

Nutzung paralleler Prozesse bei der Umweltsimulation

RALF Wieland

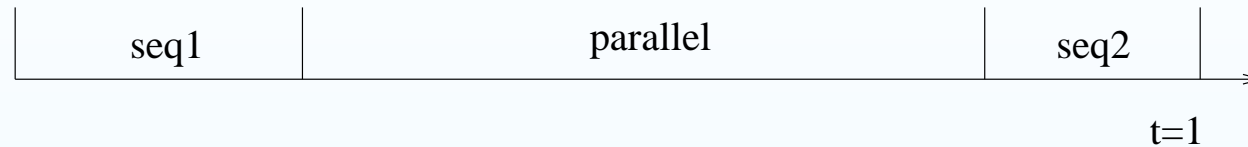
`rwieland@zalf.de`

ZALF/LSA

Warum parallele Prozesse?

- Die Steigerung der Taktfrequenz von Prozessoren ist zunehmend ein thermisches Problem (8086: $f < 5$ MHz, heute: $f > 3$ GHz)
- Viele Cores in einem Prozessor sind billig
- Umweltsimulationen basieren oft auf zeitaufwändigen stochastischen Komponenten

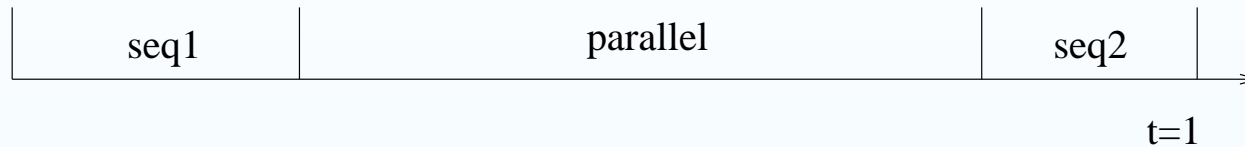
Amdahlsche Gesetz



- Ein Problem kann nie vollständig parallelisiert werden:
 $t = 1 = Seq + P = (1 - P) + P$

Für ein gegebenes Problem gibt es eine optimale Anzahl von Prozessoren!

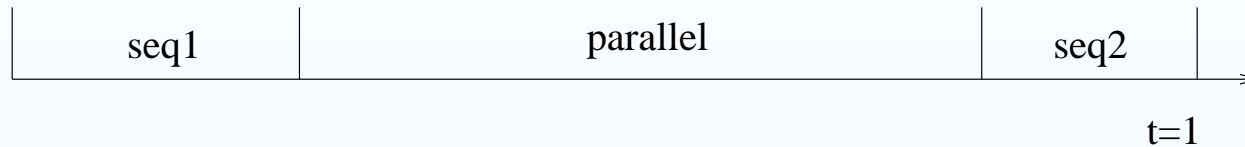
Amdahlsche Gesetz



- Ein Problem kann nie vollständig parallelisiert werden:
 $t = 1 = Seq + P = (1 - P) + P$
- Speedup S bei N Prozessoren: $S = \frac{1}{(1-P) + \frac{P}{N}} \leq \frac{1}{1-P}$

Für ein gegebenes Problem gibt es eine optimale Anzahl von Prozessoren!

Amdahlsche Gesetz



- Ein Problem kann nie vollständig parallelisiert werden:

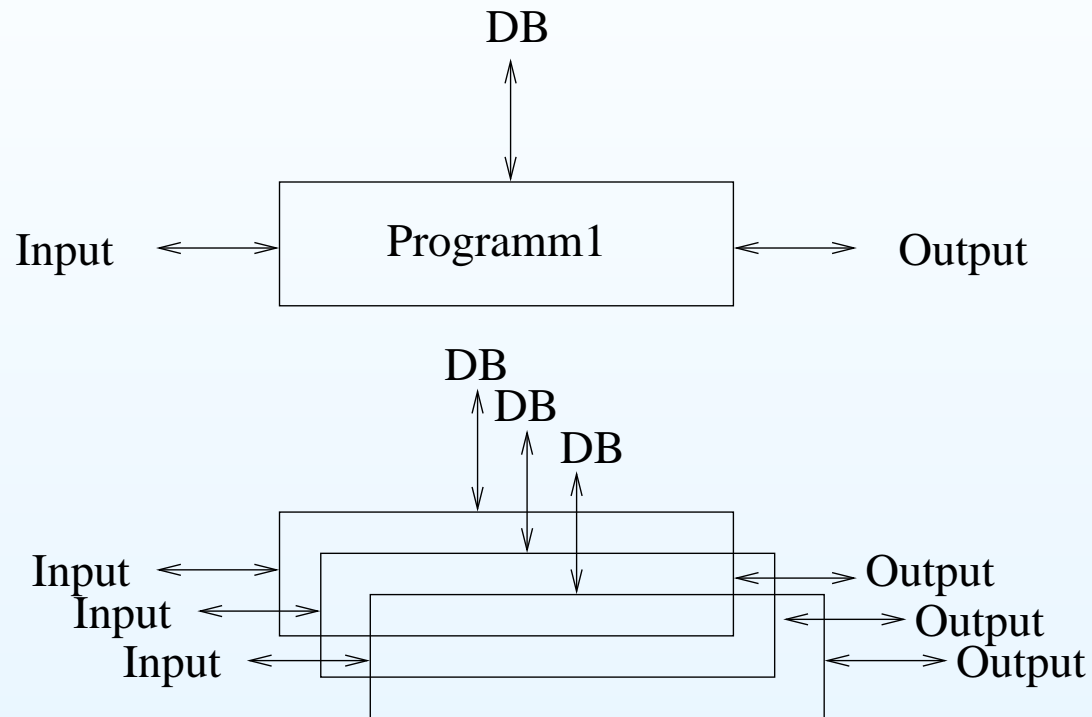
$$t = 1 = Seq + P = (1 - P) + P$$

- Speedup S bei N Prozessoren: $S = \frac{1}{(1-P) + \frac{P}{N}} \leq \frac{1}{1-P}$

- Kommunikationskosten $o(N)$: $S = \frac{1}{(1-P) + o(N) + \frac{P}{N}} \leq \frac{1}{1-P}$

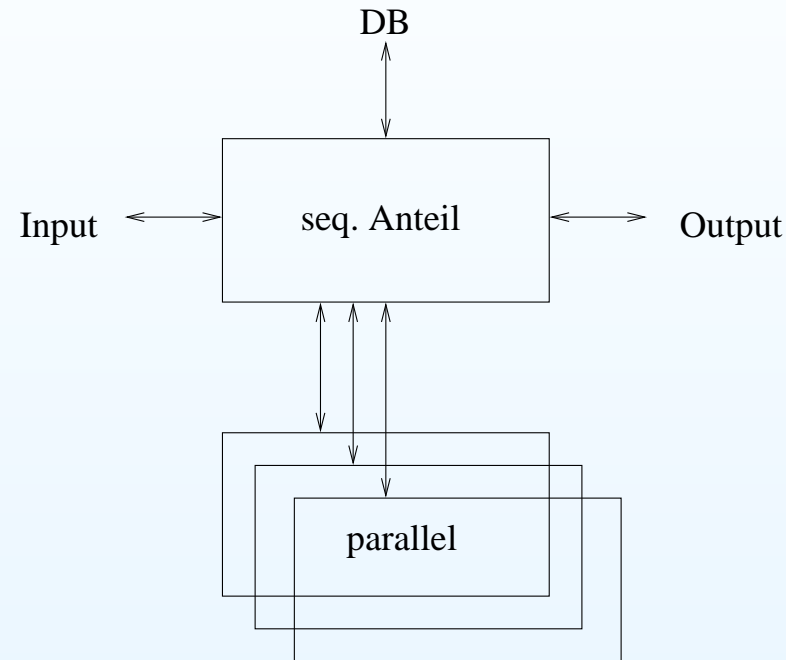
Für ein gegebenes Problem gibt es eine optimale Anzahl von Prozessoren!

Erster primitiver Ansatz



Bei 8 Prozessoren: Rechenzeit von 22.83s auf 6.14s gesenkt (S=3.72) aber so geht das nicht! Offene Connections zum Filesystem und zur Datenbank sind der Flaschenhals.

Trennung in sequenziellen und parallelen Part



Der sequenzielle Part übernimmt die Kommunikation, der parallele Anteil die Kalkulation. Problem: Synchronisation der Prozesse: Wann ist wer fertig, wann schreibt wer auf die Platte?

Python als Bindeglied zum Prozess

- Python ist eine gut strukturierte Sprache

Python als Bindeglied zum Prozess

- Python ist eine gut strukturierte Sprache
- Python bietet eine unüberschaubare Zahl an Modulen (matplotlib, numeric)

Python als Bindeglied zum Prozess

- Python ist eine gut strukturierte Sprache
- Python bietet eine unüberschaubare Zahl an Modulen (matplotlib, numeric)
- Python bietet eine ausgezeichnete Basis zur Nutzerkommunikation (qt, tk)

Python als Bindeglied zum Prozess

- Python ist eine gut strukturierte Sprache
- Python bietet eine unüberschaubare Zahl an Modulen (matplotlib, numeric)
- Python bietet eine ausgezeichnete Basis zur Nutzerkommunikation (qt, tk)
- C/C++ Code kann einfach in Python integriert werden (SWIG) - Python dient dann als Steuersprache für Modelle.

Python als Bindeglied zum Prozess

- Python ist eine gut strukturierte Sprache
- Python bietet eine unüberschaubare Zahl an Modulen (matplotlib, numeric)
- Python bietet eine ausgezeichnete Basis zur Nutzerkommunikation (qt, tk)
- C/C++ Code kann einfach in Python integriert werden (SWIG) - Python dient dann als Steuersprache für Modelle.
- Python bietet eine gute Anbindung an OpenMPI (pypar, mpi4py)

Einfaches Beispiel in Python I

```
#!/usr/bin/env python
```

```
import grid
```

```
import pypar
```

```
# control
```

```
numproc = pypar.size() # Number of processes as specified by mpirun
```

```
myid = pypar.rank() # Id of of this process (0..n-1)
```

```
node = pypar.get_processor_name() # Host name current process
```

Aufruf: `mpirun -np 4 pypar1.py`

Einfaches Beispiel in Python II

```
# parallel part: fill it with random values
if myid==0:
    g1=grid.grid(100,100)
    g1.rand_value()
    g1.write_ascii("g1.asc")
if myid==1:
    g2=grid.grid(100,100)
    g2.rand_value()
    g2.write_ascii("g2.asc")
    ...
if myid==3:
    g4=grid.grid(100,100)
    g4.rand_value()
    g4.write_ascii("g4.asc")
```

S=2, aber mpirun -np 4 verbraucht zur Intialisierung ca. 1.3s
Rechenzeit. Problem: Konkurrenz um Harddisk.

Synchronisation durch Kommunikation

Prozesse laufen in unterschiedlichen Adressräumen.
Kommunikation zwischen den Prozessen dient:

- dem Austausch von Informationen (Parameter und Ergebnisse werden als Nachrichten ausgetauscht)
- der Synchronisation von Prozessen (warten auf eine Nachricht)

Synchronisation Example

```
if myid==2:
    g2.set_value(1.0)
    g2.mul_value(4.0)
    x=pypar.receive(0)
    print "write_g2:%d" %(x)
    g2.write_ascii("g2.asc")
    pypar.send(2,0)

...

if myid==0:
    for id in range(1,numproc-1):
        x=pypar.receive(id)
        print "prozess_%d:%d" %(id,x)
        pypar.send(id,id+1)

...
```

Erzwinge rein sequenzielle Abspeicherung (S=3.14)

Zusammenfassung prozessbasierte Programmierung

- OpenMPI erlaubt Prozesse auf einem und lokalen oder verteilten Rechner zu verwalten

Prozesse sollten möglichst viel rechnen und wenig interagieren.

Zusammenfassung prozessbasierte Programmierung

- OpenMPI erlaubt Prozesse auf einem und lokalen oder verteilten Rechner zu verwalten
- Prozesse laufen in einem Prozesskontext und interagieren via Messages

Prozesse sollten möglichst viel rechnen und wenig interagieren.

Zusammenfassung prozessbasierte Programmierung

- OpenMPI erlaubt Prozesse auf einem und lokalen oder verteilten Rechner zu verwalten
- Prozesse laufen in einem Prozesskontext und interagieren via Messages
- Es gibt keine Speicherinteraktion zwischen Prozessen

Prozesse sollten möglichst viel rechnen und wenig interagieren.

Zusammenfassung prozessbasierte Programmierung

- OpenMPI erlaubt Prozesse auf einem und lokalen oder verteilten Rechner zu verwalten
- Prozesse laufen in einem Prozesskontext und interagieren via Messages
- Es gibt keine Speicherinteraktion zwischen Prozessen
- Messages tragen Informationen (Parameter, Ergebnisse etc.)

Prozesse sollten möglichst viel rechnen und wenig interagieren.

Zusammenfassung prozessbasierte Programmierung

- OpenMPI erlaubt Prozesse auf einem und lokalen oder verteilten Rechner zu verwalten
- Prozesse laufen in einem Prozesskontext und interagieren via Messages
- Es gibt keine Speicherinteraktion zwischen Prozessen
- Messages tragen Informationen (Parameter, Ergebnisse etc.)
- Messages können zur Synchronisation eingesetzt werden

Prozesse sollten möglichst viel rechnen und wenig interagieren.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern
- sehr schwierig zu programmieren

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern
- sehr schwierig zu programmieren
- OpenMP als Industriestandard

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern
- sehr schwierig zu programmieren
- OpenMP als Industriestandard
- es werden z.B. Schleifen parallelisiert

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern
- sehr schwierig zu programmieren
- OpenMP als Industriestandard
- es werden z.B. Schleifen parallelisiert
- Einsatzgebiet ist vor allem die interaktive Graphikprogrammierung

OpenMP kann gut mit OpenMPI verbunden werden.

Thread basierte Programmierung

- Threads realisieren Parallelität im gemeinsamen Speicher
- vermeidet den Overhead der Interprozesskommunikation auf Kosten der Resistenz von Programmen gegenüber Programmierfehlern
- sehr schwierig zu programmieren
- OpenMP als Industriestandard
- es werden z.B. Schleifen parallelisiert
- Einsatzgebiet ist vor allem die interaktive Graphikprogrammierung

OpenMP kann gut mit OpenMPI verbunden werden.

Beispiel in OpenMP

```
#pragma omp parallel
{
#pragma omp for private(j)

    for(i = 0; i < stt->nrows; i++) {
        for(j = 0; j < stt->ncols; j++) {
            corr->feld[i][j] = 0.0;
        }
    }
    ...
}
```

Führt im Beispiel auf ein $S=1.2$

Nachteilig: Race Condition müssen erkannt und mit “#pragma omp critical” geschützt werden.

Beispiel einer Race Condition

```
#pragma omp parallel
{
#pragma omp for private(j)
  for(int i=0; i<corine->nrows; i++){
    for(int j=0; j<corine->ncols; j++){
      if((int)corine->feld[i][j]==corine->nodata){
        evaporation->feld[i][j]=evaporation->nodata;
        infiltration->feld[i][j]=infiltration->nodata;
        continue;
      }
      gx.eta=gx.infil=0.0;
      xstein=(int)(steino->feld[i][j]+1);
      ...
    }
  }
}
```

Beispiel einer Race Condition mit pragma

```
#pragma omp parallel
{
#pragma omp for private(j)
  for(int i=0; i<corine->nrows; i++){
    for(int j=0; j<corine->ncols; j++){
      if((int) corine->feld[i][j]==corine->nodata){
        evaporation->feld[i][j]=evaporation->nodata;
        infiltration->feld[i][j]=infiltration->nodata;
        continue;
      }
      gx.eta=gx.infil=0.0;
      #pragma omp critical
      xstein=(int)(steino->feld[i][j]+1);
      ...
    }
  }
}
```

Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler

Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler
- OpenMP kann durch die Pragmas auch oft auf Einprozessormaschinen kompiliert werden

Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler
- OpenMP kann durch die Pragmas auch oft auf Einprozessormaschinen kompiliert werden
- kann sehr schnellen Code erzeugen, da keine Nachrichten ausgetauscht werden müssen (Netzwerk)

Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler
- OpenMP kann durch die Pragmas auch oft auf Einprozessormaschinen kompiliert werden
- kann sehr schnellen Code erzeugen, da keine Nachrichten ausgetauscht werden müssen (Netzwerk)
- kann gezielt zur Schleifenoptimierung eingesetzt werden

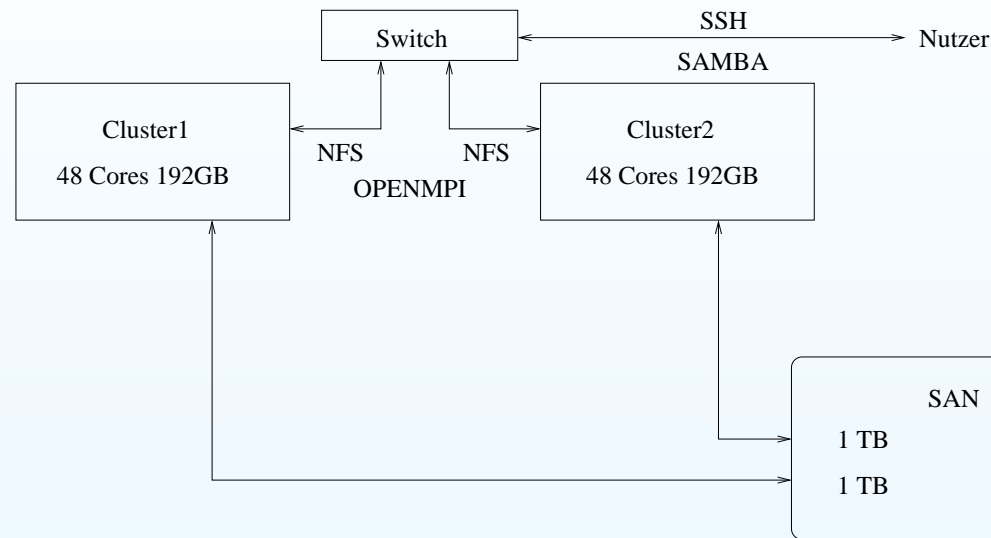
Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler
- OpenMP kann durch die Pragmas auch oft auf Einprozessormaschinen kompiliert werden
- kann sehr schnellen Code erzeugen, da keine Nachrichten ausgetauscht werden müssen (Netzwerk)
- kann gezielt zur Schleifenoptimierung eingesetzt werden
- nichttriviale Programmierung, zum Teil sehr schwierig

Zusammenfassung Threads

- Threads bedürfen einer Unterstützung durch den Compiler
- OpenMP kann durch die Pragmas auch oft auf Einprozessormaschinen kompiliert werden
- kann sehr schnellen Code erzeugen, da keine Nachrichten ausgetauscht werden müssen (Netzwerk)
- kann gezielt zur Schleifenoptimierung eingesetzt werden
- nichttriviale Programmierung, zum Teil sehr schwierig
- verantwortungsvoll eingesetzt kann OpenMP sehr gute Speedups erreichen

Aufbau des Zalf-Clusters



- 2 redundante Rechner mit 48 Cores und 192 GB
- Nutzer: SAMBA, LDAP und SSH
- 2*1TB Zwischenspeicher aus dem SAN
- OS: Debian Lenny Linux, OpenMPI, OpenMP, gcc, python

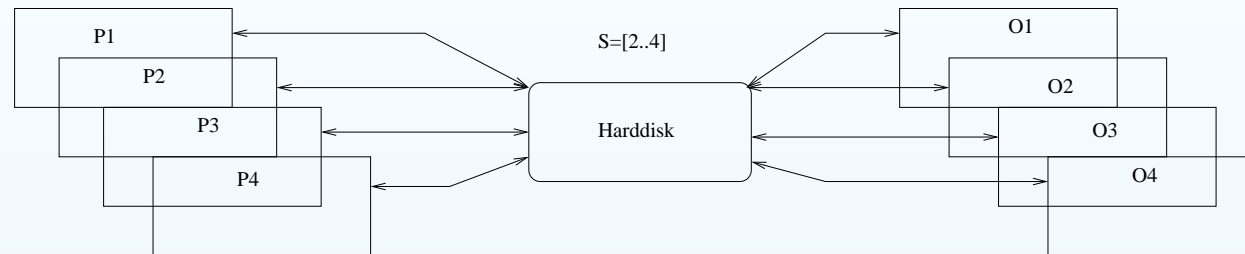
HPCC-Benchmark + Gerris

Type	4Proc	16Proc	32Proc
FFT	2.23865Gf	6.89493Gf	10.3655Gf
Copy	6658MB/s	9456MB/s	11998MB/s
Scale	6378MB/s	8445MB/s	11834MB/s
Add	6785MB/s	9569MB/s	13900MB/s
Triad	6882MB/s	10189MB/s	13441MB/s

Gerris	1Proc	4Proc	16Proc	32Proc
Time	134.58s	26.124s	9.369s	6.433s
Speedup	1	5.15	14.36	20.92

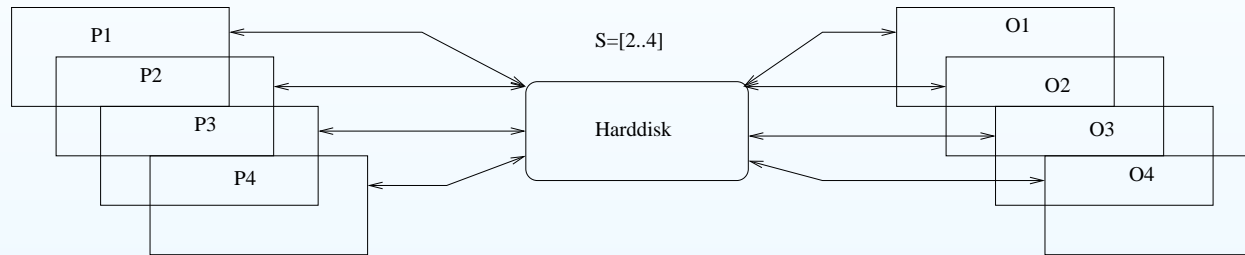
Parallelisierungsstrukturen

- Optimierung mit Hopspack

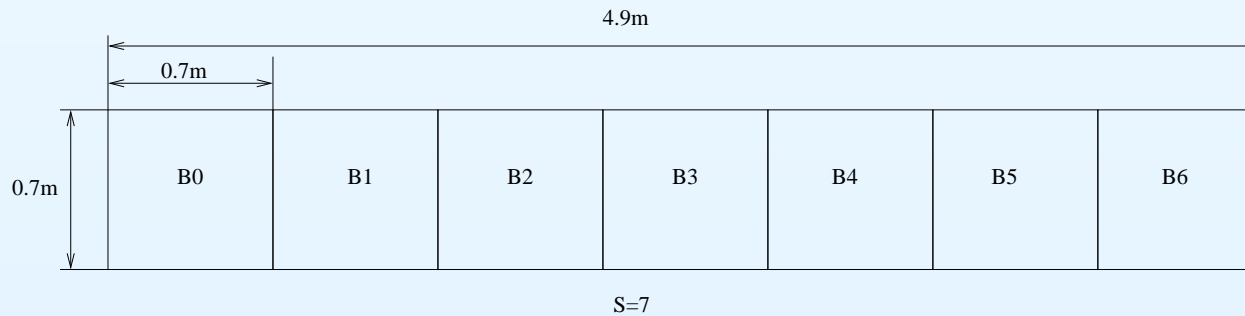


Parallelisierungsstrukturen

- Optimierung mit Hopspack

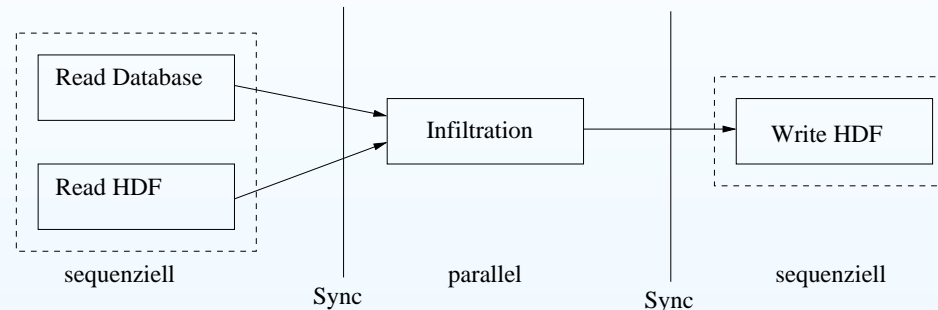


- CFD



Abimo als Beispiel

ABIMO (AbflussBildungsMOdell) nach Bagrov-Verfahren



- Read Routinen müssen sequenziell (Eimerkettenverfahren) abgearbeitet werden
- Infiltration wird parallel verarbeitet
- Schreibroutinen wieder sequenziell

Gridsize= 565600 t=10 Jahre : $t_{iter}=28.2s$ $t_{sum}=8.6s$ $S=3.3$
($t_{seq}=0.34s$, $t_{par}=2.9s$)

Zusammenfassung

- Parallelisierung birgt Potenziale ist aber auch anspruchsvoller als sequenzielle Programmierung.
- Nur ein problemspezifisches und durchdachtes Design der Software führt zum Erfolg.
- Think!