C++: Boost.Thread und Intel TBB

Martin Freund

Fakultät für Informatik und Mathematik
Universität Passau

3. Februar 2011

Inhaltsverzeichnis

Wozu Threadbibliotheken?

- Einfachheit
- Abstraktion
- Portabilität
- Stand der Technik
- Erweiterung der Möglichkeiten

Inhaltsverzeichnis

Welche Vorteile ergeben sich durch Portabilität?

- geringere Einarbeitungszeit
- bessere Vermarktungschancen
- Abstraktion

Inhaltsverzeichnis

Was soll parallel ablaufen?

- verschiedene Aufgaben (Taskparallelität)
- Datenströme in einer Aufgabe (Datenparallelität)

Inhaltsverzeichnis

Synchronisation zwischen Threads

- Wechselseitiger Ausschluss - Mutual Exclusion
- Ereignisse - Condition Variables

Inhaltsverzeichnis

Was ist Boost?

- Sammlung freier und portabler C++-Bibliotheken
- im Jahr 2000 durch Mitglieder des C++-Standardisierungskomitees gegründet
- unter einer sehr liberalen Lizenz verfügbar
- Ideenschmiede für zukünftige Spracherweiterungen

Was ist Boost.Thread?

- wurde ursprünglich von William E. Kempf entwickelt
- reflektiert die Neuerungen durch C++0x
- bietet nur grundlegende Threading-Funktionalität

Inhaltsverzeichnis

Erstellung eines Threads

```
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
14   return 0;
15 }
```

Inhaltsverzeichnis

Mutex Konzepte

Aufeinander aufbauende, komplexer werdende Funktionalität:

- Lockable - blockierende und einmalige Sperranfragen
- TimedLockable - blockierende Sperranfragen mit Timeout
- SharedLockable - gleichzeitige Sperrung durch mehrere Threads
- UpgradeLockable - Umwandlung einer geteilten in eine exklusive Sperre

Mutextypen

Verfügbare Mutexklassen:

- boost::mutex - nur Lockable
- boost::timed_mutex - zusätzlich TimedLockable

Mutextypen

Verfügbare Mutexklassen:

- boost::mutex - nur Lockable
- boost::timed_mutex - zusätzlich TimedLockable
- boost::shared_mutex - außerdem SharedLockable und UpgradeLockable

Mutextypen

Verfügbare Mutexklassen:

- boost::mutex - nur Lockable
- boost::timed_mutex - zusätzlich TimedLockable
- boost::shared_mutex - außerdem SharedLockable und UpgradeLockable
- boost::recursive_mutex - wiedereintrittsfähig
- boost::recursive_timed_mutex - wiedereintrittsfähig

Locktypen

RAII

Resource acquisition is initialization

Locktypen

RAII

Resource acquisition is initialization

Verfügbare Locks:

- boost::lock_guard<Lockable>
- boost::unique_lock<Lockable>

Locktypen

RAII

Resource acquisition is initialization

Verfügbare Locks:

- boost::lock_guard<Lockable>
- boost::unique_lock<Lockable>
- boost::shared_lock<Lockable>
- boost::upgrade_lock<Lockable>
- boost::upgrade_to_unique_lock<Lockable>

Anwendung

```
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      boost::lock_guard<boost::mutex> lock( oMutex);
10     std::cout << id << ": " << i << std::endl;
11   }
12 }
13
14 int main( int argc, char* argv[]) {
15   boost::thread oThread1( boost::bind( myThread, 1));
16   boost::thread oThread2( boost::bind( myThread, 2));
17   oThread1.join(); oThread2.join();
18
19   return 0;
20 }
```

Inhaltsverzeichnis

Voraussetzungen

boost::condition_variable_any

Benötigt ein Mutexobjekt sowie ein beliebiges Lockobjekt.

boost::condition_variable

Benötigt ein Mutexobjekt sowie ein boost::unique_lock<> Lockobjekt.

Funktionsweise

```
1  boost::mutex oMutex;
2  boost::condition_variable oCondition;
3  std::vector<int> vecNumbers;
4
5  void fill() {
6    for( int i = 0; i < 3; i++) {
7      boost::unique_lock<boost::mutex> lock( oMutex);
8      vecNumbers.push_back( i);
9      oCondition.notify_all();
10     oCondition.wait( lock);
11   }
12 }
13 void print() {
14   std::size_t next = 1;
15   for( int i = 0; i < 3; i++) {
16     boost::unique_lock<boost::mutex> lock( oMutex);
17     while( vecNumbers.size() != next)
18       oCondition.wait( lock);
19     std::cout << vecNumbers.back() << std::endl;
20     next++;
21     oCondition.notify_all();
22   }
23 }
```

Inhaltsverzeichnis

Was ist Intel TBB?

- Intel Thread Building Blocks
- im Jahr 2006 veröffentlicht, mittlerweile in Version 3.0
- basiert sehr stark auf Metaprogrammierung
- unter kommerzieller Lizenz sowie unter GPLv2 erhältlich
- ermöglicht Datenparallelisierung

Inhaltsverzeichnis

Initialisierung

Initialisierung zwingend notwendig bei

- Verwendung des Schedulers
- Verwendung der Parallel Templates

```
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  }
```

Inhaltsverzeichnis

Aufteilung des Eingabebereichs

- Aufteilung notwendig für parallele Verarbeitung
- Automatisierung des Prozesses durch Range-Konzept
- Vordefinierte Ranges:
  - blocked_range<Value>
  - blocked_range2d<RowValue,ColValue>
  - blocked_range3d<PageValue,RowValue,ColValue>

Beispiel für eine benutzerdefinierte Range

```
1  struct TrivialIntegerRange {
2    int lower;
3    int upper;
4
5    bool empty() const {
6      return lower == upper;
7    }
8
9    bool is_divisible() const {
10     return upper > lower + 1;
11   }
12
13   TrivialIntegerRange( TrivialIntegerRange& r, split) {
14     int m = ( r.lower + r.upper) / 2;
15     lower = m;
16     upper = r.upper;
17     r.upper = m;
18   }
19 };
```

Granularität von Ranges

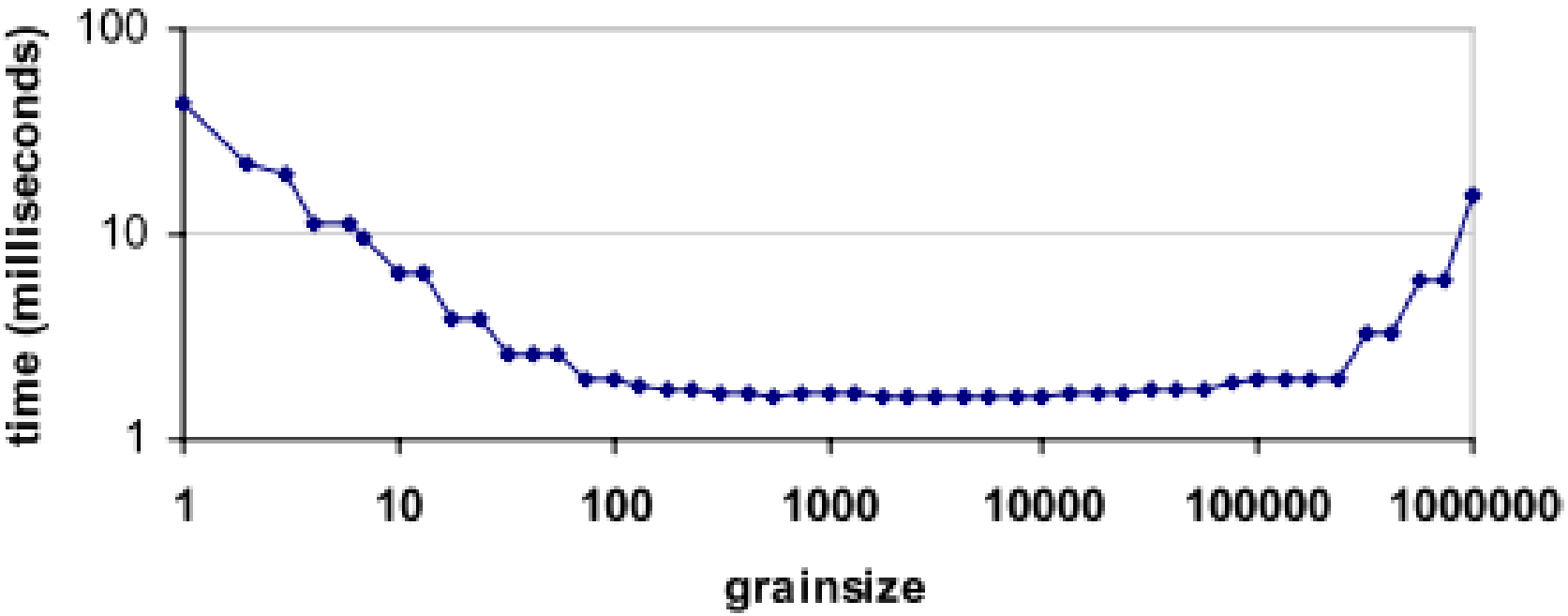

Abbildung: Beziehung zwischen Rechenzeit und Granularität

Inhaltsverzeichnis

Funktionsobjekte

```
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 }
```

Inhaltsverzeichnis

Grundlagen der Metaprogrammierung

```
1  template<typename Type>
2  struct GenericWrapper {
3    Type val;
4    GenericWrapper( Type& initial) : val( initial) {}
5    Type& get() { return val; }
6  }
7
8  template<typename Type>
9  Type& min( Type& a, Type& b) {
10   return a < b ? a : b;
11 }
12
13 int foo() {
14   GenericWrapper<int> oInt( 13);
15   // return min<long>( oInt.get(), 20);
16   return min( oInt.get(), 20);
17 }
```

Templatespezialisierung

```
1  template<unsigned int N>
2  struct Faculty {
3    enum {
4      Result = N * Faculty<N - 1>::Result
5    };
6  }
7
8  template<>
9  struct Faculty<0> {
10   enum {
11     Result = 1
12   };
13 }
14
15 void foo() {
16   unsigned int temp = Faculty<5>::Result;
17   ...
18 }
```

Auswahl verschiedener Parallel Templates

- parallel_for
- parallel_reduce
- parallel_scan
- parallel_sort

Signatur des parallel_for Templates:

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

Parallelisierung mittels parallel_for

```
1  void applyFoo( int a[], unsigned int n) {
2    for( int i = 0; i < n; i++)
3      a[i] = foo( a[i]);
4  }
5
6  #include <tbb/blocked_range.h>
7
8  class Functor {
9    int* const m_a;
10 public:
11   void operator()( const tbb::blocked_range<unsigned int>& r) {
12     int* a = m_a;
13     for( unsigned int i = r.begin(); i != r.end(); i++)
14       a[i] = foo( a[i]);
15   }
16   Functor( int a[]) : m_a( a) { }
17 };
18
19 void applyFoo( int a[], unsigned int n) {
20   tbb::parallel_for( tbb::blocked_range<unsigned int>(
21     0, n, tbb::IdealGrainSize), Functor( a));
22 }
```

parallel_for mit Lambda-Ausdruck

```
1  void applyFoo( int a[], unsigned int n) {
2    tbb::parallel_for( tbb::blocked_range<unsigned int>( 0, n),
3      [=]( const blocked_range<unsigned int>& r) {
4        for( unsigned int i = r.begin(); i != r.end(); i++)
5          a[i] = foo( a[i]);
6        }
7    );
8  }
```

Parallelisierung mittels parallel_reduce

```
1  class Functor {
2    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, 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
22   return oSum.m_sum;
23 }
```