HIGH PERFORMANCE PERFORMANCE MEMORY ALLOCATOR

CODE

Valoc est un compilateur qui utilise mmap et munmap pour alouer de la mémoire. Ces opérations systemes bas niveau et implique un coup non-négligeable sur la performance de l’allocateur. En revanche, elle propose une versatilité des optimisations consequentes et peut dans certaines conditions d’optimisation être plus performant.

Valloc sert à alloué de la mémoire

Vfree sert à libérer la mémoire.

Implémentation simple de l’allocateur.

```c
// Allocation simple avec mmap
void* valloc(size_t size) {
    void* ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (ptr == MAP_FAILED) {
        perror("valloc failed");
        return NULL;
    }
    return ptr;
}

// Libération de la mémoire
void vfree(void* ptr, size_t size) {
    if (ptr != NULL && size > 0) {
        if (munmap(ptr, size) == -1) {
            perror("vfree failed");
        }
    }
}
```

OPTIMISATION

SEGMENTATION PAR BLOCS

CODE

1

```c
typedef struct MemoryBlock {
    void* adress;
    size_t size;
    bool status;  // true = libre, false = occupé
} MemoryBlock;
```

Notre implémentation de la **ségmentation par bloc** est réaliser via la structure **MemoryBlock.**
Qui prend en argument **l'adresse** vers la mémoire alouée. La **taille** alouée. Et enfin un **état** qui indique si l'espace est libre ou non.

2

```c
typedef struct MemoryAllocator {
    MemoryBlock* blocks;
    size_t total_blocks;
    size_t used_blocks;
    size_t recycled_blocks;
    pthread_mutex_t mutex;
    bool initialized;
    ThreadCache thread_caches[MAX_THREADS];
    int num_threads;
} MemoryAllocator;
```

La structure **MemoryAllocator** est la structure principale qui gère toute l'allocation de mémoire dans le projet.

3

```c
int valloc_init(MemoryAllocator* allocator, size_t initial_blocks, int num_threads) {
    if (allocator == NULL || initial_blocks == 0 || num_threads <= 0 || num_threads > MAX_THREADS) {
        return -1;
    }

    allocator->blocks = (MemoryBlock*)malloc(initial_blocks * sizeof(MemoryBlock));
    if (allocator->blocks == NULL) {
        return -1;
    }

    // Initialisation des blocs
    for (size_t i = 0; i < initial_blocks; i++) {
        allocator->blocks[i].adress = NULL;
        allocator->blocks[i].size = 0;
        allocator->blocks[i].status = true;  // true = libre
    }
```

Ici nous retrouvons une des utilisations de la structure **MemoryBlock** et **MemoryAllocator** dans l'allocation d'un bloc mémoire...

4

```c
void free_valloc(MemoryAllocator* allocator, void* ptr) {
    if (allocator == NULL || !allocator->initialized || ptr == NULL) {
        return;
    }

    pthread_mutex_lock(&allocator->mutex);

    for (size_t i = 0; i < allocator->total_blocks; i++) {
        if (allocator->blocks[i].adress == ptr) {
            size_t size = allocator->blocks[i].size;

            // Essayer d'abord de mettre dans le cache
            ThreadCache* cache = get_thread_cache(allocator);
            if (cache) {
                cache_free(cache, ptr, size);
            } else {
                munmap(ptr, size);
            }
```

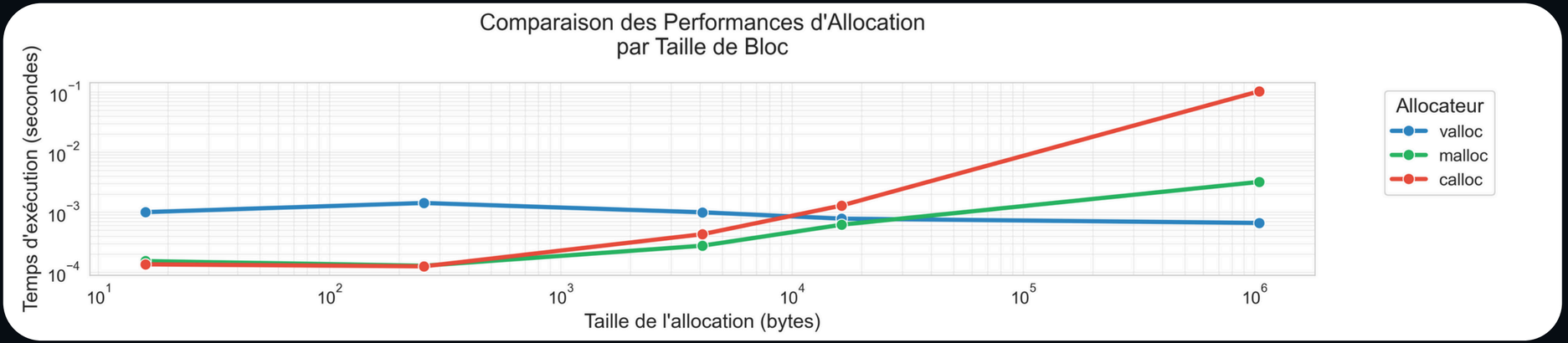
...et la libération d'un bloc mémoire.

Valoc

BENCHMARK

Pour les petites allocations:
- malloc et calloc sont plus rapides

Pour les grandes allocations:
- valloc est plus rapides



De plus, on remarque une régularité dans la performance, quelle que soit la taille du bloc pour valloc. Ceci est dû à la méthode de segmentation.

Les résultats sont cohérent avec l'implémenation par blocs

Valoc

OPTIMISATION

MULTI-THREADING

CODE

1

```
static __thread int thread_id = -1;
static int next_thread_id = 0;
static pthread_mutex_t thread_id_mutex = PTHREAD_MUTEX_INITIALIZER;

static int get_thread_id() {
    if (thread_id == -1) {
        pthread_mutex_lock(&thread_id_mutex);
        thread_id = next_thread_id++;
        pthread_mutex_unlock(&thread_id_mutex);
    }
    return thread_id;
}
```

Insertion d’un système d’id pour chaque thread avec next_thread_id.L’attribution des ids est géré par le mutex thread_id_mutex qui synchronise chaque id.

2 ALLOCATION

L’allocateur va d’abord faire une tentative d'allocation depuis le cache local du thread Si le cache est vide, allocation depuis le MemoryAllocator avec synchronisation

3 SÉCURITÉ

Des mesures de sécurités sont nécessaires pour une meilleur fiabilité. Cependant elle pose des problèmes de scalabilités. Tout d'abord, l'initialisation crée un nombre spécifié de caches de threads. Chaque cache est initialisé avec son propre mutex. Enfin, une limite configurable MAX_THREADS définit le nombre maximum de threads.

4

```
void free_valloc(MemoryAllocator* allocator, void* ptr) {
    if (allocator == NULL || !allocator->initialized || ptr == NULL) {
        return;
    }

    pthread_mutex_lock(&allocator->mutex);
```

La libération de la mémoire tente d’abord la mise en cache dans le cache local du thread.Si le cache est plein, libération directe avec munmap

BENCHMARK

TEST EFFECTUER :

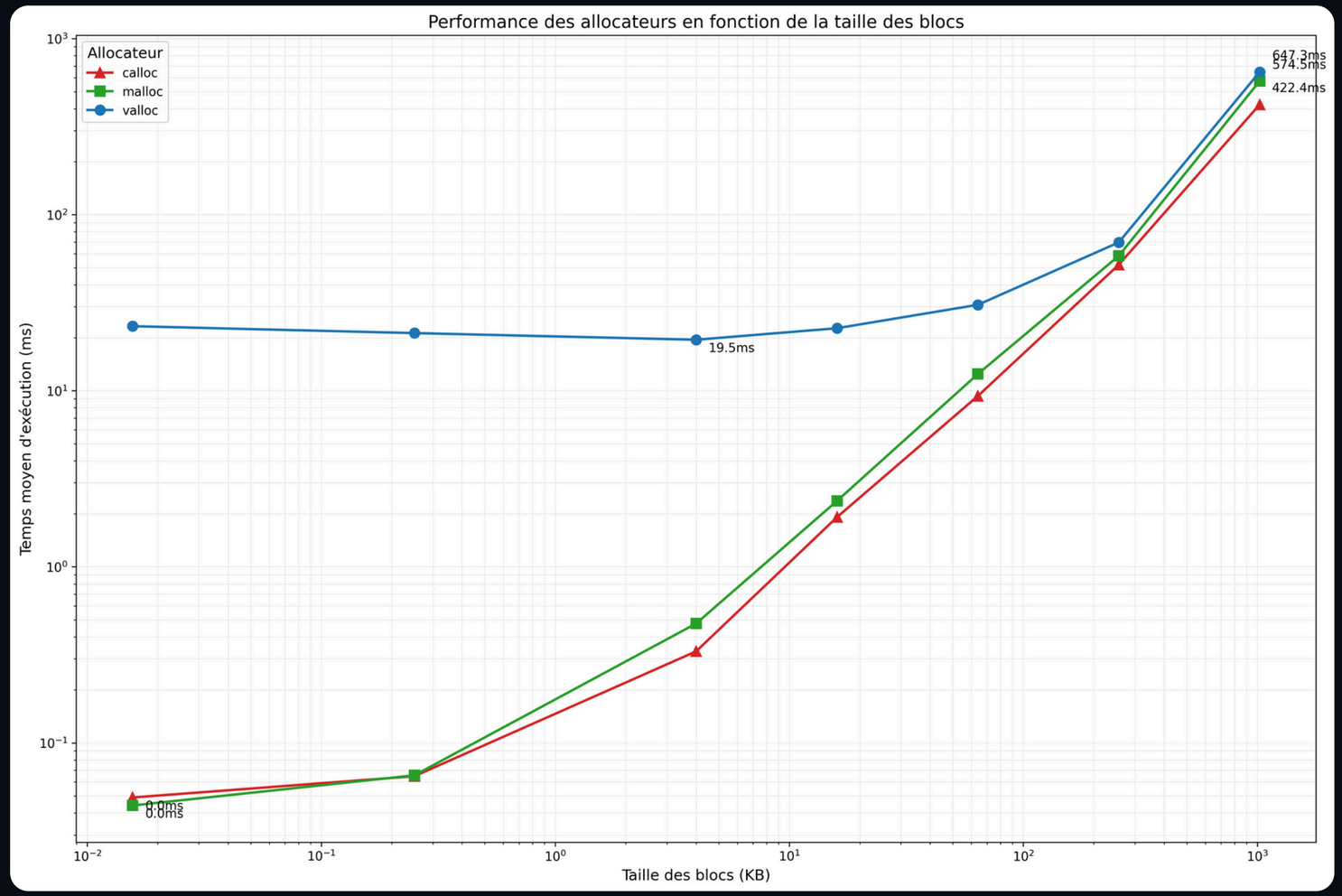
Sur 4 threads avec 1000 allocations par threads (choix arbritraire) et sur 5 iterations afin d'obtenir une moyenne.

IMPLÉMENTATION DU TEST :

Les tests utilisent tout d'abord memset pour remplir la mémoire allouée. Ils utilisent également un mutex, qui agit comme un gestionnaire de mémoire en ordonnançant les threads afin qu'ils accèdent au CSV, garantissant une gestion cohérente des données.

RESULTAT DU TEST :

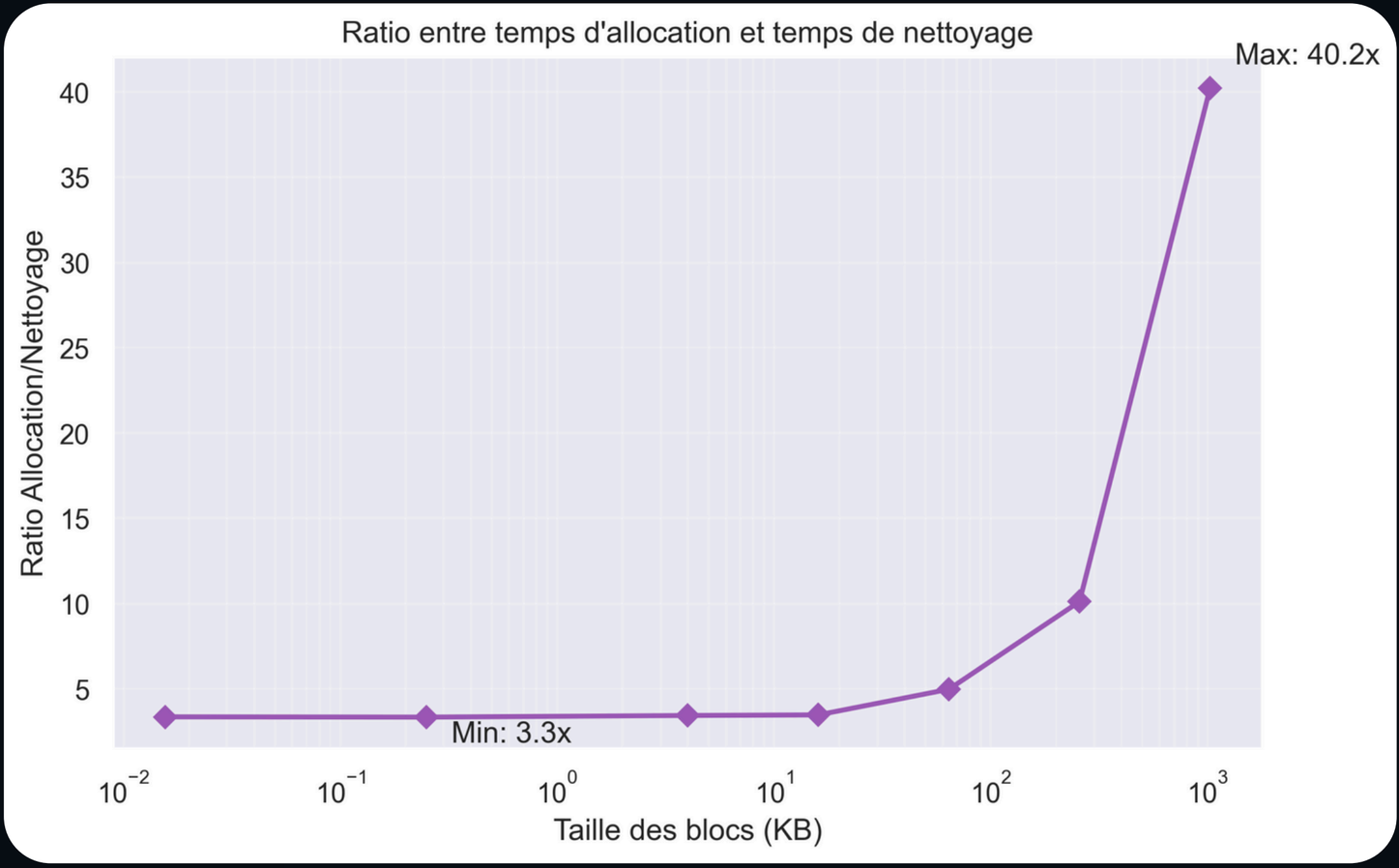
Pour des petites allocations malloc et calloc sont plus performant.
Cependant pour des allocations qui dépensant les 100 KB.On retrouve une convergence des allocateurs.



Temps moyen = (Temps d'allocation + Temps de libération) / 2

Valoc

BENCHMARK



Nous pouvons constater une différence exponentielle entre l'allocation et la libération de mémoire. Dans certains cas, cela peut engendrer un goulot d'étranglement significatif.

Une des méthodes d’optimisation serait d’implémenter une méthode de cache

Valoc

OPTIMISATION

CACHE

CODE

1

```
ThreadCache* get_thread_cache(MemoryAllocator* allocator) {
    int id = get_thread_id();
    if (id >= allocator->num_threads) return NULL;
    return &allocator->thread_caches[id];
}
```

Chaque thread a son propre cache local
Stocke temporairement des blocs récemment libérés
Accès rapide sans contention avec les autres threads
Taille limitée (MAX_CACHE_BLOCKS)

2 Allocation

```
void* cache_allocate(ThreadCache* cache, size_t size) {
    if (!cache) return NULL;

    pthread_mutex_lock(&cache->mutex);

    // Recherche d'un bloc de taille appropriée dans le cache
    for (int i = 0; i < cache->count; i++) {
        if (cache->blocks[i].size == size) {
            void* ptr = cache->blocks[i].ptr;
            // Déplacer le dernier bloc à cette position
            cache->blocks[i] = cache->blocks[--cache->count];
            pthread_mutex_unlock(&cache->mutex);
            return ptr;
        }
    }

    pthread_mutex_unlock(&cache->mutex);
    return NULL;
```

3 Libération

```
void cache_free(ThreadCache* cache, void* ptr, size_t size) {
    if (!cache || !ptr) return;

    pthread_mutex_lock(&cache->mutex);

    if (cache->count < MAX_CACHE_BLOCKS) {
        cache->blocks[cache->count].ptr = ptr;
        cache->blocks[cache->count].size = size;
        cache->count++;
        pthread_mutex_unlock(&cache->mutex);
        return;
    }

    pthread_mutex_unlock(&cache->mutex);
    munmap(ptr, size);
}
```