# Intel Threading Building Blocks

Seminar Effiziente Programmierung

Robin Mirow

22.11.2018

# Gliederung

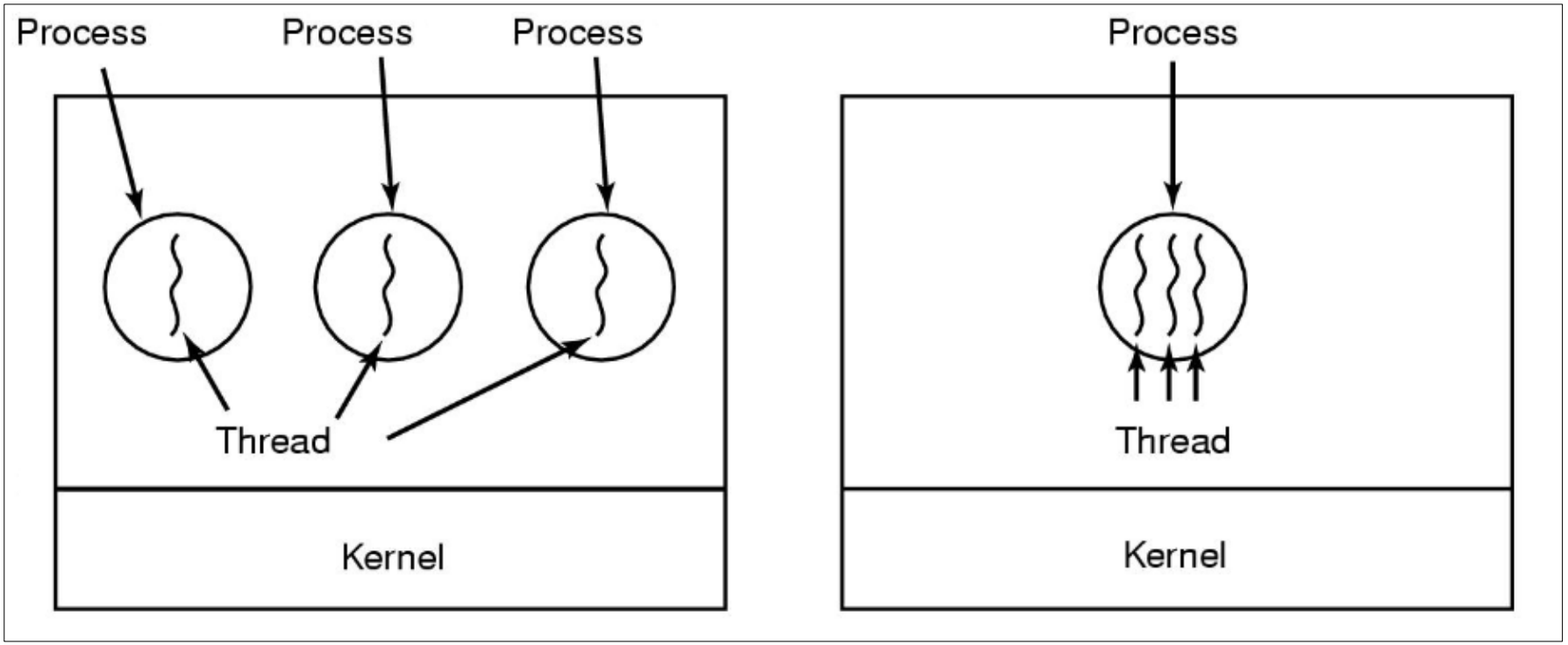
- Was ist überhaupt Multithreading?
- Gefahren von Multithreading
- Intel Threading Building Blocks
- Datenparallelismus
- Task-Parallelismus
- Vergleich mit OpenMP
- Nebenläufige Datenstrukturen
- Zusammenfassung
- Quellen

# Was ist überhaupt Multithreading?

- Mehrere Ausführungsstränge in einem Prozess
  - Eigene Stacks
  - Eigene Programmzähler
- Gemeinsamer Adressraum
- Gemeinsame Ressourcen (File Handles, Sockets, etc.)

Tanenbaum, Bos (2014), S. 102-106.

# Was ist überhaupt Multithreading?




Andrew S. Tanenbaum, Herbert Bos (2014). „Modern Operating Systems, Fourth Edition“, S. 104.

Mehrere voneinander isolierte Prozesse mit jeweils einem Thread

Ein Prozess mit mehreren Threads

# Was ist überhaupt Multithreading?

## Wann ist Multithreading sinnvoll?

- Kann Programme deutlich beschleunigen
- Multithreading bedeutet immer Overhead durch Synchronisation
- Erhöht die Komplexität des Programms
- Eine Algorithmen sind inhärent sequentiell

**Wenn immer möglich, sollte das Programm anderweitig optimiert werden**

# Gefahren von Multithreading

## Race conditions

Ein Ergebnis, dass vom zeitlichen Verhalten von mindestens zwei Operationen abhängig ist.

Tanenbaum, Bos (2014), Seite 119-121.

# Gefahren von Multithreading

## Race conditions

```
1  uint32_t num_jobs_remaining(void) {
2    // ...
3  }
4
5  void* get_job(void) {
6    // ...
7  }
8
9  void run_job(void* job) {
10   // ...
11 }
```

# Gefahren von Multithreading

## Race conditions

```
1  std::thread threads[4];
2
3  for (size_t i = 0; i < 4; i++) {
4      threads[i] = std::thread([]() {
5          while (num_jobs_remaining() > 0) {
6              run_job(get_job());
7          }
8      });
9  }
10
11 for (size_t i = 0; i < 4; i++) {
12     threads[i].join();
13 }
```

# Gefahren von Multithreading

## Data races

- Mindestens zwei Threads greifen auf die gleiche Stelle im Speicher zu
- Mindestens ein Thread hat schreibenden Zugriff

Eine race condition ist ein logische Fehler, data races sind **undefined behaviour**

# Gefahren von Multithreading

## Data races

```
 1  int32_t data_race(void) {
 2      int32_t count = 0;
 3
 4      std::thread incr([&]() {
 5          count++;
 6      });
 7
 8      count++;
 9      incr.join();
10
11      if (count == 2) {
12          return 42;
13      } else {
14          return -1;
15      }
16  }
```

# Gefahren von Multithreading

## Deadlocks

- Zwei gemeinsame Ressourcen R1 und R2, die durch Locks geschützt sind
- Thread A hält ein Lock auf R1 und möchte R2 benutzen
- Thread B hält ein Lock auf R2 und möchte R1 benutzen

Beide Threads warten aufeinander

Tanenbaum, Bos (2014), S. 435.

# Gefahren von Multithreading

## Deadlocks

```
1  class Hat {
2    public:
3      uint64_t id;
4  };
5
6  class User {
7    public:
8      std::vector<Hat> hats;
9      std::mutex mutex;
10 };
```

# Gefahren von Multithreading

## Deadlocks

```
1  void donate_hat(User& from, User& to, size_t hat_idx) {
2      std::scoped_lock<std::mutex> from_guard(from.mutex);
3
4      if (hat_idx < from.hats.size()) {
5          std::scoped_lock<std::mutex> to_guard(to.mutex);
6          to.hats.push_back(from.hats[hat_idx]);
7          from.hats.erase(from.hats.begin() + hat_idx);
8      }
9  }
```

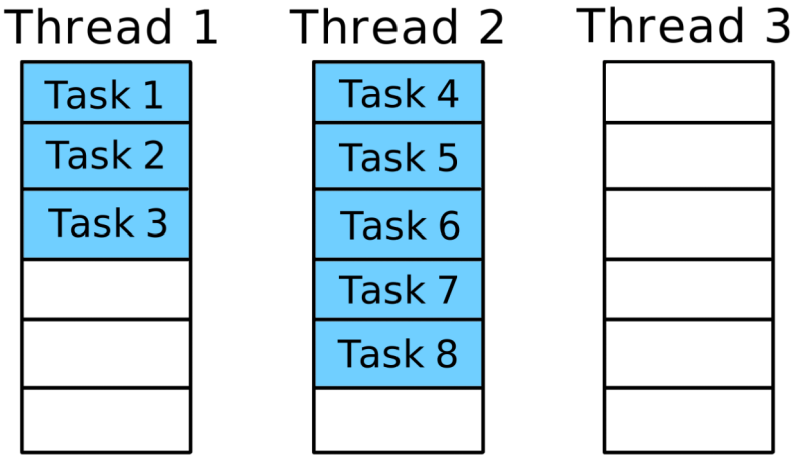
# Gefahren von Multithreading

## Deadlocks

```
1  // Thread 1
2  donate_hat(bob, alice, 0);
3
4  // Thread 2
5  donate_hat(alice, bob, 0);
```

# Intel Threading Building Blocks (TBB)

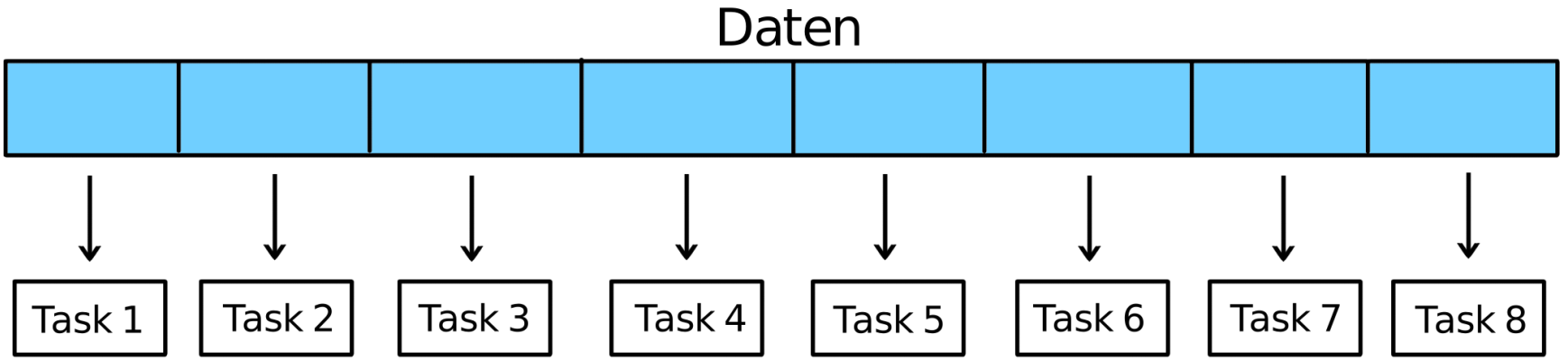
- C++ Bibliothek für Multithreading-Programmierung
- Unterstützt Windows/Linux/Android/MacOS auf x86/AMD64/ARM/PowerPC
- Lizensiert unter Apache 2.0
- Basiert auf einem Threadpool mit Workstealing




Jeder Thread besitzt eine eigene Task-Queue

# Datenparallelismus

- Verteilung von Arbeit durch Teilung der Daten
- Auf allen Teildaten wird die gleiche Rechnung ausgeführt
- Setzt voraus, dass das Problem in größtenteils unabhängige Teilprobleme zerlegbar ist




en.wikipedia.org/wiki/Data_parallelism.

# Datenparallelismus

## parallel_sort

```
1  std::vector<int32_t> random_vec = { /* ... */ };
2  tbb::parallel_sort(random_vec.begin(), random_vec.end());
```

# Datenparallelismus

## parallel_for

```
1  tbb::parallel_for(
2    tbb::blocked_range<size_t>(0, nums.size()),
3    [&](const tbb::blocked_range<size_t>& range) {
4        for (size_t i = range.begin(); i < range.end(); ++i) {
5              nums[i] = sqrt(nums[i]);
6        }
7    }
8  );
```

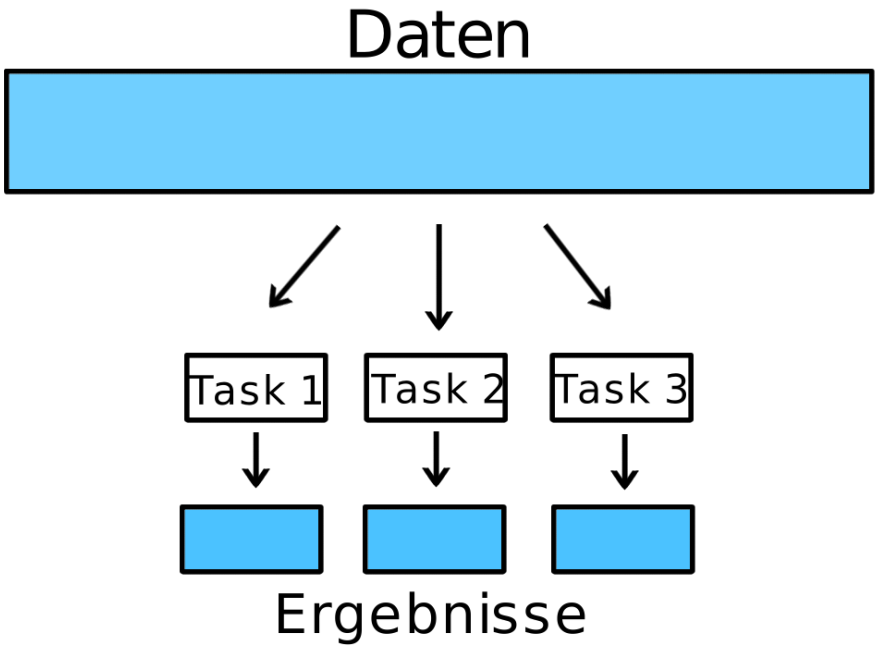
# Datenparallelismus

## parallel_reduce

```
1  int64_t sum = tbb::parallel_reduce(
2    tbb::blocked_range<size_t>(0, nums.size()),
3    int64_t(0),
4    [&](const tbb::blocked_range<size_t>& r, const int64_t& s) {
5      int64_t sum = s;
6      for (size_t i = r.begin(); i < r.end(); ++i) {
7        sum += nums[i];
8      }
9      return sum;
10   },
11   [](const int64_t& left, const int64_t& right) {
12     return left + right;
13   }
14 );
```

# Task-Parallelismus

- `task_group` ermöglicht parallele Ausführung von unterschiedlichen Aufgaben
- Tasks werden auf Threads verteilt




en.wikipedia.org/wiki/Task_parallelism.

# Task-Parallelismus

## task_group

```
1  int64_t compute_expensive_op(const std::vector<int64_t>&);
2
3  int64_t compute_other_op(const std::vector<int64_t>&);
4
5  int64_t median_of_ops(const std::vector<int64_t>& nums) {
6      auto first_value = compute_expensive_op(nums);
7      auto second_value = compute_other_op(nums);
8
9      return (first_value + second_value) / 2;
10 }
```

# Task-Parallelismus

## task_group

```
1  int64_t first_value;
2  int64_t second_value;
3
4  tbb::task_group group;
5  group.run([&]() {
6      first_value = compute_expensive_op(nums);
7  });
8  group.run([&]() {
9      second_value = compute_more_expensive_op(nums);
10 });
11 group.wait();
```

# Task-Parallelismus

## Einschränkungen von `task_group`

- Kein Ersatz für Threads
- Niemals blockende Operationen durchführen
  - Blockende IO-Operationen
  - Locks mit hohem Wettbewerb

# Vergleich mit OpenMP

- Compiler-Erweiterung
- Unterstützt von GCC, Clang, ICC
- Nur OpenMP 2.0 wird von MSVC unterstüzt
- Annotationen für Compiler geben Hinweise zur Parallelisierung

# Vergleich mit OpenMP

## Datenparallelismus

Einfacher for-Loop:

```
1  #pragma omp parallel for
2  for (size_t i = 0; i < nums.size(); i++) {
3      nums[i] = sqrt(nums[i]);
4  }
```

Mit Reduktion:

```
1  int64_t sum = 0;
2
3  #pragma omp parallel for reduction(+: sum)
4  for (size_t i = 0; i < nums.size(); i++) {
5      sum += nums[i];
6  }
```

# Vergleich mit OpenMP

## Task-Parallelismus

```
1  int64_t first_value;
2  int64_t second_value;
3
4  #pragma omp parallel
5  {
6      #pragma omp single
7      {
8          #pragma omp task
9          first_value = compute_expensive_op(nums);
10         #pragma omp task
11         second_value = compute_more_expensive_op(nums);
12     }
13 }
```

# Vergleich mit OpenMP

## Task-Parallelismus

- Tasks können rekursiv gestartet werden
- Jeder `task` kann parallel ausgeführt werden
- Tasks werden auf Threads verteilt
- Ausführungsreihenfolge ist undefiniert
- Explizite Wartepunkte mit `taskwait`

# Nebenläufige Datenstrukturen

- Datenstrukturen wie `std::queue` oder `std::unordered_map` erlauben keinen unsynchronisierten Zugriff
- Externe Synchronisation macht den Gewinn durch Parallelisierung zunichte

Nebenläufige Datenstrukturen machen Zugriff möglich

software.intel.com/en-us/intel-tbb,
en.cppreference.com/w/cpp/container/queue,
en.cppreference.com/w/cpp/container/unordered_map.

# Nebenläufige Datenstrukturen

## concurrent_unordered_map

```
1  std::thread threads[4];
2
3  for (size_t i = 0; i < 4; i++) {
4      threads[i] = std::thread([]() {
5          while (true) {
6              auto user_id = wait_for_status_request();
7              auto status = fetch_status(user_id);
8              send_status(user_id, status);
9          }
10     });
11 }
```

# Nebenläufige Datenstrukturen

## concurrent_unordered_map

```
1  tbb::concurrent_unordered_map<uint64_t, std::string> CACHE;
2
3  std::string fetch_status(uint64_t user_id) {
4      auto status = CACHE.find(user_id);
5
6      if (status != CACHE.end()) {
7          return status->second;
8      } else {
9          auto loaded_status = load_status_from_db(user_id);
10         CACHE.insert({user_id, loaded_status});
11         return loaded_status;
12     }
13 }
```

# Nebenläufige Datenstrukturen

## concurrent_queue

```
1  uint64_t hash(const std::vector<uint8_t>& buf) {
2      uint64_t hash = 0;
3
4      for (auto byte : buf) {
5          hash += byte + 1 % 42;
6      }
7
8      return hash << 2;
9  }
```

# Nebenläufige Datenstrukturen

## concurrent_queue

```
1  tbb::concurrent_queue<uint64_t> PENDING_HASHES;
2
3  void calc_hashes(std::vector<uint8_t>* bufs, size_t num_bufs) {
4    tbb::parallel_for(
5      tbb::blocked_range<size_t>(0, num_bufs),
6      [&](const tbb::blocked_range<size_t>& range) {
7        for (size_t i = range.begin(); i < range.end(); ++i) {
8          PENDING_HASHES.push(hash(bufs[i]));
9        }
10     }
11   );
12 }
```

# Nebenläufige Datenstrukturen

## concurrent_queue

```
1  void write_pending_hashes(void) {
2      std::ofstream file;
3      file.open("hashes");
4      uint64_t current_hash;
5      while (true) {
6          if (PENDING_HASHES.try_pop(current_hash)) {
7              file << current_hash << std::endl;
8          } else {
9              std::this_thread::sleep_for(
10                 std::chrono::milliseconds(20)
11             );
12         }
13     }
14 }
```

# Zusammenfassung

- Threading Building Blocks bietet wichtige Bausteine
- Noch viel mehr als besprochen: Mutexe, RwLocks, Atomics, Task-Graphen, ...
- Benötigt keine Compilererweiterungen

- Für einfachen Datenparallelismus ist OpenMP völlig ausreichend
- Standardbibliothek ab C++11 bietet auch viele Primitive

# Quellen (1)


- https://en.cppreference.com/w/cpp/container/queue [15.11.2018]
- https://en.cppreference.com/w/cpp/container/unordered_map [15.11.2018]
- https://en.cppreference.com/w/cpp/language/memory_model [11.11.2018]
- https://en.wikipedia.org/wiki/Data_parallelism [11.11.2018]
- https://en.wikipedia.org/wiki/Task_parallelism [11.11.2018]
- https://software.intel.com/en-us/node/506130 [11.11.2018]
- https://github.com/01org/tbb [11.11.2018]
- https://software.intel.com/en-us/intel-tbb [11.11.2018]
- https://msdn.microsoft.com/de-de/library/fw509c3b.aspx [14.11.2018]
- https://www.openmp.org/resources/openmp-compilers-tools/ [14.11.2018]

# Quellen (2)


- https://www.threadingbuildingblocks.org/faq/11 [17.11.2018]
- http://www.nersc.gov/users/software/programming-models/openmp/openmp-tasking/openmp-task-intro/ [17.11.2018]
- https://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods [14.11.2018]
- Andrew S. Tanenbaum, Herbert Bos (2014). „Modern Operating Systems, Fourth Edition“, S. 102-106, S. 199-121, S. 435.