t"
    - entrer en attente sur le semaphore "attente_t"

SARROCHE Nicolas
SI4 G1

Résultats



On remarque que la version avec les sémaphores est légèrement plus rapide que son homologue avec la variable de condition, et, par conséquent, plus rapide que la version POSIX.

SARROCHE Nicolas
SI4 G1

Conclusion

Ce projet permet donc de voir l'évolution de la chauffe d'une plaque simulée dans une matrice carré dont le centre serait à température constante. L'approche incrémentale permet de bien aborder les différents problèmes (compréhension de l'algorithme, compréhension des threads, compréhension des barrières et mutex, etc..).