# Parallel cp

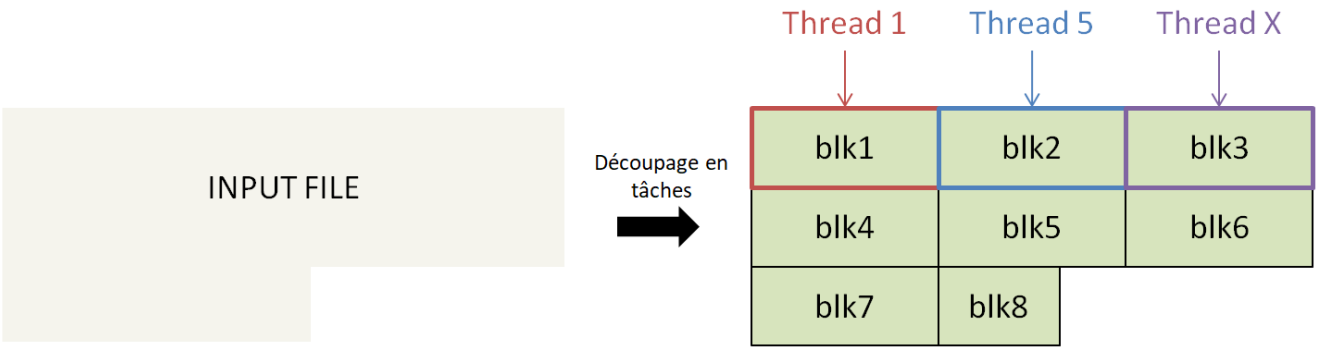
Copier un fichier *source* vers un fichier *destination* en utilisant plusieurs threads. Pour cela le fichier *source* est découpé en blocks de N octets, chaque block est copié dans le fichier *destination* par un thread différent. *pcp* utilise en interne un pool de threads afin de réutiliser les threads.

**Important**:

Chaque thread devra utiliser une position différente (offset) et un stream différent pour permettre de paralléliser l'écriture dans un même fichier.

Le fichier de *destination* doit être créé puis changé de taille (identique à la taille du fichier *source*) avant de pouvoir y écrire avec les threads.

S'aider de *std::ifstream*, *std::ofstream* et *std::filesystem*.




## Utilisation

Créer un fichier (ex: 1G) en utilisant /dev/urandom et utiliser *pcp*. Vérifier que les fichiers *source* et *destination* sont identiques avec la commande *diff*. Une fois la copie terminée, *pcp* doit indiquer le temps pris pour faire la copie du fichier (s'aider de *std::chrono*).

```
$ pcp <source> <destination> [-b <N>] [-t <T>] [-p]
```

L'option '-b' force à utiliser des blocks de N octets. Si 8 threads et '-b 4096' alors *pcp* utilisera X blocks de 4096 octets. Par défaut la taille des blocks dépendra du nombre threads, si 8 threads alors *pcp* créera 8 blocks de N octets.

L'option '-p' affiche le nombre de tâches réalisées par chaque thread.

L'option '-t' indique à *pcp* le nombre de threads à utiliser (par défaut std::thread::hardware_concurrency threads)

Exemples:

```
# force la taille des blocks a 4096
# utilisation de 2 threads
# afficher thread stats
$ pcp /tmp/toto.txt /tmp/titi.txt -b 4096 -t 2 -p
Took ...

Threads | tasks
---------------
0       | 10
1       | 28

# force des blocks de 4096 octets
$ pcp /tmp/toto.txt /tmp/titi.txt -b 4096
Took ...

# le nombre de block est egale au nombre de threads
# et la taille des blocks est (taille fichier) / (nb threads), attention au
dernier block
$ pcp /tmp/toto.txt /tmp/titi.txt
Took ...
```

## class Worker

La classe *Worker* encapsule un *std::thread* qui exécute les tâches. Le thread est créé dans le constructeur (ou dans une méthode *start*) et il est détruit dans le destructeur. Le *Worker* ne peut exécuter qu'une seule tâche à la fois. Si le thread n'a pas de tâche alors il doit s'endormir et sera réveillé dès qu'une nouvelle tâche est disponible. Le mécanisme de sommeil et de réveil du thread est implémenté grâce à *std::mutex* et à *std::condition_variable*.

Interface *Worker.h*:

```
class Worker
{
public:

    using task_t = std::function<void()>;

    Worker(WorkerPool& pool);

    // detruire m_thread (std::thread::join)
    ~Worker();

private:
    // m_thread loop,
    // le thread est bloquant si pas de tache
    void loop() {

        // ...

        while(true)
        {
            WorkerPool::worker_data_t data = m_pool.get_task();

            // data contient deux informations:
            // 1 - la task_t a executer (si presente)
            // 2 - est ce que m_thread doit sortir de sa boucle (break)

            // ...

            m_pool.finalize_task();
        }

        // ...
    }

    std::string m_name; // identifiant/nom du Worker
    std::thread m_thread; // thread utiliser pour executer une tache

    WorkerPool& m_pool;
};
```

## class WorkerPool

Le *WorkerPool* initialize un nombre *nb_threads* de thread *Worker* (*std::thread::hardware_concurrency*) et stocke les tâches à exécuter.

Interface *WorkerPool.h*:

```
class WorkerPool
{
public:

    // retourne un task_t et/ou un boolean permettant de sortir de sa boucle
    // s'aider de std::optional, std::variant, std::any, std::tuple (au choix)
    using worker_data_t = ...

    // permet au Worker d'acceder aux membres/methodes privees de WorkerPool
    friend class Worker;

    // creer les Worker
    WorkerPool(unsigned int nb_threads);

    // detruire les Workers
    // 1 - passer m_should_stop a true
    // 2 - notifier aux threads le changement
    ~WorkerPool();

    // 1 - ajoute la tache a m_tasks
    // 2 - notifier la presence d'une nouvelle tache pour les Workers (s'aider de
m_task_cv)
    template<typename F, typename... A>
    void add_task(F&& task, A&&... args);

    // attends la terminaison de toutes les taches,
    // en cours et en attente dans m_tasks.
    // L'appelant est bloque tant que (m_running_tasks != 0) && (m_tasks.empty()
== false)
    void wait_all();

private:

    // Appeler par Worker pour recuperer une tache dans m_tasks:
    // 1 - attend qu'une tache soit disponible (bloquant si pas de tache dispo)
    // 2 - incremente m_running_tasks
    // 3 - retourne la tache au Worker et/ou boolean indiquant la sortie de la
boucle Worker::loop
    worker_data_t get_task();

    // Appeler par Worker pour indiquer la terminaison d'une tache:
    // 1 - decrementer m_running_tasks
    // 2 - notifier qu'une tache est terminee (pour wait_all)
    void finalize_task();
```

```
    std::queue<std::function<void()>> m_tasks;

    // mutex protege l'acces a m_tasks
    // et m_task_available_cv
    mutable std::mutex m_tasks_mutex;

    // utilise pour get_task
    std::condition_variable m_task_cv;

    std::vector<std::unique_ptr<Worker>> m_workers;

    // true si les threads doivent s'arreter (sortir de la boucle Worker::loop)
    bool m_should_stop;

    // nb tache en cours
    unsigned int m_running_tasks;
};
```

## Main

```
int main()
{
    // 1 - creer les taches

    // 2 - demarrer chrono

    // 3 - ajouter toutes les taches
    pool.add_task(...);

    // 4 - attendre terminaison des taches
    pool.wait_all();

    // 5 - arreter chrono

    // 6 - afficher chrono
    // 7 - afficher stats (si besoin)
}
```