Funktionale Programmierung

Parallele Funktionale Programmierung

Hans-Wolfgang Loidl, Andreas Abel

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

4. Juni 2009

Grundlagen

Paralleles Rechnen ermöglicht eine schnellere Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren.

Neue Architekturen wie **Multi-core** Maschinen sowie **Grid- und Cloud-Architekturen** machen dieses Thema wieder aktuell.

Bisherige Ansätze für paralleles Rechnen sind sehr "low-level" und schwierig zu handhaben.

Parallele funktionale Sprachen bieten einen hochsprachlichen Ansatz, in dem nur wenige Aspekte der parallelen Berechnung bestimmt werden müssen.

Grundlagen

Im Allgemeinen müssen folgende Aspekte der parallelen Ausführung kontrolliert werden:

- **Partitionierung**: Die Aufspaltung des Programs in unabhängige Teile, die parallel berechnet werden können.
- **Kommunikation**: Die Identifizierung von Datenabhängigkeiten zwischen den Programmen, und der Austausch der Daten.
- **Agglomeration**: Die Verknüpfung mehrerer Berechnungen in Threads, die parallel ausgeführt werden können.
- **Mapping**: Die Zuordnung der Threads zu Prozessoren, die die Auswertung durchführen.
- **Scheduling**: Die Auswahl eines lauffähigen Threads auf einem Prozessor.

Koordinierung

Existierende Sprachen unterscheiden sich stark im Grad der Kontrolle dieser Aspekte.

Manche Sprachen verwenden eigene **Koordinierungssprachen** die diese Aspekte kontrollieren.

Die meisten heute verwendeten Sprachen bieten lediglich Bibliotheken o.ä. mit denen eine explizite Kontrolle möglich ist: z.B. MPI für message passing, oder OpenMP für shared-memory Programmierung.

Parallele Funktionale Sprachen

Rein funktionale Programmiersprachen haben keine Seiteneffekte und sind daher **referentiell transparent**.

Stoy (1977):

> *The only thing that matters about an expression is its value, and any sub-expression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.*

Insbesondere ist die Auswertungsreihenfolge beliebig, und verschiedene Teile des Programms können parallel berechnet werden.

Ansätze zur Parallelität

Basierend auf Haskell werden folgende Ansätze zur Parallelität unterstützt:

- **Semi-explizite Parallelität**: Im Programm muss lediglich die Partitionierung ausgedrückt werden. Die Kontrolle der Parallelität erfolgt automatisch im Laufzeitsystem.
- **Software Transactional Memory:** Die Sprache wird um eine Bibliothek erweitert, die es ermöglicht **spekulativ** parallele Berechnungen durchzuführen. Erst am Ende der Berechnung wird auf mögliche Konflikte mit anderen Berechnungen getestet ("lock-free").
- **Nested Data Parallelism:** Parallelität ist beschränkt auf gleichzeitiges Ausführen einer Operation auf (großen) Datenstrukturen, z.B. mittels Array-comprehensions. Diese können geschachtelt werden.

Semi-explizite Parallelität: GpH

Glasgow parallel Haskell ist eine konservative Erweiterung von Haskell und definiert 1 neues Primitiv: `par`.

`x ‘par‘ e` definiert die parallele Auswertung von `x` und von `e`.

`x ‘seq‘ e` definiert die sequentielle Auswertung von `x` und von `e`.

Beispiel für “pipeline parallelism”: `x ‘par‘ f x`

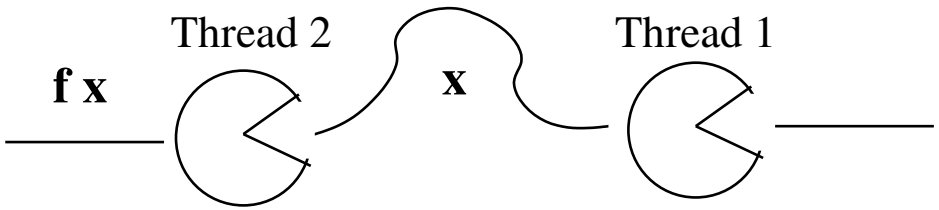
Semi-explizite Parallelität: GpH

Glasgow parallel Haskell ist eine konservative Erweiterung von Haskell und definiert 1 neues Primitiv: `par`.

`x ‘par‘ e` definiert die parallele Auswertung von `x` und von `e`.

`x ‘seq‘ e` definiert die sequentielle Auswertung von `x` und von `e`.

Beispiel für “pipeline parallelism”: `x ‘par‘ f x`



Beispiel: parfact

Wir wollen die *factorial* Funktion parallelisieren, sodass *parfact* 1 *n* $= n!$:

```
parfact :: Integer → Integer → Integer
parfact m n | m ≡ n      = m
            | otherwise  = left `par` right `seq`    -- par dnc version
                           (left * right)
                           where mid = (m + n) `div` 2
                                 left  = parfact m mid
                                 right = parfact (mid + 1) n
```

Beispiel: parfact

Eine Verbesserung des Codes verwendet "thresholding":

```
parfact :: Integer → Integer → Integer → Integer
parfact m n t | (n − m) ⩽ t = product [m . . n]   -- seq version below
              | otherwise   = left `par` right `seq`   -- par dnc versio
                              (left ∗ right)
                              where mid = (m + n) `div` 2
                                    left = parfact m mid t
                                    right = parfact (mid + 1) n t
```

Beispiel: Quicksort

Eine naive Version von Quicksort erzeugt nur geringe Parallelität:

$$
\begin{aligned}
qsort\ [\,] &= [\,] \\
qsort\ [x] &= [x] \\
qsort\ (x : xs) &= qlo\ `par`\ qhi\ `par` \\
&\quad (qlo +\!\!+ (x : qhi)) \\
&\quad \textbf{where}\ qlo = qsort\ [y \mid y \leftarrow xs, y < x] \\
&\qquad\qquad qhi = qsort\ [y \mid y \leftarrow xs, y \geqslant x]
\end{aligned}
$$

Beispiel: quick-sort

Wir müssen die Auswertung der Teillisten anstoßen:

$$
\begin{aligned}
&\mathit{forcelist} :: [a] \to () \\
&\mathit{forcelist}\ [\,] \quad\ = () \\
&\mathit{forcelist}\ (x : xs) = x\ \text{`}\mathit{seq}\text{`}\ \mathit{forcelist}\ xs \\
&\mathit{qsort}\ [\,] \quad\ = [\,] \\
&\mathit{qsort}\ [x] \quad\ = [x] \\
&\mathit{qsort}\ (x : xs) = \mathit{forcelist}\ \mathit{qlo}\ \text{`}\mathit{par}\text{`}\ \mathit{forcelist}\ \mathit{qhi}\ \text{`}\mathit{par}\text{`} \\
&\qquad\qquad\qquad\quad (\mathit{qlo} +\!\!+ (x : \mathit{qhi})) \\
&\qquad\qquad\qquad\quad \textbf{where}\ \mathit{qlo} = \mathit{qsort}\ [y \mid y \leftarrow xs, y < x] \\
&\qquad\qquad\qquad\qquad\quad\ \mathit{qhi} = \mathit{qsort}\ [y \mid y \leftarrow xs, y \geqslant x]
\end{aligned}
$$

Evaluation Strategies

Im Allgemeinen werden "forcing" Funktionen auf vielen verschiedenen Datenstrukturen benötigt.

Evaluation strategies bieten eine Abstraktion der parallelen Koordination, die diese von der Berechnung trennt.

Eine Strategie definiert lediglich Koordination, keinen Wert:

$$\textbf{type}\ \mathit{Strategy}\ a = a \to ()$$

Wir verwenden folgende Funktion um eine Strategie auf einen Ausdruck anzuwenden:

$$\mathit{using} :: a \to \mathit{Strategy}\ a \to a$$
$$\mathit{using}\ x\ s = s\ x\ `\mathit{seq}`\ x$$

Strategien mit Auswertungsgrad

Der Auswertungsgrad einer Datenstruktur kann mit folgenden Strategien definiert werden:

- `r0` führt keine Auswertung durch;
- `rwhnf` führt eine Auswertung zur "weak head normal form" durch (default);
- `rnf` führt eine komplette Auswertung durch.

Zur Definition von `rnf` verwenden wir Typklassen und Overloading.

Strategien mit Parallelität

Wir können nun eine Strategie zur parallelen Auswertung einer Liste wie folgt definieren:

$$
\begin{aligned}
&\mathit{parList} :: \mathit{Strategy}\ a \to \mathit{Strategy}\ [a] \\
&\mathit{parList}\ s\ [\,] \qquad\quad = () \\
&\mathit{parList}\ s\ (x : xs) = s\ x\ \text{`}\mathit{par}\text{`}\ \mathit{parList}\ s\ xs
\end{aligned}
$$

Damit können wir sehr einfach eine parallele Variante von `map` implementieren:

$$
\begin{aligned}
&\mathit{parMap} :: \mathit{Strategy}\ b \to (a \to b) \to [a] \to [b] \\
&\mathit{parMap}\ s\ f\ xs = \mathit{map}\ f\ xs\ \text{`}\mathit{using}\text{`}\ \mathit{parList}\ s
\end{aligned}
$$

Beispiel: Quicksort

Eine Version von Quicksort mit einer Strategie:

$$
\begin{aligned}
qsort\ [\,] &= [\,] \\
qsort\ [x] &= [x] \\
qsort\ xs &= qlo +\!\!+ (x : qhi)\ \text{`}using\text{`}\ strat \\
&\quad \textbf{where}\ qlo = qsort\ [y \mid y \leftarrow xs, y < x] \\
&\qquad\quad qhi = qsort\ [y \mid y \leftarrow xs, y \geqslant x] \\
&\qquad\quad strat = rnf\ qlo\ \text{`}par\text{`} \\
&\qquad\qquad\quad rnf\ qhi\ \text{`}par\text{`} \\
&\qquad\qquad\quad rnf\ x
\end{aligned}
$$

Evaluation Strategies

Mit Strategien können wir die Berechnung von der Koordination trennen.

Strategien spezifizieren:

- Die Auswertungsreihenfolge (mittels *seq*)
- Den Auswertungsgrad (mittels *rnf*, *rwhnf* etc)
- Die Parallelität (mittels *par*)

Beispiel: sumEuler

Die *sumEuler* Funktion summiert die Euler-Funktion, angewendet auf eine Liste von Zahlen. Die Euler Funktion berechnet zu einer gegebenen Zahl, die Anzahl der relativ primen Zahlen:

Given: $n \in \mathbb{N}$
Find: $\Sigma_{i=1}^{n}\ \varphi(i)$, where $\varphi(i) = \mid \{m \in \mathbb{N} \mid m < i \wedge m \perp i\} \mid$
(i.e. the number of values relative prime to i)

Beispiel: sumEuler

Die *sumEuler* Funktion summiert die Euler-Funktion, angewendet auf eine Liste von Zahlen. Die Euler Funktion berechnet zu einer gegebenen Zahl, die Anzahl der relativ primen Zahlen:

$$
\begin{aligned}
&\mathit{euler} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{euler}\ n = \mathit{length}\ (\mathit{filter}\ (\mathit{relprime}\ n)\ [1\,..\,(n-1)]) \\
&\mathit{sumEuler} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{sumEuler} = \mathit{sum} \circ \mathit{map}\ \mathit{euler} \circ \mathit{mkList}
\end{aligned}
$$

sumEuler: Pipeline

Wir können Pipeline-Parallelität wie folgt nutzen:

$$(.||)\ f\ g = \lambda x \rightarrow \textbf{let}\ gx = g\ x\ \textbf{in}\ gx\ `par`\ f\ gx$$

$$\begin{aligned}
&\mathit{euler} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{euler}\ n = \mathit{length}\ (\mathit{filter}\ (\mathit{relprime}\ n)\ [1\,..\,(n-1)])
\end{aligned}$$

$$\begin{aligned}
&\mathit{sumEuler} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{sumEuler} = \mathit{sum}\ .||\ \mathit{map}\ \mathit{euler}\ .||\ \mathit{mkList}
\end{aligned}$$

Allerdings nur geringe Parallelität: 3 Phasen in der Pipeline.

sumEuler: Daten-Parallelität (feinkörnig)

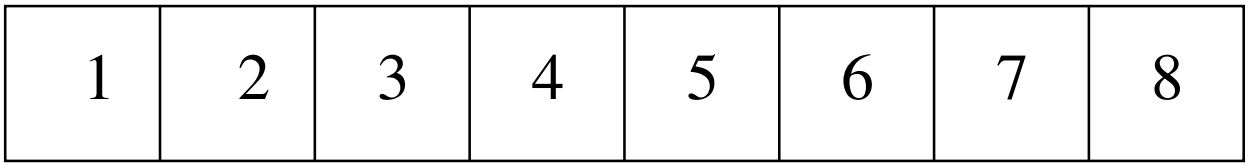
Wir können Daten-Parallelität wie folgt nutzen:

$$
\begin{aligned}
&\mathit{parMap} :: \mathit{Strategy}\ b \to (a \to b) \to [a] \to [b] \\
&\mathit{parMap}\ s\ f\ xs = \mathit{map}\ f\ xs\ \text{`}\mathit{using}\text{`}\ \mathit{parList}\ s \\
&\mathit{euler} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{euler}\ n = \mathit{length}\ (\mathit{filter}\ (\mathit{relprime}\ n)\ [1\,..\,(n-1)]) \\
&\mathit{sumEuler} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{sumEuler} = \mathit{sum} \circ \mathit{parMap}\ \mathit{rnf}\ \mathit{euler} \circ \mathit{mkList}
\end{aligned}
$$

Allerdings sehr feinkörnige Parallelität: jeder Thread berechnet nur *euler* für eine Zahl.

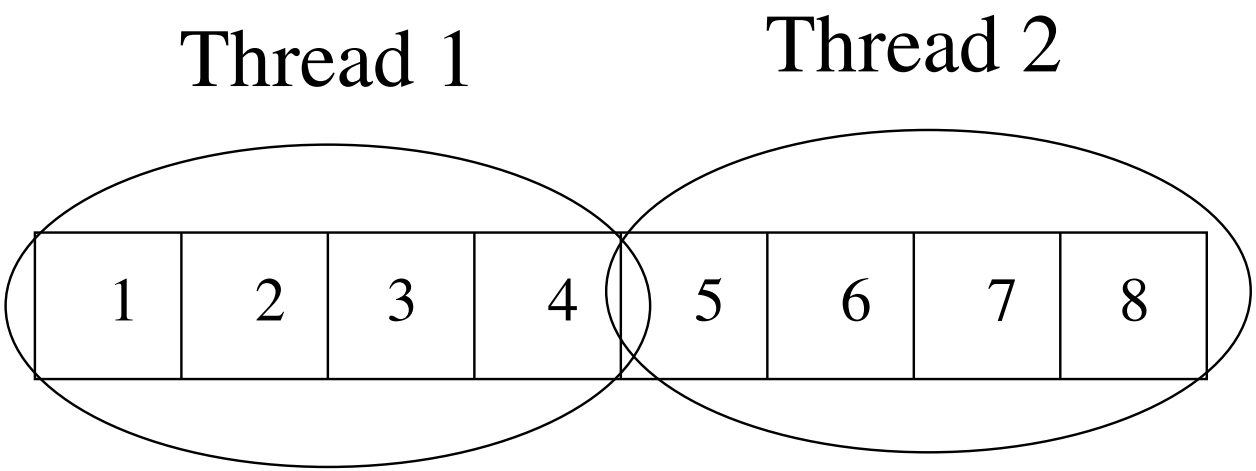
sumEuler: Daten-Parallelität (grobkörnig)

Idee: Gruppiere mehrere Berechnungen von Listenelementen in ein **Cluster** um die Granularität zu erhöhen:



sumEuler: Daten-Parallelität (grobkörnig)

Idee: Gruppiere mehrere Berechnungen von Listenelementen in ein **Cluster** um die Granularität zu erhöhen:



sumEuler: Daten-Parallelität

Wir definieren dafür eine Strategie *parListChunk*:

```
parListChunk :: Int → Strategy a → Strategy [a]
parListChunk c strat [] = ()
parListChunk c strat xs = seqList strat (take c xs) `par`
                          parListChunk c strat (drop c xs)

sumEuler :: Int → Int → Int
sumEuler z n = sum (map euler (mkList n)
                    `using`
                    parListChunk z rnf)
```

Allerdings führt die *sum* Operation zu einer sequentiellen End-Phase.

sumEuler: Daten-Parallelität

Eine verbesserte Version zieht die Summe in die parallelen Threads hinein, um die sequentielle End-Phase zu minimieren:

$$
\begin{aligned}
&\mathit{sumEuler} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{sumEuler}\ z\ n = \mathit{sum}\ ((\mathit{lift}\ \mathit{worker})\ (\mathit{cluster}\ z\ (\mathit{mkList}\ n)) \\
&\qquad\qquad `\mathit{using}`\ \mathit{parList}\ \mathit{rnf}\,) \\
&\quad \textbf{where}\ \mathit{worker} = \mathit{sum} \circ \mathit{map}\ \mathit{euler}
\end{aligned}
$$

Dies verwendet Overloading um *cluster* und *lift* zu definieren.

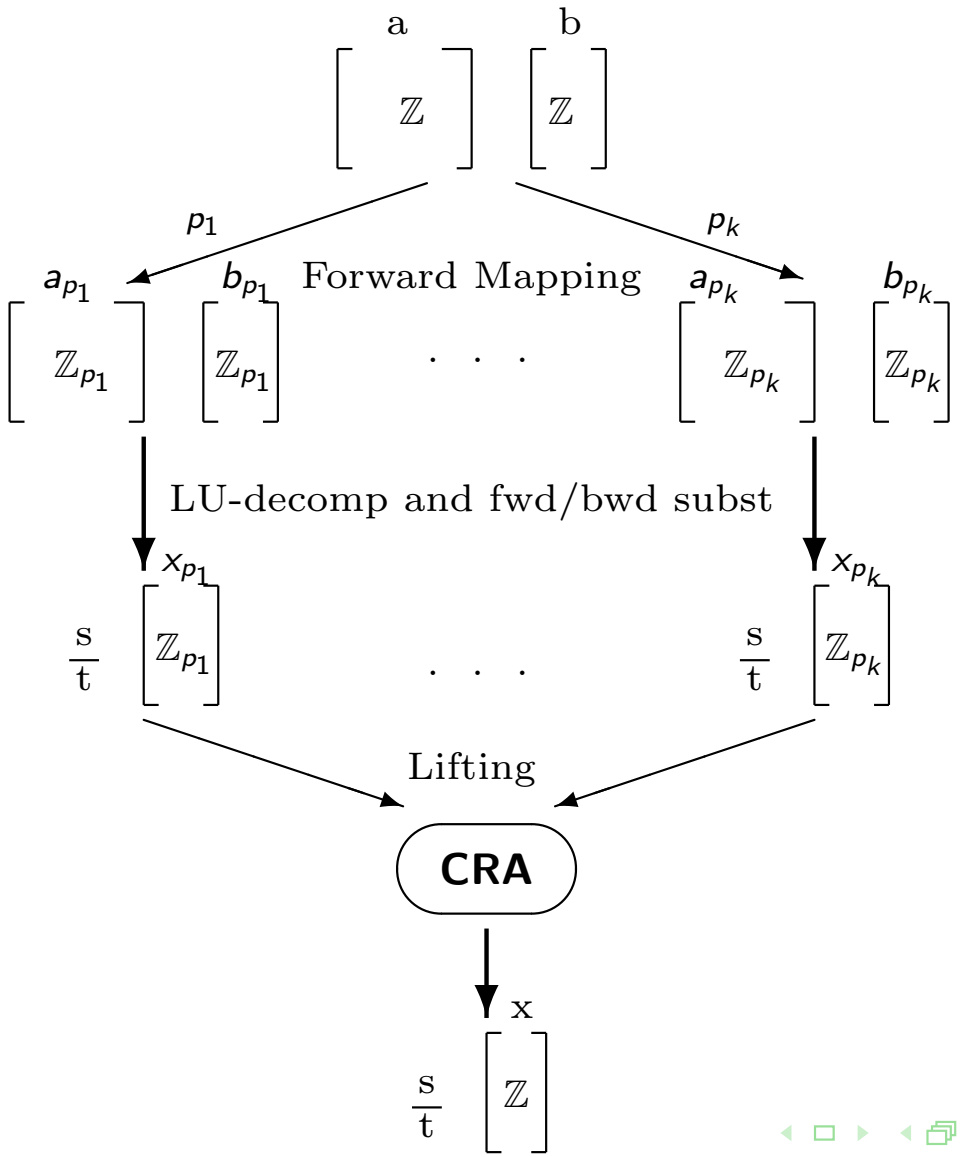
Beispiel: Linear System Solving

Given: $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n, n \in \mathbb{N}$
Find: $x \in \mathbb{Z}^n$, s.t. $Ax = b$

Methode: multiple homomorphe Bilder:

1. bilde die Eingabedaten in mehrere homomorphe Bilder ab,
2. berechne die Lösung in diesen homomorphen Bildern, und
3. kombiniere die Lösung in den homomorphen Bildern zu einer Gesamtlösung.

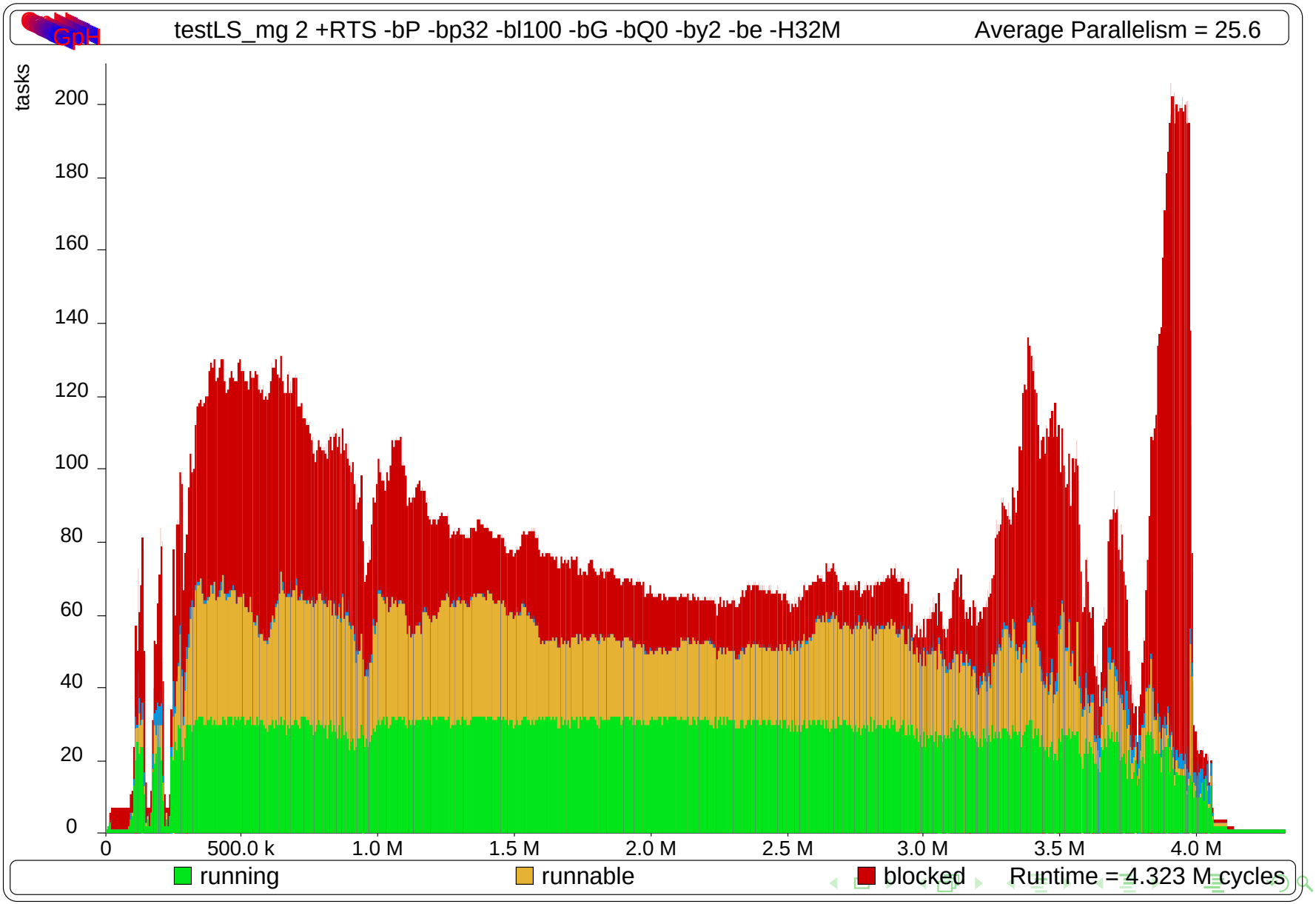
Beispiel: Linear System Solving



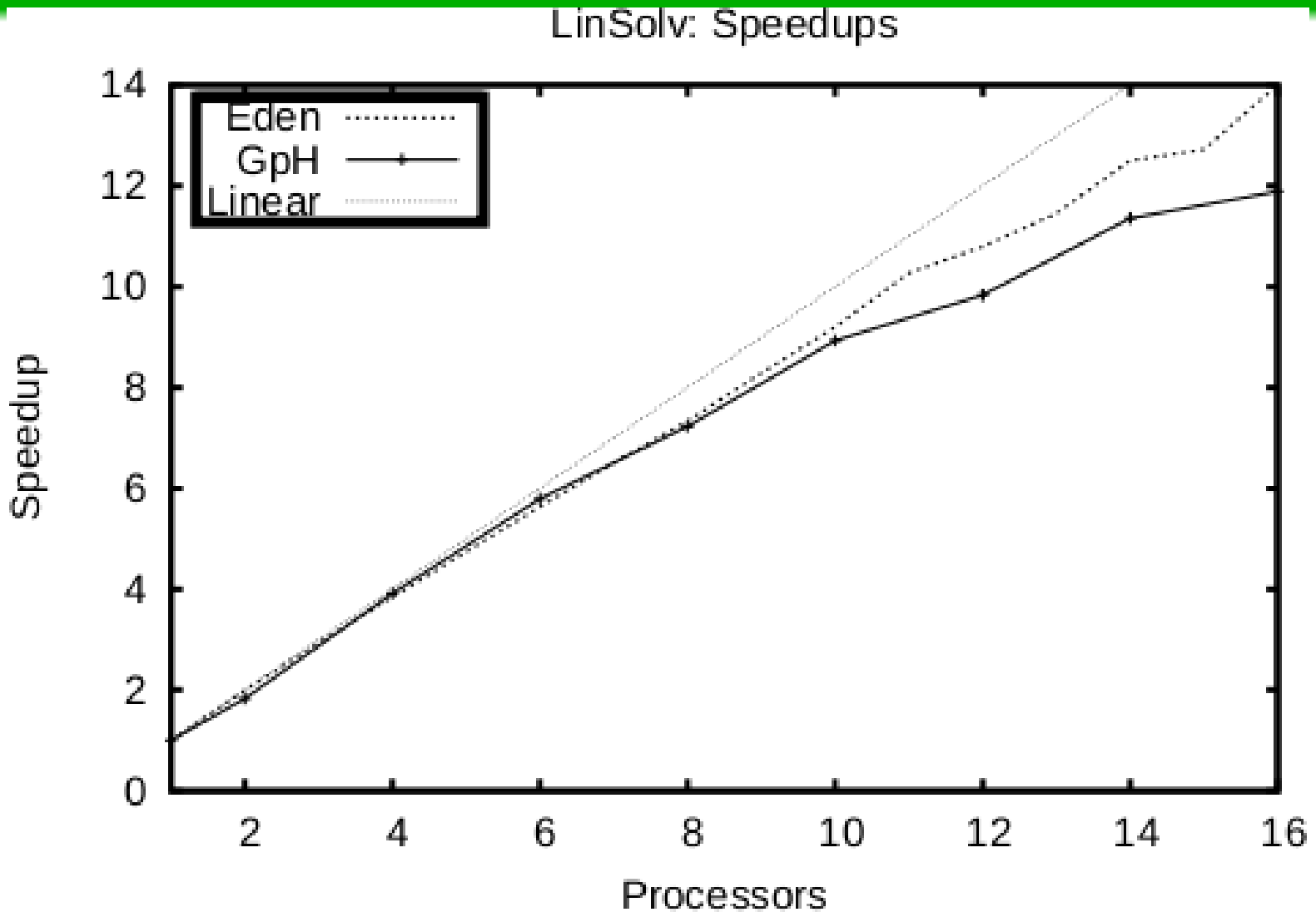
Improved Parallel Strategy

Improved parallelism over the homomorphic images:

```
strat =
λres →
  rnf noOfPrimes                                  `seq`
  parListN noOfPrimes par_sol_strat xList `par`
  parList rnf x
  where par_sol_strat :: Strategy [Integer]
        par_sol_strat = λ(p : modDet : pmx) → rnf modDet `seq`
                                              if modDet ≢ 0
                                                then parList rnf pmx
                                                else ()
```


GpH
testLS_mg 2 +RTS -bP -bp32 -bl100 -bG -bQ0 -by2 -be -H32M
Average Parallelism = 25.6
tasks
200
180
160
140
120
100
80
60
40
20
0
0
500.0 k
1.0 M
1.5 M
2.0 M
2.5 M
3.0 M
3.5 M
4.0 M
running
runnable
blocked
Runtime = 4.323 M cycles

Speedups on a 16 Processor Network



Zusammenfassung: GpH

- GpH bietet ein Modell **semi-expliziter** Parallelität.
- Autoren: P.W. Trinder, K. Hammond, H-W. Loidl, Simon L. Peyton Jones.
- Es müssen lediglich Programmteile zur parallelen Auswertung annotiert werden.
- Die Verwaltung der Parallelität erfolgt zur Gänze im Laufzeitsystem.
- "Evaluation strategies" erlauben es Koordination und Berechnung sauber zu trennen.
- Dadurch kann rein funktionaler Code mit nur wenigen Änderungen parallelisiert werden.

Explizite Parallelität

Mittels `par` kann rein funktionaler Code einfach parallelisiert werden.

In großen Anwendung findet man aber oft zustandsbasierten (monadischen) Code, den man parallel abarbeiten will.

Dazu wird **Software Transactional Memory** verwendet.

Concurrent Haskell

Concurrent Haskell bietet Bibliotheksfunktionen zum Erzeugen und zur Kontrolle von nebenläufigen Berechnungen (IO-Thread).

Im Gegensatz zu GpH sind diese IO-Threads explizite Objekte, die im Code kontrolliert werden.

`forkIO :: IO () -> IO ThreadId` erzeugt einen IO-Thread, der mittels `ThreadId` identifiziert wird.

MVars

Zur Kommunikation zwischen IO-Threads werden MVar, synchronisierte shared-memory Variablen, verwendet

- `newEmptyMVar :: IO (MVar a)` erzeugte eine neue MVar.
- `newMVar :: IO MVar` erzeugte eine uninitialisierte MVar.
- `takeMVar :: MVar a -> IO a` nimmt eine MVar, wenn verfügbar
- `putMVar :: MVar a -> a -> IO ()` gibt eine MVar zurück.
- `readMVar :: MVar a -> IO a` liest den Wert einer MVar
- `tryTakeMVar :: MVar a -> IO (Maybe a)` non-blocking takeMVar
- `tryPutMVar :: MVar a -> a -> IO Bool` non-blocking putMVar
- `isEmptyMVar :: MVar a -> IO Bool` testet ob MVar leer ist

Beispiel für MVars

```
module Main where
import Control.Concurrent
import Control.Concurrent.MVar

threadA :: MVar Int → MVar Float → IO ()
threadA valueToSendMVar valueReceiveMVar
 = do    -- some work
         -- now perform rendevous by sending 72
       putMVar valueToSendMVar 72   -- send value
       v ← takeMVar valueReceiveMVar
       putStrLn (show v)

threadB :: MVar Int → MVar Float → IO ()
threadB valueToReceiveMVar valueToSendMVar
 = do    -- some work
         -- now perform rendevous by waiting on value
       z ← takeMVar valueToReceiveMVar
       putMVar valueToSendMVar (1.2 * z)
         -- continue with other work
```

Beispiel für MVars: Rendevous

```
main :: IO ()
main = do aMVar ← newEmptyMVar
          bMVar ← newEmptyMVar
          forkIO (threadA aMVar bMVar)
          forkIO (threadB aMVar bMVar)
          threadDelay 1000   -- wait for threadA and threadB to finish (s
```

Software Transactional Memory

Software Transactional Memory (STM) erlaubt die Ausführung von IO-Threads auf gemeinsamen Variablen (TVars) ohne diese bei Beginn der Benutzung zu sperren ("lock-free").

Am Ende der Benutzung wird getestet ob Konflikte auftraten.

In dem Fall wird die Berechnung "zurückgespult".

Dazu müssen alle Operationen auf TVars in der STM Monade ausgeführt werden.

STM Operationen

```
data STM a   -- A monad supporting atomic memory transactions
atomically :: STM a → IO a   -- Perform a series of STM actions atomicall
retry      :: STM a          -- Retry current transaction from the beginni
orElse     :: STM a → STM a → STM a   -- Compose two transactions

data TVar a   -- Shared memory locations that support atomic memory op
newTVar  :: a → STM (TVar a)      -- Create a new TVar with an initial
readTVar  :: TVar a → STM a       -- Return the current value
writeTVar :: TVar a → a → STM ()  -- Write the supplied value
```

Beispiel für STM

```
   -- in thread 1:
atomically (do v ← readTVar acc
               writeTVar acc (v + 1))
   -- in thread 2:
atomically (do v ← readTVar acc
               writeTVar acc (v − 3))
```

Weitere Funktionen der STM Monade

Die Funktion *retry* ermöglicht es ein roll-back zu erzwingen. Dies ist hilfreich wenn im Programm erkannt wird, dass eine Transaktion nicht erfolgreich abgeschlossen werden kann.

Beispiel: abheben von einem leeren Konto

```
withdraw :: TVar Int → Int → STM ()
withdraw acc n = do
  bal ← readTVar acc
  if bal < n
    then retry
    else writeTVar acc (bal − n)
```

Weitere Funktionen der STM Monade

Die Funktion *orElse* ermöglicht es eine Auswahl ("choice") zwischen 2 STM Transaktionen zu implementieren. Wenn eine Transaktion fehlschlägt, so wird die andere ausgeführt. Nur wenn beide Transaktionen fehlschlagen, schlägt auch die gesamte Transaktion fehl.

Beispiel: abheben von einem von zwei Konten

```
withdraw2 :: TVar Int → TVar Int → TVar Int → Int → STM ()
withdraw2 acc1 acc2 acc3 n = do
  (withdraw acc1 n
    `orElse`
   withdraw acc2 n)
  deposit acc3 n
```

Zusammenfassung STM

- STM bietet ein Modell **expliziter Parallelität**.
- Autoren: Tim Harris, Simon L. Peyton Jones.
- STM eignet sich zur Parallelisierung von monadischem Code
- Es ermöglicht die Erstellung von “lock-free” Code auf gemeinsamen Variablen (TVars)
- Die Implementierung führt ein “log”, erkennt mögliche Konflikte und spult, wenn nötig, Berechnungen zurück

Nested Data Parallelism

Im "**nested data parallelism**" Ansatz wird nur eine Form von Parallelität unterstützt: Datenparallelität, d.h. eine Funktion wird parallel auf eine Menge (meist Liste) von Daten angewendet.

Dies schränkt die Art der Parallelität ein.

Durch die Möglichkeit, Datenparallelität zu verschachteln, erhält man dennoch mächtige Konstrukte zum Beschreiben paralleler Berechnungen.

Weiters bietet Datenparallelität ein einfaches Kostenmodell, das bei der Vorhersage der parallelen Performanz hilft.

Beispiel für Datenparallelität

Summe der Quadrate:

$$sumSq :: [:Float:] \to Float$$
$$sumSq\ a = sumP\ [:x * x \mid x \leftarrow a:]$$

Vektor Multiplikation:

$$vecMul :: [:Float:] \to [:Float:] \to Float$$
$$vecMul\ a\ b = sumP\ [:x * y \mid x \leftarrow a \mid y \leftarrow b:]$$

Pros und Cons von Datenparallelität

- Für eine n Prozessor Maschine werden n Threads erzeugt, die jeweils ein Segment der Liste berechnen
- Dadurch erzeugt man gute Granularität (Größe der Berechnung)
- Da immer benachbarte Listenelemente zusammengefasst werden, erzeugt man gute Datenlokalität.
- Da genauso viele Threads erzeugt werden wie Prozessoren vorhanden sind, ist die Lastverteilung sehr gut.

Beachte:

- Parallele Listen haben eine strikte Semantik.
- Diese Implementierung ist zunächst nur für flache Datenparallelität möglich.

Beispiel für verschachtelte Datenparallelität

$$\textbf{type } \mathit{Vector} = [:\mathit{Float}:]$$
$$\textbf{type } \mathit{Matrix} = [:\mathit{Vector}:]$$
$$\mathit{matMul} :: \mathit{Matrix} \to \mathit{Vector} \to \mathit{Vector}$$
$$\mathit{matMul}\ m\ v = [:\mathit{vecMul}\ r\ v \mid r \leftarrow m:]$$

Beachte: *vecMul* ist selbst auch daten-parallel.

Beispiel: Sieb des Eratosthenes

$$
\begin{aligned}
&\mathit{primesUpTo} :: \mathit{Int} \to [:\mathit{Int}:] \\
&\mathit{primesUpTo}\ 1 = [::] \\
&\mathit{primesUpTo}\ 2 = [:2:] \\
&\mathit{primesUpTo}\ n = \mathit{smallers} +\!: +\!: \\
&\qquad\qquad [:x \mid x \leftarrow [:\mathit{ns} + 1 \,..\, n:] \\
&\qquad\qquad\quad , \neg\ (\mathit{anyP}\ (`\mathit{divides}`x)\ \mathit{smallers}):] \\
&\qquad\qquad \textbf{where}\ \mathit{ns} \quad\quad = \mathit{intSqrt}\ n \\
&\qquad\qquad\qquad\quad \mathit{smallers} = \mathit{primesUpTo}\ \mathit{ns}
\end{aligned}
$$

Beachte: Die Schachtelungs-Tiefe der Daten-Parallelität ist nun dynamisch bestimmt.

Beispiel: Quicksort

$$
\begin{aligned}
&\mathit{qsort} :: [:\mathit{Double}:] \to [:\mathit{Double}:] \\
&\mathit{qsort}\ \mathit{xs} \mid \mathit{lengthP}\ \mathit{xs} \leqslant 1 = \mathit{xs} \\
&\qquad\quad \mid \mathit{otherwise} \qquad\quad = \mathit{rs}\,!:0 +: +\, \mathit{eq} +: +\, \mathit{rs}\,!:1 \\
&\qquad\quad \textbf{where}\ p = \mathit{xs}\,!:(\mathit{lengthP}\ \mathit{xs}\ `\mathit{div}`\ 2) \\
&\qquad\qquad\qquad \mathit{lt} = [:x \mid x \leftarrow \mathit{xs}, x < p:] \\
&\qquad\qquad\qquad \mathit{eq} = [:x \mid x \leftarrow \mathit{xs}, x \equiv p:] \\
&\qquad\qquad\qquad \mathit{gt} = [:x \mid x \leftarrow \mathit{xs}, x > p:] \\
&\qquad\qquad\qquad \mathit{rs} = \mathit{mapP}\ \mathit{qsort}\ [:\mathit{lt}, \mathit{gt}:]
\end{aligned}
$$

Beachte: Der divide-and-conquer Algorithmus wird durch Daten-Parallelität auf der Liste $[:\mathit{lt}, \mathit{gt}:]$ ausgedrückt.

Data Parallel Haskell

- Data Parallel Haskell bietet ein Modell **impliziter** Parallelität.
- Data Parallel Haskell ist eine Haskell Erweiterung, die verschachtelte Datenparallelität (in obiger Syntax) erlaubt.
- Autoren: Manuel Chakravarty, Gabriele Keller.
- Für alle gängigen higher-order Funktionen gibt es parallele Varianten (*mapP*, *zipWithP*, etc).
- Kern der Implementierung ist die Umwandlung von verschachtelter Daten-Parallelität in flache Daten-Parallelität ("vectorisation"), im 2 Phasen
  - Umwandlung von verschachtelten Arrays in flache Arrays mit primitiven Elementen. Dabei sollen verwandte Daten in benachbarten Speicherstellen zu finden sein.
  - Entsprechende Umwandlung des Codes ("code vectorisation"). Um Zwischen-Datenstrukturen zu vermeiden, wird Code-Fusion verwendet.
- Dies beruht auf Arbeiten von Guy Blelloch zu NESL.

Literatur

- S.L. Peyton Jones, S. Singh, "A Tutorial on Parallel and Concurrent Programming in Haskell" (see `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/`)
- P.W. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones. "Algorithm + Strategy = Parallelism." *J. of Functional Programming*, 8(1):23–60, Jan 1998. (see `http://www.macs.hw.ac.uk/~dsg/gph/papers/`)
- P.W. Trinder, H-W. Loidl, R. Pointon. "Parallel and Distributed Haskells." *J. of Functional Programming*, 12(5):469–512, Jul 2002.
- GHC Users Guide, `http://www.haskell.org/ghc/docs/latest/html/users_guide`