Advanced Software Development

Parallele Programmierung

Intel TBB

- http://www.threadingbuildingblocks.org/
- Plattformunabhängige Parallelisierungs-Library
- Verfügbare Versionen
  - kommerzielle Version
  - Open Source Version (GPL)
- Aktuelle Version: 4.4

Microsoft PPL

- „Parallel Pattern Library“: Microsofts Alternative zu TBB
- Verfügbar ab Visual C++ 2010
- Teilweise sehr ähnlich

Übersicht

- Parallele Algorithmen
- Pipelines
- Synchronisation
- TBB Container
- Tasks

Parallele Algorithmen (TBB)

- tbb::parallel_sort
- tbb::parallel_for
- tbb::parallel_reduce
- tbb::parallel_do
- tbb::parallel_for_each
- tbb::parallel_invoke

parallel_sort

- Parallele Sortierung eines Bereichs (nicht stabil)
- Verwendung wie std::sort

```
parallel_sort(v.begin(), v.end());
parallel_sort(v.begin(), v.end(), greater<int>());
parallel_sort(v.begin(), v.end(),
              [](int x, int y) { return x > y; });
```

parallel_for

- Parallele Anwendung einer Funktion auf alle Elemente eines Bereichs

```
vector<int> v;
parallel_for(size_t(0), v.size(), [&v](size_t i) {
  v[i] = v[i] * v[i];
});
```

parallel_for: Varianten

- parallel_for(index1, index2, function)
- parallel_for(index1, index2, step, function)
- parallel_for(range, function, partitioner)
  - range: zu durchlaufender Bereich
  - function: wird für jeden Teilbereich aufgerufen
  - partitioner (optional): gibt an, wie Bereich aufgeteilt wird

parallel_for mit Range

```
class Squares {
public:
  Squares(vector<int> &n): numbers(n) {}
  void operator()(blocked_range<size_t> const &range) const {
    for (size_t i = range.begin(); i != range.end(); i++) {
      numbers[i] = numbers[i] * numbers[i];
    }
  }
private:
  vector<int> &numbers;
};

vector<int> v;
…
parallel_for(blocked_range<size_t>(0, v.size()), Squares(v));
```

parallel_for mit Range (2)

```
// Variante mit Iteratoren, Lambdas und std::transform

typedef vector<int>::iterator TIter;

vector<int> v;

parallel_for(blocked_range<TIter>(v.begin(), v.end()),
  [](blocked_range<TIter> const &range) {
     transform(range.begin(), range.end(), range.begin(),
               [](int x) { return x * x; });
});
```

parallel_reduce

- Zur Berechnung einer Funktion (z.B. Summe) über alle Elemente
- Parameter
  - Range
  - Initialwert
  - Berechnungsfunktion
  - Reduktionsfunktion

parallel_reduce – Beispiel

```
vector<int> v;
using Range = blocked_range<vector<int>::const_iterator>;
// ...
int sum = parallel_reduce(Range(v.begin(), v.end()), 0,
  [] (Range const &range, int v) {
    return v + accumulate(range.begin(), range.end(), 0);
  },
  plus<int>());
```

combinable

- Zur Kombination von parallel berechneten Teilergebnissen
- Ermöglicht Reduktion mittels parallel_for

```
vector<int> v;
...
combinable<int> c;
parallel_for(size_t(0), v.size(), [&](size_t i) {
  c.local() += v[i];
});
int sum = c.combine(plus<int>());
```

combinable: Methoden

- local(): liefert Referenz auf lokalen Wert
- combine(): kombiniert alle Werte mittels zweistelliger Funktion
- combine_each: iteriert über alle Werte

```
int sum = 0;
c.combine_each([&sum] (int x) { sum += x; });
```

parallel_reduce mit eigener Klasse

- Für größere Datenstrukturen (zur Vermeidung unnötiger Kopien)
- Anforderungen an Klasse
  - Überladener operator() für die Berechnung
  - „Splitting constructor“ (zum Aufteilen auf mehrere Funktionsobjekte)
  - join-Methode (Zusammenfügen der Ergebnisse mehrerer Funktionsobjekte)

parallel_reduce mit Klasse – Beispiel (1)

```
struct Count {  // Zählen der letzten Ziffer
  size_t count[10];

  Count() { fill(begin(count), end(count), size_t(0)); }  // default constructor
  Count(Count const &s, split) : Count() {}   // splitting constructor

  void operator()(blocked_range<TIterConst> const &range) {
    for (auto x : range) {
      count[x % 10]++;
    }
  }

  void join(Count const &c) {
    transform(begin(count), end(count), begin(c.count), end(c.count),
              std::plus<size_t>());
  }

};
```

parallel_reduce mit Klasse – Beispiel (2)

```
vector<int> v;
...
Count c;
parallel_reduce(blocked_range<TIterConst>(v.begin(), v.end()), c);
// Ergebnis in c.count
```

Partitionierung

- Gesamtbereich wird in kleinere Teilbereiche aufgeteilt
- Minimale Größe eines Teilbereichs (grain size) kann explizit festgelegt werden

```
// Grain size 1000
parallel_for(blocked_range<size_t>(0, n, 1000), …);
```

- Default Grain Size: 1
- Unterschiedliche Partitionierungsstrategien (Auswahl über Partitioner)

Partitioner

- Explizite Auswahl eines Partitioners, z.B.:

```
parallel_for(blocked_range<size_t>(…), f(), simple_partitioner());
```

- Verfügbare Partitioner-Klassen
  - simple_partitioner: Aufteilung, bis die Grain Size erreicht wird
  - auto_partitioner: automatische Aufteilung (Default)
  - affinity_partitioner: automatische Aufteilung unter Berücksichtigung von Cache-Lokalität

Ranges

- blocked_range: für eindimensionale Bereiche
- blocked_range2d: für zweidimensionale Bereiche

```
void run2d(blocked_range2d<size_t, size_t> const &range) {
  for (size_t i = range.rows().begin(); i != range.rows().end(); i++) {
    for (size_t j = range.cols().begin(); j != range.cols().end(); j++) {
      …
    }
  }
}
```

- blocked_range3d

parallel_do

- Erlaubt das Hinzufügen zusätzlicher Elemente während der Verarbeitung
- Auch für sequentielle Container (z.B. Listen)

```
list<Item> items;
…
parallel_do(items.begin(), items.end(), [] (Item const &item) {
  …
});
```

parallel_do_feeder

- Zum Hinzufügen zusätzlicher Elemente
- Beispiel:

```
parallel_do(items.begin(), items.end(),
  [] (Item const &item, parallel_do_feeder<Item> &feeder) {
    …
    Item newItem = …;
    feeder.add(newItem);
});
```

parallel_for_each / parallel_invoke

- parallel_for_each: wie std::for_each
- parallel_invoke: Paralleler Aufruf von mehreren Funktionen (ohne Parameter)

```
parallel_invoke(f1, f2);

parallel_invoke(
  [] { f1(1, 2); },
  [] { f2(0); }
);
```

task_group

- Erlaubt die parallele Ausführung einer beliebigen (nicht zur Compilezeit bekannten) Anzahl von Tasks

```
void RecursiveFunction(int x, int y) {
  int n = ...;
  task_group tg;
  for (int i = 0; i < n; i++) {
    tg.run([=] { RecursiveFunction(x, i); });
  }
  tg.wait();
}
```

Parallele Algorithmen (PPL)

- Header <ppl.h>, Namespace Concurrency
- parallel_sort: wie tbb::parallel_sort
- parallel_for: wie tbb::parallel_for (nur indexbasierte Iteration, keine Ranges)
- parallel_reduce: mit Iteratoren statt Range
- combinable: wie tbb::combinable
- parallel_for_each: wie std::for_each und tbb::parallel_for_each
- parallel_invoke: wie tbb::parallel_invoke
- task_group: wie tbb::task_group

Pipelines

- Verarbeitung von Daten in mehreren Stufen
- Ergebnisse einer Stufe sind die Eingaben der nächsten Stufe
- Mehrere Stufen können parallel ausgeführt werden
- Parallelisierung innerhalb einer Stufe kann unterschiedlich festgelegt werden

Pipeline – Beispiel (1)

- 1. Stufe: Lesen der Eingabedaten (seriell)
- 2. Stufe: Verarbeitung der Daten (parallel)
- 3. Stufe: Schreiben der Ausgabedaten (seriell)

Pipeline – Beispiel (2)

```
#include <tbb/pipeline.h>
…

class InputReader {  // 1. Stufe der Pipeline: Lesen der Eingabedaten
public:
  InputReader(ifstream &in) : mIn(in) {}

  string *operator()(flow_control &fc) const {
    string line;
    if (getline(mIn, line)) {
      return new string(move(line));
    }
    fc.stop();  // Verarbeitung in der Pipeline beenden
    return 0;
  }

private:
  ifstream &mIn;
};
```

Pipeline – Beispiel (3)

```
class DataProcessor {  // 2. Stufe der Pipeline: Verarbeiten der Dateien
public:
  int operator()(std::string *line) const {
    unique_ptr<string> pLine(line);
    int result = …;
    // … Ergebnis berechnen …
    return result;
  }
};


class OutputWriter {  // 3. Stufe der Pipeline: Ausgabe der Ergebnisse
public:
  void operator()(int result) const {
    cout << result << endl;
  }
};
```

Pipeline – Beispiel (4)

```
int main() {
  ifstream in("input.txt");
  // Ausführen der Pipeline
  parallel_pipeline(100,
    make_filter<void, string*>(filter::serial_in_order, InputReader(in)) &
    make_filter<string*, int>(filter::parallel, DataProcessor()) &
    make_filter<int, void>(filter::serial_in_order, OutputWriter));
  return 0;
}
```

Verarbeitungsreihenfolge

- filter::serial_in_order: serielle Verarbeitung, gleiche Reihenfolge
- filter::serial_out_of_order: serielle Verarbeitung, beliebige Reihenfolge
- filter::parallel: parallele Verarbeitung

TBB-Mutexes

- mutex: Standard-Mutex
- recursive_mutex: rekursives Mutex
- spin_mutex: nicht blockierendes Mutex („busy wait“), unfair
- queuing_mutex: nicht blockierend, fair
- Read-/Write-Mutexes: spin_rw_mutex, queuing_rw_mutex

scoped_lock

- Jede Mutex-Klasse enthält einen Typ scoped_lock
- Konstruktion mit oder ohne Mutex möglich
- Destruktor gibt Mutex frei (falls nötig)
- Methoden
  - acquire
  - try_acquire (boolescher Rückgabewert)
  - release

TBB Container

- Auf effizienten parallelen Zugriff optimiert
- Container-Typen
  - concurrent_vector
  - concurrent_queue
  - concurrent_bounded_queue
  - concurrent_hash_map

concurrent_vector

- Verwendung ähnlich wie std::vector
- Elementzugriff und Hinzufügen und von Elementen parallel möglich
- Operationen auf gesamtem Vektor (Kopieren, Zuweisung, clear) sind nicht thread-sicher
- Vektor kann intern aus mehreren Blöcken im Speicher bestehen

concurrent_vector: Methoden

- push_back, emplace_back: Anfügen eines Elements
- grow_by: Anfügen mehrerer Elemente
  - grow_by(nr): nr Default-Elemente
  - grow_by(nr, value): nr Elemente mit Wert
  - grow_by(begin, end): Elemente aus Bereich
- kein insert und erase

concurrent_queue

- FIFO-Queue
- Methoden (parallele Ausführung möglich)
  - void push(x): Element am Ende hinzufügen
  - bool try_pop(&x): Erstes Element entfernen, falls verfügbar
  - void clear(): Alle Elemente löschen
  - bool empty(): Prüfen, ob Queue leer ist
- Iteratoren nur für Debugging-Zwecke gedacht (nicht threadsicher, langsam)

concurrent_bounded_queue

- Ermöglicht die Angabe einer maximalen Kapazität (standardmäßig unbeschränkt)
- Methoden (parallele Ausführung möglich):
  - void push(x): wartet, bis Einfügen möglich
  - void pop(&x): wartet, bis Element vorhanden
  - bool try_push(x): fügt nur hinzu, falls möglich
  - bool try_pop(&x): siehe concurrent_queue
  - bool empty()
  - void set_capacity(x): Max. Kapazität setzen

concurrent_hash_map

- Hash-Tabelle, die parallele Zugriffe ermöglicht
- concurrent_hash_map<TKey, TValue, THash>
  - TKey, TValue: Key- bzw. Value-Typ
  - THash: Klasse mit Hashfunktion und Vergleichsfunktion
- Zugriff auf Einträge über „accessor“ (sperrt Eintrag in Hash-Tabelle)