Parallele Programmierung mit OpenMP und MPI

oder

Wieso früher alles besser war

Robin Geyer

08.10.2010

Fahrplan

Einführung / Umfrage

Parallelrechner

OpenMP

(Open)MPI

Fazit

Folien

In ein paar Tagen

- `pinguinstall.org` oder
- `nrtm.de` oder
- `clug.de` ?

Umfrage I

Hände hoch!

Programmiersprache

- Welche Programmiersprachen werden benutzt?

  C / C++ FORTRAN
  Java Python .NET

Umfrage II

Multicore System

- Wer hat mehr als 2 Cores in seinem Rechner?
- Wer hat den Eindruck das dies wirklich was bringt?

Parallele Programmierung

- Wer hat schon parallel Programmiert?
  - Mit (Open)MPI
  - Mit OpenMP
  - Andere Methode

Geschichte der Workstations/PCs I

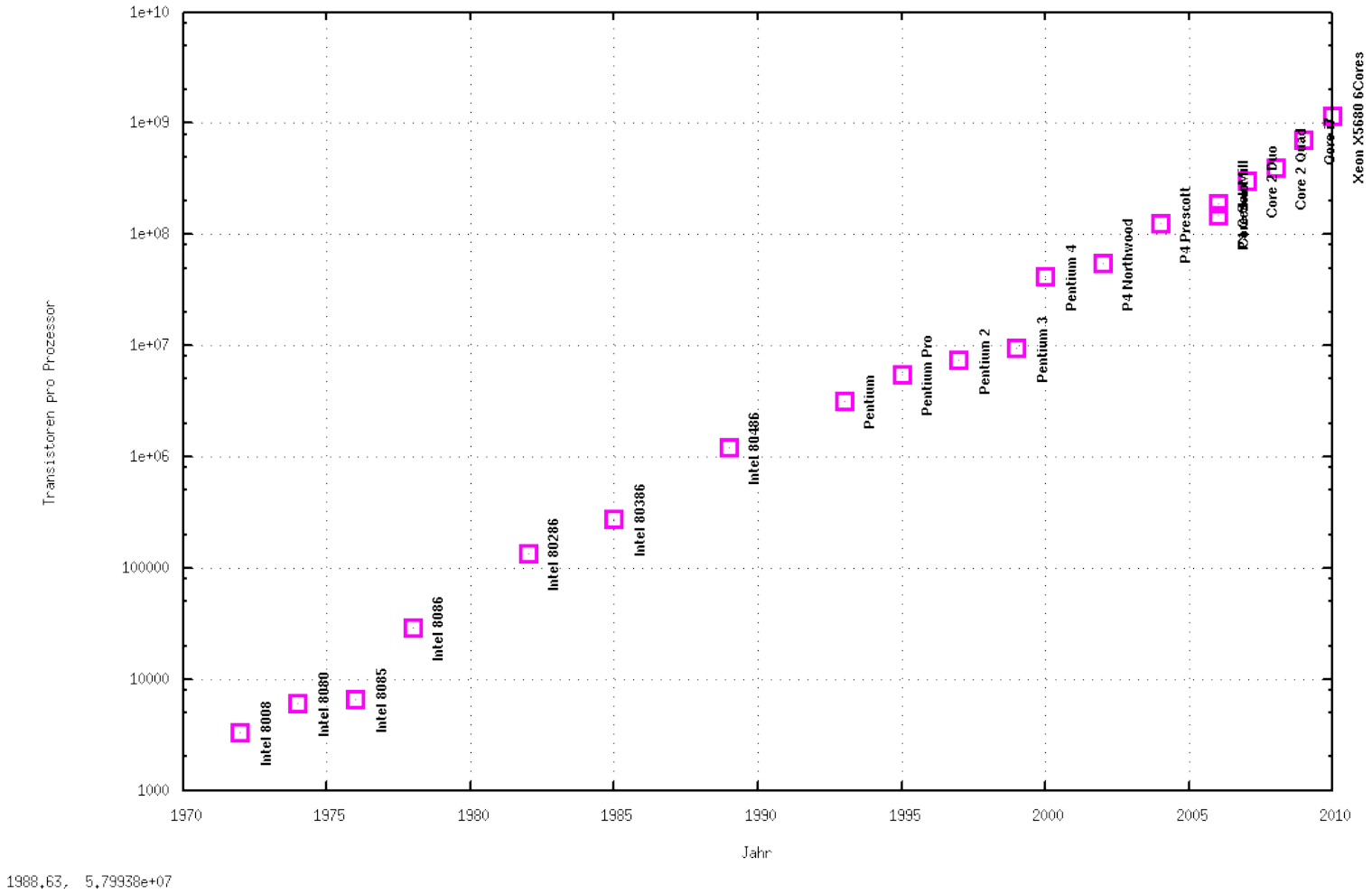


Figure: Moores Law

Geschichte der Workstations/PCs II

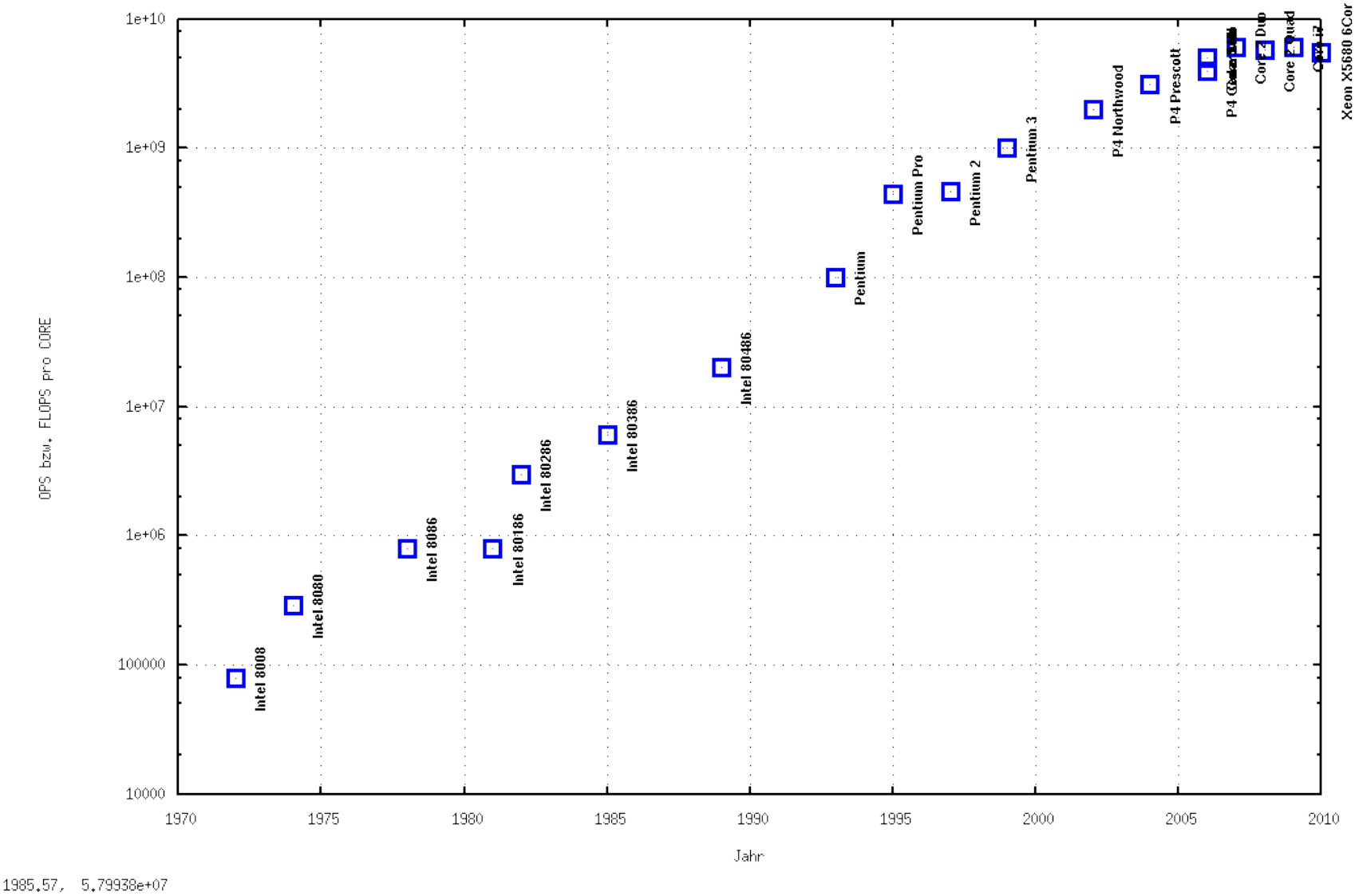


Figure: Leistung pro CPU-**Core**

Geschichte der Supercomputer I

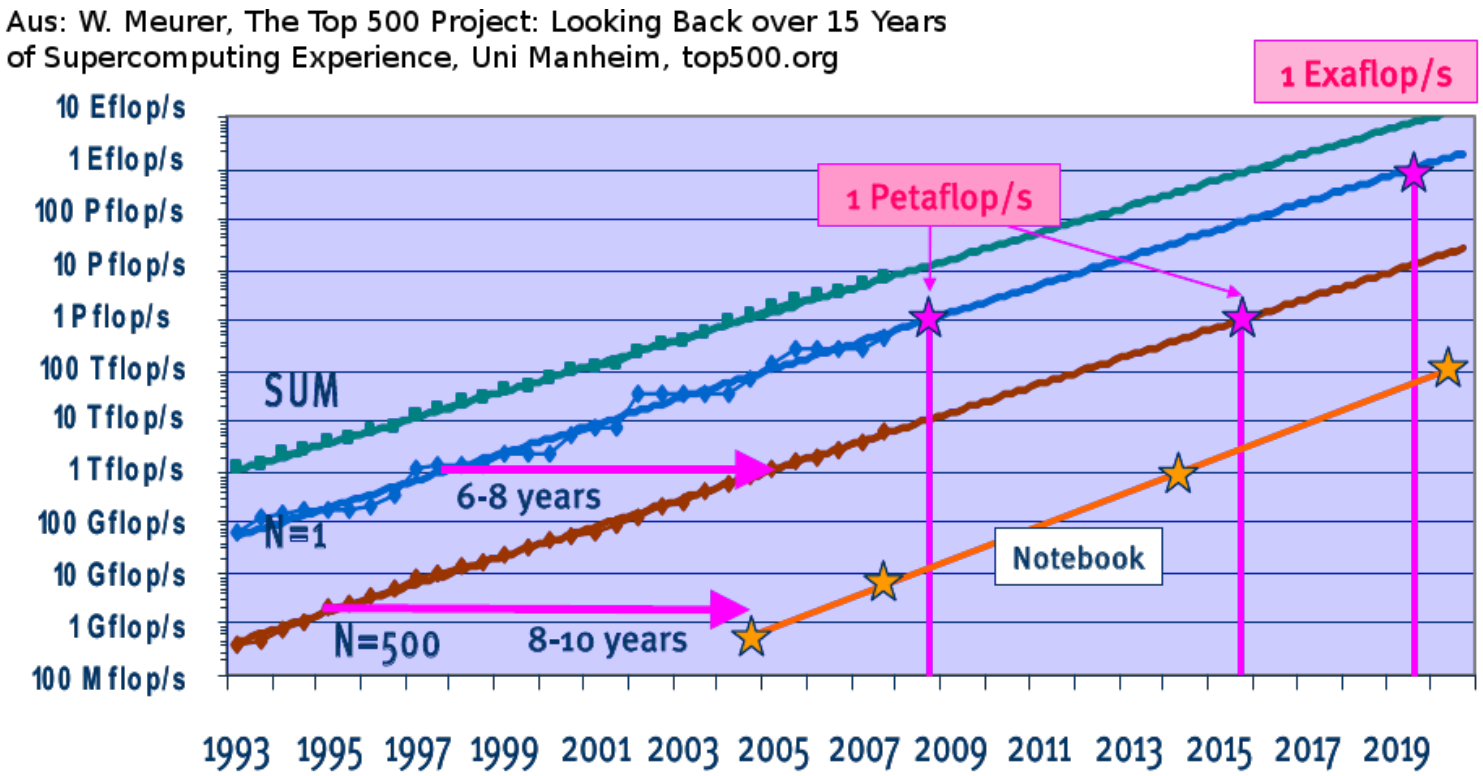


Figure: Entwicklung der Top500

Geschichte der Supercomputer II

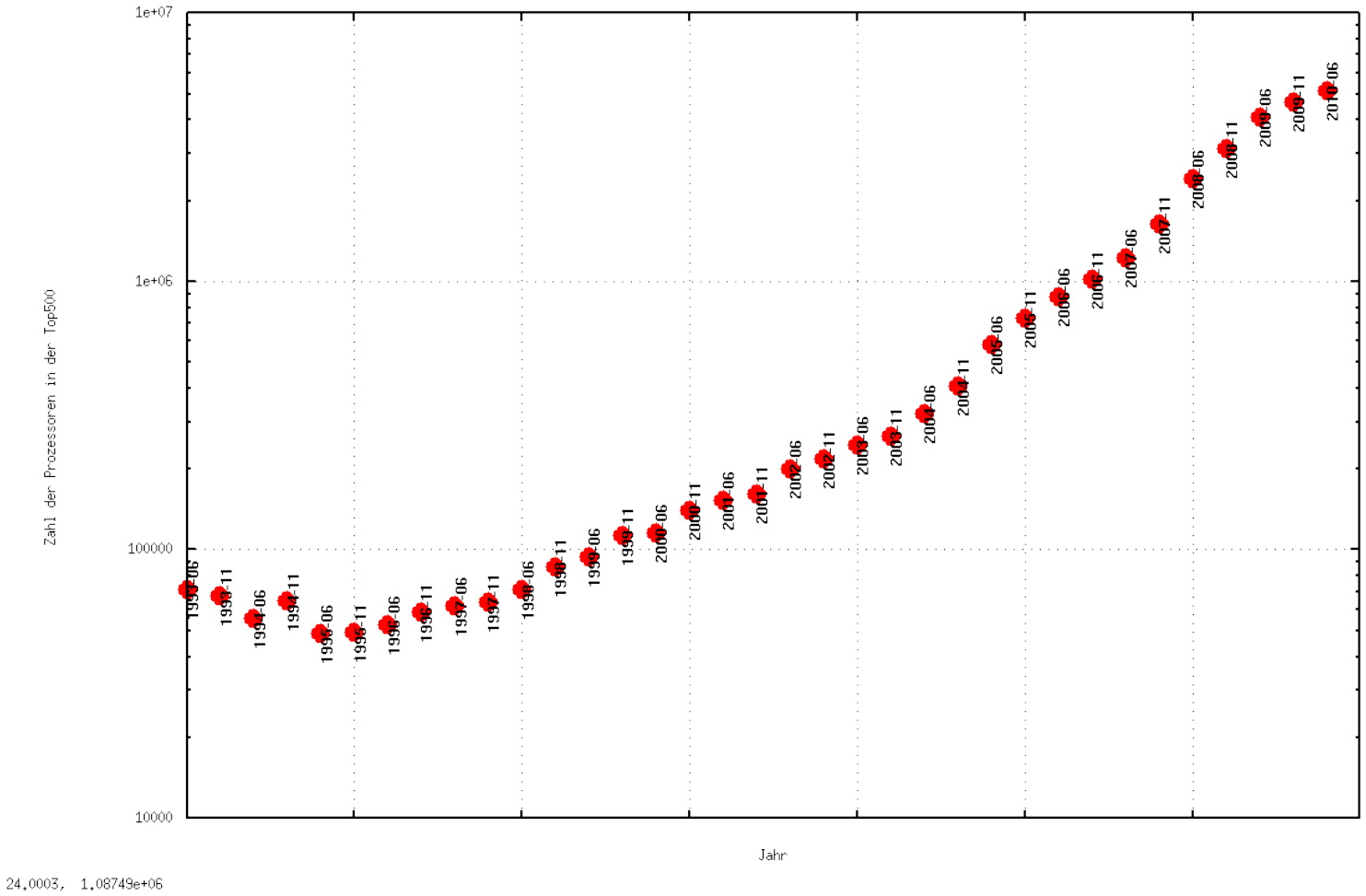


Figure: Summe der Prozessoren in der Top500

Geschichte der Supercomputer III

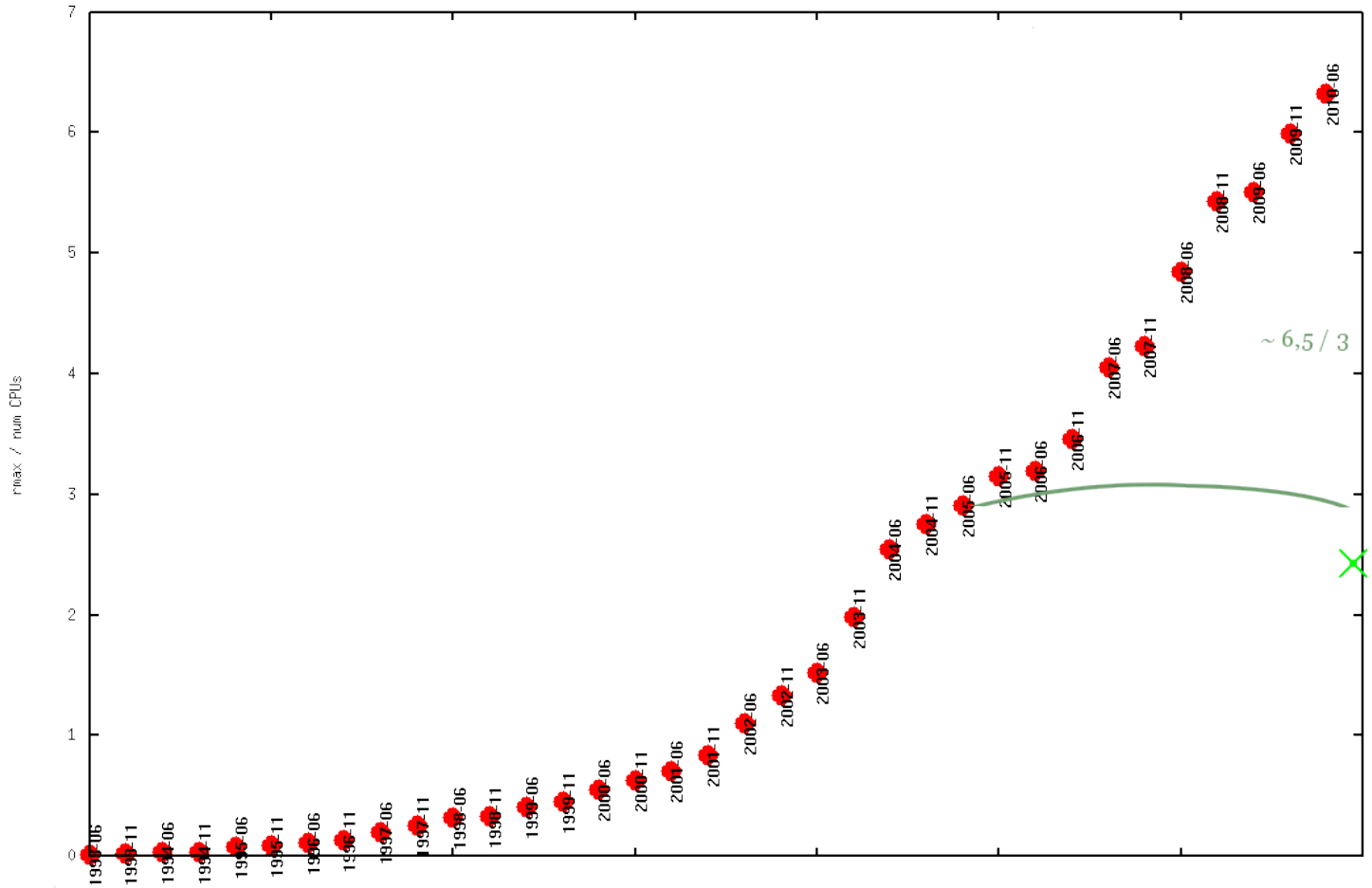


Figure: $\approx \sum$-Rechenleistung / $\sum$-Prozessoren, mit Korrektur pro Core

Probleme

- Konsequenzen von Moores Law treffen nicht mehr für einzelne Cores zu
- Große Teile der heute genutzen Software ist nicht parallelisiert
- Programmierer müssen sich mit (ganz neuer) Klasse von Bugs auseinandersetzen
  - Race Conditions
  - Deadlocks in Kommunikation
  - Heisenbugs
  - False Sharing
- Optimierung ist bei parallelen Programmen um ein Vielfaches komplexer
- "Optimierung" durch warten ist nicht mehr möglich
  - Ein serielles Programm läuft in Zukunft vielleicht eher langsamer
- Zahl der Cores pro PC übersteigt bald die Anzahl der normalerweise laufenden Prozesse
  - Konsequenz: Ohne Parallelisierung idlen bald die meisten Cores in meinem Laptop vor sich hin

Aber Eins nach dem Anderen ...

Begriffe I

Parallelrechner

Computer oder Rechnerverbund dem mehrere eigenständige Prozessoren zur Verfügung stehen, die alle zeitgleich benutzt werden können. Ein “Core” soll hier als eigenständiger Prozessor gesehen werden.

Verbindung zwischen den Prozessoren

Es gibt grundsätzlich zwei verschiedene Möglichkeiten die Prozessoren eines Parallelrechners kommunizieren zu lassen:

- Mit einer -wie auch immer gearteten- Form eines Netzwerkes
- Über einen gemeinsamen Speicher

Begriffe II

Parallelisierung / Parallelverarbeitung

- Methode um schneller zu Ergebnissen zu kommen
- Nutzung mehrere Recheneinheiten gleichzeitig zum lösen eines Problems
- Im Gegensatz zur Optimierung (serieller Programme) werden hier zusätzliche Ressourcen benutzt

Wallclock Time vs. CPU Time

- Wallclock Time = Rechenzeit eines Jobs unabhängig von der Anzahl verwendeter Prozessoren
  - Wird bei Parallelisierung (meist) kürzer
- CPU Time = $\sum$ Rechenzeit jeder CPU für den Job[1]
  - Steigt bei Parallelisierung immer an, da nie perfekte Skalierung

Begriffe III

Speedup, Effizienz, Skalierung

- $\text{Speedup}_n = \frac{T_1}{T_n}$ mit $(1 \leq \mathit{Speedup} \leq n)$
- $\text{Effizienz} = \frac{\text{Speedup}_n}{n}$
- Man sagt: Ein Programm skaliert bis n Prozessoren wenn seine Effizienz erst bei mehr als n Prozessoren unter 0.9 fällt.[2]

[1] einfach: Wallclock $\times$ Anzahl CPUs
[2] 0.9 hier frei von mir gewählt

Unsere Vorteile als Linux-Normal-User

- Wir kämpfen (noch) nicht mit mehr als 20 CPUs
- Wir haben immer einen gemeinsamen Speicher mit Cache Koherenz (noch)
- Unsere Rechner kosten weder im Betrieb noch in der Anschaffung so viel das wir auf 99% Effizienz kommen müssen

Hura?

Die Qual der Wahl

Message Passing

- Kommunikation zw. CPUs per Nachrichtenaustausch
- Programmiermodell besitzt keinen gemeinsamen Speicher
- Eigenständiger Prozess auf jeder CPU/Core
- Unabhängig von der Verbindung zw. CPUs (zur Not ginge auch IPoAC ;-))
- Distribution über Bibliothek + Tools (OpenMPI, MPICH, MVAPICH, ...)
- Programmierung durch Nutzung der Bibliotheksfunktionen
- Gilt allgemein als skalierbarer

OpenMP

- Programmiermodell ausschließlich für gemeinsamen Speicher
- Kommunikation über gemeinsame Variablen, Speicherbereiche
- Parallelität durch Threads
- Programmierung durch setzten von Compiler-Direktiven um Codeblöcke

Allgemeines zu OpenMP

- Im Grunde Standard für Shared Memory Parallelisierung in C/C++ und FORTRAN
- Besteht aus: Compiler Direktiven, ENV Vars, Laufzeitroutinen
- Aktuell in Version 3.0
- Viel mehr Möglichkeiten als bei Compiler-Parallelisierung
  - Datenabhängigkeiten können manuell gelöst werden
  - Grobgranulare Parallelisierung überblicken die Compiler nicht
  - Optimierung
  - Bessere Skalierung (falls gut Programmiert ;-))
- OpenMP Programme sind portabel zwischen einer Vielzahl von Compilern (gcc, intel, cray, ibm, sun, nec, ...)
- Schon vorhandene Programme können ohne viel Aufwand angepasst werden

Fork-Join Modell

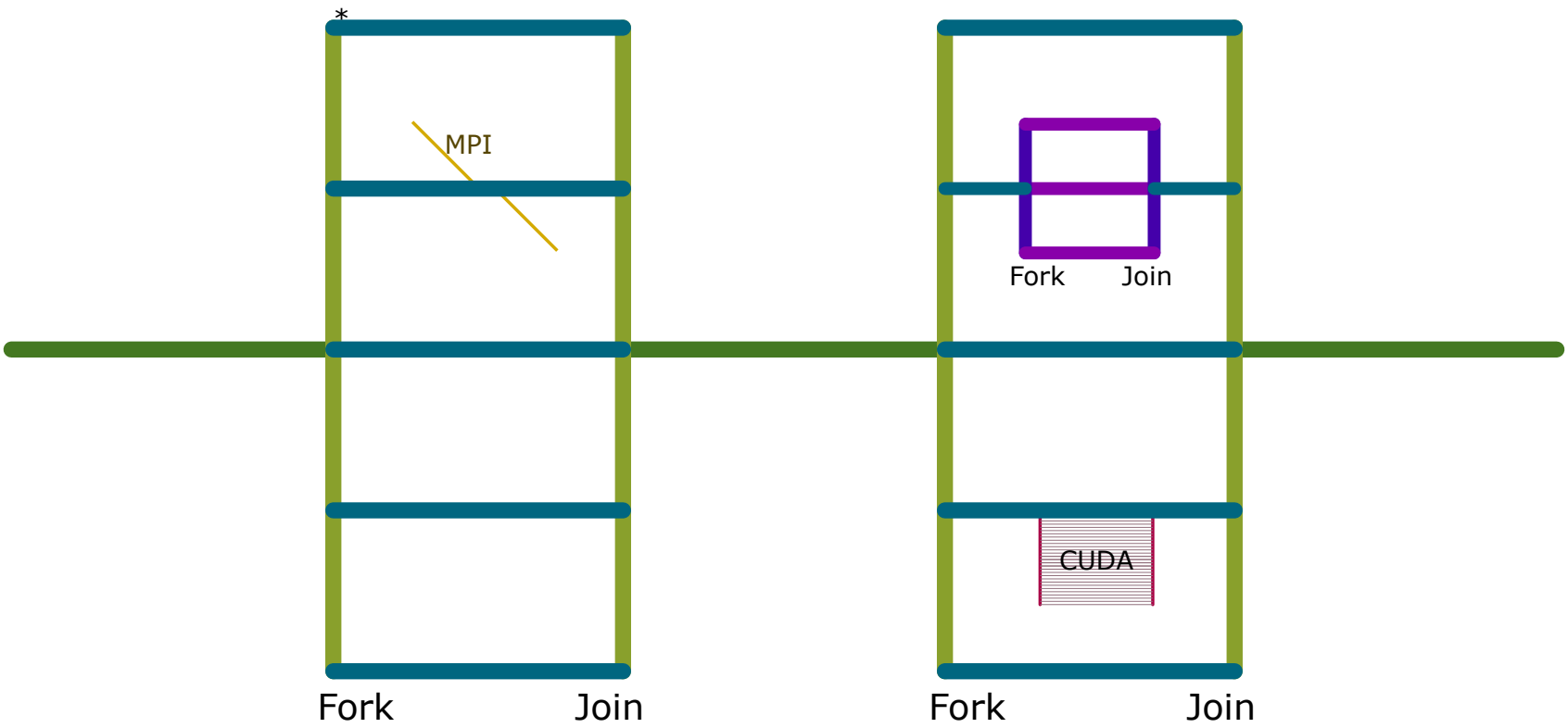


Figure: Fork-Join Modell von OpenMP, sei $*$ ein Masterthread d.h. immer TID 0

Einführungsbeispiel OpenMP

```
1 for(i=0;i<n;i++){
2   a[i] = b[i] + c[i];
3 }
```

Figure: Schleife

```
1 #pragma omp parallel for
2 for(i=0;i<n;i++){
3   c[i] = a[i] + b[i];
4 }
5
6 ## gcc -fopenmp -... bsp-omp.c
7 ## export OMP_NUM_THREADS=4
8 ## ./a.out
```

Figure: OpenMP parallelisierte Schleife

Direktiven

C vs. FORTRAN

- **C**
  - Direktiven und Clauses sind Case-Sensitive
  - `#pragma omp directive [clause [clause] ...]`
  - Fortsetzung des `pragma` durch "\"
  - `_OPENMP` Macro
- **FORTRAN**
  - `!$OMP directive [clause [clause] ...]`
  - Direktiven und Clauses sind Case-Insensitive
  - Fortsetzung des `pragma` durch "&"

Grundlegende clauses für `parallel`-Direktive

- `if(scalar expr)`: Nur Parallel wenn "scalar expr" gleich wahr
- `private(list)`: Jeder Thread bekommt Variablen aus Liste als eigene, für andere unsichtbare, uninitialisierte Variable
- `shared(list)`: Jeder Thread sieht Variablen aus Liste und kann sie verändern
- `firstprivate(list)`: Wie `private` nur das jeder Thread den Originalwert (vor parallel) der Variable übernimmt
- `lastprivate(list)`: Wie `firstprivate` nur das die Originalvariable nach der Parallel-Region geupdatet wird
- `reduction(operator:list)`: Wie `lastprivate` nur das die Originalvariable nach der Parallel-Region unter anwendung des Operators auf jede Variable der Liste geupdatet wird

Zurück zum Beispiel

```
#pragma omp parallel for
for(i=0;i<n;i++){
  c[i] = a[i] + b[i];
}

## gcc -fopenmp -... bsp-omp.c
## export OMP_NUM_THREADS=4
## ./a.out
```

=

```
#pragma omp parallel for shared(n,a,b,c) private(i)
for(i=0;i<n;i++){
  c[i] = a[i] + b[i];
}
```

private, shared, or what?

Aus den OpenMP 3.0 Specs

- Variables appearing in threadprivate directives are `threadprivate`.
- Variables with automatic storage duration that are declared in a scope inside the C/C++ construct are `private`.
- Variables with heap allocated storage are `shared`.
- Static data members are `shared`.
- The loop iteration variable(s) in the associated for-loop(s) of a for or parallel for construct is(are) `private`.
- Variables with const-qualified type having no mutable member are `shared`.
- Static variables which are declared in a scope inside the construct are `shared`.

Im Zweifelsfall **immer** explizit angeben welche Eigenschaft eine Variable besitzen soll.

Einige Beispiele

```c
#include <stdio.h>
#include <omp.h>

int main(){
int i;
i = 42;
printf("serial says: i = %i\n",i);
omp_set_num_threads(6);
#pragma omp parallel private(i)
 {
   printf("id %i says: i=%i\n",omp_get_thread_num(), i);
 }
return 0;
}
```

Einige Beispiele

Ausgabe

serial says: i = 42
id 1 says: i=0
id 0 says: i=0
id 2 says: i=0
id 3 says: i=0
id 4 says: i=0
id 5 says: i=0

- `private` Variablen werden nicht initialisiert
- Reihenfolge der Threads nicht terministisch

Einige Beispiele II

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(){
6  int *tmp;
7
8  tmp = malloc(3*sizeof(int));
9
10 tmp[0] = 1;
11 tmp[1] = 2;
12 tmp[2] = 3;
13
14 omp_set_num_threads(6);
15 #pragma omp parallel private(tmp)
16  {
17    tmp[0] = rand();
18  }
19 return 0;
20 }
```

Einige Beispiele II

Ausgabe

Segmentation fault

`malloc` für dynamische `private`-Variablen in parallelen Bereichen ist zwingend. Alternativ `firstprivate` benutzen.

Einige Beispiele II

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(){
int *tmp;

tmp = malloc(3*sizeof(int));

tmp[0] = 1;
tmp[1] = 2;
tmp[2] = 3;

omp_set_num_threads(6);
#pragma omp parallel firstprivate(tmp)
 {
   tmp[0] = rand();
   printf("%i, %i, %i\n", tmp[0], tmp[1], tmp[2]);
 }
return 0;
}
```

Einige Beispiele III

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <omp.h>
5
6  int iter(double cx, double cy, double max_bq, double max_iter){
7    /* ganz viel code in for-schleife  */
8  return iter;
9  }
10
11 int main(int argc, char *argv[]){
12 omp_set_num_threads(3);
13 ...
14 #pragma omp parallel for private(pix_y,pix_x,cx,cy) shared(max_iter,
       max_pix_y,max_pix_x,x_min,y_min,max_bq)
15 for(pix_y=1 ; pix_y <= max_pix_y; pix_y++){
16   for(pix_x=1; pix_x <= max_pix_x; pix_x++){
17     cx = x_min + ((x_max - x_min)/max_pix_x) * pix_x;
18     cy = y_min + ((y_max - y_min)/max_pix_y) * pix_y;
19
20     field[pix_x][pix_y] = iter(cx, cy, max_bq, max_iter);
21   }
22 }
23 ...
24 }
```

Einige Beispiele III

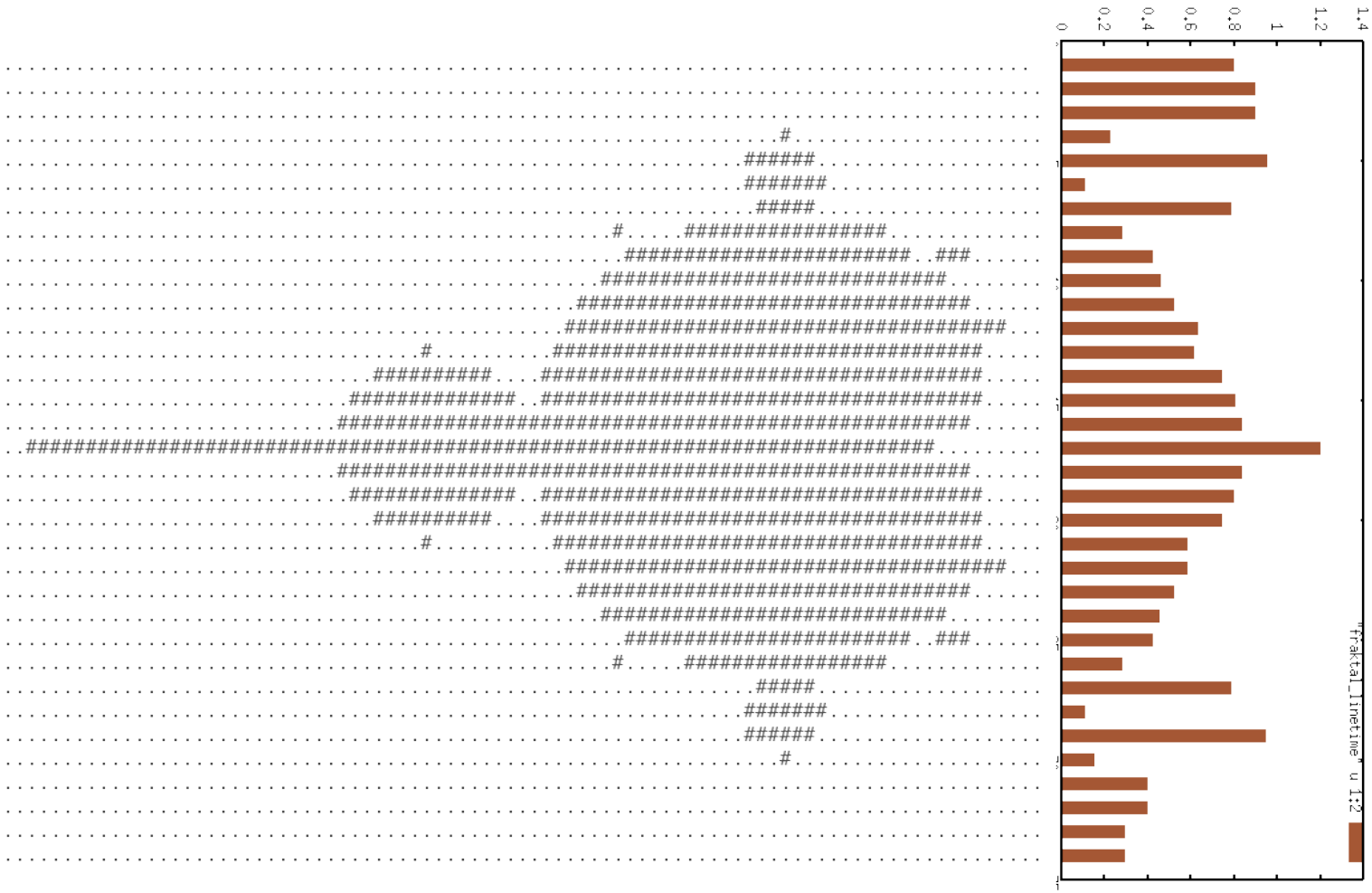


Figure: Unterschiedliche Zeiten für je einen Schleifendurchlauf und Abhängigkeit der Zeile

Die Scheduling Clauses

- `schedule(static | dynamic | guided [,chunk])`
  - **static:** Feste Zuordnung am Anfang welcher Thread welchen Block der Größe `chunk` bekommt. Ohne `chunk`, implizit `chunk` = N / num_threads
  - **dynamic:** Feste Blöcke aber weitere Verteilung erst nach Beendigung eines Blockes. (Default = 1)
  - **guided:** Gleiches Verhalten wie dynamic, nur das chunk die Startgröße vorgibt und die chunksize dann exponentiell verkleinert wird. (Default = 1)
- `schedule(runtime | auto)`
  - **runtime:** Festlegung zur Laufzeit über Variable `OMP_SCHEDULE`
  - **auto:** Alles dem Compiler überlassen, quasi unbestimmt
- Abwägung und genaue Analyse des Scheduling Aufwands bzw. der Zeitvariantz der Iterationen nötig

Scheduling

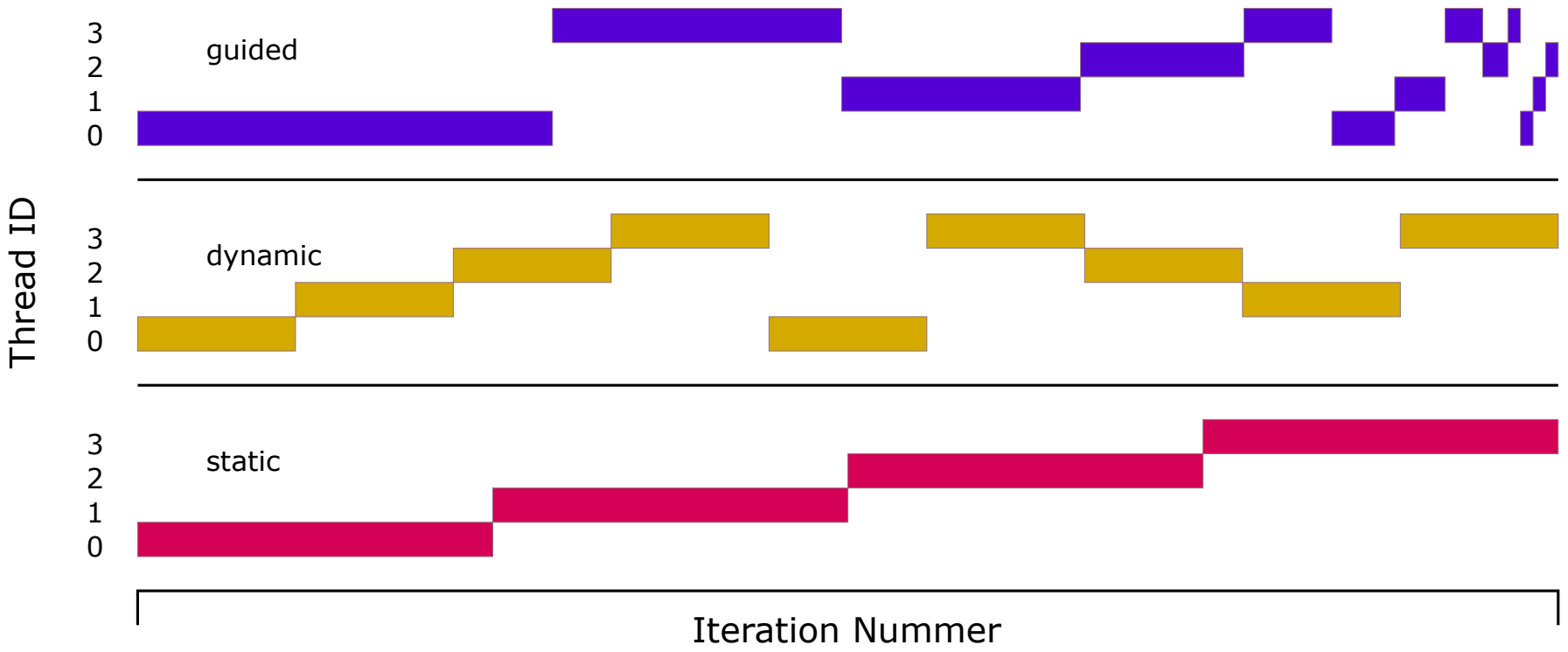


Figure: Scheduling-Methoden grafisch

Data Dependencies

Typen von Datenabhängigkeiten und deren Wirkung auf die Parallelisierung

- Einfache Abhängigkeiten $\delta_{=}$: Gleicher Speicherplatz wird in einer Schleifen mehrmals "angefasst"
- Vorwärts-Abhängigkeit $\delta_{\rightarrow}$: Speicherplatz und sein Nachfolger (bzw. mit Offset n) wird in einer Schleife "angefasst"
- Rückwärts-Abhängigkeit $\delta_{\leftarrow}$: Speicherplatz und sein Vorgänger (bzw. mit Offset n) wird in einer Schleife "angefasst"
- ...

Behandlung von Datenabhängigkeiten in `for`-Schleifen

- Einfache Abhängigkeiten $\delta_{=}$: Für Parallelisierung über die Zählvariable (`i`) ohne Einfluss
- Vorwärts-Abhängigkeit $\delta_{\rightarrow}$ bzw. Rückwärts-Abhängigkeit $\delta_{\leftarrow}$
  - In *umgebenen* Schleifen der zu parallelisierenden können Ignoriert werden
  - In möglicherweise vorhandenen *inneren* Schleifen der zu parallelisierenden können ignoriert werden
  - In der zu Parallelisierenden *selbst* **müssen gelöst werden** oder synchronisiert werden

Beispiele für Datenabhängigkeiten I

```
1  /* Datenabhaengigkeiten die nicht Synchronisiert werden muessen*/
2
3  // S1 (=) S2
4  for(i=0;i<n;i++){
5    a[i] = c[i]; //S1
6    b[i] = a[i]; //S2
7  }
8
9  // S1 (=,<) S2 UND S2 (=,=) S1
10 // Abhaengigkeit betrifft nur innere Schleife,
11 // aeussere kann Paralellisiert werden
12 for(i=1;i<n;i++){
13     for(k=1;k<n;k++){
14         a[i][k] = b[i][k];    //S1
15         b[i][k] = a[i][k-1]; //S2
16     }
17 }
```

Beispiele für Datenabhängigkeiten II

```
1  /* Datenabhaengigkeit mit Sync erforderlich */
2
3  for(i=1;i<n;i++){
4      a[i] = b[i] + c[i];    //S1
5      d[i] = a[i] + e[i-1]; //S2
6      e[i] = e[i] + 2*b[i]; //S3
7      f[i] = e[i] + 1;       //S4
8  }
9
10 // S1 (=) S2
11 // S3 (=) S4
12 // S3 (<) S2
```

3

2 Möglichkeiten

- **1.** Serialisieren
- **2.** Clever lösen (nach WOLFE)

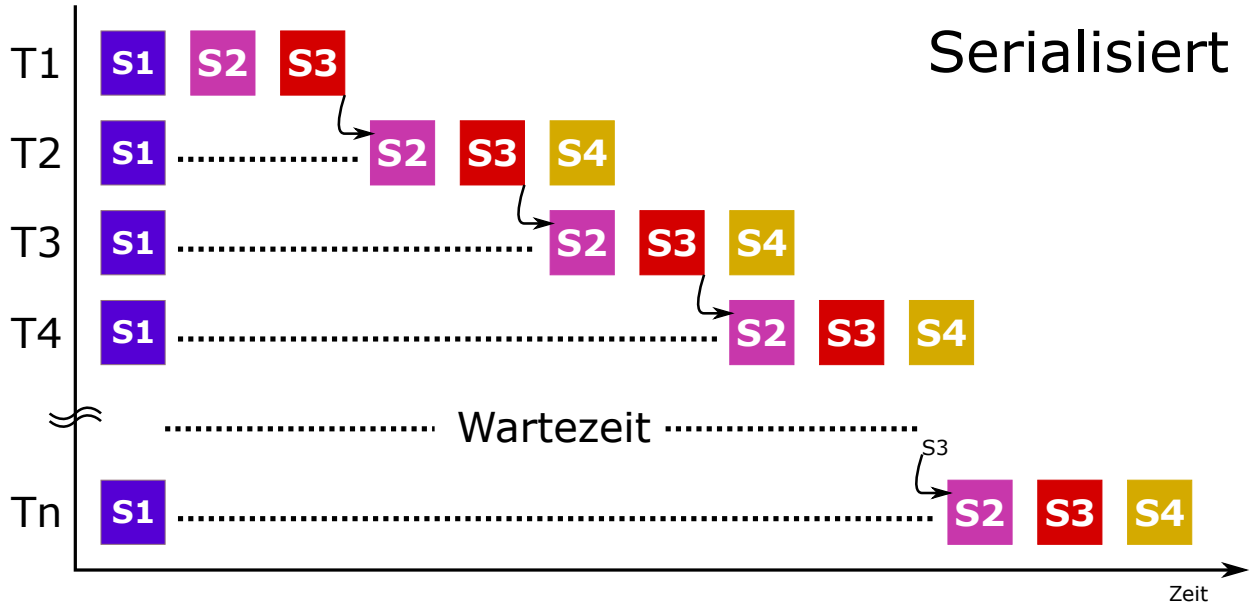
[3] Beispiel nach WOLFE

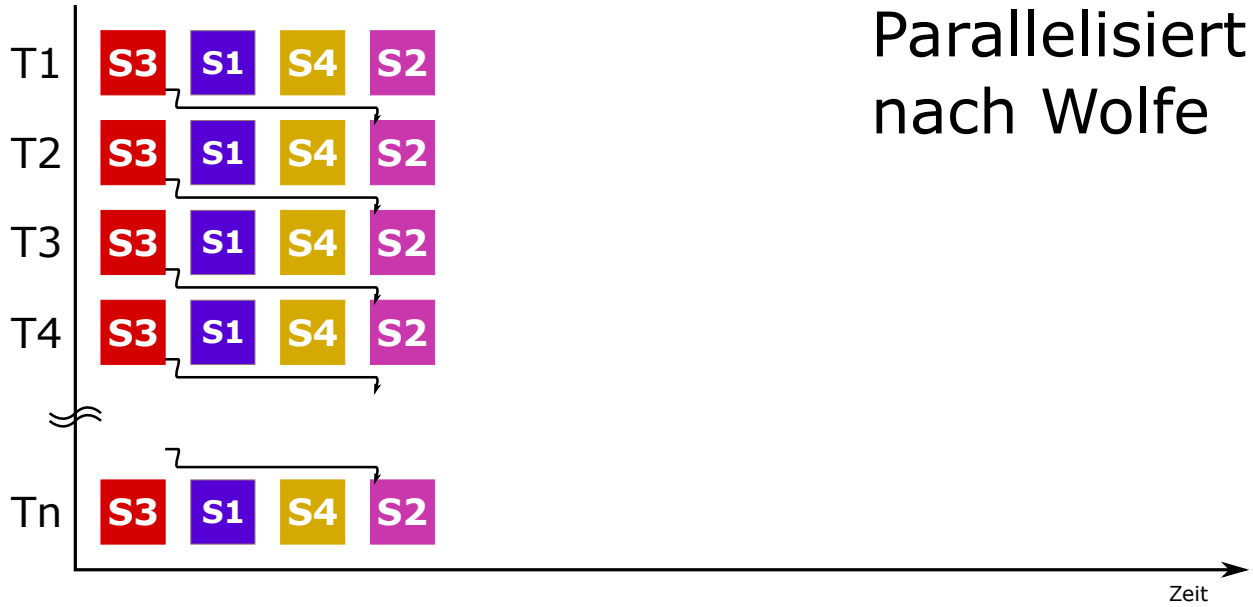
Lösen von Datenabhängigkeiten

- `#pragma omp critical [(name)]` Konstrukt: The `critical` construct restricts execution of the associated structured block to a single thread at a time.
- `#pragma omp atomic`: Besser für Updates nur einer spezifischen Variable (atomic gilt auch immer nur für die genau nachfolgende (Zu)Anweisung)
- *Semaphoren:* Bietet OpenMP in Form der `omp_*_*_lock()` Funktionen an

Lösen von Datenabhängigkeiten

```
1  // lock Initialisierung etc beachten!
2
3  omp_lock_t *lock = new omp_lock_t[n];
4
5  for(i=1;i<n;i++){
6     e[i] = e[i] + 2*b[i];   // S3
7
8     omp_unset_lock(&lock[i]);
9
10    {
11      a[i] = b[i] + c[i]; // S1
12      f[i] = e[i] + 1;    // S4
13    }
14
15    if(i>1)
16      omp_set_lock(&lock[i-1]);
17
18    d[i] = a[i] + e[i-1]; // S2
19 }
```





Beispiele aus: T. Bräunl; Parallele Programmierung; Vieweg 1993

Weitere nützliche Konstrukte

- `single`: Nur ein Thread führt aus, alle anderen überspringen
- `copyin`: Von Masters `threadprivate` Variable zu "meiner" `threadprivate` Variable kopieren
- `barrier`: Warten an genau der Stelle bis alle angekommen sind
- `sections`-Konstrukt: Blöcke des Codes definieren, die dann parallel ausgeführt werden können
- `task`: Fire and forget Möglichkeit
- `...`

Abschluss / Literatur

Bücher

- Das wichtigste überhaupt: `http://www.openmp.org/mp-documents/spec30.pdf`
- Parallele Programmierung von Thomas Rauber und Gudula Rünger (ISBN: 3540465499)

- **Kurse:** `https://fs.hlrs.de/projects/par/par_prog_ws/` (online) und `http://www.hlrs.de/no_cache/organization/sos/par/services/training/course-list/` (Rabenseifner-Kurse)

Allgemeines zu (open)MPI

- MPI = Standard (aktuell 2.2) für Distributed Memory Programmierung
- Besteht aus Bibliothek und Laufzeitumgebung
- OpenMPI als aktuelle und modulare Implementierung (Version 1.4.3)
- Beruht auf dem Versenden von Nachrichten/Anweisungen im (Hochgeschwindigkeits)Netz
- OpenMPI lässt sich ebenfalls mit einer Vielzahl von Compilern benutzen
- Anpassung von schon vorhandenen Programmen teilweise sehr kompliziert
- Sauber MPI-Implementiertes Programm sollte von vornherein dafür konzipiert werden
- Programme werden üblicher Weise mit Zusatzprogramm `mpirun` gestartet

Die Bücher zuerst

- `http://www.mpi-forum.org/docs/docs.html`
- Parallel Programming with MPI. von Peter Pacheco bei: Morgan Kaufmann (ISBN: 1558603395)
- Parallele Programmierung von Thomas Rauber und Gudula Rünger (s.O.)

Uns schockt jetzt nichts mehr – Beispiel am Anfang I

```
1  program fractal
2  implicit none
3  include 'mpif.h'
4
5  ! picture size, counters
6  integer(kind=8) :: max_x_len, max_y_len, iter, i, j, n
7
8  ! mpi variables, mpi status
9  integer :: mpierror, mpisize, mpirank
10 integer(kind=8), dimension(:), allocatable :: sendvec
11 integer :: probestatus(MPI_STATUS_SIZE), recvstat(MPI_STATUS_SIZE)
12 integer, dimension(1) :: addrbuf
13
14 ! the result array
15 integer(kind=8), dimension(:,:), allocatable :: fracarray
16
17 ! mathematical boundarys
18 real(kind=8) :: x_max, x_min, y_max, y_min, cx,cy,nil
19
20 ! set mathematical maximum coordinates
21 x_max = 1; x_min = -2.3; y_max = 1.5; y_min = -1.5
22
23 ! set picture size
24 max_x_len = XSIZE; max_y_len = YSIZE
25
```

Uns schockt jetzt nichts mehr – Beispiel am Anfang II

```
26 nil = 0
27
28 ! allocate array of results
29 allocate(fracarray(max_x_len,max_y_len))
30
31 ! vector for sending with line number
32 allocate(sendvec(max_y_len+1))
33
34 ! do mpi initialization
35 call MPI_Init(mpierror)
36 call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)
37 call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)
38
39 if (mpirank == 0) then
40 !if i'm the master, distribute work
41   do i=1, max_x_len
42     call MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
         probestatus, mpierror)
43     call MPI_Recv(sendvec,max_y_len+1,MPI_INTEGER8, probestatus(
         MPI_SOURCE), &
44    & probestatus(MPI_TAG), MPI_COMM_WORLD, recvstat, mpierror)
45
46     if (sendvec(1) == 0) then
47       sendvec = 0
48     else
49       fracarray(sendvec(1),:) = sendvec(2:)
```

Uns schockt jetzt nichts mehr – Beispiel am Anfang III

```
50      end if
51
52      addrbuf(1) = i
53
54      call MPI_Send(addrbuf,1,MPI_INTEGER8,probestatus(MPI_SOURCE), &
55     & probestatus(MPI_SOURCE),MPI_COMM_WORLD,mpierror)
56
57    end do
58    addrbuf(1) = nil
59    do i=1,mpisize-1
60      call MPI_Send(addrbuf,1,MPI_INTEGER8,i,i,MPI_COMM_WORLD)
61    end do
62  else
63  !if i'm slave ... receive work!
64    i = 1
65    sendvec = 0
66
67    call MPI_Send(sendvec,max_y_len+1,MPI_INTEGER8,0,mpirank,
          MPI_COMM_WORLD,mpierror)
68
69    do while (.TRUE.)
70      ! mpirank+(sendvec(4)+1) for tag uniqueness
71      call MPI_Recv(addrbuf,1,MPI_INTEGER8,0,mpirank,MPI_COMM_WORLD,
            recvstat,mpierror)
72
73      if (addrbuf(1) == nil) then
```

Uns schockt jetzt nichts mehr – Beispiel am Anfang IV

```
74          exit
75        end if
76
77        sendvec(1) = addrbuf(1)
78        do j=1, max_y_len
79          cx = x_min + ((x_max - x_min)/max_x_len) * addrbuf(1)
80          cy = y_min + ((y_max - y_min)/max_y_len) * j
81
82          call iterate(cx,cy, n)
83          sendvec(j+1) = n
84        end do
85
86        call MPI_Send(sendvec,max_y_len+1,MPI_INTEGER8,0,mpirank,
              MPI_COMM_WORLD,mpierror)
87
88      end do
89    end if
90
91    !close mpi stuff
92    call MPI_Finalize(mpierror)
93
94    !clean up array
95    deallocate(fracarray)
96
97    end program
```

Die wichtigsten MPI Routinen I

Punk zu Punkt

- `MPI_Send` / `MPI_Recv`: Einfaches Blockierendes Senden mit zugehörigen empfangen
- `MPI_Bsend`: Gepuffertes Senden
- `MPI_Ssend`: Synchrones Senden

Punkt zu Alle

- `MPI_Bcast`: Ein (und das selbe) Datenpaket an alle
- `MPI_Scatter`: Ein Prozess sended an alle beteiligten Prozesse (unterschiedliche Pakete)
- `MPI_Reduce` zum aufsammeln der Datenpakete
- + Vektorbasierte Varianten, blockierende, nicht-blockierende, gepufferte, etc.

Die wichtigsten MPI Routinen II

Alle zu Alle

- MPI_Allgather: Jeder gathered, ein Element von jedem Beteiligtem Prozess
- MPI_Alltoall: Jeder schickt Alles zu Allen (Multibroadcast)

Sonstiges

- MPI_Win_*: Für Locks
- MPI_Barrier: Barriere zur Synchronisation
- Hunderte andere Calls für, Zeitsynchronisation, Puffer, Datentypen-Transfer, ...

Fazit I

Früher war nicht alles besser!

- Uns steht heute eine unglaublich hohe Rechenleistung selbst bei billigster Hardware zur Verfügung
- Für schlecht zu parallelisierende Probleme gibt es Spezialhardware
- Vieles lässt sich sehr gut Parallelisieren und skaliert gut
- OpenCL als vielleicht neuer und einfacher Standard

Probleme an denen wir arbeiten müssen

- Wir müssen parallel Programmieren lernen
- Wir müssen wieder anfangen unsere Software zu optimieren[4]

[4] Früher wurden Game-Engines in reinen Assembler geschrieben, heute ist das nicht mehr nötig ... dafür müssen wir aber genau so viel Aufwand in die Parallelisierung stecken!

Ende ...

Danke

Fragen?

Quellen

- Hennessy & Patterson; Computer Architecture: A Quantitative Approach; 4th Edition; Morgan Kaufmann
- Thomas Bräunl; Parallele Programmierung: Eine Einführung; Vieweg
- OpenMP, OpenMPI und MPI Referenzen wie in den Folien genannt
- `intel.com`
- `top500.org`