ENSEIRB CISD 2022-2023, Analyses et Post-Traitements In-Situ

TP n°2 :

Avant tout, il faut finir le TP n°1. Et notamment le dernier exercice.

1) **Étude avec plusieurs analyses de type histogramme (Exercice N°3 TP N°1)**

Maintenant copier le fichier
~/TPs_In_Situ/exaStamp/data/samples/tutorial_insitu_histo2_par_freq.msp

Exercices :

1. Étudier le jeu de données. Combien d'analyses vont être exécutées et à quelles fréquences ? Vérifier votre hypothèse avec des traces vite.
2. Faites varier le nombre de threads. Que constatez-vous ?
3. Modifier la fréquence d'appel des analyses et éventuellement des samples et vérifier que les traces vite valident bien votre hypothèse.
4. Qu'est-ce qui peut perturber les mesures ?

Début des exercices du TP n°2

---

2) **Réservation d'une partie des cœurs pour l'analyse (Exercice N°1 TP N°2)**

Créer un répertoire Exercice2.

Récupérer le fichier de données : tutorial_insitu_histo2_space_sharing.msp
toujours dans ~/Tps_In_Situ/exaStamp/data/samples/.

Exercices :

1. Exécuter ce jeu de données avec 4 puis 8 threads OpenMP. Étudier les traces vite. Que constatez-vous ? Est-ce que ce comportement est bien conforme avec ce qui est demandé dans le jeu de données.
2. Qu'est-ce qui peut perturber les performances de la simulations ? Modifier le jeu de données en remplaçant : cores: [3 , 1] par cores: [3 , 2] (ce qui change le ratio simulation vs analyse). Est-ce que cela confirme vos hypothèses ?

   Attention cela fonctionne parce que le code utilise MPI_THREAD_MULTIPLE.

3) **Parallélisation avec l'utilisation d'un thread C++ (Exercice N°2 TP N°2)**

L'objectif de cet exercice est de commencer à développer une version très simple d'un système In Situ utilisant des threads C++ pour réaliser le calcul le l'analyse histogram_energy.

Vous allez devoir modifier les fichiers histogram_seq_naive_thread.cpp et histogram_worker.cpp situés dans ~/TPs_In_Situ/exaStamp/src/tutorial.

Exercices :

- Modifier/compléter le code source des fichiers .cpp pour que les deux phases de calcul de l'histogram soit fait dans un thread C++. Pour cela, une partie du travail a déjà été réalisé. En effet, vous trouverez dans histogram_worker.h, histogram_worker.cpp et histogram_seq_naive_thread.cpp une partie de la fonctionnalité développée. Histogram_worker contient une classe qui calcul l'histogram (sous traité au thread). Et histogram_seq_naive_thread.cpp contient les fonctionnalités pour lancer un thread, lui demander un travail et en récupérer le résultat.
- Il faudra bien faire attention aux éventuels problèmes liés à l'écriture concurrente dans des variables « shared » de l'algorithme. Vous pouvez consulter la documentation https://en.cppreference.com/w/cpp/thread/thread. L'exemple de std::condition_variable est très intéressant pour vous.
  (cf. https://en.cppreference.com/w/cpp/thread/condition_variable)
- Une fois que votre développement fonctionne, que pouvez-vous dire du comportement in situ de ce nouveau système ? N'hésitez pas à utiliser les traces vite ou le résumé des temps d'exécution des différentes fonctions.
- Vérifier que votre développement fonction bien en version MPI : salloc -p miriel -N 1 -n 2 /bin/env OMP_NUM_THREADS=8 mpirun -n 2 /home/$HOME/local/ExaStamp/x86_64/bin/ustamp tutorial_insitu_histo_seq_naive_thread.msp --profiling-vite trace.vite
  ExaStamp utilisant l'option MPI_THREAD_MULTIPLE par défaut pour cette version MPI d'ExaStamp, votre code doit fonctionner.

3) Parallélisation avec l'utilisation d'un thread C++ (Exercice N°3 TP N°2)

Suite à l'étude effectuée dans le cadre de l'exercice N°2 du TP N°2 concernant l'implantation d'une version utilisant un C++ thread, il est possible de rendre plus générique le système utilisant un thread C++ pour éventuellement permettent l'activation de plusieurs analyses.

Pour cela, le système doit être découpé en trois phases. La première est l'initialisation du thread C++ qui va gérer les analyses. La seconde récupère les données dans une structure indépendante, sélectionne l'analyse à exécuter et bien sûr réveil le thread C++ pour lui faire exécuter l'analyse souhaitée. La dernière phase est la récupération des données pour les enregistrer dans une structure gérée par ExaStamp.

Pour vous faciliter un peu le travail, les « squelettes » des fichiers .cpp pour ces trois phases sont disponibles dans tutorial histogram_seq_thread_init.cpp, histogram_seq_thread_run.cpp et histogram_seq_thread_finish.cpp. Ainsi qu'un jeu de données type tutorial_insitu_histo_seq_3parts.msp dans ~/TPs_In_Situ/exaStamp/data/samples/ qui test simplement la première étape qui consiste faire l'histogram.

Exercice :

Le premier développement à réaliser dans le cadre de ce Mini Projet est donc d'obtenir l'exécution de l'histogramme « energy » dans ce nouveau système avec un « sample » différent pour les itérations paire et impaire. Avec les fichiers « squelettes »fournis et la réalisation du dernier exercice du TP N°2, cela devrait être très simple.

Annexe : quelques petits exemples d'utilisation de threads C++

Utilisation de lock_guard :

```cpp
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex m;  // protects g_i

void safe_increment()
{

  // Lorsqu'un objet lock_guard est créé, il tente de s'approprier le mutex
  // qui lui est donné. Lorsque le contrôle quitte le scope C++ dans lequel
  // l'objet lock_guard a été créé, le lock_guard est détruit et
  // le mutex est libéré.

    const std::lock_guard<std::mutex> lk(m);
    ++g_i;

    std::cout << std::this_thread::get_id() << ": " << g_i << '\n';

    // g_i_mutex is automatically released when lock
    // goes out of scope
}

int main()
{
    std::cout << "main: " << g_i << '\n';

    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();

    std::cout << "main: " << g_i << '\n';
}
```

Utilisation de condition_variable :

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

// La classe condition_variable définie  une primitive de synchronisation qui peut être utilisée pour
// bloquer un ou plusieurs threads en même temps. Et cela jusqu'à ce qu'un autre thread modifie
// une variable partagée (la condition) et notifie la condition_variable.

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;}); // Blocks the current thread until the condition variable is woken up

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one(); // If any threads are waiting on *this, calling notify_one unblocks one of the waiting threads.
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```