Architektur und Programmierung von Grafik- und Koprozessoren

Performanz von Computerprogrammen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Komplexität paralleler Algorithmen

Parallelisierungspotential Einschätzen

- Wir haben nun Methodiken und Werkzeuge kennengelernt, um Parallelismus auf Prozessoren auszunutzen.
- Im nächsten Schritt wollen wir einschätzen lernen, welche Parallelisierungspotentiale ein Computerprogramm überhaupt hat.
- Diese Einschätzung soll zunächst theoretischer Natur sein.

Amdahlsches Gesetz (1)

- Gene Myron Amdahl
- 16. Nov. 1922 - 10. Nov. 2015
- Entwicklung Mainframe Computer IBM

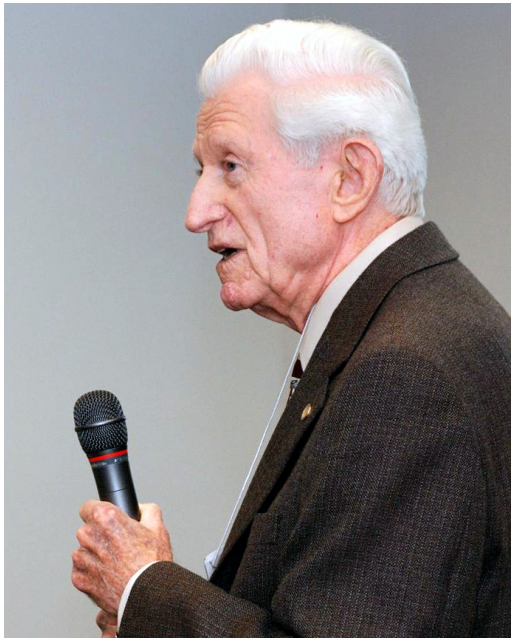


Abbildung: Gene M. Amdahl, ©Perry Kivolowitz, 2008, CC BY 3.0 Lizenz[1]

[1] https://creativecommons.org/licenses/by/3.0/

Amdahlsches Gesetz (2)

- Computerprogramme haben Bestandteile, die parallel ausgeführt werden können, und andere Bestandteile, die lediglich sequentiell durchgeführt werden können.
- Solche Bestandteile können bspw. C++ Befehle sein, wobei der Programmierer explizit bestimmt, ob diese in unterschiedlichen Prozessen, Threads etc. ausgeführt werden.
- Parallelismus auch auf Instruktionslevel möglich, dann bestimmt der Compiler oder die CPU zur Laufzeit, welche Instruktionen parallel ausgeführt werden.
- Wir abstrahieren davon und nehmen schlicht an, dass sich eindeutig quantifizieren lässt, zu welchen Teilen ein Computerprogramm sich aus sequentiellen und parallelen Bestandteilen zusammensetzt.

Amdahlsches Gesetz (3)

Bezeichne $P \in [0..1]$ den *Anteil* des Computerprogramms, der sich parallel ausführen lässt. Ferner bezeichne S_{par} den *Speedup*, den wir uns *für den parallelen Anteil* durch eine Parallelisierung erhoffen:

$$S_{par} = \frac{T(1)}{T(N)}, \tag{8}$$

wobei $T(1)$ die Ausführungszeit auf einem Rechenkern/Prozessor etc. und $T(N)$ die Ausführungszeit auf N Kernen bezeichne. Dann ergibt sich der *maximal erzielbare Speedup* als

$$S = \frac{1}{(1-P) + \frac{P}{S_{par}}}. \tag{9}$$

Der maximal erzielbare Speedup stellt eine theoretische obere Grenze dar, um die sich die Programmausführung maximal beschleunigen lässt.

Amdahlsches Gesetz (4)

Der Speedup für den parallelen Anteil des Programms, S_{par}, hängt von der Ausführungszeit des Programms auf N Prozessoren $T(N)$ ab. Diese setzt sich zusammen aus der tatsächlichen Rechenzeit T_c, der eventuellen Wartezeit T_w, und der Zeit T_o, die durch Kommunikationsoverhead entsteht.

$$T(N) = T_c + T_w + T_o. \tag{10}$$

Amdahlsches Gesetz Beispiel (1)

- Wir schätzen, dass der parallelisierbare Teil unseres Programms 80% beträgt. Wir vermuten, dass wir diesen Programmbestandteil auf einem Multi-Core Prozessor um einen Faktor 4 beschleunigen können.
  - Wir berechnen den maximal erzielbaren Speedup für das ganze Programm als

$$S = \frac{1}{(1 - 0.8) + \frac{0.8}{4}} = 2.5. \tag{11}$$

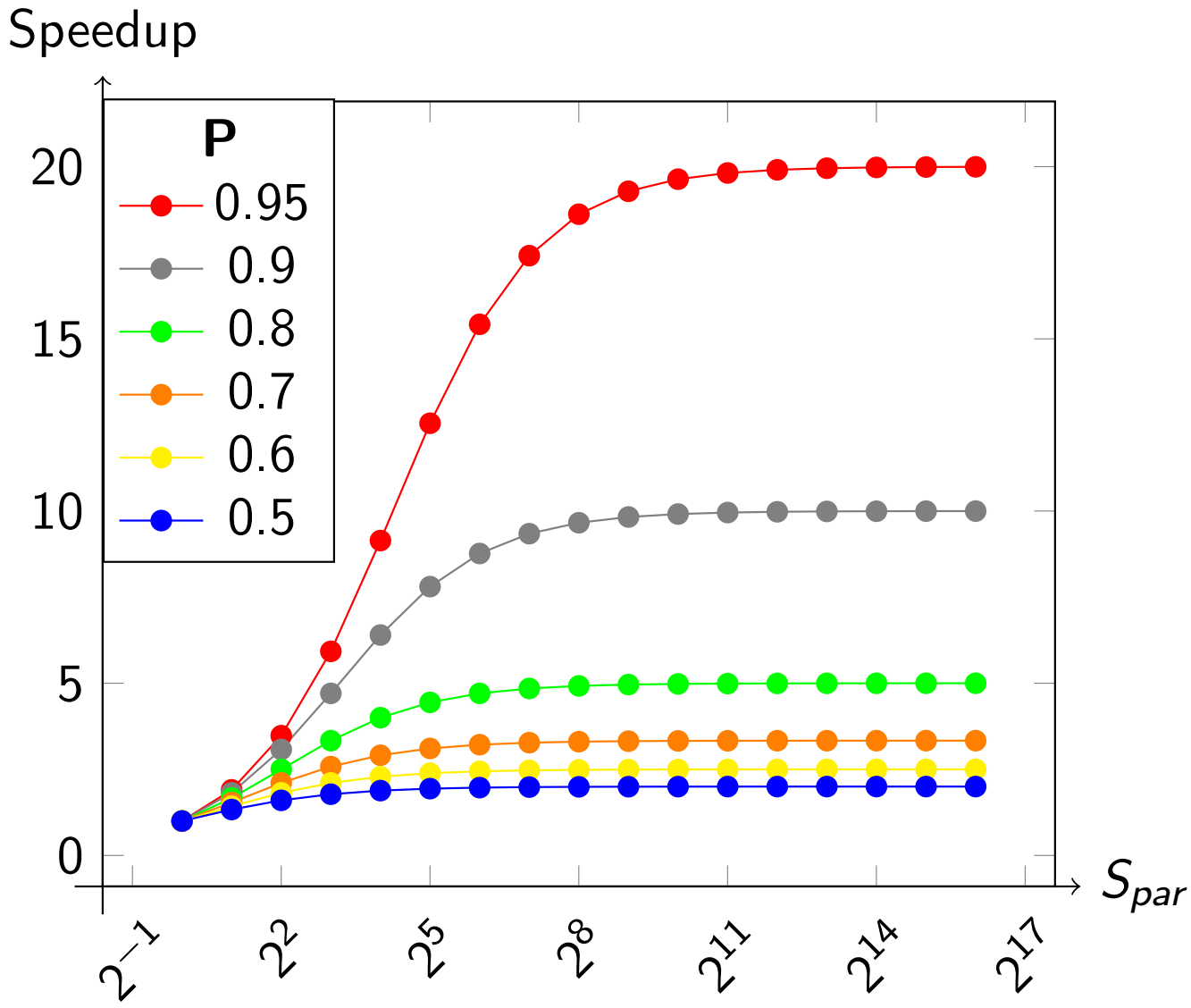
- Auf einem Supercomputer schätzen wir das Parallelisierungspotential deutlich höher ein, wir erwarten, dass $S_{par} = 200$.
  - Wir berechnen den maximal erzielbaren Speedup für das ganze Programm als

$$S = \frac{1}{(1 - 0.8) + \frac{0.8}{200}} = 4.9. \tag{12}$$

Amdahlsches Gesetz Beispiel (2)

- Am Beispiel erkennt man sehr anschaulich, dass selbst dann, wenn der Anteil des Computerprogramms, der parallel ausgeführt werden kann, im Vergleich zum sequentiellen Bestandteil sehr hoch ist, das Beschleunigungspotential durch Parallelisierung tatsächlich eher gering ist.
- Da das Amdahlsche Gesetz eine theoretische obere Schranke vorgibt, wird dieses Beschleunigungspotential von tatsächlichen Implementierungen nicht voll ausgeschöpft werden.

Amdahlsches Gesetz



Parallele Effizienz

- Die Effizienz eines parallelen Programms, oder kurz die "Parallele Effizienz", eines Programms, das auf N Prozessoren ausgeführt wird, ist definiert als:

$$E(N) = \frac{S_{par}}{N}, \tag{13}$$

- Während das Amdahlsche Gesetz Aufschluss über das Parallelisierungpotential für das *gesamte Programm gibt*, stellt Effizienz den *Speedup für die parallel ablaufenden Programmbestandteile* ins Verhältnis zu den eingesetzten Ressourcen.
- Im Idealfall (der in der Praxis nicht eintritt) gilt $S_{par} = N$ und $E(N) = 1$.

Einfussfaktoren für Parallele Effizienz

- **Latency** - Warten auf Speicherzugriffe oder auf andere Ressourcen (z. B. Netzwerkstack, File IO etc.)
- **Overhead** - Synchronisationsmechanismen wie Mutexe oder atomare Variablen bedingen Overhead, der in sequentiellen Programmen nicht auftritt.
- **Starvation** - Parallele Prozessoren warten ("idle"), während andere parallele Prozessoren noch arbeiten. In solchen Fällen: *Lastverteilung*.
- **Contention** - Parallele Prozessoren greifen gleichzeitig auf geteilte Ressourcen zu, die Zugriffe müssen synchronisiert werden. Beispiele sind Hauptspeicherzugriffe, Netzwerkzugriffe etc.

Kosten

Die Kosten eines parallelen Programms sind definiert als das Produkt aus Ressourceneinsatz N und der Ausführungszeit auf N Prozessoren:

$$C(N) = N \cdot T(N). \tag{14}$$

Die Kosten sind immer höher oder gleich der seriellen Komplexität:

$$T(1) \leq C(N). \tag{15}$$

Das folgt aus der Definition des parallelen Anteils des Speedups:

$$S_{par} = \frac{T(1)}{T(N)} \leq N \Leftrightarrow T(1) \leq N \cdot T(N). \quad \square \tag{16}$$

Kostenoptimalität

Falls die Kosten eines parallelen Algorithmus der seriellen Komplexität entsprechen, nennen wir das Programm *kostenoptimal*:

$$C(N) = T(1). \tag{17}$$

Es ist also das Ziel, das *beste* serielle Programm (bzgl. Komplexität) zu kennen und ein *bzgl. dessen* kostenoptimales paralleles Programm zu finden.

Parallel Random Access Machine (PRAM)

- Bei sequentiellen Algorithmen schätzen wir die Komplexität mittels asymptotischer Betrachtungen ein. Dabei legt man serielle Maschinenmodelle zu Grunde.
- Um die Komplexität paralleler Algorithmen einschätzen zu können, muss man Maschinenmodelle zu Grunde legen, die die asymptotische Ausführungszeit im Verhältnis zum Ressourceneinsatz betrachten.
- Wir wollen im folgenden kurz ein solches Modell behandeln. (Eine ausführliche Betrachtung solcher Modelle ist Gegenstand einer “Parallele Algorithmen” Vorlesung.)

Parallel Random Access Machine (PRAM)

PRAM Maschinenmodell

Wir legen ein Maschinenmodell zugrunde, das über beliebig viele Prozessoren verfügt. Die Prozessoren verfügen über geteilten Speicher. Dieser ist unbegrenzt verfügbar, außerdem benötigt jeder Speicherzugriff eine Zeiteinheit. Die Prozessoren arbeiten Instruktionsfolgen *synchron* ab.

Das Modell programmiert man, indem man *parallele Funktionen* schreibt, die auf P Prozessoren ausgeführt werden. In der Funktion steht der *Prozessorindex* **p** zur Verfügung.

Parallel Random Access Machine (PRAM)

PRAM Speichermodell

Man nimmt jedoch *i. Allg. nicht* an, dass alle Prozessoren gleichzeitig auf den geteilten Speicher zugreifen können. Vielmehr unterscheidet man drei Arten von PRAMs gemäß ihrer Speicherzugriffscharakteristen.

- **EREW** - **E**xclusive **R**ead, **E**xclusive **W**rite: alle Speicherzugriffe erfolgen exklusiv.
- **CREW** - **C**oncurrent **R**ead, **E**xclusive **W**rite: lesender Zugriff erfolgt gleichzeitig, schreibender Zugriff exklusiv.
- **CRCW** - **C**oncurrent **R**ead, **C**oncurrent **W**rite: alle Prozessoren können gleichzeitig lesend und schreibend auf den Speicher zugreifen.

Parallel Random Access Machine (PRAM)

PRAM Speichermodell

Algorithmen, die auf einer EREW PRAM lauffähig sind, sind auch auf allen anderen PRAM Modellen lauffähig. CREW Algorithmen lassen sind auch auf CRCW PRAMs ausführbar.

CRCW PRAMs unterscheiden sich bzgl. der Reihenfolge, in der Schreibzugriffe durchgeführt werden. Dies betrachten wir im Rahmen der Vorlesung nicht näher.

Arbeit vs. Zeit

Arbeitskomplexität $W(n)$: Anzahl an Operationen, die der Algorithmus insgesamt ausführt.

Zeitkomplexität (auch *Schrittkomplexität*) $S(n)$: Anzahl an Einzelschritten, die der Algorithmus ausführt.

Es gilt die Identität

$$W(n) = \sum_{i=1}^{S(n)} W_i(n), \tag{18}$$

wobei $W_i(n)$ die Anzahl der parallelen Einzelschritte zum Schritt i bezeichnet.

Arbeit vs. Zeit

Beispiel

Paralleles saxpy auf P Prozessoren $\mathbf{p} \in [0..P-1]$, wobei Array Länge $N > P$.

function SaxpyPRAM(y,a,x,N) ▷ Par. Funktion auf P Proz.
  $np \leftarrow \lceil \frac{N}{P} \rceil$ ▷ Anteil, den p bearbeitet
  $first \leftarrow np * \mathbf{p}$ ▷ Range, die p bearbeitet
  $last \leftarrow first + np$
  for all $i \in [first..last)$ **do** ▷ saxpy für N/P Elemente
    $y[i] \leftarrow a[i] * x[i] + y[i]$
  end for
end function

Arbeit vs. Zeit

Beispiel

Wir bestimmen zunächst die Schrittkomplexität:

function SAXPYPRAM(y,a,x,N)
  $np \leftarrow \lceil \frac{N}{P} \rceil$ ▷ $O(1)$ auf P Prozessoren
  $first \leftarrow np * \mathbf{p}$ ▷ $O(1)$ auf P Prozessoren
  $last \leftarrow first + np$ ▷ $O(1)$ auf P Prozessoren
  for all $i \in [first..last)$ **do**
    $y[i] \leftarrow a[i] * x[i] + y[i]$ ▷ $O(\lceil \frac{N}{P} \rceil)$ auf P Prozessoren
  end for
end function

$\Rightarrow S(n) = 4 + \lceil \frac{N}{P} \rceil = O(\lceil \frac{N}{P} \rceil)$ Einzelschritte.

Arbeit vs. Zeit

Beispiel

Wir zählen die *Einzelschritte* $W_i(n)$, um die Arbeitskomplexität $W(n)$ zu berechnen.

function SAXPYPRAM(y,a,x,N)
  $np \leftarrow \lceil \frac{N}{P} \rceil$ ▷ $W_1 = P$
  $first \leftarrow np * \mathbf{p}$ ▷ $W_2 = P$
  $last \leftarrow first + np$ ▷ $W_3 = P$
  for all $i \in [first..last)$ **do**
    $y[i] \leftarrow a[i] * x[i] + y[i]$ ▷ $W_4 = \lceil \frac{N}{P} \rceil \times P$
  end for
end function

$\Rightarrow W(n) = 3P + \lceil \frac{N}{P} \rceil \times P = O(N)$ Operationen.

Arbeit vs. Zeit

Vorgehen im Allgemeinen:

1. Finde den *seriellen* Algorithmus mit der vorteilhaftesten Komplexität.
2. Finde einen parallelen Algorithmus, der bei *gleichbleibender Arbeitskomplexität* die *Zeitkomplexität minimiert*.

Wir suchen also Algorithmen, die *insgesamt auf allen Prozessoren* genauso viel Arbeit verrichten, wie der beste serielle Algorithmus, und die trotzdem bzgl. der reinen Ausführungszeit des Algorithmus asymptotisch auf mehr Prozessoren schneller werden.

Brentsches Theorem

Beim Arbeit vs. Zeit Paradigma betrachtet man im Gegensatz zur PRAM parallele Algorithmen, die auf *unendlich vielen Prozessoren* ablaufen. Parallele Algorithmen formuliert man mit einem generischen **for all**..**do in parallel** Konstrukt, ohne explizit das Programm hinsichtlich P Prozessoren formulieren zu müssen. Das saxpy Beispiel lässt sich etwa vereinfachen zu

function SaxpyWorkTime(y,a,x,N)
  for all $i \in N$ **do in parallel**
    $y[i] \leftarrow a[i] * x[i] + y[I]$
  end for
end function

Wir wollen dieses Paradigma daher verwenden, um parallele Algorithmen zu formulieren.

Brentsches Theorem

Dazu machen wir uns die Eigenschaft zu Nutze, dass ein paralleler Algorithmus, der mit Arbeitskomplexität $W(n)$ und Schrittkomplexität $S(n)$ auf einer Maschine mit unendlich vielen Prozessoren ausgeführt wird, auf einer PRAM mit P Prozessoren mit höchstens

$$\left\lfloor \frac{W(n)}{P} \right\rfloor + S(n) \tag{19}$$

parallelen Schritten simuliert werden kann.

Brentsches Theorem: Beweis

Wir zeigen, dass

$$\left\lfloor \frac{W(n)}{P} \right\rfloor + S(n)$$

obere Schranke. Dazu betrachten wir die parallelen Operationen $W_i(n)$, wobei $1 \leq i \leq S(n)$. Bei P Prozessoren kann jede parallele Operation in $\left\lceil \frac{W_i(n)}{P} \right\rceil$ parallelen Schritten durchgeführt werden.

Offensichtlich gilt:

$$\sum_{i=1}^{S(n)} \left\lceil \frac{W_i(n)}{P} \right\rceil \leq \sum_{i=1}^{S(n)} \left(\left\lfloor \frac{W_i(n)}{P} \right\rfloor + 1 \right) \leq \left\lfloor \frac{W(n)}{P} \right\rfloor + S(n) \quad \square \quad (20)$$

Brentsches Theorem

Wegen des Brentschen Theorems kann man jeden parallelen Algorithmus, der gemäß des Arbeit vs. Zeit Paradigmas mit unendlich vielen Prozessoren formuliert ist, auch als PRAM Algorithmus auf P Prozessoren formulieren. Es gilt die oben genannte Schranke für die Anzahl benötigter paralleler Operationen.

Dies erlaubt uns eine allgemeinere Betrachtung paralleler Algorithmen.

Für die praktischen Beispiele im Verlauf der Vorlesung wollen wir der Einfachheit halber parallele Algorithmen nur noch im Sinne des Arbeit vs. Zeit Paradigmas formulieren.

Beispiel: Reduktion

Problem

Gegeben: Reellwertiges Array $a[1,..,n]$, $n \in \mathbb{N}$
Gesucht: $S_n := \sum_{i=1}^{n} a[i]$.

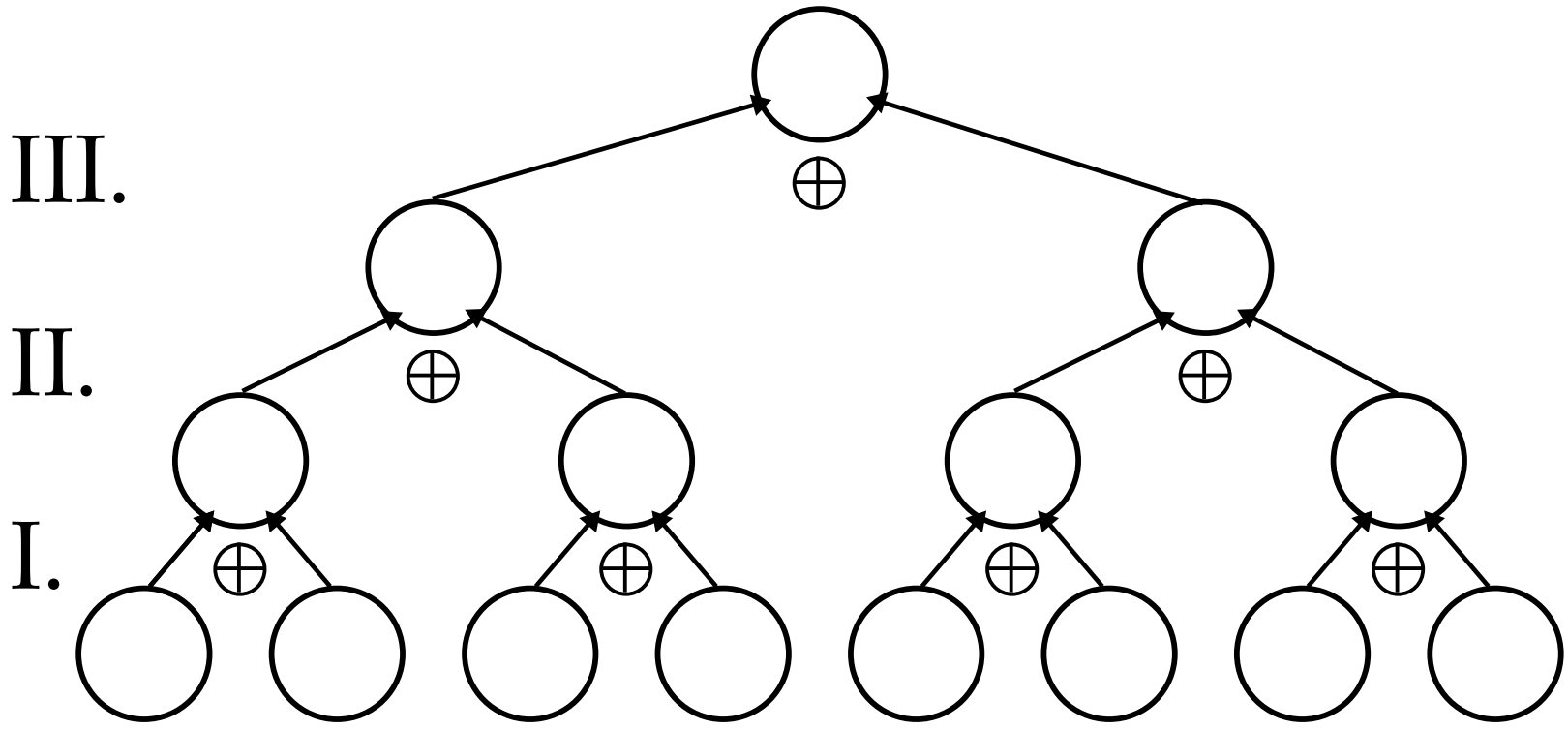
Serieller Algorithmus

function REDUZIERE(a,n)
    $S \leftarrow a[1]$
    for all $i = 2..n$ **do**
        $S \leftarrow S + a[i]$
    end for
end function

Komplexität: $O(n)$.

Beispiel: Parallele Reduktion

Prinzip: $log\,(n)$ Iterationen, paarweise Operationen.



Beispiel: Parallele Reduktion

Vereinfachende Annahme: $n = 2^k, k \in \mathbb{N}$. Wir betrachten den folgenden parallelen Algorithmus auf einer P-Prozessor PRAM. Der Einfachheit halber nehmen wir an, dass $P = n$. Die Funktion wird für P Prozessoren $\mathbf{p} \in [1..P]$ parallel ausgeführt.

```
function ReduzierePRAM(a,n)
    t[p] ← a[p]                                  ▷ Kopie in temp. Array
    for all h = 1..logn do
        if p ≤ n/2^h then
            t[p] ← t[2p − 1] + t[2p]
        end if
    end for
    if p = 1 then
        S ← t[p]
    end if
end function
```

Beispiel: Parallele Reduktion

- Der Algorithmus $\textsc{ReduziereP}\mathrm{RAM}$ ist auf der EREW PRAM ausführbar.
- Die Komplexität bzgl. Laufzeit ist $O(\mathit{logn})$ auf P Prozessoren.
- Die Formulierung auf der PRAM erfordert, dass der Algorithmus explizit im Sinne von P Prozessoren formuliert wird.

Beispiel: Parallele Reduktion

Umformulierung des Algorithmus gemäß Arbeit vs. Zeit Paradigma.

function ReduziereWorkTime(a,n)
  for all $p \in n$ **do in parallel**
    $t[p] \leftarrow a[p]$
  end for
  for all $h = 1..logn$ **do**
    for all $p = 1..n/2^h$ **do in parallel**
      $t[p] \leftarrow t[2p-1] + t[2p]$
    end for
  end for
  $S \leftarrow t[1]$
end function

Beispiel: Parallele Reduktion

Zeitkomplexität: Zähle Zeiteinheiten

for all $p \in n$ **do in parallel** ▷ $O(1)$ Zeiteinheiten
  $t[p] \leftarrow a[p]$
end for
for all $h = 1..logn$ **do** ▷ $O(logn)$ Zeiteinheiten
  for all $p = 1..n/2^h$ **do in parallel** ▷ $O(1)$ Zeiteinheiten
    $t[p] \leftarrow t[2p-1] + t[2p]$
  end for
end for
$S \leftarrow t[1]$ ▷ $O(1)$ Zeiteinheiten

$\Rightarrow S(n) = O(1) + O(logn) \times O(1) + O(1) = O(logn)$ Zeiteinheiten.

Beispiel: Parallele Reduktion

Arbeitskomplexität: Zähle Operationen

for all $p \in n$ **do in parallel** ▷ $O(n)$ Operationen
  $t[p] \leftarrow a[p]$
end for
for all $h = 1..logn$ **do** ▷ $O(logn)$ Operationen
  for all $p = 1..n/2^h$ **do in parallel** ▷ $O(n/2^h)$ Operationen
    $t[p] \leftarrow t[2p-1] + t[2p]$
  end for
end for
$S \leftarrow t[1]$ ▷ $O(1)$ Operationen

$\Rightarrow W(n) = O(n) + O(logn \times O(n/2^h)) + O(1) = O(n)$
Operationen.

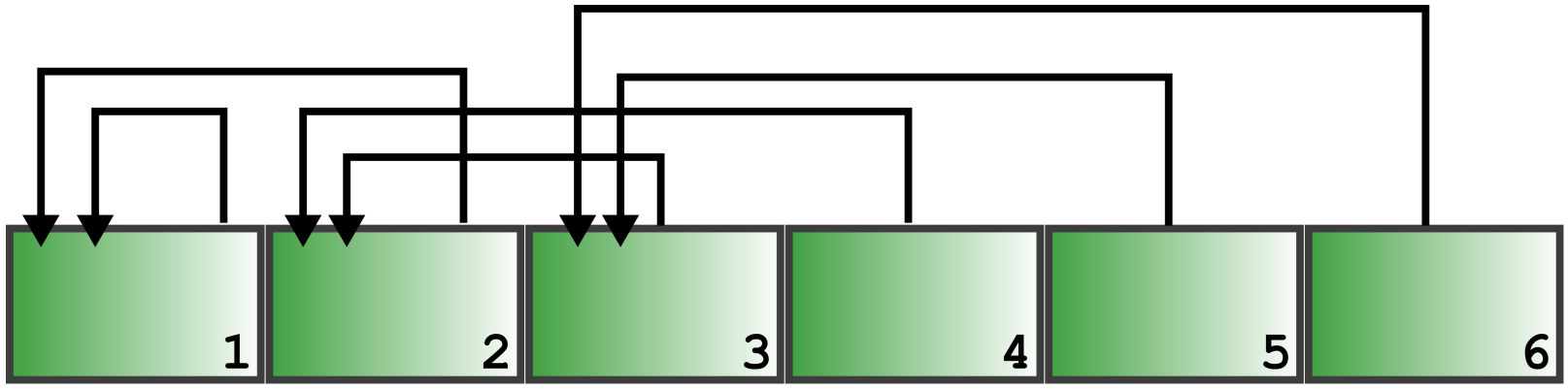
Beispiel: Parallele Reduktion

Bemerkungen

- Die *Zeitkomplexität* beträgt $S(n) := O(logn)$.
- Die *Arbeitskomplexität* beträgt $W(n) := O(n)$.
- Algorithmus ReduziereWorkTime berechnet S auf einer EREW PRAM mit P Prozessoren.
- Der Algorithmus hat eine Besonderheit:
  - Keine der Zuweisungen aus $t\,[p] \leftarrow t\,[2p-1] + t\,[2p]$ kann durchgeführt werden, bevor nicht alle Ausdrücke auf der rechten Seite ausgewertet wurden.
- Was bedeutet das in der Praxis?

Beispiel: Parallele Reduktion

Speicherzugriffe des Algorithmus



Offensichtlich liegt eine Inter Iterations-Datenabhängigkeit vor. Dennoch ist ReduziereWorkTime ein valider EREW Algorithmus. Welche Implikationen ergeben sich bspw. für eine Multi-Threading Implementierung?

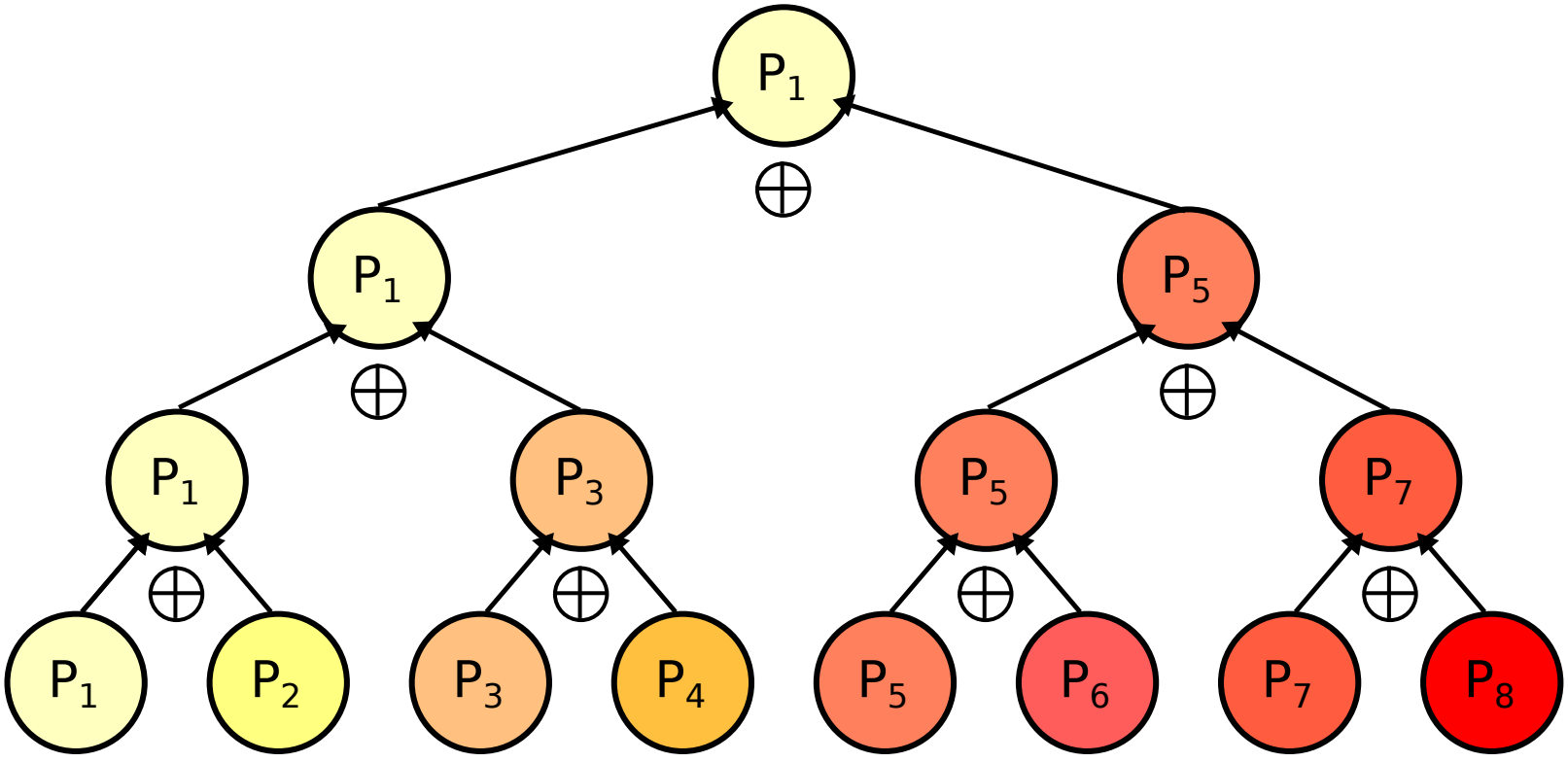
Beispiel: Parallele Reduktion

PRAM Reduktion: Implikationen

- In der Praxis, also bspw. einer Implementierung in C, muss die 2. parallele for Schleife sequentiell abgearbeitet werden, da die Reihenfolge relevant ist, in der Zuweisungen und Additionen ausgeführt werden können.
- Frage: können wir den parallelen Algorithmus so anpassen, dass er auch auf einer Multi-Threading Maschine lauffähig ist?

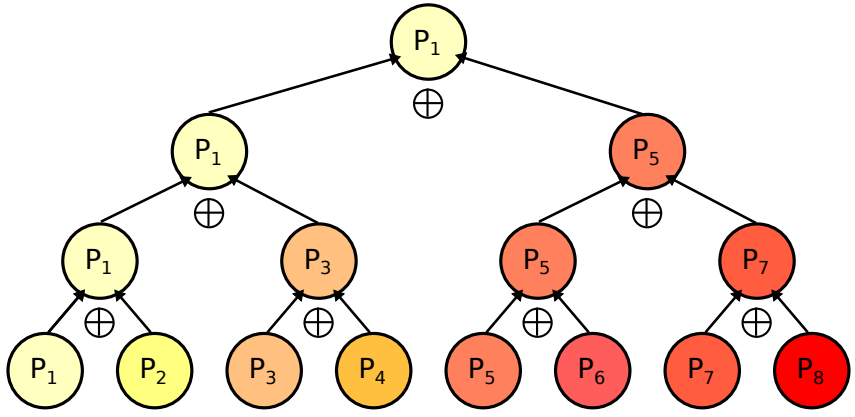
Beispiel: Parallele Reduktion

Shared Memory Datenpfad für Multi-Threading Reduktionsalgorithmus



Beispiel: Parallele Reduktion

Shared Memory Datenpfad für Multi-Threading Reduktionsalgorithmus



Man sieht leicht:
Wir können die Datenabhängigkeit auflösen, indem jeder Prozessor die Eingabedaten von Speicheradressen liest, die eine Distanz von $log2(h-1)$ haben, und das Ergebnis an eine bestimmte der beiden Speicheradressen schreibt.

Beispiel: Parallele Reduktion

Mit dieser einfachen Anpassung könnte der Algorithmus ReduziereWorkTime auf einer Multi-Threading Architektur ausgeführt werden.

function ReduziereWorkTime(a,n)
  for all $p \in n$ **do in parallel**
    $t[p] \leftarrow a[p]$
  end for
  for all $h = 1..logn$ **do**
    for all $p = 1..n, p \leftarrow p + 2^h$ **do in parallel** ▷ <<<
      $t[p] \leftarrow t[p] + t\left[p + 2^{h-1}\right]$ ▷ <<<
    end for
  end for
  $S \leftarrow t[1]$
end function

Beispiel: Parallele Reduktion

Passt man den Algorithmus ReduziereWorkTime leicht an, passt er besser zu einem in der Praxis relevanten Maschinenmodell. Speicher- sowie Zeit- und Arbeitskomplexität ändern sich ggü. dem ersten parallelen Algorithmus nicht.

Generell ist es nicht immer möglich, parallele Algorithmen für das PRAM oder das Arbeit vs. Zeit Modell entsprechend umzuformulieren. Das Beispiel veranschaulich, dass es wichtig ist, die Modelle, die man verwendet, zu hinterfragen. Die Datenabhängigkeit im parallelen Algorithmus führt dazu, dass eine tatsächliche Implementierung in wesentlichen Teilen sequentiell wäre.

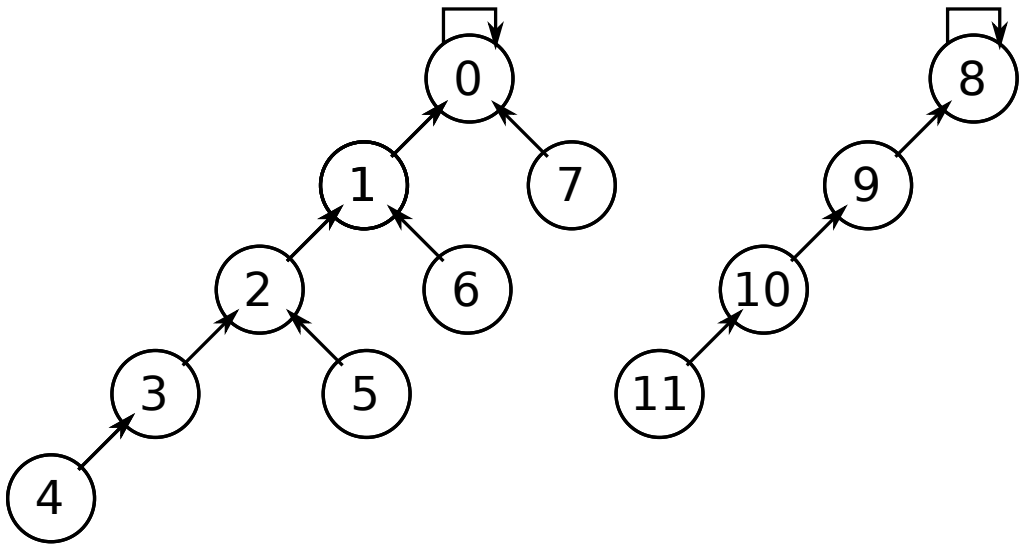
Beispiel: Parallele Reduktion

Bemerkung

Das Reduktionsproblem lässt sich generalisieren, sodass beliebige assoziative Binäroperationen unterstützt werden. Wir suchen dann $S_n := \bigoplus_{i=1}^{n} a[i]$, wobei $\oplus$ eine beliebige assoziative Binäroperation ist, sodass $\bigoplus_{i=1}^{n} a[i] = a[1] \oplus ... \oplus a[n]$.

Beispiel: Pointer Jumping

Gegeben: *Wald* von paarweise disjunkten, wurzelgerichteten Bäumen, d. h. jeder Knoten ist über seinen *Vorgänger* charakterisiert, der Vorgänger der Wurzel ist die Wurzel selbst.

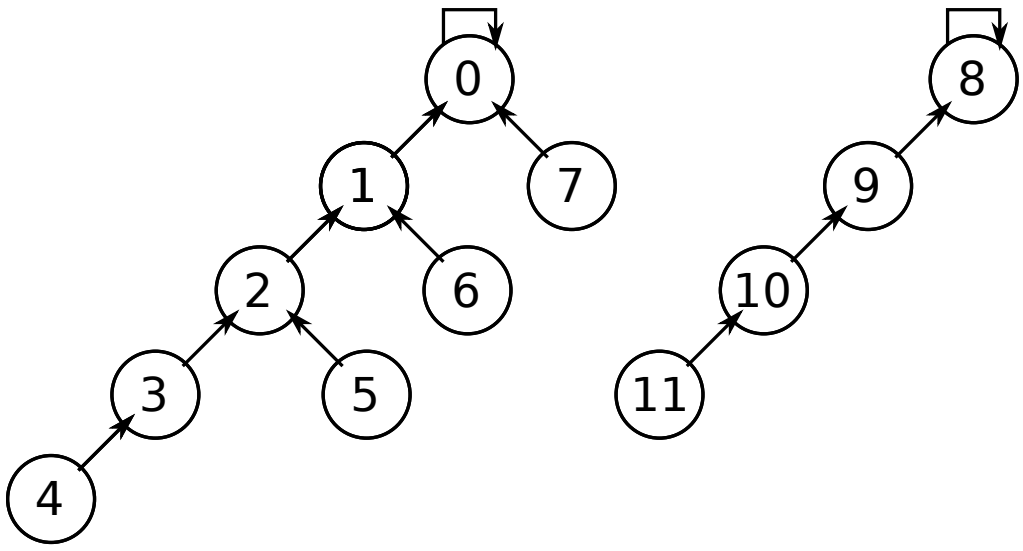


Gesucht: Wurzelknoten jedes Baums.

Beispiel: Pointer Jumping

Baum gegeben als Liste von Vorgängern F, wobei $F[i] = j$, falls j Vorgänger von i:

0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	3	2	1	0	8	8	9	10



Beispiel: Pointer Jumping

Paralleler Algorithmus

Eingabe: Wald von wurzelgerichteten Bäume als Vorgängerliste F der Länge n.
Ausgabe: Liste S der Länge n, sodass $S[i] = j$, falls j die Wurzel des Baumes ist, zu dem i gehört.

```
function PointerJumping
    for all i := 0..n − 1 do in parallel
        S[i] ← F[i]
        while S[i] ≠ S[S[i]] do
            S[i] ← S[S[i]]
        end while
    end for
end function
```

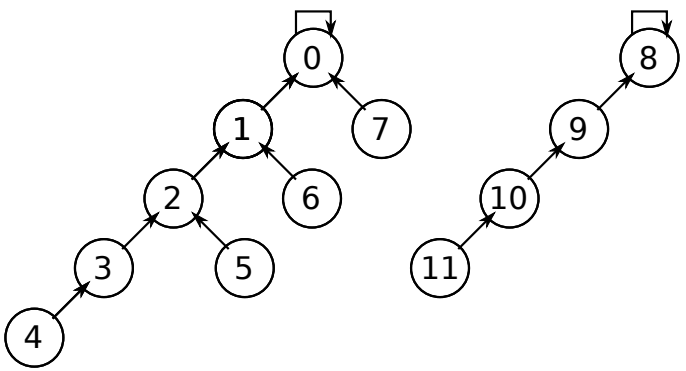
Iteration 1

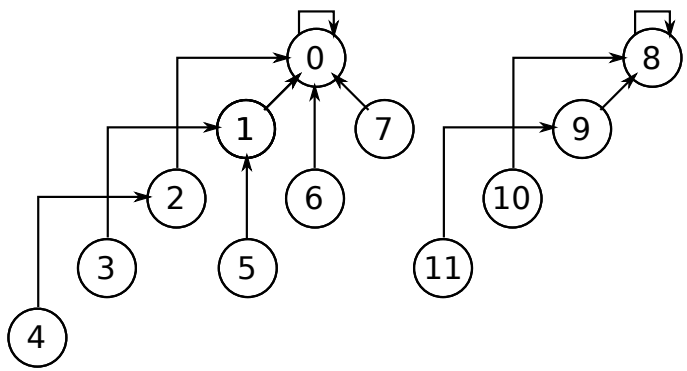
S:

0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	3	2	1	0	8	8	9	10

S':

0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	1	2	1	0	0	8	8	8	9





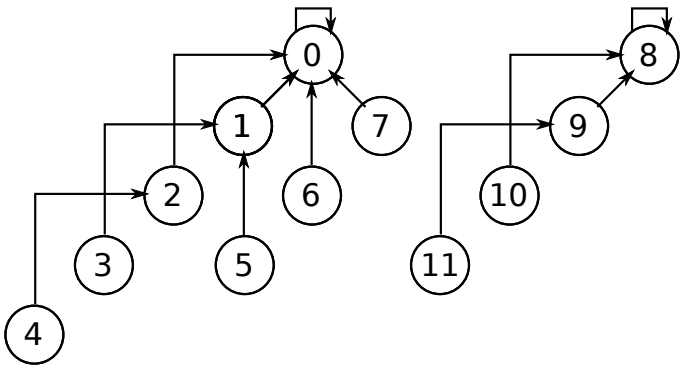
Iteration 2

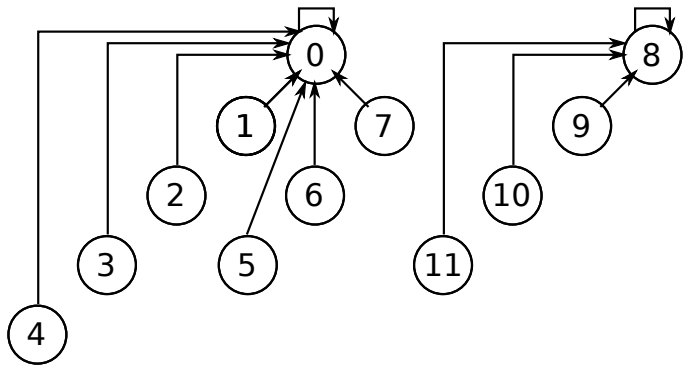
S':

0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	1	2	1	0	0	8	8	8	9

S'':

0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	8	8	8	8





Pointer Jumping Analyse

In der 1. Iteration zeigen alle Knoten auf den Vorgänger 2. Grades. Da dieses Ergebnis in der nächsten Iteration wiederverwertet wird, ermitteln sie im nächsten Schritt nicht den Vorgänger 3. Grades, sondern denjenigen 4. Grades. Im nächsten Schritt ist der Vorgänger dieses Vorgängers der Vorgänger 8. Grades usw. Die Arbeitskomplexität ist $O(n \cdot logh)$, die Zeitkomplexität ist $O(logh)$, wobei h die Höhe des höchsten Baums ist.

Der Algorithmus benötigt concurrent reads und schreibt exklusiv an eine Speicherstelle $\Rightarrow$ CREW PRAM.

Pointer Jumping Bemerkung

Mit Pointer Jumping kann man einfach die *Präfixsumme* für eine Liste von Zahlen parallel bestimmen (Übungsaufgabe!).

Bemerkungen zum PRAM Modell

- PRAM ist ein einfaches Modell, um Zeit- und Arbeitskomplexität eines Algorithmus gegenüberzustellen.
- Annahmen (Speicherzugriffslatenz von 1, unendlich viele Prozessoren) sind allerdings realitätsfern. Algorithmen, die bzgl. des idealisierten Modells effizient sind, sind in der Praxis häufig langsam.
- Konkurrierende Speicherzugriffe (**CREW** und **CRCW**) sind in der Praxis das primäre Bottleneck eines parallelen Algorithmus.

Recap (1)

- Mooresches Gesetz + Dennard Scaling vs. Powerwall. Architekturen werden tendentiell paralleler, keine triviale Erhöhung der Taktfrequenz mehr.
- Performanzbegriff: $\mathit{CPU\ time} = \frac{\#I \times CPI}{Taktfrequenz}$.
  - #I und CPI werden statistisch ermittelt.
  - Durch die Wahl von Algorithmen, Hochsprachenkonstrukten und Compilern kann man #I und CPI beeinflussen.
- Caches: es gibt statischen und dynamischen Speicher in CPU Systemen, es ergeben sich Speicherhierarchien. Beim Programmieren von Speicherhierarchien ist Lokalität sowie Kohärenz von Speicherzugriffen wichtig.

Recap (2)

- Nebenläufigkeit auf verschiedenstem Niveau.
  - ILP schon immer, daher Hardware und Software Pipelining.
  - SIMD: spezielle Register, auf denen eine Operation auf ganzem Vektor anstatt auf Skalar ausgeführt wird.
  - Multi-Threading: Nebenläufigkeit auf Prozessor- und Core Ebene (aber: NUMA Systeme – Divergenz bzgl. Daten- und Instruktionspfad schlecht für Performanz).

Recap (3)

- Hochparallele Architekturen: Instrumente wie Amdahlsches Gesetz, Parallele Effizienz oder Kosten helfen einzuschätzen, wie hoch das Parallelisierungspotential von Algorithmen ist.
- Laufzeiteinschätzungen für parallele Algorithmen kann man mit PRAMs oder mit dem Arbeit vs. Zeit Paradigma durchführen. Diese Modelle treffen womöglich Annahmen, die in der Realität nicht vertretbar sind.

Literaturempfehlungen

- David A. Patterson, John L. Hennessy: Computer Organization and Design, The Hardware/Software Interface, 5th ed. (2014)
- Henri Casanova, Arnaud Legrand, Yves Robert: Parallel Algorithms, 1st ed. (2008)
- Ulrich Drepper: What Every Programmer Should Know About Memory, Technical Report (2007)
- B. Rau, Iterative Modulo Scheduling: An Algorithm for Software Pipelining, Proceedings of the 27th Annual International Symposium on Microarchitecture (1994) Loops (2008)