Multilevel Monte Carlo

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Multithreading en C++11

- Classe `thread`
- Exemples
- Un mot sur la librarire `chrono`
- Utilisation avancée : mutex, `promise`, `future`...
- Monte Carlo et multithreading

Multilevel Monte Carlo

- Monte Carlo biaisé
- Multilevel Monte Carlo
- Retour sur l'extrapolation de Richarson-Romberg
- Multilevel Richardson-Romberg
- Exemples

Multithreading en C++11

- Classe `thread`
- Exemples
- Un mot sur la librarire `chrono`
- Utilisation avancée : mutex, `promise`, `future`...
- Monte Carlo et multithreading

Multilevel Monte Carlo

- Monte Carlo biaisé
- Multilevel Monte Carlo
- Retour sur l'extrapolation de Richarson-Romberg
- Multilevel Richardson-Romberg
- Exemples

Librairie `thread`, syntaxe

La classe thread représente un seul thread d'exécution ou processus léger. Les threads permettent à plusieurs morceaux de codes de s'exécuter simultanément et de manière asynchrone.

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

Attention, les arguments de la fonction de thread sont copiées par valeur. Si un argument doit être passé par référence à la fonction de thread, il doit être enveloppé avec `std::ref` ou `std::cref`.

- constructeur par défaut qui ne fait rien
- pas de contructeur de copie / d'opérateur d'affectation
- constructeur de déplacement / opérateur de déplacement =

Méthodes de le classe `thread`

- **`static unsigned`** `hardware_concurrency();`
  renvoie le nombre de threads qui peuvent s'executer en parallèle (dépend de la machine)

- `std::thread::id get_id()` **`const`**`;`
  renvoie le numéro d'identifiant du thread (objet de la classe membre `id`)

- **`void`** `join();`
  attend la fin du thread courant (appelant la méthode `join()`)

- **`bool`** `joinable()` **`const`**`;`
  `true` si le thread est lancé et non fini

- **`void`** `detach();`
  permet de détacher un `thread`

- **`void`** `swap( thread& other );`
  échange deux threads

Premier exemple multithreading

```
#include <iostream>
#include <thread>

void f1() {
    for (int i = 0; i < 6; ++i)
        std::cout << "exp " << std::exp(i) << std::endl;
}
void f2(double x) {
    for (int i = 0; i < 6; ++i)
        std::cout << "log " << std::log(x/(i+1)) << std::endl;
}

int main() {
    std::thread first (f1);         // creation d'un thread appelant f1()
    std::thread second (f2, 0.5);  // creation d'un thread appeland f2(0.5)

    std::cout << "f1 et f2 s'executent en parallele..." << std::endl;

    first.join();                  // on attend la fin du thread first
    second.join();                 // on attend la fin du thread second

    std::cout << "l'execution de f1 et de f2 est terminee" << std::endl;

    return 0;
}
```

Résultat de l'exemple

```
f1 et f2 s'executent en parallele...exp
1
exp log 2.71828
exp 7.38906
-0.693147exp
log 20.0855-1.38629

log exp -1.79176
54.5982log
exp -2.07944
log 148.413
-2.30259
log -2.48491
l'execution de f1 et de f2 est terminee
```

L'execution n'est pas séquentielle : les appels des instructions de `f1` sont mélangées avec celles de `f2`.

Un autre résultat (après une autre execution)

```
f1 et f2 s'executent en parallele...exp log
1
-0.693147exp
log 2.71828-1.38629
exp
log 7.38906-1.79176
log
exp -2.07944
log 20.0855
-2.30259
log exp -2.48491
54.5982
exp 148.413
l'execution de f1 et de f2 est terminee
```

L'ordre d'execution des instructions n'est pas prévisible !

Un deuxième exemple

On peut utiliser un thread avec un objet fonctionnel et avec une lambda fonction. Dans l'exemple suivant on lance 24 threads et chaque thread est une lambda fonction qui execute l'affichage de `"thread"` suivi du numéro de thread.

```
using namespace std;
int main()
{
    vector<thread> ths;
    for (int i = 0; i < 24; ++i) {
        ths.push_back(thread([=]() {
            cout << "thread:" << i << endl;
        }));
    }
    for (auto & t : ths) t.join();
    return 0;
}
```

Une autre façon pour appeler `join` sur tous les threads de `ths`.

```
std::for_each(threads.begin(), threads.end(), std::mem_fn(&std::thread::join));
```

Un autre résultat (après une autre execution)

```
thread:thread:10

thread:2
thread:3
thread:5
thread:6
thread:7
thread:9
thread:thread:12
thread:11
thread:15
thread:13
thread:thread:16
thread:21
thread:19
thread:thread:17
23
thread:thread:14
4
thread:20
18
thread:2210
```

Méthodes detach

Que fait ce programme ?

```
void independentThread() {
    std::cout << "Starting concurrent thread.\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Exiting concurrent thread.\n";
}

void threadCaller() {
    std::cout << "Starting thread caller.\n";
    std::thread t(independentThread);
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Exiting thread caller.\n";
}

int main() {
    threadCaller();
}
```

this_thread est un **namespace** qui contient notamment :

- **template**< **class** Rep, **class** Period >
  void sleep_for(**const** std::chrono::duration<Rep, Period>& sleep_duration);
- std::thread::id get_id();

Execution

Tests et modifications du code en live...

Un mot sur la librairie chrono

- Il y a 3 types d'horloge :
  - `std::chrono::system_clock`
    Représente l'horloge système. Celle qui peut être modifiée à tout moment. Cette horloge permet généralement l'affichage de la date sur un système.
  - `std::chrono::steady_clock`
    Horloge monotonique, certainement la plus adaptée pour représenter une période. Elle assure une cadence fiable.
    `std::chrono::high_resolution_clock`
    L'horloge la plus précise disponible sur l'environnement d'exécution.

  Une horologe est interrogée via la méthode `now()` qui renvoie un `time_point`

- `time_point`

```
template< class Clock,
          class Duration = typename Clock::duration >
class time_point;
```

  On peut faire des opérations sur les `time_point`.
  La différence entre 2 `time_point` est une `duration`

- duration
  Classe générique qui permet de manipuler les données de temps dans de nombreuses échelles.
  Pour compter en secondes (non entières), on utilisera la spécialisation `duration<double>`

promise et future

A faire

Une classe `linear_estimator`

```
template <typename T, typename S>
class linear_estimator {
public:
    linear_estimator() { reinit(); }
    linear_estimator(unsigned size)
    : _sum(size), _sum_of_squares(size), _sample_size(size) { reinit(); }
    void reinit() {
        _sum = 0; _sum_of_squares = 0; _sample_size = 0;
        _time_span = std::chrono::duration<double>(0.0);
    }
    double time() const { return _time_span.count(); }
    linear_estimator & operator+=(linear_estimator const & other);
protected:
    T _sum, _sum_of_squares;
    S _sample_size;
    std::chrono::duration<double> _time_span;
};

template <typename T, typename S>
linear_estimator<T, S> &
linear_estimator<T, S>::operator+=(linear_estimator<T, S> const & other) {
    _sum += other._sum;
    _sum_of_squares += other._sum_of_squares;
    _sample_size += other._sample_size;
    _time_span += other._time_span;
    return (*this);
};
```

Une classe monte_carlo

```
template <typename Generator>
class monte_carlo : public linear_estimator<double, double> {
public:
    typedef Generator TGenerator;
    monte_carlo(std::function<double(Generator &)> X) : _random_variable(X) {}

    double operator()(Generator & gen, unsigned M); // definition slide suivant

    double mean() const { return this->_sum / this->_sample_size; }
    double mean_of_squares() const { // ... }
    double var() const { return // variance TCL }
    double var_est() const { return // variance estimateur }
    double st_dev() const { return sqrt(var()); }
    double ic() const { return 1.96*sqrt(var_est()); }
    template <typename G2>
    friend std::ostream & operator<<(std::ostream & o, monte_carlo<G2> const & M
private:
    std::function<double(Generator &)> _random_variable;
};
```

La classe est générique par rapport à un type `Generator` qui représente un type de générateur de nombre aléatoire par exemple `mt19937_64`.
C'est nécessaire à cause du champ `_random_variable`.

Opérateur fonctionnel de monte_carlo

```
template <typename Generator>
double monte_carlo<Generator>::operator()(Generator & gen, unsigned M) {
    auto time_start = std::chrono::high_resolution_clock::now();
    for (unsigned m = 0; m < M; ++m) {
        double x = _random_variable(gen);
        _sum += x;
        _sum_of_squares += x*x;
    }
    _sample_size += M;
    auto time_end = std::chrono::high_resolution_clock::now();
    _time_span = time_end - time_start;
    return mean();
};
```

Il est important de noter que cet opérateur prend 2 arguments : un générateur passer par référence (comme il faut toujours le faire) et le nombre M de tirages.

Exemple d'appel

```
typedef mt19937_64 generator;
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
generator gen(seed);

double x0=100, r=0.06, sigma=0.4, K=80, T=1;
BlackScholes BS(x0, r, sigma);
unsigned n = 1;
double h = T / (double) n;

auto X = make_euler(BS, h, n); // X(gen) renvoie une realisation

auto call = [=](double x) { return x > K ? exp(-r*T)*(x - K) : 0; };

auto mc = monte_carlo([=](gen & g) { return call(X(gen)); });

cout << mc(gen, 1e6) << endl;
```

Une fonction générique `parallelize`

```
template <typename TLinearEstimator, typename ForwIt>
TLinearEstimator parallelize(TLinearEstimator X,
                             ForwIt first, ForwIt last, unsigned M) {
    std::list<TLinearEstimator> Xs;
    std::list<std::thread> threads;
    unsigned nb_threads = std::distance(first, last);
    unsigned q = M / nb_threads;
    unsigned r = M % nb_threads;
    unsigned i = 0;
    --last;
    for (ForwIt it = first; it != last; ++it, ++i) {
        Xs.push_back(X);
        unsigned Mi = q + (i < r ? 1 : 0);
        threads.push_back(std::thread(ref(Xs.back()), ref(*it), Mi));
    }
    X(ref(*last), q + (i < r ? 1 : 0));
    for (auto & th : threads) th.join();
    TLinearEstimator result = X;
    for (auto const & Y : Xs) result += Y;
    return result;
};
```

Que fait cette fonction ?

Exemple d'utilisation

```
typedef mt19937_64 generator;
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
unsigned long const hw_threads = std::thread::hardware_concurrency();
std::vector<generator> gens(hw_threads);

std::seed_seq seq({seed, seed+1, seed+2, seed+3, seed+4});
std::vector<std::uint32_t> seeds(hw_threads);
seq.generate(seeds.begin(), seeds.end());

for (unsigned i = 0; i < hw_threads; ++i)
gens[i] = generator(seeds[i]);

double x0=100, r=0.06, sigma=0.4, K=80, T=1;
BlackScholes BS(x0, r, sigma);
unsigned n = 1;
double h = T / (double) n;
auto X = make_euler(BS, h, n);
auto call = [=](double x) { return x > K ? exp(-r*T)*(x - K) : 0; };
auto mc = monte_carlo([=](gen & g) { return call(X(gen)); });

cout << parallelize(mc, gens.begin(), gens.end(), 1e6) << endl;
```

Changer de fichier...