Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang

Abgabe: 12. Dezember 2013 in der Übung

Übung 15 Resourcen-Verwaltung (7 Punkte)

In einem System gebe es zwei identische, belegbare Ressourcen und N Bewerber für die Belegung einer der beiden Resourcen. Implementieren Sie mit den ThreadTools ein Programm, das sicherstellt, dass höchstens zwei Bewerber gleichzeitig die Resourcen belegen können, aber auch, dass jeder Bewerber irgendwann Zugriff erhält. Jeder Bewerber soll seine Resource unterschiedlich lange belegen. Verwenden Sie dazu eine zufällige Belegungszeit:

```
1 sleep(rand() % 3);
```

Für die arbeitenden Threads leiten Sie wie gewohnt eine Klasse `Consumer` vom `BasicThread` ab, welche die Methode `run` überlädt.

Verwenden Sie den bereitgestellten Kernel `resources.cc`. Diesen können Sie mit dem Befehl `make` schon übersetzen. Testen Sie Ihr Programm mit $N = 22$: Hier müssen z.B. 22 Fussballer nach dem Spiel 2 Duschen teilen ☺. Testen Sie per Ausgabe, ob tatsächlich jeder Fussballer geduscht hat.

Übung 16 ThreadTools: Travelling Salesman Problem (8 Punkte)

Ein Handlungsreisender (travelling salesman) muss Kunden in n untereinander verbundenen Städten besuchen. Dabei soll er einen möglichst kurzen Weg zurücklegen. Die einfachste Lösung ist, ausgehend von einer beliebigen Stadt systematisch alle möglichen Wege abzugehen und die „Kosten“ zu berechnen. Die (besuchten) Pfade lassen sich in den Knoten eines Baumes abgespeichern, der beste Weg kann mit einer Tiefensuche gefunden werden. Für große n ist diese Lösung nicht praktikabel, da es $(n-1)!$ mögliche Pfade gibt. Eine Verbesserung verwendet *branch-and-bound*: Hierbei wird bei der Traversion der möglichen Pfade geprüft, ob der aktuelle Pfad die Länge des bisherigen optimalen Pfades überschreitet, und eventuell abgebrochen. Ein Suchbaum mit nun unregelmäßig abgeschnittenen Ästen ist in Abbildung 0.4 dargestellt. Die unterschiedliche Pfadlänge ist einer der Hauptschwierigkeiten bei der Parallelisierung.

Das Verfahren könnte etwa mittels Rekursion implementiert werden. Günstiger ist jedoch, eine Variante mit Stacks zu verwenden:

```
1  // travelling salesman mit stack: branch-and-bound-Suche
2  path = {0}
3  stack.push(path)
4
5  while (stack.pop(path)) {
6    if (path hat die maximale Laenge n)
7      if (path ist neuer bester Pfad) best = path;
8    else
9      for (alle noch nicht besuchten Ziele i)
10       if (Laenge (path ∪ i) < bester bekannter Pfad)
11         stack.push(path ∪ i);
12 }
```

Der Stack wird mit dem Pfad, der nur die Start-Stadt enthält, initialisiert. In jeder Iteration wird der oberste Pfad entnommen. Hat der entnommene Pfad die Länge n (Anzahl der zu besuchenden Städte), so wird er bewertet, falls nicht, wird für jede noch nicht besuchte Stadt ein neuer zu bearbeitender Pfad auf den Stack gelegt, falls dieser nicht länger als der bis dahin eventuell gefundene beste Pfad ist.

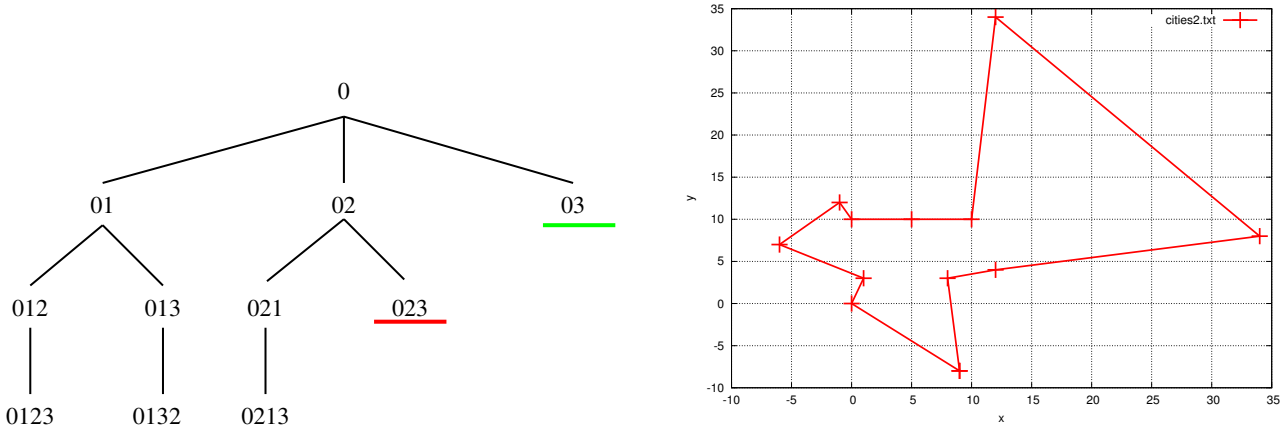

Abbildung 0.4: Links: Branch-and-bound-Suche mit unregelmäßiger Suchtiefe. Rechts: Beispielhafter optimaler Weg für 12 Städte.

Parallelisierungss-Strategie mit Threads:

Die Knoten des Suchbaumes können parallel von einem Thread bearbeitet werden, da jeder Knoten seine Nachfolger eindeutig charakterisiert. Jeder Thread erhält einen lokalen Stack mit zu bearbeitenden Knoten. Um die Arbeit zwischen den Threads zu verteilen, wird zusätzlich ein globaler Stack verwendet. Der globale Stack wird zu Beginn mit der Wurzel des Baumes initialisiert und die Arbeits-Threads mit zunächst leeren lokalen Stacks gestartet. Arbeiter mit leerem Stack versuchen, einen Knoten vom globalen Stack zu nehmen und mit branch-and-bound zu bearbeiten. Ist der globale Stack leer, warten wartende Prozesse, die Arbeit anfordern, bis arbeitende Threads einen Teil Ihres Suchbaums wieder abgeben. Der globale Stack enthält dazu einen Zähler, wieviele Prozesse warten, sowie eine Semaphore, an denen freie Threads durch eine P()-Operation warten können.

Jeder arbeitende Thread prüft während seiner branch-and-bound-Phase nach einer gewissen Anzahl `MAXI` an Knoten, die er bearbeitet hat, ob er Arbeit an freie Arbeiter abgeben kann, d.h. einen Knoten aus dem lokalen auf den globalen Stack pushen kann. Der abgegebene Knoten sollte möglichst viel Arbeit enthalten. Daher sollten die abzugebenden Knoten vom unteren Ende des lokalen Stacks entnommen werden, da die im Suchbaum unteren Knoten, für die nur noch wenig Rest-Arbeit zu leisten ist, im lokalen Stack oben liegen. Nach dem Push auf den globalen Stack weckt der abgebende Thread einen wartenden auf.

Hinweise zum Code-Gerüst

In den `threadtools` finden sie den Kernel `tsp.cc`, ein Makefile mit dem entsprechenden Target und den Ordner `tsp_cities` mit Städte-Daten. In der Datei `tsp.cc` finden Sie ein Programm-Gerüst, dass Sie zur Umsetzung dieser Aufgabe verwenden können (aber nicht müssen).

Im Gerüst finden Sie schon Klassen für einen Knoten, Z. 125, den besten bisher gefundenen Knoten, Z. 151, den lokalen (Z. 283) und den globalen Stack (Z. 190) sowie Methoden zum Einlesen (Z. 74) und Schreiben (Z. 25) einer Städte-Datei. Die Entfernungen werden in einem Doppel-`std::vector<float>` namens `dist` gespeichert, die Variable `nCities` speichert die Gesamtzahl Städte und `nThreads` die Zahl P der Threads. Ein Knoten des Baum speichert seinen Pfad (Nummern der Städte als `char`-Feld), die Länge des Pfades (Anzahl der Städte) und die bisher angefallenen Kosten (Entfernung der Punkte in der euklidischen Norm). Die Klasse `BestNode`, Z. 151, für den besten bisher gefundenen Knoten ist von der Knoten-Klasse abgeleitet und enthält zusätzlich eine Funktion zum Update des besten Knoten, der Zugriff wird durch einen Mutex abgesichert.

Der globale Stack enthält eine Mutex-Variable `mutex`, Z. 270, um exklusiven Zugriff sicherzustellen, sowie eine Semaphore `waitSem`, Z. 272, die zum Warten und für das Aufwecken arbeitsloser Threads verwendet wird. Durch den Aufruf von `wait()` (globaler Stack, Z. 221) erhöht ein Thread die Anzahl der wartenden Arbeiter an der Semphore. Außerdem gibt diese Methode die Anzahl der wartenden

Threads zurück (oder -1, falls keiner mehr wartet, in diesem Fall wird auch das Flag zur Einleitung der Terminierungsphase gesetzt). Die Methode `sharedSignal` (Z. 210) ist das Gegenstück, sie dekrementiert die Zahl der wartenden Prozesse und weckt einen anderen Thread. Die Klasse für den lokalen Stack enthält zwei Funktionen `popRear` und `peekRear`, (Z. 309 bzw. 318), die Zugriff auf den bottom of stack bieten, um die unteren Elemente des lokalen Stacks auf den globalen Stack schieben zu können.

Die Arbeiterklasse `Worker`, Z. 344, realisiert einen arbeitenden Thread. Sie ist von `TT::BasicThread` abgeleitet und überlädt daher die Methode `run`. Die Methode `branchAndBound` soll die oben abgebildete Suche realisieren. Eine wichtige Hilfsmethode ist `expandNode` (Z. 437), die für einen gegeben Knoten die Nachfolger im Baum bestimmt und auf dem lokalen Stack stapelt.

Aufgabe

Implementieren Sie die `run`-Methode der Arbeits-Threads und die Methode für das `branchAndBound` der `Worker`, wie in der Parallelisierungs-Strategie oben beschrieben:

- Die `run`-Methode versucht in einer Endlos-Schleife, Arbeit vom globalen Stapel zu bekommen. Kann Sie einen Knoten vom globalen Stack holen, legt sie in auf dem lokalen Stack ab und startet das branch-and-bound. Falls sie keinen Knoten vom globalen Stack bekommt, soll mittels der `wait`-Funktion des globalen Stack getestet werden, ob bereits $P-1$ Threads warten (Rückgabewert -1). In diesem Fall darf der letzte Thread nicht blockieren, sondern muss die anderen aufwecken (Methode `sharedSignal`). Wird ein Thread geweckt, d. h. er kehrt aus `wait` zurück, muss er nochmals mit `terminate()` prüfen, ob die Terminierungsphase eingeleitet ist: Dann darf er terminieren, ansonsten versucht er wieder, Arbeit zu erhalten.
- In der Methode `branchAndBound` wird solange gearbeitet, bis der lokale Stapel leer ist. Alle `MAXC` Iterationen wird geprüft, ob Arbeit abgegeben und auf den globalen Stack gepusht werden kann. Ein Knoten soll nur abgegeben werden, wenn die restliche Baumgröße noch größer als `MIND` ist. Der abzugebende Knoten soll vom bottom of stack des lokalen Stapels entnommen werden. Hierzu gibt es die Methoden `popRear` und `peekRear` (lesender Zugriff) in der Klasse des lokalen Stacks. Wurde ein Knoten an den globalen Stack abgegeben, müssen wartende Threads aufgeweckt werden.

Ihr Programm können Sie über den Aufruf von `make` kompilieren. Kompilieren Sie am Besten mit der Optimierung `-O3`. Die erste Städte-Datei `cities1.txt` enthält 5 Städte auf einer Linie und ist dafür gedacht, Ihr Programm zu verifizieren. Messen Sie nun die Programm-Laufzeiten (*User-time* in s) für $1, 2, 4$ und 8 Threads und für die vier verbleibenden Dateien `cities[2-5].txt` und berechnen Sie den erreichten Speed-Up. Wenn Sie die Möglichkeit haben, messen Sie die Zeiten auch auf Rechnern, die mehr als zwei sinnvolle Threads wie im Pool erlauben. Zur Zeitmessung können Sie den Befehl `time` in der Konsole verwenden:

```
1 /usr/bin/time -f %e ./tsp ./tsp_cities/cities4.txt
```

misst die elapsed User-time in Sekunden (Schalten Sie dazu den Output durch Setzen der Variable `output=false` ab). Diskutieren Sie die erreichten Speed-Ups, erstellen Sie wie bisher Plots zur besseren Übersicht, falls nötig. Wenn Sie dem Programm übrigens als weiteren Parameter einen Datei-Namen übergeben, wird eine Output-Datei mit dem optimalen Pfad erstellt, die Sie mit Gnuplot über den Befehl `plot '<datei.dat>'w lp` für einen Plot wie in Abbildung 0.4 rechts plotten können. Geben sie einen Plot mit dem optimalen Pfad für die Datei `cities5.txt` mit ab, um zu zeigen, welches (lokale) Optimum ihre Implementierung findet!