

Aufgabenblatt 6

Musterlösung

Vorlesung Verteilte Systeme

Wintersemester 2025/26

Aufgabe 1: Programmierung: JNDI

- a) JNDI (Java Naming and Directory Interface) stellt Methoden mit Standard-Verzeichnisoperationen bereit (lesen, schreiben, löschen, hinzufügen von Attributen zu Objekten und die Suche nach Objekten auf Basis dieser Attribute). Darüber hinaus definiert es eine einheitliche Schnittstelle für den Zugriff auf verschiedene Verzeichnisdienste.
- Java Applikationen bieten mithilfe von JNDI zwei Möglichkeiten mit Namens und Verzeichnisdiensten zu Arbeiten. Einmal auf traditionellem Wege - traditionell bedeutet in diesem Zusammenhang auf tatsächlich existierende Verzeichnisdienste (Dateisystem, Mail, etc.) zuzugreifen. Die zweite Möglichkeit besteht darin, das Directory als einen Speicherort und eine Organisationsform für die Ablage von Objekten zu verwenden.
- b) Die Lösung dieser Aufgabe finden Sie im Archiv [106Files.zip](#)¹ auf der Vorlesungswebseite.
- c) Die Lösung dieser Aufgabe finden Sie im Archiv [106Files.zip](#)² auf der Vorlesungswebseite.

Aufgabe 2: Lastverteilungsstrategien

- a) statische Lastverteilung: vor der Ausführung wird ein Ablaufplan erstellt; Nachteil: alle Prozesse und deren Laufzeiten müssen a-priori bekannt sein, Dynamik der Prozesse und Fremdlast können nicht berücksichtigt werden.
- dynamische Lastverteilung: Prozesse werden (erst) bei ihrer Erzeugung auf einen Knoten plaziert und bleiben dort; Nachteil: spätere Änderung der Lastsituation (durch Dynamik der Prozesse oder Fremdlast) bleibt unberücksichtigt.
- dynamisch-präemptive Lastverteilung: Prozesse können während ihrer Laufzeit (auch mehrfach) den Knoten wechseln; Nachteile: Migrations-Problematik, Thrashing möglich; Vorteil: sehr adaptiv, vor allem bei Fremdlast.
- b) Nicht alle rechenbereiten Prozesse verursachen gleich viel Last (z.B. Prozesse, die häufig auf I/O warten). Ein besseres Lastmaß könnte die CPU-Auslastung sein (also wieviel % der Zeit rechnet die CPU), die aber nicht immer einfach zu ermitteln ist und auch stark schwanken kann. Auch Speicherbedarf, Seitenfehlerraten, E/A-Raten, Kommunikationsraten etc. könnten in Lastmetriken mit einfließen. Die (häufige) Ermittlung dieser Metriken bedeutet aber zusätzlichen Overhead, der sich durch die „bessere“ Metrik nicht immer amortisiert. In Experimenten hat sich herausgestellt, daß (aufgrund dieses Overheads) durch solche Metriken i.a. auch kein besserer Lastausgleich erzielt wird (d.h. die Laufzeit eines Programmsystems nicht reduziert wird).

Aufgabe 3: Graphpartitionierung und List-Scheduling

Lastverteilung durch Graph-Partitionierung:

Gegeben ist eine Anzahl von nebenläufigen, kommunizierenden Prozessen und die Kommunikationslast zwischen je zwei Prozessen. Dies wird als Graph dargestellt. Die Knoten stellen die Prozesse dar und die Kanten zeigen die Kommunikationlast. Ziel: die auftretende Last möglichst gleichmäßig zu verteilen, um Überlast auf einzelnen Knoten zu verhindern. Der Graph wird so partitioniert, daß die Gewichte-Summe der geschnittenen Kanten minimal ist, d.h. die Kommunikation zwischen Knoten minimiert wird. Alle Lastverteilungsstrategien gründen ihre Entscheidungen auf irgendeine *Lastmetrik*.

¹<http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/vs/106Files.zip>

²<http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/vs/106Files.zip>

List-Scheduling:

Das Programm bestehend aus mehreren abhängigen Tasks, die ggf. auf Ausgabedaten ihrer Vorgängertasks arbeiten, aber *während* der Abarbeitung nicht kommunizieren. Dies wird als DAG dargestellt. Die Knoten beinhalten Tasks mit Ausführungszeiten und die Kanten zeigen die notwendige Kommunikation mit Übertragungsdauer. Ziel: Scheduling der Tasks so, daß die Bearbeitungszeit (engl. *makespan*) minimiert wird.

Gemeinsamkeiten:

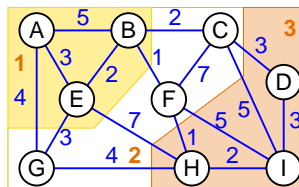
Graphen mit Tasks als Knoten und Kommunikation als Kanten, Kantengewichte, Taskstruktur ist fest, Anzahl von CPUs ist vorgegeben, Schedule (Zuordnung von Prozessen zu CPUs) wird vor Programmstart berechnet, Ziel i.d.R. Minimierung der Laufzeit, ...

Unterschiede:

Programmstruktur (bei Graph-Partitionierung: nebenläufige Prozesse, die während ihrer Ausführung kommunizieren; bei List-Scheduling: Tasks, die ggf. Ausgabedaten anderer Tasks benötigen, aber *während* der Ausführung nicht kommunizieren), Bedeutung der Knotengewichte, ungerichteter vs. gerichteter Graph, ...

Aufgabe 4: Lastverteilung durch Graphpartitionierung

Die neue Aufteilung sieht wie folgt aus:



In dieser Aufteilung hat Knoten 1 die Prozesse A, B und E, Knoten 2 hat die Prozesse C, F, und G, und Knoten 3 hat die Prozesse D, H und I. Der Schnitt zwischen Knoten 1 und 2 enthält die Kanten AG, EG, BF und BC mit einer Kommunikationslast von $4+3+1+2 = 10$. Der Schnitt zwischen Knoten 2 und 3 enthält die Kanten CD, CI, FI, FH und GH mit einer Kommunikationslast von $3+5+5+1+4 = 18$. Der Schnitt zwischen Knoten 1 und 3 enthält die Kante EH mit einer Kommunikationslast von 7. Die Summe ist $10 + 18 + 7 = 35$