Wie viel Speed gewinnt man

In diesem Beispiel untersuchen wir, wie sehr man die Summation von 1 Milliarde Integer durch Multithreading beschleunigen kann. Da man eine solche Datenmenge nur mehr dynamisch erzeugen kann, verwenden wir dafür die C++20-Konstruktion:

```
constexpr int sz = 1'000'000'000;
auto vi = std::make_unique_for_overwrite<int[]>(sz);
```

Dadurch erhält man ein nicht-initialisiertes dynamisches `int` C-Array, das am Scope-Ende durch den Destruktor von `vi` automatisch wieder zurückgegeben wird, sodass kein Speicherleck entsteht. Dieses Array füllen wir mit den Zahlen 0, 1, … und addieren diese. Das korrekte Ergebnis ist `1'000'000'000*(1'000'000'000-1)/2`, was man nur mit einer 64bit-Summation erhalten kann. Wir stoppen mit einer automatischen Stoppuhr `Timer` die Zeit der einzelnen Schritte.
mehr_speed-1.cpp:

```
int main()
{   auto vi = std::make_unique_for_overwrite<int[]>(sz);
    {   Timer t{"Vektor fuellen"};
        for (int i{}; i < sz; ++i)
            vi[i] = i;
    }

    {   Timer t{"Vektor summieren"};
        cout << accumulate(&vi[0], &vi[sz], 0LL) << '\n';
    }
}
```

Um das Programm durch Threads zu beschleunigen, müssen wir zunächst die anfallenden Aufgaben in kleiner Häppchen zerlegen. Dazu schreiben wir 2 Lambdas `fill()`, `add()`, die einen Teilbereich `from`, `to` bearbeiten können.

```
auto fill= [&vi](int from, int to) {
     for (int i{from}; i < to; ++i)   vi[i] = i;
};
```

Das Lambda `add()` realisieren wir zu Demonstrationszwecken im ersten Versuch nicht mit `accumulate()`, sondern als Addition zu einer aufgezeichneten Variable `sum`:

```
 auto add = [&sum, &vi](int from, int to) {
        for (int i{from}; i < to; ++i)  sum += vi[i];
    };
```

Das Summe könnten wir nun singlethreaded in **mehr_speed-2.cpp** so berechnen:

```
    {   Timer t{"Vektor fuellen"};
        fill(0, sz);
    }

    {   Timer t{"Vektor summieren"};
        add(0, sz);
    }
```

Wir teilen das gleich in einzelne Häppchen auf mit
`int threads = 10, szt = sz/threads);` **mehr_speed-3.cpp**

```
    {   Timer t{"Vektor fuellen"};
        for (int i{}; i < threads; ++i)
            fill(i*szt, (i+1)*szt);
    }

    {   Timer t{"Vektor summieren"};
        for (int i{}; i < threads; ++i)
            add(i*szt, (i+1)*szt);
    }
```

In **mehr_speed-4.cpp** werden die Häppchen durch eigene Threads erledigt:

```
    {   Timer t{"Vektor fuellen"};
        vector<jthread> worker;
        for (int i{}; i < threads; ++i)
            worker.emplace_back(fill, i*szt, (i+1)*szt);
    }

    {   Timer t{"Vektor summieren"};
        vector<jthread> worker;
        for (int i{}; i < threads; ++i)
            worker.emplace_back(add, i*szt, (i+1)*szt);
    }
```

Die Füllzeit wurde dadurch den Faktor 3 beschleunigt, die Additionszeit **verlängerte sich um den Faktor 8! Und obendrein war das Ergebnis auch noch falsch!** Als Grund dafür kam nur ein Data-Race in Frage, den man durch `atomic<long long> sum{}` verhindern muss. Locking würde noch deutlich mehr Zeit verlieren!

mehr_speed-5.cpp: Die Addition **verlängerte sich dadurch um einen weiteren Faktor 4**, immerhin war das Ergebnis nun richtig!

Aber was ist der Grund für den extremen Zeitverlust?

Der Grund liegt in der modernen Computerarchitektur begründet, die die langen Zugriffszeiten auf das Memory durch extrem schnelle Zwischenspeicher (= Caches) ausgleicht. Daten gelangen vom Memory über den/die Caches in die CPU und werden in den Caches noch einige Zeit weitergespeichert, um weitere Zugriffe darauf zu beschleunigen. Die CPU arbeitet sozusagen nur mit Datenkopien in den schnellen Caches.

Was bei einem Thread super ist, wird bei vielen Threads zum Flaschenhals, weil jeder Thread diese Kopien in seinem eigenen Cache ablegt und die Speicherzelle somit Kopien in verschiedenen Zwischenspeichern hat. Damit alle Threads denselben Wert sehen, muss die CPU diese Kopien aufwendig synchronisieren (Cache-Kohärenz-Protokoll), was extrem viel Zeit kostet: Verändert ein Thread diese Variable, wird sie in den anderen Threads als ungültig markiert und die übrigen Threads müssen sie langsam aus dem Speicher holen.

Die einfache Lösung ist daher, **nicht immer** in die gemeinsame Summenvariable zu summieren, sondern den eigenen Bereich über eine eigene Summenvariable zu erledigen und erst am Schluss diese zur Gesamtsumme zu addieren. Dasselbe erreicht man auch, wenn man im Lambda `add()` wieder den `accumulate`-Algorithmus einsetzt.
mehr_speed-6.cpp:

```
auto add = [&sum, &vi](int from, int to) {
     sum += accumulate(&vi[from], &vi[to], 0LL);
};
```

Dadurch beschleunigt sich die Addition um den Faktor 110 und man ist ca. 3 Mal schneller als am Anfang. Übrigens muss man nicht auf dieselbe Variable zugreifen, um diesen negativen Effekt zu verspüren. Die Caches sind in sogenannten Cache-Lines von 32 Byte organisiert und es reicht aus, wenn ein Thread in dieselbe Cache Line schreibt, um diese bei allen übrigen Threads rauszuwerfen!

`std::async` statt `std::thread/std::jthread`

C++ hat schon länger eine Alternative zur Erzeugung von Threads: Die Funktion `std::async()` (definiert in `future`) kann ebenfalls Threads erzeugen, die im Gegensatz zu `std::thread/jthread` einen Rückgabewert haben können und bei nicht aufgefangenen Exceptions das Programm am Leben lassen.

`async()` erwartet als ersten Parameter die Launch-Policy, die man sinnvollerweise auf `std::launch::async` setzen sollte, damit eine parallele Ausführung stattfindet. Die weiteren Argumente sind dieselben wie bei Threads.

Der Rückgabewert von `async()` ist vom Typ `std::future<T>`, wenn `T` der Rückgabetyp des Threads ist. In unserem Fall gibt `add()` nichts zurück, wir können das aber leicht zu `long long` ändern, indem wir die Teilsumme zurückgeben.

```
auto add = [&sum, &vi](int from, int to) {
     return accumulate(&vi[from], &vi[to], 0LL);
};
```

Anstatt eines `vector<jthread> worker;` nehmen wir einen
`vector<future<long long>> teilsummen;`

Die einzelnen Threads speichern ihre Teilergebnisse in der Zukunft (daher der Name!) in ihrer `future<long long>` ab. Der Haupt-Thread muss diese Ergebnisse dort abholen und zur Summenvariablen addieren. Da dies nur mehr eine Single-Thread Angelegenheit ist, braucht man hierzu auch keine `atomic`-Variable.

Das Ergebnis eines Threads bekommt man über die `get()`- Methode der `future`. Diese blockiert auch so lange, bis der Thread dieses Resultat auch zurückgegeben hat.

Achtung: Beim Aufruf der `get()`−Methode wird auch eine etwaige Exception, die der Thread nicht aufgefangen hat, ausgeliefert. Vorsichtige Leute packen daher diesen Methodenaufruf in einen try-catch-Block.
mehr_speed-7.cpp:

```
{   Timer t{"Vektor summieren"};
    vector<future<long long>> teilsummen;
    for (int i{}; i < threads; ++i)
        summen.push_back(async(launch::async, add,
            i*szt, (i+1)*szt));

        for (auto& i: teilsummen)
            sum += i.get();           // hier ist kein try-catch nötig
 }
```

Diese Programmvariante war die insgesamt schnellste.