Praktikum Algorithm Engineering: Multicore-Programmierung mit der (MC)STL

Organisatorisches und Einführung

Peter Sanders, Johannes Singler

Institut für Theoretische Informatik
Universität Karlsruhe

20. April 2007

Gliederung

Ziele eines Praktikums

- selbständiges Arbeiten / Arbeiten im Team
- Erlernen und üben von Sprachen und APIs, hier speziell
  - C++
  - STL
  - OpenMP
- Programmiertechniken, Debugging, SVN
- Dokumentation von Implementierung, Ergebnissen
- Beitrag zur Forschung

Ziele dieses Praktikums

- Implementierung paralleler Algorithmen
- shared memory: besonderes Augenmerk auf Multicore-Prozessoren
- STL: Integration in die C++-Standardbibliothek $\longrightarrow$ einfache Verwendung
  - Verwendung von vorhandenen MCSTL-Routinen
  - Erweiterung der MCSTL
- Algorithm Engineering: Experimente, Tuning, Leistungsfähigkeit in der praktischen Anwendung

Ablauf

- zweiwöchentliche Treffen: voraussichtlich 27.4., 4.5., 18.5., 1.6., 15.6., 29.6., 13.7.
  - Einführung MCSTL / OpenMP
  - Vorstellung von Ergebnissen der Teilnehmer
  - allgemeiner Erfahrungsaustausch
- Team-Arbeit in Gruppen von zwei (max. drei) Leuten
  - Beginn: zwei gleiche Arbeitsblätter für alle $\rightarrow$ Ü-Eier für alle besten / schnellsten Lösungen
  - danach: ein Projekt pro Gruppe, eventuell zwei konkurrierende Gruppen pro Projekt
- Dokumentation aller Implementierungen / Ergebnisse
- kurze Vorträge über Projektergebnis

Ergebnis

- Erweiterungen werden in die MCSTL-Distribution übernommen (freie Boost-Lizenz)
- Schein bei vollständiger Anwesenheit (einmal fehlen erlaubt) und bearbeitetem Projekt

Zu Grunde liegende Architektur

- primär Linux, im Prinzip Windows-kompatibel
- Programmiersprache C++
  - generische und/oder objekt-orientierte Programmierung
  - Standardbibliothek (wichtigster Teil hier STL)

Rechner-Ausstattung des Lehrstuhls

- 4-Dualcore-Opteron (`i10pc108`—`i10pc111`)
- 4x-Opteron (`i10pc103`)
- Dualcore-Core-2-Duo (`i10pc98`)
- 2x-Dualcore-Xeon (`i10pc120`)
- 2x-Quadcore-Xeon (`i10pc121`)
- eventuell 8-Core Sun T1 (`i41s3`)
- obere beide getrennter L2-Cache
- untere vier gemeinsamer L2-Cache

Mögliche Aufgaben / Projekte

- Mengen-Operationen (gemeinsame Aufgabe)
- Vektor-Operationen (`valarray`)
- Ganzzahl-Sortieren
- Suffix-Array-Konstruktion
- Maximum-Matching / Dominating Set
- Minimaler Spannbaum mittels Kruskal-Filter-Algorithmus
- hocheffizienter sequentieller/IL-paralleler Sortierer: Super Scalar Sample Sort
- Dynamisches Programmieren
- XML-Parser

Kontakt / Rechner-Arbeitsplätze

- Fragen
  - per E-Mail: `singler@ira.uka.de`
  - persönlich: Raum 221
- Rechner-Arbeitsplätze
  - 3 Rechner in Raum 205, davon ein Core 2 Duo
  - Accounts `prakt01` bis `prakt10`
  - Zugriff auf andere Multicore-Rechner mittels SSH
    `[Rechnername].iti.uni-karlsruhe.de`
  - Kompilieren:
    ```
    g++-4.2.0 -fopenmp ...
    icpc -openmp ...
    ```

Gliederung

Überblick STL 1: Funktionalität

- Standard Template Library
- gehört zur C++-Sprachdefinition nach den ISO-Standard

Generische Datenstrukturen mit Operationen

`vector<T>`, `list<T>`, `set<T>`, `map<K,V>`, `priority_queue<T>`, ...

Algorithmen

`for_each`, `sort`, `random_shuffle`, `merge`, `find`, `accumulate`, `partial_sum`, `copy`, `unique`, Mengenoperationen ...

Überblick STL 2: Wichtige Konzepte

Iteratoren

Objekte, die Positionen in einer Datenstruktur anzeigen,
Verallgemeinerung von Zeigern

```
s.begin(),s.end(),
vector<int>::iterator i = v.begin() + 5;
i[3] = 42; i++;
```

verschiedene Iterator-Konzepte

Funktoren

Klasse mit überladenem Operator,
zur Parametrisierung mit Funktionalität

```
class Functor {
void operator()(const T& e)
{ bearbeite e } };
```

Überblick OpenMP 1: Grundkonzept

- Compiler-Erweiterung für C++ und Fortran
- ermöglicht Parallelisierung von Code-Stücken
- `#pragma omp ...`
  (von unwissenden Compilern ignoriert)
- plattform-übergreifend, von allen großen Compiler inzwischen unterstützt:
  gcc, Intel, Microsoft, Sun, IBM, Spezial-Hersteller
- Fork-Join-Parallelismus

Überblick OpenMP 2: Funktionalität

- `#pragma omp parallel num_threads(nt) {}` wichtigster Grundblock
- `omp_get_num_threads()`, `omp_get_thread_num()`: zur Identifizierung
- `#pragma omp sections`: “Task-Parallelismus”
- `#pragma omp barrier`: Synchronisierung
- `#pragma omp critical`, `#pragma omp atomic`: kritische Abschnitte
- `omp_*_lock`: Ereignisse
- `shared`, `private`: Variablen-Scope, als relativ unwichtig herausgestellt

Überblick OpenMP 3: Beispiel

```
#include <omp.h>
#include <iostream>
int main()
{
  #pragma omp parallel
  {
    std::cout << omp_get_thread_num() << " von "
          << omp_get_num_threads() << std::endl;
  }
  return 0;
}
```

Überblick atomare Funktionen

- `fetch_and_add` atomare In-/Dekrementieren
- `compare_and_swap`: atomares Austauschen, (Miss-)Erfolgsrückmeldung
- darauf aufbauend:
  Lock Free Data Structures, z. B. Warteschlange

Wie geht es weiter?

- Fragen?
- Gruppeneinteilung
- OpenMP-Präsentation
- kurze MCSTL-Präsentation
- lange MCSTL-Präsentation (algorithmisch)