thread-overhead.cpp

```
#include <chrono>
#include <iostream>
#include <ratio>
#include <thread>

int main() {
   unsigned int counter = 0;
   unsigned int nof_threads = 1<<15;

   auto start = std::chrono::high_resolution_clock::now();
      for (unsigned int i = 0; i < nof_threads; ++i) {
         auto t = std::thread([&]() { ++counter; });
         t.join();
      }
   auto finish = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double, std::micro> elapsed = finish - start;
   std::cout << "avg time per thread creation = " <<
      elapsed.count() / nof_threads << " us" << std::endl;
}
```

```
thales$ g++ -std=c++11 -Ofast -o thread-overhead thread-overhead.cpp
thales$ thread-overhead
avg time per thread creation = 34.4164 us
thales$
```

- Das Erzeugen und Abbauen von Threads ist nicht sehr billig. Auf Thales kostet dies ca. 34 μs pro Thread.
- Es stellt sich daher die Frage, ob dies vermieden werden kann.
- Über *std*::*thread*::*hardware_concurrency*() lässt sich ermitteln, wieviele Threads auf der aktuellen Hardware gleichzeitig operieren können.
- Entsprechend erscheint es naheliegend, entsprechend viele Threads zu Beginn anzulegen und diese dann mit den anfallenden Aufgaben zu beschäftigen.
- Das Master/Worker-Pattern ist hier hilfreich.

Das Master/Worker-Pattern operiert mit mindestens einem Master-Thread, beliebig vielen Worker-Threads und einer Warteschlange für anfallende Aufgaben, auf die alle Parteien konkurrierend zugreifen:

- Die anfallenden Aufgaben werden analog wie beim Fork-And-Join-Pattern in Teilaufgaben zerlegt und durch den Master-Thread an das Ende der Warteschlange angefügt.
- Jeder Worker-Thread holt in einer immerwährenden Schleife immer eine Aufgabe aus der Warteschlange und arbeitet diese ab. Wenn die Warteschlange leer ist, wird darauf gewartet, dass die Warteschlange sich wieder füllt.
- Die Warteschlange muss konkurrierende Zugriffe durch den Master und die Worker zulassen und das Warten auf eine sich wieder füllende Warteschlange zulassen.

Aufgaben können wie bei Threads durch Funktionsobjekte repräsentiert werden:

- Da Funktionsobjekte alle möglichen Typen haben können bzw. bei Lambda-Ausdrücken diese ohnehin anonym sind, lohnt sich die Verwendung der Template-Klasse *std*::*function*, die beliebige Funktionsobjekte mit einer einheitlichen Parameterliste und einem vorgegebenen Return-Typ verwalten kann.
- Beispiel: *std*::*function*<**double**(**int**, **float**)> repräsentiert beliebige Funktionsobjekte, mit zwei Parametern (**int** und **float**) und dem Return-Typ **double**.
- Zu Beginn versuchen wir es ganz einfach ohne Parameter und mit dem Return-Typ **void**: *std*::*function*<**void**()>.
- Beispiel:

```
std::function<void()> task = [=,&result]() {
    result[i] = simpson(f, a, b, n);
};
```

tpool1-simpson.cpp

```
struct ThreadPool {
   public:
      using Job = std::function<void()>;
      // ...
   private:
      unsigned int nof_threads;
      bool finished;
      std::vector<std::thread> threads;
      std::mutex mutex;
      std::condition_variable cv;
      std::list<Job> jobs;
      // ...
};
```

- Ein Thread-Pool verknüpft eine Warteschlange für Aufgaben mit Zugriffsmethoden mit einem Array von Threads für die Worker. Die **bool**-Variable *finished* dient dem Abbau des Thread-Pools.

tpool1-simpson.cpp

```
ThreadPool(unsigned int nof_threads) :
      nof_threads(nof_threads), finished(false),
      threads(nof_threads) {
   for (auto& t: threads) {
      t = std::thread([=]() { process_jobs(); });
   }
}
```

- Die private Methode *process_jobs* repräsentiert den Algorithmus eines Workers. Die Capture [=] umfasst hier **this**, das implizit benötigt wird, um die Methode aufzurufen, da *process_jobs*(); zu **this**−>*process_jobs*(); expandiert wird.

tpool1-simpson.cpp

```
~ThreadPool() {
   {
      std::unique_lock<std::mutex> lock(mutex);
      finished = true;
   }
   cv.notify_all();
   for (auto& t: threads) {
      t.join();
   }
}
```

- Beim Abbau wird *finished* auf **true** gesetzt und danach werden alle Worker benachrichtigt, dass die Arbeit einzustellen ist, sobald alle Aufträge abgearbeitet sind. Mit *join* werden dann alle Threads ordnungsgemäß beendet.
- Der Abbau eines Thread-Pools bietet somit auch immer eine Möglichkeit der Synchronisierung.

tpool1-simpson.cpp

```
void submit(Job job) {
   std::unique_lock<std::mutex> lock(mutex);
   jobs.push_back(std::move(job));
   cv.notify_one();
}
```

- Die Methode *submit* nimmt einen Auftrag für die Worker entgegen und fügt an das Ende der Warteschlange an. Mit *std::move* wird ein unnötiges Klonen des Funktionsobjekts vermieden.
- Wenn Worker auf neue Aufträge warten, wird durch *cv.notify_one*() einer davon aufgeweckt.

Algorithmus des Worker

tpool1-simpson.cpp

```
void process_jobs() {
   for(;;) {
      Job job;
      /* fetch job */
      {
         std::unique_lock<std::mutex> lock(mutex);
         while (jobs.empty() && !finished) {
            cv.wait(lock);
         }
         if (jobs.empty() && finished) break;
         job = std::move(jobs.front());
         jobs.pop_front();
      }
      /* execute job */
      job();
   }
}
```

- Worker arbeiten kontinuierlich Aufträge aus der Warteschlange ab. Wenn die Warteschlange leer ist, wird gewartet. Wenn zudem *finished* gesetzt ist, beendet der Worker seine Tätigkeit.

tpool1-simpson.cpp

```
int main() {
   unsigned int nofintervals = 1<<12;
   unsigned int jobsize = 1<<6;
   unsigned int noftasks = nofintervals / jobsize;
   std::vector<double> results(noftasks);

   {
      ThreadPool tpool(std::thread::hardware_concurrency());

      auto f = [](double x) { return 4 / (1 + x*x); };
      double a = 0; double b = 1;

      // submit simpson for each interval
      for (int i = 0; i < noftasks; ++i) {
         tpool.submit([=,&results]() {
            double ai = a + i * (b - a) / noftasks;
            double bi = a + (i + 1) * (b - a) / noftasks;
            results[i] = simpson(f, ai, bi, jobsize);
         });
      }
   }

   // sum up the results
   double sum = 0;
   for (auto res: results) {
      sum += res;
   }
   std::cout << std::setprecision(14) << sum << std::endl;
   std::cout << std::setprecision(14) << M_PI << std::endl;
}
```

- In dem vorgestellten Beispiel war eine einfache Synchronisierung nur mit dem gesamten Thread-Pool möglich.
- Es gab keine Vorkehrung zur Synchronisierung mit der Fertigstellung eines Auftrags.
- Prinzipiell ist eine Synchronisierung immer möglich mit Hilfe von Bedingungsvariablen.
- Die C++-Standardbibliothek offeriert hier eine fertige Lösung mit *std*::*promise* und *std*::*future*.

- C++ bietet mit *std::promise* und *std::future* eine einfache Kommunikations- und Synchronisierungsmöglichkeit an.
- Mit *std*::*promise*<*T*> *p*; wird ein „Versprechen“ gegeben, irgendwann einmal mit *p.set_value*(*value*) einen Wert zu setzen.
- Mit *std*::*future*<*T*> *f* = *p.get_future*() darf daraus genau einmal(!) ein zugehöriges *std::future*-Objekt abgeleitet werden.
- Typischerweise „gehören“ dann die beiden Objekte unterschiedlichen Threads.
- Mit *f.get*() wartet der Aufrufer darauf, dass der Wert mit *p.set_value*(*value*) gesetzt wird und liefert dann diesen zurück.
- Normale Zuweisungen sind weder für *std::promise*- noch *std::future*-Objekte zulässig. Ein Paar dient dazu, einen Wert genau einmal synchronisert weiterzugeben.

simpson-fp.cpp

```
double mt_simpson(double (*f)(double), double a, double b, int n,
      int nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   int nofintervals = n / nofthreads; int remainder = n % nofthreads;
   int interval = 0;

   std::future<double> results[nofthreads];
   // ... fork ...
   // join threads and sum up their results
   double sum = 0;
   for (int i = 0; i < nofthreads; ++i) {
      sum += results[i].get();
   }
   return sum;
}
```

- Das Array der hier deklarierten *std::future*-Objekte ist zu Beginn noch „leer“, d.h. die Objekte sind noch nicht mit *std::promise*-Objekten verbunden.
- Am Ende werden die Ergebnisse eingesammelt. Die Synchronisierung erfolgt hier implizit bei der *get*-Methode.

simpson-fp.cpp

```
// fork off the individual threads
double x = a;
for (int i = 0; i < nofthreads; ++i) {
   int intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   double xleft = x; x = a + interval * (b - a) / n;
   std::promise<double> promise;
   results[i] = promise.get_future();
   auto t = std::thread([=, promise=std::move(promise)]() mutable {
      promise.set_value_at_thread_exit(simpson(f, xleft, x, intervals));
   });
   t.detach(); // no longer joinable, t object can be destructed
}
```

- In der Schleife werden lokal temporäre *std::promise*-Objekte erzeugt und daraus jeweils das zugehörige *std::future*-Objekt abgeholt.
- Das *std::future*-Objekt wird in *results* abgespeichert, das *std::promise*-Objekt geht an den zu erzeugenden Thread.

simpson-fp.cpp

```
std::promise<double> promise;
results[i] = promise.get_future();
auto t = std::thread([=, promise = std::move(promise)]() mutable {
   promise.set_value_at_thread_exit(simpson(f, xleft, x, intervals));
});
t.detach(); // no longer joinable, t object can be destructed
```

- Hier wird zunächst ein *std::promise*-Objekt lokal erzeugt und von diesem ein *std::future*-Objekt abgeleitet, das in dem *results*-Array abgespeichert wird.
- Dann wird ein Thread gestartet, bei dem als Funktions-Objekt ein Lambda-Ausdruck übergeben wird.
- Der Lambda-Ausdruck übernimmt diverse Parameter implizit durch Zuweisung und im Falle von *promise* durch *std::move*, da eine normale Zuweisung nicht zulässig ist. (Dies geht so erst ab C++14, bei C++11 wird noch *std::bind* benötigt.)
- Ohne **mutable** kann der Lambda-Ausdruck *promise* nicht verändern.

simpson-fp.cpp

```
t.detach(); // no longer joinable, t object can be destructed
```

- Auf einen Thread muss entweder mit *join* irgendwann gewartet werden oder dieser muss explizit mit *detach* „vergessen“ werden. Nach *detach* ist eine Synchronisierung mit *join* nicht mehr möglich. Das ist hier auch nicht notwendig, da die Synchronisierung über die *std::future*-Objekte erfolgt.
- Allerdings sollte dann bei *std*::*promise* die Methode *set_value_at_thread_exit* verwendet werden, damit der vollständige Abbau des Threads abgewartet wird.
- Wenn ein Thread-Objekt dekonstruiert wird, für den weder *join* noch *detach* aufgerufen wurde, dann wird die gesamte Programmausführung gewaltsam mit *std::terminate* beendet.

simpson-async.cpp

```
results[i] = std::async([=]() -> double {
   return simpson(f, xleft, x, intervals);
});
```

- Den Standard-Fall, dass ein Thread genau einen Wert berechnet, der über ein *std::future*-Objekt synchronisiert abgeholt werden kann, wird mit *std::async* vereinfacht.
- *std::async* benötigt ein Funktionsobjekt (hier ein Lambda-Ausdruck), das den gewünschten Wert mit **return** zurückliefert.
- *std::async* erzeugt ein *std::promise*-Objekt, leitet davon das *std::future*-Objekt ab, das zurückgegeben wird, ruft dann in einem neu erzeugten Thread das Funktionsobjekt auf und weist den zurückgegebenen Wert dem *std::promise*-Objekt zu.

Funktionsobjekte lassen sich mit *std::future* und *std::promise* verknüpfen:

- Die C++-Standardbibliothek bietet hierfür *std*::*packaged_task* an, das genauso wie *std*::*function* parametrisiert wird. Beispiel: *std*::*packaged_task*<**double**()>
- *std*::*packaged_task* bietet die Methode *get_future* an, mit der ein entsprechendes *std*::*future*-Objekt geliefert wird.
- *std*::*packaged_task*-Objekte sind wiederum Funktionsobjekte mit Return-Typ **void**. Der Return-Wert des ursprünglichen Funktionsobjekts wird somit implizit dem *std*::*promise*-Objekt zugewiesen.
- Achtung: *std*::*package_task*-Objekte sind nicht kopierbar, nur *std*::*move* wird unterstützt.