C++: Boost.Thread und Intel TBB

Martin Freund

2. Februar 2011

Zusammenfassung

Aufgrund der hohen Verfügbarkeit von Multicorearchitekturen ist die Entwicklung nebenläufiger Programme schon jetzt gängige Praxis. Um die Entwicklung solcher Programme zu beschleunigen, unterstützen manche Hochsprachen Parallelität bereits nativ. Indes schreitet die Evolution von C++ auf diesem Gebiet nur langsam voran, wodurch sich die Notwendigkeit von Threadbibliotheken ergibt, um die Konkurrenzfähig zu wahren.

Im Rahmen dieser Arbeit werden zwei bekannte Bibliotheken - *Boost Thread* und *Intel Thread Buildung Blocks* - praxisbezogen vorgestellt.

Inhaltsverzeichnis

1 Motivation **3**

2 Anforderungen an C++ Thread-Bibliotheken **3**
2.1 Plattformunabhängigkeit 3
2.2 C++0x Kompatibilität 3
2.3 Parallelisierung 4
2.4 Synchronisationsprimitive 4
2.5 Threadsichere Container 5

3 Boost.Thread **5**
3.1 Überblick 5
3.2 Verwaltung 5
3.3 Mutex- und Locktypen 6
3.4 Conditional Variables 8

4 Intel Thread Building Blocks **9**
4.1 Überblick 9
4.2 Verwaltung 10
4.3 Splittable- & Range-Konzept 10
4.4 Funktoren 11
4.5 Parallel Templates 12
4.6 Thread Local Storage 14
4.7 Speicherverwaltung 15
4.8 Threadsichere Container 15

Literatur **16**

1 Motivation

Die fortschreitende Spezialisierung auf Mehrkernprozessoren, speziell die Steigerung der Kernzahl (sog. Manycores[1]), erfordert ein gut durchdachtes Design der darauf auszuführenden Programme, um den Leistungsgewinn durch diese Architekturen auch bestmöglich nutzen zu können.

Neben der Lokalität der Daten ist das Konzept der Ausführungsfäden - den Threads - hierbei essentiell, weshalb Hochsprachen wie *C#*, *Delphi* und *Java* diese bereits nativ unterstützen. Hierbei findet eine Abstraktion vom Betriebssystem satt, wodurch der Aufwand durch die Threadverwaltung und somit auch mögliche Fehlerquellen minimiert werden.

Einzig C++ unterstützte bis zum Standard *C++0x* Multithreading nur durch die Abwärtskompatibilität mit Hilfe der C Laufzeitumgebung, wodurch sich die Notwendigkeit für externe Threadbibliotheken ergibt. Auch wenn der neue Standard *C++0x* bereits Threads unterstützt, so verlieren diese Bibliotheken nicht an Wert, da die gängigen Compiler wie *Comeau C++*, *GCC G++*, *Intel C++* und *MSVC* den neuen Standard bisher nur partiell implementiert haben.[Comeau, 2010][Meyers, 2010]

Sowohl *Boost.Thread* als auch *Intel Thread Building Blocks (TBB)* orientieren sich dabei stark an *C++0x* wodurch ein nahtloser Wechsel möglich ist, solange nur standardisierte Funktionalitäten verwendet werden. Ausgehend von einem allgemeinen Anforderungsprofil werden diese Bibliotheken anwendungsbezogen vorgestellt.

2 Anforderungen an C++ Thread-Bibliotheken

Neben der vorausgesetzten Korrektheit spielen weitere Kriterien eine wichtige Rolle bei der Wahl der zu verwendenden Bibliothek.

2.1 Plattformunabhängigkeit

Plattformunabhängigkeit impliziert zugleich eine höhere Abstraktionsebene, was z. B. das Erstellen eines Prototyps wesentlich verkürzt, da kein betriebssystemspezifisches Fachwissen nötig ist.

Speziell in einem heterogenen Hard- und Softwareumfeld ermöglicht eine plattformunabhängige Threadbibliothek kürzere Entwicklungszeiten und sauberen Code, da Eigenheiten der Architektur und des Betriebssystems nicht gesondert berücksichtigt werden müssen. So kann vor allem auf bedingte Kompilierung verzichtet werden.

Auch eine nachträgliche, kostengünstige Portierung (z. B. von Windows auf Linux) ist eine Option, die wettbewerbsentscheidend sein kann.

2.2 C++0x Kompatibilität

Sobald *C++0x* etabliert ist, werden einige, wenn nicht sogar alle Funktionalitäten mancher Bibliothek obsolet. Folglich wird man versuchen, die Abhängigkeiten von diesen Bibliotheken zu minimieren oder dieses gar ganz zu entfernen.

[1] Auf Skalierung getrimmte Mehrkernprozessoren, `http://software.intel.com/en-us/articles/many-core-processor/`

Wenn dabei alle Schnittstellen standardisiert sind, reduziert sich der Arbeitsaufwand auf das bloße Wechseln des Namensraums und dem Einbinden der entsprechenden Header.

Außerdem trägt die syntaktische Ähnlichkeit zur Verständlichkeit bei, wodurch zusätzliche Einarbeitungszeit entfällt. So sollten grundlegende Funktionalitäten wie die Kapselung von Threads in Objekte und die Sicherung kritischer Abschnitte äquivalent zu den standardisierten Klassen `std::thread` und `std::mutex` verfügbar sein.[Becker, 2010][Williams, 2008]

Ein zusätzlicher Leistungsgewinn kann durch die Verwendung *C++0x* Neuerungen wie z. B. den Lambda-Ausdrücken erzielt werden, da keine zusätzlichen Wrapperklassen erstellt werden müssen.

2.3 Parallelisierung

Die Aufgabenparallelisierung (Taskparallelität) ist eine grundlegende Anforderung an jede Threadbibliothek. Sie abstrahiert von der betriebssystemspezifischen Threadverwaltung durch Kapselung von Threads in Objekte. Diese Art der Parallelisierung sollte schon während der Entwicklung integriert werden, da eine nachträgliche Transformation Probleme mit sich bringt, die bei einem sequentiellen Programm nicht bedacht werden mussten. Darunter fallen z. B. die threadsichere Ressourcenverwaltung oder die Terminierung aller Threads, da ein Prozess nur dann korrekt beendet wird, wenn er keine ausführenden Threads mehr besitzt.

Datenparallelisierung spielt vor allem bei Manycores eine wichtige Rolle. Sie erlaubt ein nahezu lineares Verhältnis zwischen zugeteilten Threads und Datendurchsatz solange die Unabhängigkeit der Daten gewährleistet ist. Um dem Programmierer dabei die Arbeit zu erleichtern, sollte die Bibliothek Konstrukte zur Verfügung stellen, die ein generisches Aufteilen der Daten auf eine beliebige Anzahl an Threads erlaubt. Eine nachträgliche Parallelisierung ist hier ohne Probleme möglich, da die Nebenläufigkeit lokal begrenzt wird und keine Zugriffskonflikte auftreten.

2.4 Synchronisationsprimitive

Um Zugriffskonflikte zwischen mehreren Threads auf einen Datenbestand auflösen zu können, werden Konstrukte benötigt, die einen wechselseitigen Ausschluss garantieren und unter dem Obergriff „Mutal Exclusion“ (Mutex) zusammengefasst werden.

Die wohl wichtigste unter ihnen ist die binäre Semaphore, welche idealerweise mit dem Scheduler des Betriebssystems interagiert. Sie umschließt die zu schützende Befehlssequenz bzw. Ressource und gibt den Zugriff darauf nur frei, wenn dieser nicht schon an einen anderen Thread vergeben wurde. Ist die Semaphore bereits belegt, so wird der anfordernde Thread suspendiert und vom Scheduler erst wieder reaktiviert, wenn die Chance besteht, selbst Zugriff zu erhalten.

Dieses Konzept lässt sich noch weiter verfeinern, indem z. B. mit Hilfe eines Timeouts die Zugriffsanfrage abgebrochen werden kann. Außerdem ist es manchmal wünschenswert, den Zugriff auf eine Ressource auf mehrere Threads aufteilen zu können, so können Leseoperationen auf einer Datenbank durchaus

parallel erfolgen, da die Konsistenz dadurch nicht gefährdet wird. Schreibzugriffe müssen jedoch exklusiv erfolgen.

Neben der bloßen Sperrung von Ressourcen ist es oft notwendig, Threads auf Ereignisse reagieren zu lassen. So wäre das ständige Auslesen einer Aufgabenwarteschlange, um diese auf neue Elemente zu prüfen, ineffizient. Stattdessen könnte ein auslesender Thread reaktiviert werden, sobald die Warteschlange mindestens ein Element enthält. Dies wird durch sogenannte „Condition Variables“ erreicht.

2.5 Threadsichere Container

Die Container aus der *Standard Template Library (STL)* sind nicht threadsicher. Eine Möglichkeit besteht darin, eine Synchronisationsprimitive zu verwenden, um die Integrität der Daten zu gewährleisten. Dieser Ansatz ist jedoch aus mehreren Gründen nicht optimal:

- Der Programmierer trägt die Verantwortung für den korrekten Ablauf.
- Die Sperren müssen möglicherweise exceptionsicher ausgelegt werden.
- Optimierungspotential geht verloren, da z. B. mehrere Threads gleichzeitig lesend auf einen Container zugreifen können, eine externe Sperre dies jedoch aufgrund der fehlenden Einsicht nicht berücksichtigen kann und somit nur einem Thread den Zugriff gewährt.

Idealerweise liefert eine Threadbibliothek also threadsichere und zusätzlich bezüglich der Zugriffsart optimierte Container wie z. B. Listen und Hashmaps.

3 Boost.Thread

3.1 Überblick

Boost[Williams, 2007] ist eine Sammlung freier C++-Bibliotheken, deren Hauptaugenmerk auf Portabilität gelegt wird. Unterstützt werden u. a. *Unix*, *GNU/-Linux*, *Mac OS* und *Windows*. Das Projekt wurde im Jahr 2000 durch Mitglieder des C++-Standardisierungskomitees gegründet, um Spracherweiterungen in der Praxis zu testen. Es basiert stark auf Metaprogrammierung und wird auch verwendet, um die Standardkonformität von Compilern auf diesem Gebiet zu testen. Die *Boost*-Lizenz ist weniger restriktiv als die *GPL* oder gar die *LGPL*, wodurch *Boost* v. a. für kommerzielle Nutzung attraktiv wird.

Boost.Thread wurde ursprünglich von *William E. Kempf* entwickelt, dann aber an den zukünftigen Standard *C++0x* angepasst und liegt mittlerweile in Version 1.41.0 vor. Die Bibliothek ist auf die wichtigsten Funktionen beschränkt, worunter u. a. die Kapselung von Threads in Objekten und Synchronisierungsprimitive fallen.

3.2 Verwaltung

Die Erstellung eines Threads bis hin zur korrekten Terminierung benötigt lediglich zwei Codezeilen. Zuerst wird ein Objekt der Klasse `boost::thread` mit der Threadroutine `myThread` erstellt. Danach beginnt der Thread bereits mit

der Ausführung. Anschließend wird mit `join()` auf dessen Terminierung gewartet. Dieser Schritt ist nötig, um eine vorzeitige Zerstörung des Objekts zu verhindern.

```
1  #include <boost/thread.hpp>
2  #include <iostream>
3
4  void myThread() {
5    for( int i = 0; i < 9; i++) {
6      std::cout << i << std::endl;
7    }
8  }
9
10 int main( int argc, char* argv[]) {
11   boost::thread oThread( myThread);
12   oThread.join();
13   return 0;
14 }
```

Listing 1: Kapselung eines Threads in `boost::thread`

Bis auf die Methoden `sleep()` und `yield()` werden aus Gründen der Portabilität keine weiteren Verwaltungsfunktionen unterstützt. Um z. B. Threads zu pausieren, müssen Synchronisationsprimitive verwendet werden. Es ist jedoch möglich, über `native_handle()` einen Verweis auf den Thread zu erhalten und diesen damit plattformabhängig zu verwalten.

Um auf dem aktuellen Kontext ohne eine Threadreferenz zu operieren, steht der Namensraum `boost::this_thread` zur Verfügung, mit dessen Hilfe z. B. der ausführende Thread per `boost::this_thread::get_id()` identifiziert werden kann. Diese Funktion liefert eine Instanz der Klasse `boost::thread::id`, welche Vergleichsoperationen unterstützt und über einen Stream die textuelle Repräsentation der Thread-ID generieren kann. Dabei spielt es keine Rolle, wie der Thread erzeugt wurde:

```
1  #include <boost/thread.hpp>
2  #include <iostream>
3
4  int main( int argc, char* argv[]) {
5    std::cout << "Thread-ID: " <<
6      boost::this_thread::get_id() << std::endl;
7    return 0;
8  }
```

Listing 2: Ausgabe der ID des ausführenden Threads

3.3 Mutex- und Locktypen

Zur Auflösung von Zugriffskonflikten auf Ressourcen verwendet *Boost.Thread* ein objektorientiertes, hierarchisch aufgebautes Mutex-System. Dieses basiert auf vier aufeinander aufbauenden Konzepten:

- **Lockable** erlaubt nur blockierende Sperranfragen sowie einmalige Sperrversuche.

- **TimedLockable** ermöglicht timeout-basierte, blockierende Sperranfragen.
- **SharedLockable** erlaubt mehreren Threads, die selbe Mutex zu sperren.
- **UpgradeLockable** ermöglicht eine geteilte Sperre in eine exklusive Sperre umzuwandeln.

Diese Konzepte werden von fünf vordefinierten Mutexen umgesetzt:

- `boost::mutex` implementiert nur **Lockable**
- `boost::timed_mutex` implementiert zusätzlich **TimedLockable**
- `boost::shared_mutex` implementiert außerdem noch **SharedLockable** und **UpgradeLockable**
- `boost::recursive_mutex` erweitert `boost::mutex` lediglich um die Wiedereintrittsfähigkeit
- `boost::recursive_timed_mutex` erweitert `boost::timed_mutex` lediglich um die Wiedereintrittsfähigkeit

Bei nicht-rekursiven Mutexen ist der Wiedereintritt undefiniert und muss daher vermieden werden. Die effizienteste Form ist auf Grund des geringsten Verwaltungsaufwands `boost::mutex`.

Kontrolliert wird eine Mutex (`Lockable`) von mindestens einem Lockobjekt. Diese lassen sich wiederum in verschiedene Typen unterteilen:

- `boost::lock_guard<Lockable>` einfache Sperre ohne Timeout
- `boost::unique_lock<Lockable>` manuell kontrollierbare Sperre
- `boost::shared_lock<Lockable>` geteilte Sperre
- `boost::upgrade_lock<Lockable>` erwirbt das alleinige Upgraderecht für einen `boost::shared_lock<Lockable>`
- `boost::upgrade_to_unique_lock<Lockable>` führt mit Hilfe des Upgraderechts ein Upgrade auf einen `boost::unique_lock<Lockable>` durch

Charakteristisch für Lockobjekte ist die durchgängige Anwendung des RAII[2]-Prinzips. Dabei wird - wenn im Konstruktor nicht anders spezifiziert - eine Mutex bereits während der Erstellung des Lockobjekts gesperrt. Außerdem wird diese Sperre durch die Zerstörung des Lockobjekts automatisch wieder freigegeben. Somit können Sperren exception-sicher umgesetzt werden, da sich der Programmierer nicht mehr selbst um die Integrität kümmern muss.

Zu beachten ist noch, dass sowohl `boost::upgrade_lock<Lockable>` als auch `boost::upgrade_to_unique_lock<Lockable>` nur in Verbindung mit einer `boost::shared_mutex` verwendet werden können, da sie auf dem **SharedLockable** bzw. dem **UpgradeLockable** Konzept aufbauen. Für die meisten Anwendungsfälle ist jedoch `boost::lock_guard<Lockable>` vollkommen ausreichend. Im folgenden Beispielcode werden zwei Threads erzeugt, die jeweils von 0 bis 9 zählen. Um dabei zu verhindern, dass der Schreibvorgang vom anderen Thread unterbrochen wird, wird das Scope der `for`-Schleife gesperrt:

[2]Ressourcenbelegung ist Initialisierung, meist abgekürzt durch RAII, für engl. resource acquisition is initialization

```
1  #include <boost/bind.hpp>
2  #include <boost/thread.hpp>
3  #include <iostream>
4
5  boost::mutex oMutex;
6
7  void myThread( int id) {
8    for( int i = 0; i < 10; i++) {
9      // Die kritische Region beginnt und endet
10     // mit der G\"ultigkeit von 'lock'.
11     boost::lock_guard<boost::mutex> lock( oMutex);
12     std::cout << id << ": " << i << std::endl;
13   }
14 }
15
16 int main( int argc, char* argv[]) {
17   boost::thread oThread1( boost::bind( myThread, 1));
18   boost::thread oThread2( boost::bind( myThread, 2));
19   oThread1.join();
20   oThread2.join();
21   return 0;
22 }
```

Listing 3: Einfache Sperrung mit `boost::mutex` durch `boost::lock_guard`

Da Threadroutinen weder Rückgabewerte noch Parameter haben dürfen, damit sie von `boost::thread` gekapselt werden können, wird auf *Boost.Bind* zurückgegriffen. Es kapselt die Routine mit einer konstanten ID in einen Funktor, welcher dann die passende Signatur liefert.

3.4 Conditional Variables

Condition Variables benötigen sowohl eine Mutex als auch ein Lockobjekt. `boost::condition_variable_any` ist die generalisierte Version, während mit `boost::condition_variable` nur `boost::unique_lock<boost::mutex>` Lockobjekt verwendet werden können. Diese Einschränkung ermöglicht jedoch intern eine höheren Optimierung.

Um auf ein Ereignis zu warten, wird die Methode `wait()` verwendet, welche atomar die aktive Sperre freigibt und auf die Signalisierung durch einen anderen Thread wartet. Nach einer Signalisierung wird solange blockiert, bis der Thread wieder im Besitz der Sperre ist.

Zur Signalisierung von wartenden Threads wird die Methoden `notify_all()` bzw. `notify_one()` aufgerufen. Erstere führt dazu, dass alle wartenden Threads versuchen, die Sperre zu erhalten, während letztere nur einen beliebigen Thread aufweckt.

Im folgenden Beispielcode wird der Container `vecNumbers` schrittweise vom ersten Thread mit den Zahlen 0, 1 und 2 gefüllt, während der zweite Thread nach jeder Einfügeoperation das neue Element ausgibt. Dabei informiert jeder Thread den anderen, sobald die jeweilige Operation beendet wurde.

```
1  #include <boost/thread.hpp>
2  #include <iostream>
3  #include <vector>
```

```
4
5  boost::mutex oMutex;
6  boost::condition_variable oCondition;
7  std::vector<int> vecNumbers;
8
9  void fill() {
10   for( int i = 0; i < 3; i++) {
11     boost::unique_lock<boost::mutex> lock( oMutex);
12     vecNumbers.push_back( i);
13
14     // Anderen Thread signalisieren, erneute Sperrung abwarten
15     oCondition.notify_all();
16     oCondition.wait( lock);
17   }
18 }
19
20 void print() {
21   std::size_t stNextSize = 1;
22   for( int i = 0; i < 3; i++) {
23     // Auf ein weiteres Element im Container warten
24     boost::unique_lock<boost::mutex> lock( oMutex);
25     while( vecNumbers.size() != stNextSize)
26       oCondition.wait( lock);
27
28     std::cout << vecNumbers.back() << std::endl;
29     stNextSize++;
30
31     // Anderen Thread signalisieren
32     oCondition.notify_all();
33   }
34 }
35
36 int main() {
37   boost::thread oThread1( fill);
38   boost::thread oThread2( print);
39   oThread1.join();
40   oThread2.join();
41   return 0;
42 }
```

Listing 4: Beispiel `boost::condition_variable`

4 Intel Thread Building Blocks

4.1 Überblick

Die *Intel Thread Building Blocks* sind ebenfalls eine Open-Source-Bibliothek. Sie wurde 2006 veröffentlicht und liegt mittlerweile in Version 3.0 vor. Im Unterschied zu *Boost.Thread* bietet *Intel* zwei verschiedene Lizenzen an. Einerseits die *GPLv2* mit ihren Vor- und Nachteilen und andererseits eine kommerzielle Lizenz. Unterstützt werden *GNU/Linux*, *Mac OS* und *Windows*.Intel [2010b]

Der Funktionsumfang übersteigt den von *Boost.Thread* bei weitem, was sich vor allem auf die Unterstützung von Datenparallelisierung zurückführen lässt.

So können funktionale Konzepte wie *MapReduce* mit Hilfe von Metaprogrammierung umgesetzt oder Programme nachträglich parallelisiert werden. Auch ein neues, skalierendes Speichermodell trägt zur Leistungssteigerung bei, da parallele Speicherbereichsanforderungen und -freigaben effizienter verarbeitet werden können. Außerdem wird ein separates Threadscheduling ermöglicht. Die Bibliothek ist kompatibel zu *C++0x* und beinhaltet zudem noch threadsichere Container. Intel [2010a]

Verwendet wird die Bibliothek beispielsweise vom Animationsstudio *DreamWorks* und der *Unreal Engine* von *Epic Games*.

4.2 Verwaltung

Jeder Thread der parallele Algorithmen oder den Scheduler verwendet, benötigt ein initialisiertes *tbb::task_scheduler_init* Objekt. Dieses erstellt man am einfachsten am Anfang des Programms, wodurch gleichzeitig dessen Gültigkeit über die Dauer des Programms gewährleistet ist:

```
1 #include <tbb/task_scheduler_init.h>
2
3 int main( int argc, char* argv[]) {
4   task_scheduler_init init;
5   ...
6   return 0;
7 }
```

Listing 5: Initialisierung der *Intel TBB*

Die Erstellung von Threads sowie die Verwendung der Synchronisationsprimitive erfolgt aufgrund der Standarisierung analog zu *Boost.thread*. Der Umfang an Mutextypen ist im Vergleich zu *Boost.Thread* etwas eingeschränkt, so werden z. B. keine wiedereintrittsfähigen Mutexe unterstützt.

4.3 Splittable- & Range-Konzept

Die Grundlage für Datenparallelisierung in *Intel TBB* bilden das Splittable- & Range-Konzept.

Ersteres ermöglicht sowohl die Aufteilung des Eingabebereichs in zwei getrennte Arbeitspakete als auch die Teilung einer laufenden Operation im Funktionsrumpf. Um dieses Konzept umzusetzen benötigt ein Objekt den Split-Konstruktor `X::X(X& x, tbb::split)`. `split` ist hier lediglich ein Dummy, um den Split-Konstruktor von einem Copy-Konstruktor unterscheiden zu können.

Das Range-Konzept wird hingegen nur für das rekursive und auch parallele Aufteilen des Eingabebereichs benötigt und erweitert das Splittable-Konzept um die Methoden `empty()` und `is_divisible()`. Für einfache ein- bis dreidimensionale Eingabebereiche können vordefinierten Ranges verwendet werden:

- `blocked_range<Value>`
- `blocked_range2d<RowValue,ColValue>`
- `blocked_range3d<PageValue,RowValue,ColValue>`

Neben dem Anfangs- und Enditerator kann an deren Konstruktor noch ein zusätzlicher Parameter übergeben werden, der die Granularität der Aufteilung steuert. Solange die Anzahl der Elemente in einer Range diesen Wert übersteigt, wird die Range aufgeteilt. Die Wahl der Granularität kann die Effizienz stark beeinflussen, da ein zu hoher Wert die Rechenkapazität nicht auslastet und ein zu niedriger Wert den Parallelisierungsgewinn durch den Verwaltungsaufwand zunichte macht. Es wird daher empfohlen, den Wert auf `tbb::IdealGrainSize` zu belassen. Dieser wurde von Intel empirisch bestimmt und beträgt 10,000. Um jedoch einen messbaren Leistungsverlust zu erhalten, müssen die Werte stark abweichen, wie auf der Grafik deutlich zu erkennen ist:

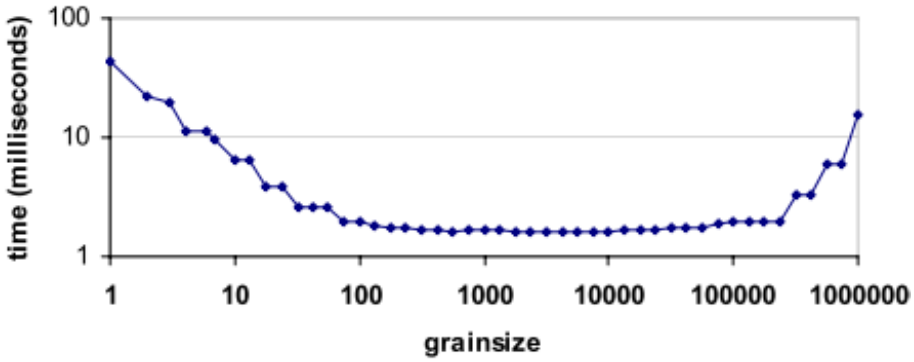

Abbildung 1: Beziehung zwischen Rechenzeit und Granularität

Ranges lassen sich leicht nach folgendem Muster implementierenIntel [2010c]:

```
1  struct TrivialIntegerRange {
2    int lower, upper;
3    bool empty() const {
4      return lower == upper;
5    }
6    bool is_divisible() const {
7      return upper > lower + 1;
8    }
9    TrivialIntegerRange( TrivialIntegerRange& r, tbb::split) {
10     int m = ( r.lower + r.upper) / 2;
11     lower = m;
12     upper = r.upper;
13     r.upper = m;
14   }
15 };
```

Listing 6: Beispiel für die Implementierung eines Rangeobjekts

4.4 Funktoren

Bei Funktionsobjekten (sog. Funktoren) handelt es sich in *C++* um Objekte, deren `operator()` überladen wurde. Dadurch entspricht der Aufruf einer solchen Methode syntaktisch dem einer Funktion, wie folgendes Beispiel verdeutlicht:

```
1  struct Sum {
```

```
2     int operator()( int x, int y) {
3       return x + y;
4     }
5   };
6
7   int sum( int x, int y) {
8     return x + y;
9   }
10
11  int main( int argc, char* argv[]) {
12    Sum oSum;
13    int temp;
14    temp = oSum( 10, 5);
15    temp = sum( 10, 5);
16    return 0;
17  }
```

Listing 7: Funktor und Funktion

Die Vorteile eines Funktors liegen einerseits in der Abstraktion von Funktionen und andererseits in der Erweiterung der Anwendungsmöglichkeiten. So lässt sich ein Funktor kopieren oder sein Zustand durch andere Methoden modifizieren. Außerdem ist eine Klasse in der Metaprogrammierung auf Grund zusätzlicher Typinformationen wesentlich nützlicher als eine Funktion. So können Klassen z. B. Typen definieren, die nach außen sichtbar sind, wovon u. a. in der *STL* Gebraucht gemacht wird.

Einige Konzepte der *Intel TBB* erfordern explizit die Verwendung von Funktoren, während an anderen Stellen Lambda-Ausdrücke genügen. Da diese jedoch erst in *C++0x* verfügbar sind, werden in den folgenden Codebeispielen bis auf eine Ausnahme Funktoren verwendet.

4.5 Parallel Templates

Intel TBB ermöglicht durch die *parallel templates* Datenparallelisierung, die teilweise der von *OpenMP* ähnelt. Zur Auswahl stehen u. a. folgende Templates:

- `parallel_for` für das Parallelisieren von Schleifen.
- `parallel_reduce` führt eine parallele `reduce`-Operation auf einer Folge durch. Dabei werden alle Elemente durch eine assoziative Operation zu einem einzigen Wert verknüpft.
- `parallel_scan` führt einen Präfixscan durch, welcher die Zwischenergebnisse, wie sie bei `reduce` auftreten würden, zurückgibt.
- `parallel_sort` zum vergleichenden Sortieren einer Folge.

Die Parallelisierung wird dabei vollständig durch das jeweilige Template gesteuert.

Um beispielsweise folgende `map`-Operation, die auf einer Folge von Elementen die Funktion `foo()` ausführt, zu parallelisieren

```
1   void applyFoo( int a[], unsigned int n) {
2     for( int i = 0; i < n; i++)
3       a[i] = foo( a[i]);
```

```
}
```

Listing 8: Sequentielle `map`-Operation

muss der Quellcode folgendermaßen ergänzt bzw. geändert werden:

```
#include <tbb/blocked_range.h>

class Functor {
  int* const m_a;
public:
  void operator()(const tbb::blocked_range<unsigned int>& r) {
    int* a = m_a;
    for( unsigned int i = r.begin(); i != r.end(); i++)
      a[i] = foo( a[i]);
  }
  Functor( int a[]) : m_a( a) { }
};

void applyFoo( int a[], unsigned int n) {
  tbb::parallel_for( tbb::blocked_range<unsigned int>(
    0, n, tbb::IdealGrainSize), Functor( a));
}
```

Listing 9: Parallelisierung mit `parallel_for`

```
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body);
```

Listing 10: `parallel_for` Signatur

Die Aufgabe der Klasse `Functor` besteht hier lediglich darin, die `map`-Operation auf jedes Element im festgelegten Eingabebereich `r` auszuführen. Da die Berechnungen voneinander unabhängig sind, können bei `parallel_for` statt Funktoren auch *C++0x* Lambda-Ausdrücke verwendet werden, wodurch sich obiger Code deutlich verkürzt:

```
#include <tbb/blocked_range.h>

void applyFoo( int a[], unsigned int n) {
  tbb::parallel_for( tbb::blocked_range<unsigned int>( 0, n),
    [=]( const blocked_range<unsigned int>& r) {
      for( unsigned int i = r.begin(); i != r.end(); i++)
        a[i] = foo( a[i]);
    }
  );
}
```

Listing 11: `parallel_for` mit Lambda-Ausdruck

Auch `parallel_reduce` benötigt ebenfalls einen Funktor, dieser muss jedoch zusätzlich einen `split`-Konstruktor und die `join()` Methode beinhalten, um triviale Teilfolgen zu erstellen und deren Ergebnisse wieder zusammenführen zu können. Folgendes Codebeispiel führt eine `map&reduce`-Operation auf einer Folge durch:

```
class Functor {
  int* m_a;
```

```
3  public:
4    int m_sum;
5    void operator()(const tbb::blocked_range<unsigned int>& r) {
6      for( unsigned int i = r.begin(); i != r.end(); i++)
7        m_sum += foo( m_a[i]);
8    }
9    void join( const Functor& y) {
10     m_sum += y.m_sum;
11   }
12
13   Functor( Functor& x, tbb::split) : m_a( x.m_a), m_sum( 0) {}
14   Functor( int a[]) : m_a( a), m_sum( 0) {}
15 };
16
17 int applyFooAndReduce( const int a[], unsigned int n) {
18   Functor oSum( a);
19   tbb::parallel_reduce( tbb::blocked_range<unsigned int>(
20     0, n, tbb::IdealGrainSize), oSum);
21   return oSum.m_sum;
22 }
```

Listing 12: Parallelisierung mit `parallel_reduce`

4.6 Thread Local Storage

Ein Problem, das während der nachträglichen Parallelisierung eines Programms auftreten kann, ist das Vorhandensein von statischen Variablen in einer zu parallelisierenden Funktion. Jeder Thread hat zwar seinen eigenen Stapelspeicher, teilt sich jedoch die statische Variable mit den anderen Threads. Falls der Inhalt der Variable modifiziert wird, kann dies zur Dateninkonsistenz führen. Um dieses Problem zu lösen, kann die Funktion einerseits komplett umgeschrieben werden oder andererseits auf *Thread Local Storage (TLS)* zurückgegriffen werden.

TLS belegt für jeden Thread, der Zugriff auf eine entsprechende Variable hat, einen von den anderen Threads getrennten Speicherbereich. Dadurch entspricht die Semantik wieder der des ursprünglichen, sequentiellen Programms. *Intel TBB* verwendet hierfür die Klasse `enumerable_thread_specific<T>`, welche einen beliebigen Datentyp im *TLS* des jeweiligen Threads ablegt. Die Besonderheit dieser Klasse ist, dass über alle Einträge iteriert werden kann. Wie bereits erwähnt, wird ein Eintrag erst beim Zugriff eines Threads erstellt, falls für diesen noch keiner existiert.

Folgendes Beispiel zeigt die Anpassung einer Funktion mit statischer Variable für die Nutzung mit mehreren Threads:

```
1  float floatingMeanSingleThreaded( float input) {
2    static float mean = 0.0f;
3    mean = mean * 0.75f + input * 0.25f;
4    return mean;
5  }
6
7  #include <tbb/enumerable_thread_specific.h>
8
9  float floatingMeanMultiThreaded( float input) {
10   static tbb::enumerable_thread_specific<float> oMean( 0.0f);
```

```
11    float& mean = oMean.local();
12    mean = mean * 0.75f + input * 0.25f;
13    return mean;
14  }
```

Listing 13: Umwandlung einer statischen Variable in eine *TLS* Variable

4.7 Speicherverwaltung

Bei massiv parallelisierten Programmen, die zudem noch speicherintensiv sind, kann die Speicherverwaltung zum Flaschenhals werden, da diese standardmäßig nur sequentiell arbeitet. Aus diesem Grund bietet *Intel TBB* den skalierenden Allokator `scalable_allocator<T>`, welcher jedoch die separate Speichermanagement Bibliothek der *Intel TBB* benötigt.

Folgender Code verdeutlicht die Integration des Allokators in einen *STL* Vektor sowie das Anlegen und Freigeben eines einfachen Arrays:

```
1  #include <tbb/scalable_allocator.h>
2  #include <vector>
3
4  void foo() {
5    std::vector<int,cache_aligned_allocator<int> > vecFoo;
6
7    // C-kompatible Allokation
8    char* data = scalable_malloc( 500);
9    ...
10   scalable_free( data);
11 }
```

Listing 14: Beispiel für skalierenden Allokator der *Intel TBB*

4.8 Threadsichere Container

Intel TBB unterstützt drei threadsichere und *STL*-kompatible Container:

- `concurrent_hash_map<Key,Value,Compare>`
- `concurrent_vector<T>`
- `concurrent_queue<T>`

Diese sind hochspezialisiert und erlauben auch gleichzeitigen Schreibzugriff. Um die Konsistenz zu wahren, müssen modifizierende Operationen Änderungen durch andere Threads erkennen und entsprechend darauf reagieren. Dies führt zu einem zusätzlichen Aufwand, weshalb man abwägen sollte, ob der tatsächliche Nutzen eines threadsicherer Container im konkreten Anwendungsfall dem einer Lösung mit externer Sicherung durch eine Mutex überwiegt.

Literatur

Pete Becker. *Working Draft, Standard for Programming Language C++*, 2010. Online verfügbar unter `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf`; heruntergeladen am 15.12.2010.

Comeau. *Features in Comeau C/C++ 4.3.9*, 2010. Online verfügbar unter `http://www.comeaucomputing.com/439features.html`; besucht am 5.1.2010.

Intel. *Intel TBB Release Notes*, 2010a. Online verfügbar unter `http://software.intel.com/sites/products/documentation/hpc/tbb/reference.pdf`; heruntergeladen am 11.10.2010.

Intel. *Intel TBB Reference Manual*, 2010b. Online verfügbar unter `http://software.intel.com/sites/products/documentation/hpc/tbb/release_notesv2.txt`; heruntergeladen am 11.10.2010.

Intel. *Intel TBB Tutorial*, 2010c. Online verfügbar unter `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/tbb/tutorial.pdf`; heruntergeladen am 11.10.2010.

Scott Meyers. *Summary of C++0x Feature Availability in gcc and MSVC*, 2010. Online verfügbar unter `http://www.aristeia.com/C++0x/C++0xFeatureAvailability.htm`; besucht am 5.1.2011.

Anthony Williams. *Boost.Thread Documentation*, 2007. Online verfügbar unter `http://www.boost.org/doc/libs/1_45_0/doc/html/thread.html`; besucht am 1.2.2011.

Anthony Williams. *Simpler Multithreading in C++0x*, 2008. Online verfügbar unter `http://www.devx.com/SpecialReports/Article/38883`; besucht am 5.1.2011.