Komplexitätsanalyse paralleler Algorithmen

Markus Wanke

Seminarausarbeitung 06.2011

Inhaltsverzeichnis

0 Vorwort

Gerade in den letzten Jahren haben sich Multiprozessorsysteme auf dem Desktopmarkt sehr stark etabliert und selbst in mobilen Devices wie Handys und Tablets finden sich ebenfalls schon Systeme mit mehr als einem Prozessor. Erfahrungen mit parallelem Algorithmendesign, das früher ausschließlich für Entwickler an Supercomputern wichtig war, sind nun für alle Programmierer von Interesse.
Dass die parallele Programmierung viele Schwierigkeiten mit sich bringt ist allgemein bekannt. Auf Probleme wie Synchronisation, Deadlocks, Race Conditions, etc. stößt man unweigerlich, aber die Tatsache, dass aus schlechtem Algorithmendesign großer Parallelitätsverlust entstehen kann, fristet eher noch ein Schattendasein.
Einen Einblick in Mittel zur Analyse und Lösung solcher Problematiken soll hier gegeben werden.

1 Notationskonvention

Um parallele Abläufe in den Pseudocodebeispielen zu ermöglichen wird dieser um die SPAWN und SYNC Statements erweitert.

```
 1 fibonacci( n )
 2
 3     if n <= 1
 4         return n
 5
 6     spawn x = fibonacci( n - 1 )
 7     y = fibonacci( n - 2 )
 8     sync
 9
10     return x + y
11 end
```

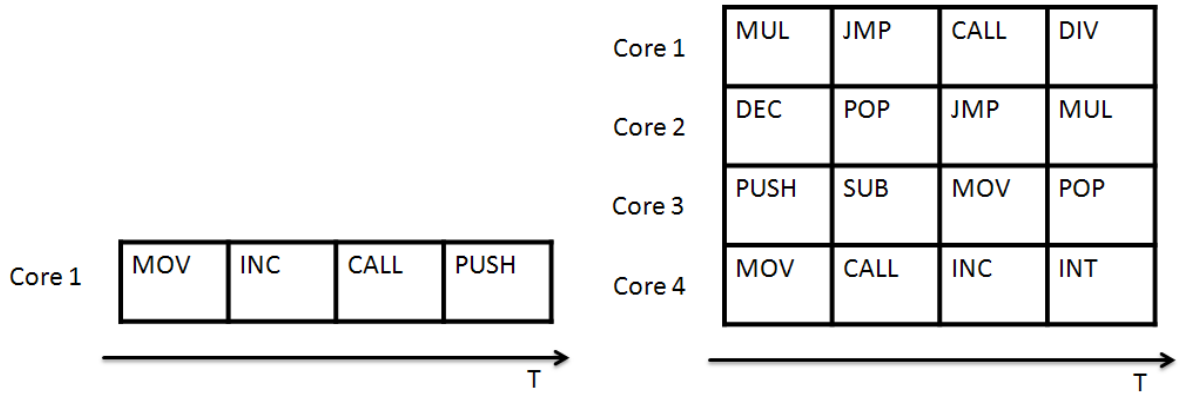
SPAWN führt einen Funktionsaufruf Parallel zum aktuellen Programmverlauf aus. Die Funktion wird aufgerufen, aber es wird nicht darauf gewartet, dass sie ihre Arbeit abschließt, sondern es wird sofort nach dem Aufruf im aktuellen Programmfluss weiter gemacht.

SYNC hält den Programmfluss solange an, bis alle mit SPAWN ausgelagerten Funktionen wiederkehren.

2 Theoretische Grundlagen

2.1 Arbeitsgesetz

Von der Vorstellung, dass unterschiedliche Maschinenbefehle unterschiedlich viel Zeit benötigen, wird abstrahiert und jeder Maschinenbefehl wird als konstante Operation angesehen.
Bei dem Vergleich zweier Computer, von denen einer Maschinenbefehle nur seriell verarbeiten kann, wo hingegen der andere die Möglichkeit hat vier Maschinenbefehle zur gleichen Zeit auszuführen, erkennt man, dass die mögliche Geschwindigkeitszunahme begrenzt ist.



In einem Takt kann das Mehrprozessorsystem höchstens so viele Operationen durchführen, wie Prozessoren/Cores vorhanden sind. Somit begrenzt sich die Geschwindigkeitszunahme auf einen Koeffizienten. Daraus lässt sich das Arbeitsgesetzt ableiten:

$$T_P \geq \frac{T_1}{P}$$

T_1: Zeit die ein Algorithmus auf einem Prozessor benötigt
T_P: Zeit die ein Algorithmus auf *P* Prozessoren benötigt
P : Anzahl der Prozessoren

Die seriell benötigte Zeit, geteilt durch die Anzahl der Prozessoren kann zwar theoretisch gleich sein, wird aber meistens doch unter dem wirklich auf parallele Weise erreichten Wert liegen. Einmal sicher durch Overhead wie Threaderzeugung, Scheduling, etc., aber auch durch falsches Algorithmendesign wie sich später zeigen wird.

Das Arbeitsgesetz nochmal salopp formuliert: Mit P Prozessoren kann ein Problem allerhöchstens P mal so schnell gelöst werden, aber auf keinen Fall schneller.

2.2 Beschleunigung

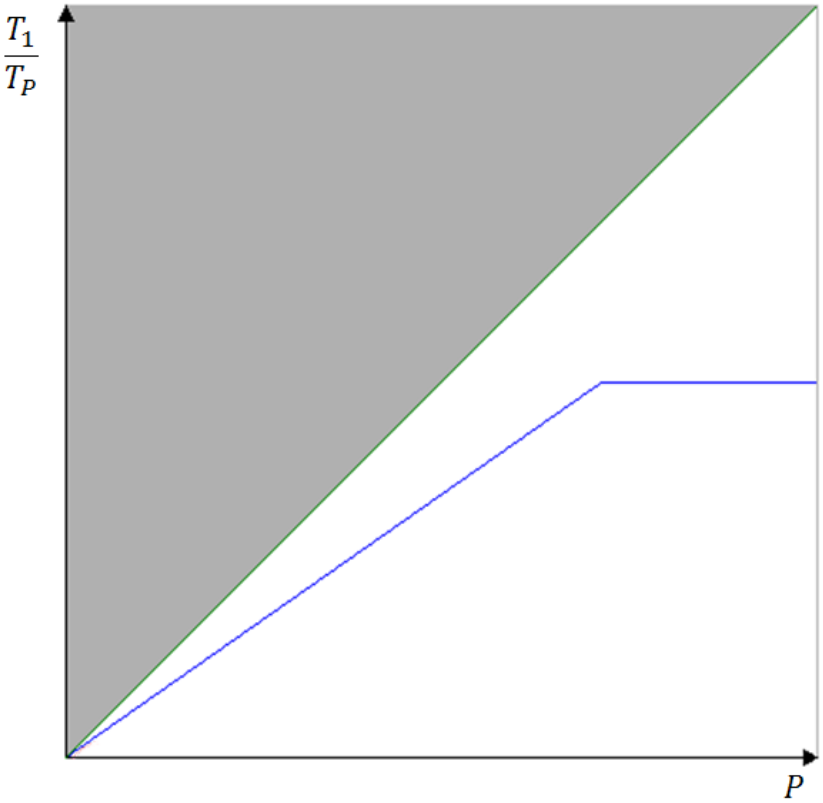
Setzt man die seriell und parallel benötigte Zeit in ein Verhältnis, ergibt dies die Beschleunigung.

$$\text{Beschleunigung} = \frac{T_1}{T_P}$$

Die Beschleunigung ergibt nun den tatsächlichen Faktor um den ein Problem schneller gelöst worden ist, aufgrund von Overhead wird dies meistens unter der Anzahl der Prozessoren liegen. Wenn die Beschleunigung gleich der Anzahl der Prozessoren ist, wird dies „perfekte lineare Beschleunigung“ genannt. Laut Arbeitsgesetz kann die Beschleunigung niemals größer werden, als die Anzahl der Prozessoren.

2.3 Verhalten der Beschleunigung bei steigender Anzahl der Prozessoren

Entscheidend für die Aussage wie gut sich ein Algorithmus parallelisieren lässt, ist das Verhalten der Beschleunigung bei steigender Anzahl der Prozessoren.



Grau: Laut Arbeitsgesetz kann dieser Bereich nicht erreicht werden.
Grün: Perfekte lineare Beschleunigung
Blau: Zu erwartendes Verhalten

Es ist zu erwarten, dass die Beschleunigung bei zunehmender Anzahl der Prozessoren unter der perfekten linearen Beschleunig liegt, bedingt durch Overhead der Threadverwaltung. Darüber hinaus wird die Beschleunigung ab einem bestimmten Punkt in eine Konstante übergehen, weil ein Zustand erreicht ist, an dem mehr Prozessoren nicht zur weiteren Beschleunigung der Problemlösung beitragen können. Dieser Punkt entscheidet wie gut sich ein Algorithmus parallelisieren lässt, umso später er eintritt, desto besser. Die Bestimmung dieses Punktes ist mittels Parallelitätsanalyse möglich.

3 Parallelität

3.1 Parallelität von Algorithmen klassifizieren

Um Algorithmen zu klassifizieren, ordnen wir den Verbrauch von Zeit und Speicherplatz in Komplexitätsklassen ein. Diese beiden Kategorien werden jetzt um den Begriff der Parallelität erweitert**,** der wie zu erwarten eine Aussage über die Parallelisierbarkeit eines Algorithmus trifft. Und genau wie Zeit und Speicher, kann auch die parallelität mittels Komplexitätsklassen klassifiziert werden.
Die Parallelität ist abhängig von der Problemgröße N. Ist für einen Algorithmus die Parallelitätsfunktion bestimmt, kann die Problemgröße eingesetzt werden, um einen Grenzwert für die maximal mögliche Beschleunigung zu erhalten. Und das entspricht der maximalen Anzahl an Prozessoren die zur Lösung benutzt werden können. Alle weiteren Prozessoren, die eventuell darüber hinaus auch noch zur Verfügung stehen, würden gar nicht erst benutzt oder die Threads die auf ihnen laufen, wären ausschließlich mit dem Warten auf Teilergebnisse beschäftig, sodass für diesen Thread ein eigener Prozessor gar nicht notwendig wäre.

3.2 Berechnen der Parallelität

Um die Parallelität nun für einen Algorithmus zu berechnen, trennt man sich zunächst von der Vorstellung, diesen auf einer konstanten Anzahl an Prozessoren ablaufen zu lassen.

T_∞ : Zeit die ein Algorithmus auf unendlich Prozessoren benötigt

$$\text{Parallelität} = \frac{T_1}{T_\infty}$$

Serieller Zeitverbrauch, geteilt durch parallelen Zeitverbrauch, auf unendlich vielen Prozessoren, ergibt die Parallelität. Der Serielle Zeitverbraucht ist T_1, dies entspricht der Komplexitätsklasse der Zeit des Algorithmus. T_∞ wird genau wie T_1 berechnet, aber alle Parallel ausgelagerten Teilfunktionen werden nicht mit in die Zeitberechnung einbezogen. Es ist nur noch der Zeitverbrauch auf dem kritischen Pfad von Interesse.

3.3 Bestimmung der Parallelität der Fibonacci Funktion

Ein Beispiel anhand der exponentiellen Fibonacci Funktion. Die Fibonacci Zahlen lassen sich natürlich unter geringerem Aufwand berechnen, aber zur Demonstration eignet sich dieses Beispiel sehr gut.

Serielle Version (ohne SPAWN und SYNC):

```
 1 fibonacci( n )
 2
 3     if n <= 1
 4         return n
 5
 6     x = fibonacci( n - 1 )
 7     y = fibonacci( n - 2 )
 8
 9     return x + y
10 end
```

$FIB_1(n) = FIB_1(n-1) + FIB_1(n-2) + \Theta(1)$

$FIB_1(n) \in \Theta(\phi^n)$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1{,}62$$

(Der Schluss von Rekursionsgleichung auf Komplexitätsklasse wird in dieser Ausarbeitung nicht behandelt. Bei Interesse lohnt sich ein Blick in [1], alternativ lassen sich viele der noch folgenden Rekursionsgleichungen auch sehr einfach mit dem Mastertheorem lösen.)

Parallele Version:

```
 1 fibonacci( n )
 2
 3     if n <= 1
 4         return n
 5
 6     spawn x = fibonacci( n - 1 )
 7     y = fibonacci( n - 2 )
 8     sync
 9
10     return x + y
11 end
```

$FIB_\infty(n) = FIB_\infty(n-1) + \sout{FIB_\infty(n-2)} + \Theta(1)$
Es wird nur noch berücksichtigt welcher der Parallel ausgeführten Funktionsaufrufe am längsten dauert.

$FIB_\infty(n) = FIB_\infty(n-1) + \Theta(1)$

$FIB_\infty(n) \in \Theta(n)$

Die Parallelität ergibt sich also zu:

$$\frac{T_1}{T_\infty} = \frac{FIB_1}{FIB_\infty} = \frac{\Theta(\phi^n)}{\Theta(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

Es ergibt sich also eine exponentielle Funktion, aus der sich schließen lässt, dass dieser Fibonacci Algorithmus sich äußerst gut parallelisieren lässt. Denn um hier maximale Beschleunigung zu erhalten, kann eine exponentielle Anzahl von Prozessoren im Verhältnis zur Problemgröße N, verwendet werden.

Für Algorithmen die sich nicht parallelisieren lassen gilt $T_1 = T_\infty$, woraus folgt, dass die Parallelität $T_1 / T_\infty = 1$ ist. Was auch die logische Schlussfolgerung ist, da diese Algorithmen aus mehr als einem Prozessor keinen Nutzen ziehen können.

4 Parallelitätsanalyse von Mergesort

Anhand eines Beispiels von Mergesort wird sich zeigen, wie ein auf den ersten Blick einfach zu parallelisierender Algorithmus nur eine schlechte Parallelität erlangt und warum es deshalb sinnvoll ist, solch eine Parallelitätsanalyse durchzuführen.

4.1 Parallelisierung der Sortfunktion

```
1  merge_sort( A, l, r )
2  {
3      if l < r
4      {
5          int m = (l + r) / 2
6
7          spawn merge_sort( A, l, m )
8          merge_sort( A, m+1, r )
9          sync
10
11         merge( A, l, m, r )
12     }
13 }
```

```
1  merge( A, l, m, r )
2  {
3      L = Array( m - l + 2 )
4      R = Array( r - m + 1 )
5      L[ L.size ] = ∞
6      R[ R.size ] = ∞
7
8      for i = 1 to L.size - 1
9          L[i] = A[ l + i - 1 ]
10
11     for i = 1 to R.size - 1
12         R[i] = A[ m + i ]
13
14     l_pos = 1
15     r_pos = 1
16     for k = l to r
17         if L[ l_pos ] <= R[ r_pos ]
18             A[k] = L[ l_pos++ ]
19         else
20             A[k] = R[ r_pos++ ]
21 }
```

Genau wie bei Fibonacci liegt es auch hier wieder nahe, die Parallelität beim rekursiven Abstieg zu implementieren. Zu den in konstanter Zeit arbeitenden Operationen in merge_sort() kommt noch der Aufruf von merge() hinzu, der in linearer Zeit aus zwei sortierten Arrays eins macht.

$MS_1(n) = 2 \cdot MS_1\left(\frac{n}{2}\right) + MERGE(n)$
$MS_1(n) = 2 \cdot MS_1\left(\frac{n}{2}\right) + \Theta(n)$
$MS_1(n) \in \Theta(n \cdot ld(n))$

$MS_\infty(n) = \quad MS_\infty\left(\frac{n}{2}\right) + MERGE(n)$
$MS_\infty(n) = \quad MS_\infty\left(\frac{n}{2}\right) + \Theta(n)$
$MS_\infty(n) \in \Theta(n)$

Parallelität $= \frac{T_1}{T_\infty} = \frac{MS_1}{MS_\infty} = \frac{\Theta(n \cdot ld(n))}{\Theta(n)} = \boldsymbol{\Theta(ld(n))}$

Wir erhalten als Parallelitätsfunktion eine logarithmische Funktion. Das spricht für ein schlechtes paralleles Verhalten, denn im Gegensatz zur Problemgröße wächst die Anzahl der sinnvoll benutzbaren Prozessoren nur sehr langsam, und die Beschleunigungsgrenze ist mit nur wenigen Prozessoren sehr schnell erreicht.

Problemgröße n	**Parallelität ld(n)**
10^3 (Tausend)	9.9657842846621
10^6 (Million)	19.931568569324
10^9 (Milliarde)	29.897352853986
10^12 (Billion)	39.863137138648
10^15 (Billiarde)	49.82892142331

Der Grund für dieses Verhalten ist die Mergefunktion. Es werden zwar sehr viele Threads erstellt, aber diese sind nur mit dem Warten auf die Ergebnisse der Mergefunktion beschäftig. Die Ergebnisse von Merge() werden also nicht schnell genug, oder besser gesagt nicht parallel genug bereitgestellt.

Merge() muss ein Array mit N Elementen füllen und tut dies auch in $\Theta(n)$. Schneller ist das nicht möglich, entscheidend ist, ob es eine Möglichkeit gibt diesen Vorgang zu parallelisieren. Allerdings stellt sich heraus, dass Merge() inhärent seriell ist. Das liegt an l_pos und r_pos, die die aktuelle Position in den zu mergenden Arrays anzeigen. Da bei jedem Schleifendurchlauf auf die Position vom vorherigen Durchlauf zurückgegriffen wird, lässt sich hier keine Parallelität implementieren.

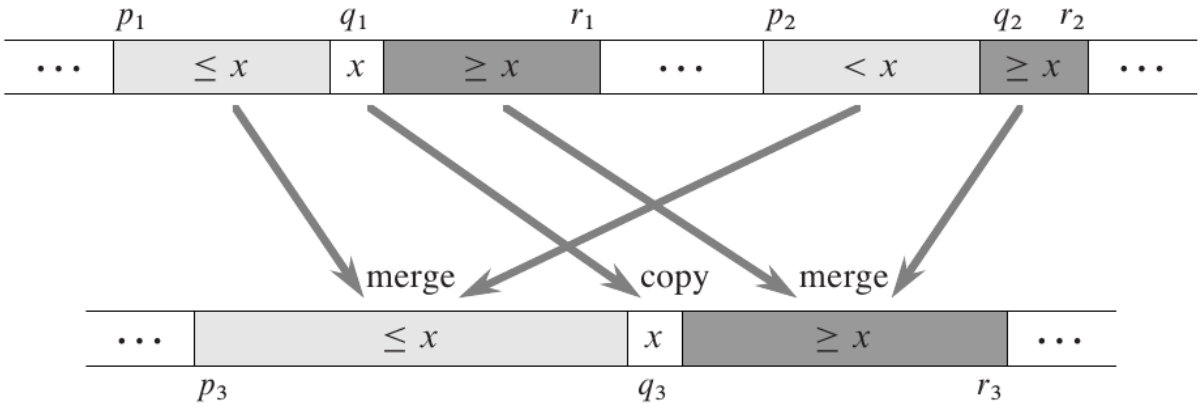
Um den Mergevorgang jetzt doch zu parallelisieren, muss die Mergefunktion neu entworfen werden. Die alte Mergefunktion ist wertlos, da sie sich nicht parallelisieren lässt, obwohl ihre serielle Geschwindigkeit mit $\Theta(n)$ nicht verbesserbar ist.

4.2 Entwicklung einer parallelen Mergefunktion

Anforderungen:

1. aus zwei sortierten Arrays ein sortiertes Array machen
2. sollte seriell nicht langsamer sein als die alte Version
3. parallel

Parallele Mergefunktion mittels Divide & Conquer:



Quelle: [1]

$p_1 - r_1$: Erstes bereits sortiertes Eingabearray.
$p_2 - r_2$: Zweites bereits sortiertes Eingabearray.
$p_3 - r_3$: Ausgabearray A.

Algorithmus:

1. Bestimmen des größeren Eingabearrays K_1 und des kleiner oder gleichgroßen K_2.
2. Bestimmen der Position des Median x in K_1. $q_1 = (r_1 - p_1) / 2$
3. Die Position in K_2 finden, in der der Median aus K_1 eingefügt werden könnte, sodass nach einfügen K_2 immer noch sortiert wäre. Diese Position q_2 wird mittels Binärersuche ermittelt.
4. q_3 ergibt sich aus $p_3 + (q_1 - p_1) + (q_2 - p_2)$
5. x wird nach q_3 kopiert. Dies wäre die letzte Operation vor dem Rekursionsabbruch, wenn die Länge der Eingabearrays 1 und 0 ist.
6. Rekursives Mischen von $K_1[\, p_1 \,..\, q_1 - 1]$ und $K_2[\, p_2 \,..\, q_2 - 1]$ deren Ergebnis nach $A[\, p_3 \,..\, q_3 - 1]$ gespeichert wird. (Hellgraue Arrays)
7. Rekursives Mischen von $K_1[\, q_1 + 1 \,..\, r_1]$ und $K_2[q_2 \,..\, r_2]$ deren Ergebnis nach $A[q_3 + 1 \,..\, r_3]$ gespeichert wird. (Dunkelgraue Arrays)

Aus diesen Ideen ergibt sich folgender Pseudocode. Zu beachten ist, dass die Eingabearrays K_1 und K_2 hier zusammen in dem Array T zu finden sind.

```
 1 para_merge(T, p1, r1 p2, r2, A, p3 )
 2 {
 3     n1 = r1 - p1 + 1
 4     n2 = r2 - p2 + 1
 5     if n1 < n2 {
 6         swap( p1, p2 )
 7         swap( r1, r2 )
 8         swap( n1, n2 )
 9     }
10     if n1 == 0
11         return
12
13     q1 = (p1 + r1) / 2
14     q2 = binary_search( T[q1], T, p2, r2 )
15     q3 = p3 + (q1 - p1) + (q2 - p2)
16     A[q3] = T[q1]
17
18     spawn para_merge( T, p1, q1-1, p2, q2-1, A, p3)
19     para_merge( T, q1+1, r1, q2, r2, A, q3+1)
20     sync
21 }
```

Der Einfachheit halber, wird mit der parallelen Version begonnen.

$PM_\infty(n) = \quad PM_\infty\left(\frac{3}{4}\,n\right) + BS\left(\frac{1}{2}\,n\right)$

Die binäre Suche liegt in der Komplexitätsklasse $\Theta\big(ld(n)\big)$.

$PM_\infty(n) = \quad PM_\infty\left(\frac{3}{4}\,n\right) + \Theta\big(ld(n)\big)$

$PM_\infty(n) \in \Theta(ld^2(n))$

Die drei Viertel im rekursiven Aufruf ergeben sich aus der Worst-Case Konstellation:

1. Beide Arrays sind gleich groß (für sich allein genommen nicht schlimm)
2. Die Position des Median x liegt ganz am Rand von K_2

Dadurch wird im schlechtesten Fall ein rekursiver Aufruf mit drei Viertel der Problemgröße eintreten, wohingegen der andere Aufruf nur ein Viertel erhält.

In der parallelen Rekursionsgleichung muss nur der teurere Aufruf miteinbezogen werden, ganz im Gegensatz zur seriellen Rekursionsgleichung:

$$PM_1(n) = \quad PM_1(\alpha \cdot n) + PM_1((1-\alpha)\cdot n) + \Theta\big(ld(n)\big) \qquad \frac{1}{4} \le \alpha \le \frac{3}{4}$$

$PM_1(n) \in \Theta(n)$

4.3 Alte und neue Mergefunktion im Vergleich

Serielle Merge Funktion:

$M_1(n) \in \Theta(n)$
$M_\infty(n) \in \Theta(n)$

Parallelität = $\frac{M_1}{M_\infty} = \frac{\Theta(n)}{\Theta(n)} = \mathbf{1}$

Parallele Merge Funktion:

$PM_1(n) \in \Theta(n)$
$PM_\infty(n) \in \Theta(ld^2(n))$
Parallelität = $\frac{PM_1}{PM_\infty} = \frac{\Theta(n)}{\Theta(ld^2(n))} = \boldsymbol{\Theta\left(\frac{n}{ld^2(n)}\right)}$

Als erstes sollte auffallen, dass sich das serielle Verhalten der neuen Mergefunktion nicht verschlechtert hat. Und im Gegensatz zur alten Mergefunktion zeigt die neue linear beschränkte Parallelität.

Um diese neue Mergefunktion benutzen zu können, muss die Sortfunktion ebenfalls angepasst werden. Logisch gesehen verändert sich hier aber nichts.

```
 1 merge_sort_2( A, p, r, B, s)
 2 {
 3     n = r - p + 1
 4     if n == 1
 5         B[s] == A[p]
 6     else
 7     {
 8         T = Array( n )
 9         m = (p + r) / 2
10         q = m - p + 1
11
12         spawn merge_sort_2( A, p, m, T, 1)
13         merge_sort_2( A, m+1, r, T, q+1)
14         sync
15
16         para_merge( T, 1, q, q+1, n, B, s )
17     }
18 }
```

4.4 Parallelitätsanalyse von parallelem Mergesort mit parallelem Merge

$$
\begin{aligned}
PMS_1(n) &= 2 \cdot PMS_1\left(\tfrac{n}{2}\right) + PM_1(n) \\
&= 2 \cdot PMS_1\left(\tfrac{n}{2}\right) + \Theta(n) \\
&\in \Theta(n \cdot ld(n))
\end{aligned}
$$

$$
\begin{aligned}
PMS_\infty(n) &= PMS_\infty\left(\tfrac{n}{2}\right) + PM_\infty(n) \\
&= PMS_\infty\left(\tfrac{n}{2}\right) + \boldsymbol{\Theta\left(ld^2(n)\right)} \\
&\in \boldsymbol{\Theta\left(ld^3(n)\right)}
\end{aligned}
$$

Parallelität = $\frac{PMS_1}{PMS_\infty} = \frac{\Theta(n \cdot ld(n))}{\Theta(ld^3(n))} = \boldsymbol{\Theta\left(\frac{n}{ld^2(n)}\right)}$

4.5 Paralleles Mergesort mit und ohne parallelem Merge im Vergleich

Parallel Mergesort mit <u>serieller</u> Merge Funktion:

$MS_1(n) \in \Theta(n \cdot ld(n))$
$MS_\infty(n) \in \Theta(n)$

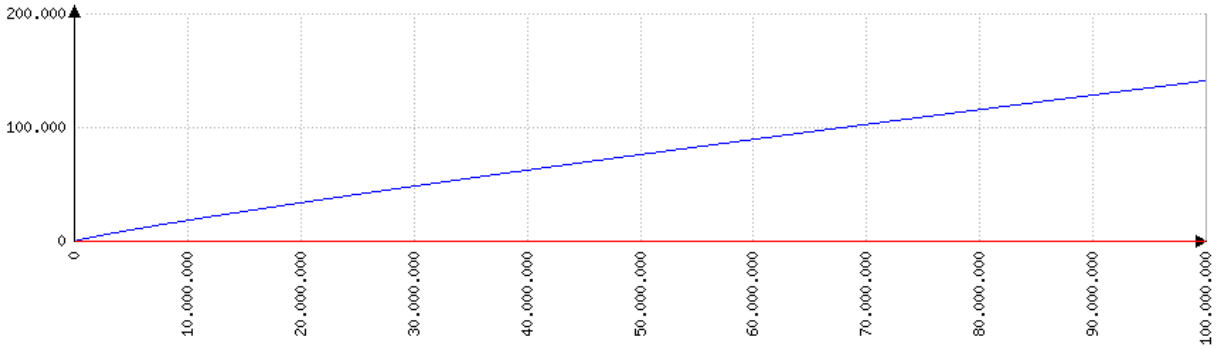
Parallelität = $\frac{MS_1}{MS_\infty} = \frac{\Theta(n \cdot ld(n))}{\Theta(n)} = \boldsymbol{\Theta(ld(n))}$

Parallel Mergesort mit <u>paralleler</u> Merge Funktion:

$PMS_1(n) \in \Theta(n \cdot ld(n))$
$PMS_\infty(n) \in \Theta(ld^3(n))$

Parallelität = $\frac{PMS_1}{PMS_\infty} = \frac{\Theta(n \cdot ld(n))}{\Theta(ld^3(n))} = \boldsymbol{\Theta\left(\frac{n}{ld^2(n)}\right)}$

Diese verbesserte Parallelität ist nicht nur theoretisch besser, sondern auch im praktischen Einsatz.



Rot: $\Theta(ld(n))$ Blau: $\Theta\left(\frac{n}{ld^2(n)}\right)$

Man sieht an der Skalierung von 100.000.000 zu 200.000, dass die Parallelität immer noch limitiert wird, aber es ergeben sich wesentlich bessere Werte als vorher.

N	ld(n)	n / ld²(n)
10^3 (Tausend)	9.9657842846621	10.068784254384
10^6 (Million)	19.931568569324	2517.196063596
10^9 (Milliarde)	29.897352853986	1118753.8060427
10^12 (Billion)	39.863137138648	629299015.899
10^15 (Billiarde)	49.82892142331	402751370175.36

Die neu erstellte Mergefunktion sorgt hier für einen gewaltigen Parallelitätszuwachs. Wo vorher nur Prozessoranzahlen im zweistelligen Bereich die Grenze der Beschleunigung waren, kommt man jetzt schon an die Grenze des technisch überhaupt Realisierbaren.

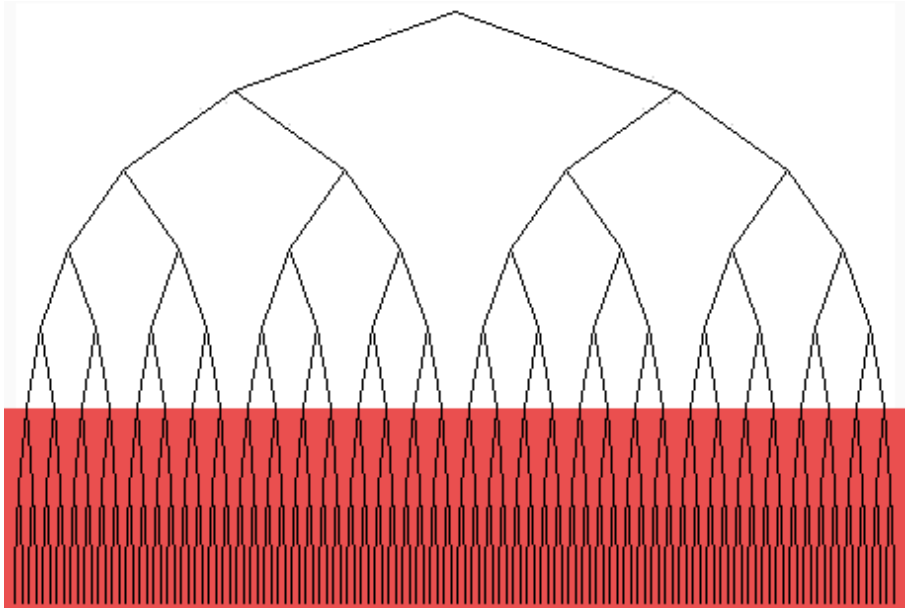
5 Probleme im praktischen Einsatz

Nachdem gezeigt wurde, welche Problematiken beim parallelen Algorithmendesign auftreten können, sollen noch einige Punkte angesprochen werden, die eine praktische Umsetzung mit sich bringen würde.

5.1 Wechsel auf einen seriellen Algorithmus bei kleiner Problemgröße

Bei dem Lösen eines größeren Problems mittels eines parallelen Divide & Conquer Ansatzes tritt tief im Berechnungsbaum der Fall ein, dass sehr triviale Probleme natürlich ebenfalls parallel gelöst werden müssen. Wo es bei großer Problemgröße sehr viel Sinn macht, das Problem auf verschiedene Prozessoren aufzuteilen, wäre es bei kleiner Problemgröße maßlos übertrieben, diese mit einem parallelen Divide & Conquer Ansatz zu lösen, weil ein riesiger Overhead durch die Threadverwaltung entstehen würde.
Um dem entgegenzuwirken, sollte im Berechnungsbaum ab einer bestimmten Problemgröße auf eine serielle Version des Algorithmus gewechselt werden. In manchen Fällen bietet es sich auch an, auf einen ganz anderen Algorithmus zu wechseln der für kleine Problemgrößen optimiert ist.

Als Beispiel wieder das Sortierproblem: 1000 Elemente zu sortieren hätte einen Berechnungsbaum mit 1999 Knoten zur Folge. Da in den Blättern des Baumes keine Threads gespawned werden, entstehen 999 Threads, also $n - 1$.
Das Problem 1000 Elemente zu sortieren ist, seriell gesehen, trivial und kann fast als konstante Operation angesehen werden. Mit parallelem Divide & Conquer Ansatz aber, entsteht riesiger Overhead durch: anlegen von Threads, Synchronisation, Scheduler Verwaltung, löschen von Threads, etc.. Dieser Vorgang ist nicht auf einen Fall beschränkt, sondern das Problem 1000 Elemente zu sortieren, zieht sich natürlich durch den kompletten unteren Abschnitt des Berechnungsbaumes.



Weiß: Großes Problem - Threadoverhead winzig im Vergleich zum Problem.
Rot: Kleine triviale Probleme - Threadoverhead gigantisch, im Vergleich zum Problem.

Die Kunst ist es jetzt natürlich, eine Problemgröße zu bestimmen, ab der auf einen seriellen Algorithmus gewechselt wird. Allerdings lässt sich dieser Punkt meistens sehr gut abschätzen. Alternativ sind aber auch Techniken denkbar, die diesen Punkt intelligent selber finden können, um sich verschiedenen Hardware- und Betriebssystemen anzupassen. Zum Beispiel durch Benchmarks die im Vorfeld Testdaten sammeln, um daraus diesen Punkt zu berechnen. Sehr wichtig ist aber das solch eine Technik eingesetzt wird, denn je nach Betriebssystem und Hardware, ist die Anzahl der möglichen Threads limitiert und diese Grenze könnte schon bei kleinen Problemgrößen erreicht werden.

5.2 Parallelität auf höherem Level implementieren

Es stellt sich die Frage, warum nicht von vornherein die Anzahl der zur Verfügung stehenden Prozessoren stärker mit einbezogen wird und nur so viele Threads gespawned werden wie Prozessoren vorhanden sind. Anstelle dessen, dass erst bei einer zu kleinen Problemgröße auf einen seriellen Algorithmus gewechselt wird, könnte dies schon passieren nachdem jedem Prozessor ein Thread zugeteilt wurde.

Um es erst einmal vorweg zu nehmen, diese Möglichkeit ist praktisch durchaus einsetzbar. Es entsteht wenig Overhead durch Threadverwaltung, da die Anzahl der Threads überschaubar bleibt. Das vorherige Problem mit dem Overhead bei zu kleinen Problemgrößen entsteht auch nicht.

Allerdings darf nicht außer Acht gelassen werden, dass mit diesem Modell ein Teil der Parallelität verschwendet wird, denn auch wenn das Problem in gleich große Teile aufgeteilt wird, ist nicht gewährleistet, dass die serielle Berechnung der Teilergebnisse immer gleich lange dauert. Da ja anfangs auch nicht bekannt ist wie lange die Berechnung dauern wird, ist dieses Teilproblem auch so im Vorfeld nicht zu kontern.

Darüber hinaus reicht es auch nicht, wenn der seriell verwendete Algorithmus aus der Koplexitätsklasse Θ ist, sondern es muss abhängig der Problemgröße immer exakt die gleiche Anzahl an Maschineninstruktionen resultieren. Und diese Eigenschaft ist nur bei sehr trivialen Algorithmen gegeben. Und zu guter Letzt muss sich auch der Scheduler absolut fair verhalten.

Sind diese Eigenschaften nicht gegeben, kann eine lineare Beschleunigung nicht erreicht werden, weil die Threads unterschiedlich viel Zeit benötigen.

5.3 Speicherplatzverbrauch: Seriell vs. Parallel

Bei Divide & Conquer Algorithmen die in jedem Berechnungsknoten neben den rekursiven Aufrufen auch noch Speicherplatz allozieren, dessen Größe im Verhältnis zur Problemgröße steht, ist ebenfalls Vorsicht geboten sollten diese parallelisiert werden. (Ein Beispiel für einen solchen Algorithmus wäre hier, die endgültige parallele Version von Mergesort. In Zeile 8 wird ein Array der Größe n angelegt).

Im seriellen Ablauf wird der Baum wie bei einer Tiefensuche traversiert, woraus so maximal ein Speicherverbrauch von $2 \cdot n$ entsteht. Sollte dieser jetzt aber parallel abgearbeitet werden, wird natürlich auch die Speicherallokationen gleichzeitig stattfinden, wodurch im Speicherplatz logarithmisch zur Tiefe des Baumes benötigt werden kann: $n \cdot ld(n)$.

Eine Möglichkeit wäre es, wenn der Algorithmus es zulässt, den Speicherplatz erst nach den rekursiven Aufrufen zu allozieren, aber das wird in den meisten Fällen nicht möglich sein. Sollte man Aufgrund dieses Problems an die Grenzen des verfügbaren Speicherplatzes stoßen, muss neben dem Prozessor Scheduler die Implementation eines Memory Schedulers in Betracht gezogen werden. Das würde sich dann natürlich negativ auf die Parallelität auswirken, denn Threads die eigentlich arbeiten könnten, weil freie Prozessoren vorhanden sind, können nicht arbeiten weil kein weiterer Speicherplatz mehr zur Verfügung steht.

6 Referenzen

[1] Th. H. Cormen, Ch. E. Leiserson, R. Rvest, C. Stein / Introduction to Algorithms / 3. Auflage

[2] W. Cohan, C. Patel, A. Seshagiri / Cost of User and Kernel Level Threads Operations on Linux

[3] C. Breshears / The Art of Concurrency