Kommentare zu Teil 4

Slide 2

Summarische Resultate wie z. B. das Maximum einer Menge von Werten können effizient durch sukzessive Reduktion auf zwei Hälften berechnet werden (divide et impera), wobei die entsprechenden summarischen Werte zuerst in den beiden Hälften parallel ermittelt und danach die beiden Teilresultate zum Gesamtresultat reduziert werden.

Slide 3

Zeigt ein entsprechendes C# Programm unter Verwendung eines parallelen Aufrufs sowie eines cutoffs.

Slide 4

Wieder einmal lässt sich der cutoff im Programm nicht vermeiden, wenn man optimale Effizienz will.

Slide 5

Die Reduktionsmethode funktioniert für alle assoziativen Operationen wie z. B. max (a, b, c, d) = max(max(a, b), max(c, d)).

Slide 6

Falls dem Reduktionsschritt ein Abbildungsschritt vorgeschaltet ist, so erhält man das Map-Reduce Berechnungsmuster. Beispiel: Mittelwert einer Reihe von Funktionswerten.

Slide 7

Map-Reduce wurde von Google als archetypisches parallelisierbares Berechnungsmuster kultiviert, propagiert und implementiert. Es basiert auf zwei durch die Benutzer gelieferten Methoden map und reduce, welche in die Map-Reduce Maschinerie „eingesteckt“ werden können. Grundsätzlich liefert map eine Liste von Paaren (key1, value1), welche dann durch die Maschinerie in eine Liste von Paaren (key2, value2) übersetzt wird. Mit reduce kann schliesslich die zu einem Schlüssel key2 gehörende Liste von v2-Werten beliebig reduziert werden.

Slide 8

Archetypisches Beispiel einer Map-Reduce Anwendung ist die Wortzählung in einer Menge von Dokumenten, wobei key1 = Dokumenten-Id, value1 = (Text-)Inhalt, key2 = Wort, value2 = Position des Wortes im Text ist. Die Umsetzung von key1 auf k2 setzt eine Umgruppierung der Listen durch die Map-Reduce Maschinerie (hinter den Kulissen) voraus.

Slide 9

C# Skizzen der beiden Plug-Ins map und reduce für das Wortzählbeispiel.

Slide 10

Map-Reduce ist ebenfalls geeignet für die Erstellung von Internet-Zugriffsstatistiken sowie für Indexinvertierungen (Uebergang von einer Indexordnung auf eine andere).

Slide 11

Das Slide zeigt dass und wie das Map-Reduce Schema in verteilten Systemen („Cloud“) verwendet werden kann. Dies ist wichtig, weil in der Cloud potenziell Hunderte oder Tausende von Prozessoren (Maschinen) zur Verfügung stehen.

Slide 13

Bisher wurden hauptsächlich Abläufe (Prozesse) diskutiert, welche in disjunkte, gegenseitig unabhängie Teilprozesse (Threads) zerlegbar sind. Synchronisation war höchstens für die Termination in Form einer „Barriere“ nötig. Im nächsten Abschnitt betrachten wir nun Prozesse, deren Threads auf (nicht immer disjunkten) Daten im gemeinsamen Speicher operieren. Wir unterscheiden Lese- und Schreibzugriffe und zwischen Schreibzugriffen auf gemeinsame sowie auf disjunkte Datenbereiche.

Slide 14

Ein zentrales Problem bei (fast) gleichzeitiger Verwendung gemeinsamer Daten durch verschiedene Threads sind Race-Conditions. Obwohl wir annehmen können, dass einzelne Schreib- und Leseanforderungen vom Memory-Controller bzw. vom Bus-Schiedsrichter sauber synchronisiert (d. h. sequenzialisiert) werden, können zwei oder mehrere Threads trotzdem ungünstig miteinander interferieren, wie das Beispiel einer Inkrementoperation auf die Variable i zeigt. Meist geht alles gut, aber in (sehr) seltenen Fällen ergibt sich ein falsches Resultat. Das Problem rührt davon her, dass die Inkrementoperation nicht atomar ist. Sie setzt sich bei mikroskopischer Betrachtung aus einer Ladeoperation, einer Addition und einer Zurückschreiboperation zusammen.

Slide 15

Für den garantiert korrekten Ablauf (nicht nur mit hoher Wahrscheinlichkeit) wird Atomizität („Isolation“) benötigt. Je nach Granularität der gewünschten Operation geschieht dies automatisch (im Falle einzelner Reads und Writes), oder es müssen Vorkehrungen getroffen werden. Einzelne „Read-Inspect-Write“ Zyklen können durch Verwendung atomarer Hardwareinstruktionen wie Test-and-Set oder Compare-and-Swap (Genaueres später) atomisiert werden. Dafür steht in .net die Interlocked Klasse zur Verfügung (s. Slide 16). Für die Atomisierung ganzer Codebereiche werden „höhere“ Software-Konstrukte benötigt (s. späteres Kapitel).

Slide 16

Zwei wichtige Interlocked Operationen: Atomares Inkrementieren (und analog Dekrementieren) und bedingter atomarer Austausch von Werten.

Slide 17

Als erstes Beispiel betrachten wir die Parallelisierung des Acht-Damen-Problems. Dies ist ein klassisches Algorithmenproblem, in welchem es darum geht, Stellungen von acht (oder mehr) Schachdamen zu finden, in welchen sich keine zwei Damen bedrohen können. (Damen können sich auf dem Schachbrett horizontal, vertikal oder diagonal bewegen).

Slide 18

Es wird ein Branch-and-Bound Verfahren verwendet, welches einen symbolischen Lösungsbaum rekursiv traversiert und erschöpfend durchsucht. Die beiden Diagonalrichtungen zeichnen sich dadurch aus, dass die Summe bzw. die Differenz der x- und y-Koordinaten der Damen auf dem Schachbrett einen gemeinsamen Wert besitzen.

Slide 19

Zeigt ein sequenzielles C#-Programm zur Lösung des Acht-Damenproblems. Der Datentyp QueensRequest kapselt einen separaten Platzierungsversuch aller acht Damen ein. Die for-Schleife läuft durch alle Platzierungsmöglichkeiten (in vertikaler Richtung) der Dame Nr. n. In der if-Anweisung wird geprüft ob die entsprechende Zeile sowie beide Diagonalen für die vorgesehene Platzierung an Position i frei sind. Falls ja, wird die Platzierung vorgenommen, und die Rekursion geht zur nächsten Position (falls das Ende noch nicht erreicht ist). Falls das Ende erreicht ist, wurde eine Lösung gefunden, und der (globale, gemeinsame) Lösungszähler wird um 1 erhöht. Danach wird die Platzierung wieder rückgängig gemacht, sodass die nächste Möglichkeit ausprobiert werden kann.

Slide 20

Parallelisierung der Platzierungsbemühungen in Form einer lokalen Task-Queue. Für jeden Platzierungsversuch in der nächsten Spalte wird eine Task erzeugt, welche den bisherigen Platzierungszustand reflektiert. Für die Erhöhung des Lösungszählers wird ein atomares Inkrementieren verwendet und für die Terminationssynchronisierung eine Task-Barriere.

Slide 21

Enttäuschendes Resultat: die parallele Version läuft trotz vier Hypercores (viel) langsamer als die sequenzielle. Grund: die einzelnen Tasks habne zuwenig Arbeit zu leisten, der Verwaltungs-Overhead ist grösser als der Gewinn.

Slide 22

Das Bild ändert sich allerdings bei grösseren Schachbrettern klar zu Gunsten der parallelen Version.

Slide 23

Neben der Parallelisierung einer sequenziellen Lösung zur Effizienzsteigerung darf natürlich die Möglichkeit der Optimierung des Algorithmus selbst nicht vergessen werden. Im Falle des Acht-Damenproblems lässt sich der Algorithmus durch Beschränkung auf Permutation von allem Anfang an optimieren.

Slide 24

Zeigt was die Optimierung sowie die Parallelisierung bzw. die Kombination der beiden bringt. Aber Achtung: Optimierungen können die Parallelisierbarkeit einschränken und dadurch die Skalierbarkeit im Mehrprozessorraum beeinträchtigen, s. früheres Beispiel des Schachprogrammes.

Slide 25

Ein noch grösserer Slowdown bei Parallelisierung der optimierten Version in der ursprünglichen Problemgrösse

Slide 26

Das Rucksackproblem ist eine weiteres Branch-and-Bound Problem, welches (zumindest auf den ersten Blick) gut parallelisierbar ist. Es geht dabei um das Finden einer optimalen d. h. möglichst wertvollen Packung eines Rucksacks mit gegebenen Gegenständen, wobei eine gegebene Gewichtslimite nicht überschritten werden darf. Die naheliegende Lösung (Nehmen des wertvollsten Gegenstandes bzw. des Gegenstandes mit höchstem spezifischen Wert zuerst funktioniert nicht).

Slide 27

Wieder steht im Hintergrund die Traversierung eines Lösungsbaumes, wobei Aeste abgeschnitten werden können, falls entweder die Gewichtsllimite überschritten würde oder der bisher höchste Wert nicht mehr erreicht werden könnte.

Slide 28

Vorgeschlagene Datenstruktur mit Klassen ItemSet und Cand (Kandidaten).

Slide 29

Die gegebenen Gegenstände werden als Paare von Konstanten (Gewicht, Wert) in der Umgebung spezifiziert.

Slide 30

Der sequenzielle Programmkern mit den beiden „Aesten“ Gegenstand i nehmen (falls die Gewichtslimite dies zulässt) oder Nichtnehmen (falls dann der bisher höchste Wert noch erreichbar wäre).

Slide 31

Die Interaktion mit der Umgebung beschränkt sich auf die (harmlose) Verwendung von gemeinsamen Konstanten sowie der (nicht harmlosen) Verwendung der globalen Variablen maxCand, welche die bisher beste gefundene Lösung speichert. Ebenso beachtet werden muss, dass die aktuelle Konstellation in Form einer Referenz s rekursiv weitergereicht wird.

Slide 32

Zeigt nochmals die sequenzielle Lösung, in welcher diesmal die Interaktion mit der Umgebung markiert ist.

Slide 33

Folgende Paralelisierungs-Hindernisse infolge der Interaktion mit der Umgebung sind zu beachten:

1) Weiterreichen der Referenz s führt zum Aliasing und macht das zugehörige Objekt zur gemeinsamen Variablen.
2) Mögliche Verwendung eines veralteten Wertes (stale value) beim Zugriff auf maxCand führt ev. zu unnötigem Mehraufwand aber nicht zu Fehlern.
3) Infolge Race Condition kann ein besseres Resultat durch ein schlechteres überschrieben werden, d.h. Fehlerpotenzial (wie?)

Slide 34

Lösungsmöglichkeiten:

1) Cloning vermeidet (wiederum) Aliasing
2) Verwendung von Interlocked CompareExchange für das Registrieren eines neuen Optimums
3) Alternativ verzögerte Maximumsbestimmung durch Einführung einer privaten Kandidatenmenge

Slide 35

Verwendung von Interlocked.CompareExchange für die Registrierung eines neuen Maximums. Benötigt eine while-Schleife!

Slides 36 - 38

Lösung mit verzögerter Maximumsbestimmung. Verwaltung privater Kandidatenmengen (je eine für den Einschlusspfad un den Ausschlusspfad) und synchrone Vereinigungsbildung. Suchen des wirklichenMaximums am Schluss.

Slide 39

Wiederum: trügerischer Parallelisierungsvorteil bei „kleiner“ Problemgrösse! Parallelisierung zahlt sich erst bei 30 – 40 Gegenständen aus. Die Variante mit privaten Kandidatenmengen ist vor allem bei kleiner Problemgrösse der atomaren Austauschlösung überlegen, da der Cache besser genutzt wird und keine „Staus“ entstehen.

Slide 40

Zeigt das Branch-and-Bound Parallelisierungsmuster. Es entspricht der Breitensuche vor dem cutoff und der Tiefensuche danach. Aber Achtung: Trotz „natürlichem“ Parallelisierungspotenzial mit Vorsicht zu benutzen. Overhead beachten. Erst bei „grossen“ Problemgrössen verwenden. Stets cutoff einbauen!

Slide 41

Race Conditions sind nicht das einzige Problem bei der Benutzung von gemeinsamem Speicher durch mehrere Threads. Neben dem „Flaschenhals“ Memory Controller sind die sogenannte Cache-Kohärenz sowie das False-Sharing effizienzvernichtende Artefakte.

Slide 42

Cache Kohärenz ist eine Voraussetzung für Korrektheit. Sie bedeutet, dass alle Caches, welche denselben Speicherauszug darstellen, stets den gleichen Inhalt anzeigen. Bei jeder Schreibeoperation muss die Kohärenz wieder hergestellt werden oder es müssen die „unsauberen“ (dirty) Caches invalidiert werden. Dies geschieht durch die Hardware automatisch via sogenanntes Cache-Kohärenzprotokoll. Es ist transparent für die Programmierung, beeinträchtigt aber die Effizienz.

Slide 43

False-Sharing entsteht dann, wenn zwei an sich disjunkte Speicherbereiche dieselbe Cacheline anschneiden. Dann muss diese Cacheline in beiden Caches enthalten sein und ist somit automatisch dem (effizienzreduzierenden) Cachekohärenzmechanismus unterworfen, obwohl dies sachlich nicht nötig wäre.