Preview

- Überschriften
  - Threads und Prozesse
  - Synchronisationsmechanismen
  - openMP
  - Implizite Parallelisierung in Java
  - MPI
  - Literatur
- Codebeispiele
  - Augenscheinlich ausschließlich in Java
- Tabellen
  - Beschreiben unter anderem Abläufe von Programmen
  - Vergleich verschiedener Programmiersprachen
- Durch Umrandung hervorgehobener Text

Question

- Welche Problemstellungen werden durch paralleles Programmieren besser gelöst?
- Welche Nachteile hat paralleles Programmieren im Gegensatz zum sequentiellen Programmieren?
- Wie löst man Konsistenzprobleme bei parallelen Zugriffen?
- Welche Implementierungen existieren bereits?
- Welche Geschwindigkeitsvorteile sind zu erwarten?
- Was sind parallele Algorithmen bzw. was gibt es bei denen zu beachten?
- Welche Ergänzungen gibt es im Vergleich zum Text aus Exzerpt 1?

Read

- http://www.rz.rwth-aachen.de/mata/downloads/paralleles_programmieren/ppj_skript.pdf

Reflect

Der vorliegende Text ergänzt den Text aus dem 1. Exzerpt um detailliertere Informationen über Threads, Problematik und mögliche Lösungen mit derer Programmierung. Andere Aspekte wie Geschwindigkeitsvorteile werden kaum behandelt.

Recite

Eine Möglichkeit Parallele Programmierung auf Prozessebene zu erreichen sind Threads. Threads teilen sich einen Prozesskontext und damit globale Variablen. Die größten Probleme bei Threads sind Race Conditions, bei denen das Ergebnis von der Reihenfolge der Threadzugriffe abhängt, und Deadlocks, bei denen mehrere Threads Ressourcen blockieren, die der jeweils andere braucht und sich damit gegenseitig theoritisch unendlich lange blockieren. Ein Mutex verhindert, dass mehrere Threads gleichzeitig auf einen Codeblock zugreifen können, evtl. müssen andere Threads warten, bis der Block vom besitzenden Thread wieder freigebeben wird. Eine Semaphore funktioniert ähnlich wie ein Mutex, nur dass eine definierte Anzahl von Threads auf eine Ressource zugreifen können, bevor diese für neue Threads blockiert wird. Eine Grundregel bei paralleler Programmierung besagt, dass immer nur ein Thread für Ein- und Ausgabe zuständig sein sollte. Ein Monitor bildet die Basis für mehrere andere Synchronisationsverfahren. Dabei wird vom gegenwärtigen Besitzer eines Codeblocks ein Signal an wartende Threads geschickt, bevor der blockierte Berreich wieder verlassen wird. Danach ist dieser Bereich wieder für andere Threads zugänglich. Barriere, Future und Pipe bauen auf diesme

System auf. Eine Barriere blockiert einen Thread so lange, bis alle anderen Threads an einem bestimmten Punkt im Code (Der Barriere) angelant sind. Eine Pipe bildet eine Möglichkeit, Daten zwischen Threads auszutauschen. Ein Worker-Thread legt Daten in die Pipe, die ein Consumer-Thread dann ausliest. Futures dienen als Platzhalter für noch nicht abgeschlossene Berechnungen. OpenMP ist ein Framework, Code zur Comilezeit zu parallelisieren. Dabei bedient man sich unter anderem der Annahme, dass die meiste Arbeit in Programmen in for-Schleifen geschieht. MPI ist eine Interface-Spezifikation, die es ermöglicht parallele Ausführung auf mehreren Rechnern auszuführen.

Review

Der Text macht an Beispielen die Probleme, die beim parallelen Programmieren auftreten können, relativ anschaulich. Negativ viel eine in Großbuchstaben geschriebene Notiz auf, dass ein Kapitel noch zu ergänzen sei. Die openMP und MPI Kapitel sind relativ oberflächlich gehalten und eher als Einführung in die Konzepte beider Technologien zu verstehen, da eine gründliche Behandlung dieser Themen sowohl Umfang als auch Intention des Textes sprengen würde. Die Sprache ist neutral gehalten, es werden Grundkonzepte bei der Progammierung mit Threads vermittelt. Die Kapitel über Probleme bei der GUI Programmierung sind sehr Java-lastig gehalten, wodurch die Allgemeingültigkeit der Aussagen etwas leidet. Für den interessierten Leser findet sich am Ende des Textes ein Literaturverzeichnis.