Verteilte Systeme

...für C++ Programmierer

Threads: Performance & Speicher

by

Dr. Günter Kolousek

Threadpools

- Parallelisierung oftmals zahlreicher, kleiner Aufgaben (engl. tasks)
- jeder Task in eigenem Thread $\rightarrow$ teuer!
  - auch Starten von Threads kostet!
    - $\rightarrow$ Wiederverwendung von Threads
  - Context switch kostet!
    - $\rightarrow$ je Kern ein Thread…
- Threadpools
  - Angaben
    - Anzahl der Threads, die bei Beginn erzeugt werden
    - maximale und minimale # an Thread
    - Größe der Queue
  - Abarbeitung von Aufgaben bis Threadpool beendet
  - z.B. bei Webservern oder Rechenaufgaben

Lastverteilung

- OS kann Threads zwischen Kernen verschieben
  - kostet Zeit
  - auch wg. Caches nicht gut (muss nachgeladen werden)
    - kleinste Verwaltungseinheit ist Cache-Zeile
    - Caches müssen aktualisiert werden!
- Mindestens so viele Threads wie Kerne
  - mehr Threads $\rightarrow$ Überbelegung (engl. oversubscription)
- viele blockierende Threads $\rightarrow$ mehr Threads
  - ansonsten Unterbelegung (engl. undersubscription)
- Annahme: 4 Threads unterschiedlicher Prioritäten
  - 4 Kerne: alles ok
  - 1 Kern:

Lastverteilung

- OS kann Threads zwischen Kernen verschieben
  - kostet Zeit
  - auch wg. Caches nicht gut (muss nachgeladen werden)
    - kleinste Verwaltungseinheit ist Cache-Zeile
    - Caches müssen aktualisiert werden!
- Mindestens so viele Threads wie Kerne
  - mehr Threads $\rightarrow$ Überbelegung (engl. oversubscription)
- viele blockierende Threads $\rightarrow$ mehr Threads
  - ansonsten Unterbelegung (engl. undersubscription)
- Annahme: 4 Threads unterschiedlicher Prioritäten
  - 4 Kerne: alles ok
  - 1 Kern: in Abhängigkeit des Scheduler $\rightarrow$ niedrig prioritisierte verhungern (engl. starvation)

Optimale # an parallelen Threads?

```cpp
#include <iostream>  // hwthreads.cpp
#include <thread>
using namespace std;
int main() {
    int hw_threads{thread::hardware_concurrency()};
    if (hw_threads) {
        cout << hw_threads << endl;
    } else {
        cout << "no info available" << endl; }}
```

```
8
```

... Intel i7-8550U: 4 Kerne mit je 2 Threads

single-threaded vs. multi-threaded

- Wann ist es sinnvoll zu parallelisieren?
- Wann nicht?

Addieren – single-threaded

```cpp
#include <iostream> // sum_single.cpp
#include <chrono>
#include <algorithm>
#include <random>
#include <vector>
using namespace std;
using ull = unsigned long long;
constexpr ull size{100000000};

int main() {
    mt19937 engine;  // always the same
    uniform_int_distribution<> dis(1,10);

    vector<int> values;  values.reserve(size);
    for (ull i{0}; i < size ; ++i)
        values.push_back(dis(engine));
```

Addieren – single-threaded – 2

```
    ull acc{0};

    auto start = chrono::system_clock::now();
    acc = accumulate(begin(values),end(values),0);
    chrono::duration<double> dur =
        chrono::system_clock::now() - start;

    cout << "Time: " << dur.count() << "s" << endl;
    cout << "Result: " << acc << endl;
}
```

```
Time: 1.1698s
Result: 549996948
```

Addieren – multi-threaded

```cpp
#include <iostream> // sum_multiple.cpp
#include <chrono>
#include <random>
#include <vector>
#include <mutex>
#include <thread>
using namespace std;
using ull = unsigned long long;
constexpr ull size{100000000};
constexpr ull b1{25000000};
constexpr ull b2{50000000};
constexpr ull b3{75000000};
constexpr ull b4{100000000};

mutex mtx;
```

Addieren – multi-threaded – 2

```
void sum(ull& acc, const vector<int>& values,
         ull beg, ull end) {
    for (auto it = beg; it < end; ++it) {
        lock_guard<std::mutex> lg(mtx);
        acc += values[it]; }
}
int main() {
    mt19937 engine;
    uniform_int_distribution<> dis(1,10);

    vector<int> values; values.reserve(size);
    for (long long i{0}; i < size; ++i)
      values.push_back(dis(engine));

    ull acc{0};
```

Addieren – multi-threaded – 3

```
    auto start = chrono::system_clock::now();

    thread t1(sum, ref(acc), ref(values), 0, b1);
    thread t2(sum, ref(acc), ref(values), b1, b2);
    thread t3(sum, ref(acc), ref(values), b2, b3);
    thread t4(sum, ref(acc), ref(values), b3, b4);
    t1.join();  t2.join();  t3.join();  t4.join();

    chrono::duration<double> dur =
      chrono::system_clock::now() - start;

    cout << "Time: " << dur.count() << "s" << endl;
    cout << "Result: " << acc << endl;
}
```

single- vs. multi-thread. – 2

- Linux, 2 Kerne je 2 Threads, gcc 5.3.0

	single-threaded	multi-threaded
ohne Opt.	1.1698s	17.5945s
max. Opt.	0.0490417s	11.0846s

single- vs. multi-thread. – 2

- Linux, 2 Kerne je 2 Threads, gcc 5.3.0

	single-threaded	multi-threaded
ohne Opt.	1.1698s	17.5945s
max. Opt.	0.0490417s	11.0846s

- Verbesserungspotenzial
  - atomare Variable (anstatt `lock_guard`, siehe später!)
    - `atomic<ull>& acc; acc += values[it];`

    Verbesserung gegenüber `lock_guard`...

single- vs. multi-thread. – 2

- Linux, 2 Kerne je 2 Threads, gcc 5.3.0

	single-threaded	multi-threaded
ohne Opt.	1.1698s	17.5945s
max. Opt.	0.0490417s	11.0846s

- Verbesserungspotenzial
  - atomare Variable (anstatt `lock_guard`, siehe später!)
    - `atomic<ull>& acc; acc += values[it];`

    Verbesserung gegenüber `lock_guard`...Faktor 3-4!

single- vs. multi-thread. – 2

- Linux, 2 Kerne je 2 Threads, gcc 5.3.0

	single-threaded	multi-threaded
ohne Opt.	1.1698s	17.5945s
max. Opt.	0.0490417s	11.0846s

- Verbesserungspotenzial
  - atomare Variable (anstatt `lock_guard`, siehe später!)
    - ```
      atomic<ull>& acc; acc += values[it];
      ```

    Verbesserung gegenüber `lock_guard`...Faktor 3-4!
  - `atomic` mit `fetch_add` & `memory_order_relaxed`
    - ```
      atomic<ull>& acc;
      acc.fetch_add(values[it],
                    memory_order_relaxed)
      ```

    Verbesserung gegenüber `lock_guard`...

single- vs. multi-thread. – 2

- Linux, 2 Kerne je 2 Threads, gcc 5.3.0

	single-threaded	multi-threaded
ohne Opt.	1.1698s	17.5945s
max. Opt.	0.0490417s	11.0846s

- Verbesserungspotenzial
  - atomare Variable (anstatt `lock_guard`, siehe später!)
    - ```
      atomic<ull>& acc; acc += values[it];
      ```

    Verbesserung gegenüber `lock_guard`...Faktor 3-4!
  - `atomic` mit `fetch_add` & `memory_order_relaxed`
    - ```
      atomic<ull>& acc;
      acc.fetch_add(values[it],
                    memory_order_relaxed)
      ```

    Verbesserung gegenüber `lock_guard`...Faktor 7-8!

  $\rightarrow$ single-threaded ca. 30 Mal schneller als schnellste multi-threaded Variante!!

Verbesserung – nicht so naiv!

besser so wenig wie möglich synchronisieren!

```
void sum(ull& acc, const vector<int>& values,
         ull beg, ull end) {
    ull acc_tmp{0};
    for (auto it = beg; it < end; ++it)
        acc_tmp += values[it];
    lock_guard<std::mutex> lg(mtx);
    acc += acc_tmp;
}
```

Verbesserung – nicht so naiv!

besser so wenig wie möglich synchronisieren!

```
void sum(ull& acc, const vector<int>& values,
         ull beg, ull end) {
    ull acc_tmp{0};
    for (auto it = beg; it < end; ++it)
        acc_tmp += values[it];
    lock_guard<std::mutex> lg(mtx);
    acc += acc_tmp;
}
```

```
Time: 0.0420116s
Result: 549996948
```

$\rightarrow$ wie single-threaded!!

Anstatt lokaler Variable: thread-lokale Variable oder Promise (beide siehe später) (aber auch nicht schneller)

Datenaustausch

- prinzipieller Austausch von Daten über Objekte und Variable (engl. shared memory programming)
- Thread muss nicht immer aktuelle Daten ”sehen”
  - da Speicherinhalt in Register (und noch nicht zurückgeschrieben)
  - Problem auch auf Single-Core-System
- Konflikte können auftreten!

Speichermodell (Memory Model)

- formale Spezifikation der Lese- und Schreiboperationen
- notwendig, für die Semantik von multi-threaded Programmen
- behandelt
  - Reihenfolge
  - Atomarität
- hat Auswirkungen auf
  - Programmierung
  - Performance
  - Portabilität

Speichermodell – 2

```
a == 0 && b == 0

Thread 1 | Thread 2
  x = a; |  y = b;
  b = 2; |  a = 1;
```

Speichermodell – 3

```
a == 0 && b == 0

Thread 1 | Thread 2
  x = a  |
  b = 2  |
         |   y = b
         |   a = 1

x == 0 && y == 2
```

Speichermodell – 4

```
a == 0 && b == 0

Thread 1 | Thread 2
  x = a  |
         |   y = b
         |   a = 1
  b = 2  |

x == 0 && y == 0
```

Speichermodell – 5

```
a == 0 && b == 0

Thread 1 | Thread 2
  b = 2; |
         |  y = b;
         |  a = 1;
  x = a; |

x == 1 && y == 2
```

Speichermodell – 5

```
a == 0 && b == 0

Thread 1 | Thread 2
  b = 2; |
         |   y = b;
         |   a = 1;
  x = a; |

x == 1 && y == 2
```

- Optimierungen der Pipeline durch Compiler & Prozessor!
- **Veränderung auch über Caches möglich** (wenn Daten vor Verwendung geladen)
- **Auch in Java und C# möglich!**

Speichermodell – 6

- Ohne Unterstützung daher kein Austausch von Daten zwischen Threads möglich!
- *Speichermodell* legt fest wann Werte von anderen Threads gesehen werden
- Java, C# und C++ haben jetzt ein definiertes Speichermodell
- *Sequenzielle Konsistenz* als Speichermodell
  - kein Umsortierungen erlaubt
  - Schreibeoperationen atomar und sofort für alle Threads sichtbar
  - aber: Performance…
  - deshalb: Speichermodell basierend auf Speicherbarrieren

Speicherbarrieren

(engl. memory barrier, memory fence)

- Full Fence: kein Verschieben über Barriere
- Store Fence: kein Verschieben von Schreiboperationen
- Load Fence: detto für Leseoperation
- Acquire Fence: keine Operationen dürfen nach vorne verschoben werden
- Release Fence: keine Operationen dürfen nach hinten verschoben werden

Speicherbarrieren – 2

```
          Thread 1 | Thread 2
             x = a; |
             b = 2; | d = 4;
lock.release();     | ...
             ...    | lock.acquire();
             c = 3; |  y = b;
                    |  a = 1;
```

Speicherbarrieren – 2

```
                Thread 1 | Thread 2
                  x = a; |
                  b = 2; | d = 4;
lock.release();          | ...
                  ...    | lock.acquire();
                  c = 3; |  y = b;
                         |  a = 1;
```

- c könnte nach vorne geschoben werden
- d könnte nach hinten geschben werden
- wird in der Anwendungsprogrammierung nicht verwendet, aber…

Speicherbarrieren – 2

```
                Thread 1 | Thread 2
                  x = a; |
                  b = 2; | d = 4;
      lock.release(); | ...
                  ...    | lock.acquire();
                  c = 3; |  y = b;
                         |  a = 1;
```

- c könnte nach vorne geschoben werden
- d könnte nach hinten geschben werden
- wird in der Anwendungsprogrammierung nicht verwendet, aber…
  - Verständnis!
  - Verwendung in Implementierung von Synchronisationsmechanismen!

Schlüsselwort `volatile`

- Bedeutung in C und C++
  - nur für Zugriff auf HW gedacht (engl. memory mapped I/O)
    - Wert wird direkt in den Speicher geschrieben
    - Kein Wegoptimieren oder Umordnen erlaubt
    - keine atomare Aktion (schreiben bzw. lesen)
  - nicht für Kommunikation zwischen Threads verwenden!
    - $\rightarrow$ Performance wird sinken und keine Sicherheit
- Bedeutung in C# und Java
  - Zugriff auf eine Variable nach der Acquire/Release Semantik!

Schlüsselwort thread_local

```cpp
#include <iostream>  // threadlocal.cpp
#include <string>
#include <thread>
#include <mutex>
using namespace std;

// not synchronized in each thread -> performance!
thread_local unsigned int cnt{10};
mutex cout_mtx;

void cnt_chars(const string& str, const int id) {
    // no race cond, cnt belongs to this thread
    cnt += str.size();
    lock_guard<std::mutex> lock(cout_mtx);
    cout << "t" << id << ": " << cnt << endl;
}
```

Schlüsselwort thread_local - 2

```
int main() {
    string str{"abcdefghi"};
    thread t1{cnt_chars,
              str.substr(0, str.size() / 2), 1};
    thread t2{cnt_chars,
        str.substr(str.size() / 2, str.size()), 2};
    {
        std::lock_guard<std::mutex> lock(cout_mtx);
        std::cout << "main: " << cnt << endl;
    }
    t1.join();  t2.join();
}
```

```
main: 10
t2: 15
t1: 14
```

False Sharing

- Problem beim Zugriff auf Caches (Teil der Speicherhierarchie)
  - bedeutet: ”scheinbar gemeinsame Nutzung”
- kleinste Einheit ist Cache-Zeile (ca. 32 bis 256 Bytes)
- mehrere Kerne können gleichzeitig Kopien derselben Zeile im lokalen Cache
- Änderung einer Zeile $\rightarrow$ Invalidierung und Aktualisierung der anderen Caches!
- Modifikation verschiedener Daten der in gleicher Zeile…
  - Änderung und daher Invalidierung und Aktualisierung
  - obwohl nicht notwendig
  - $\rightarrow$ Ping Pong!

False Sharing – 2