

# 4. Parallelprogrammierung

ALDaBi Praktikum

David Weese  
© 2010/11

Enrico Siragusa  
WS 2011/12

# Inhalt

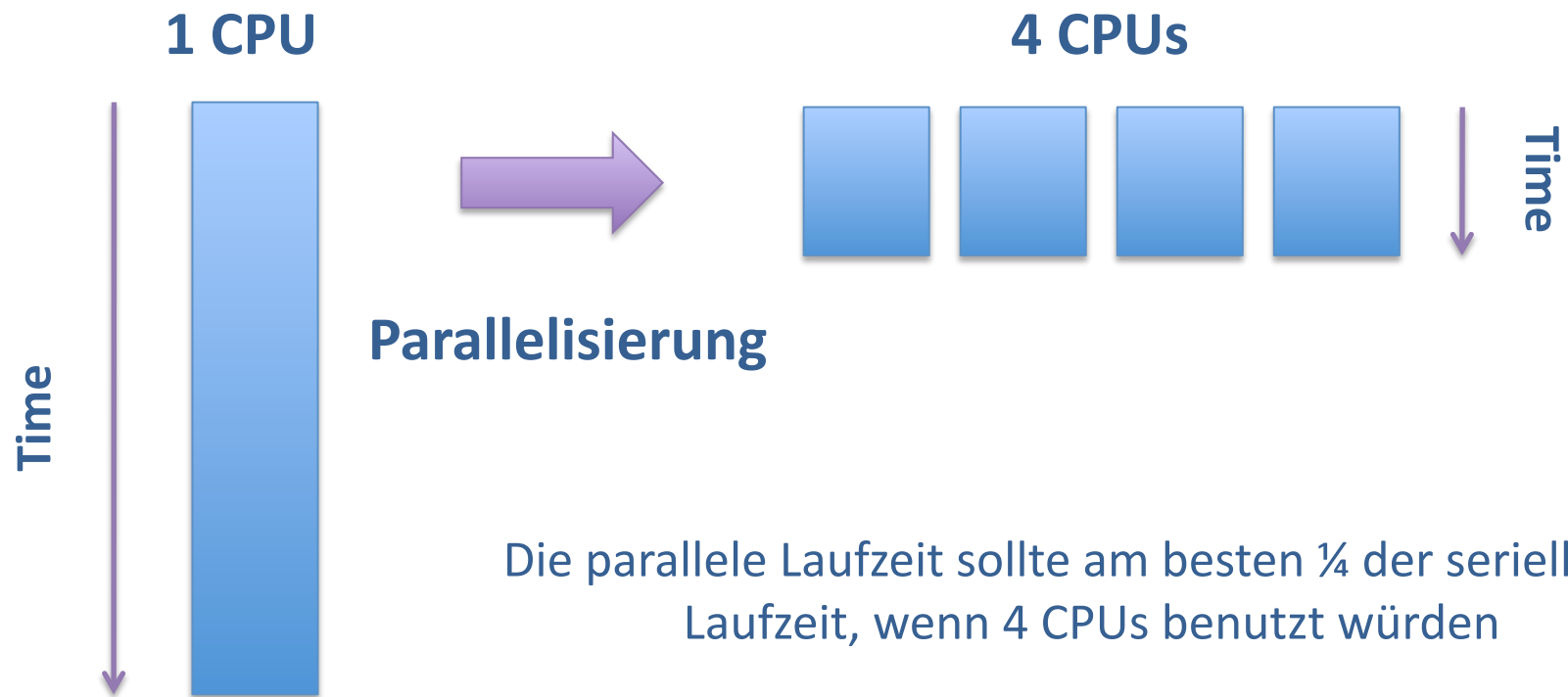
- Einführung in Parallelität
- OpenMP
- Bemerkungen zur P-Aufgabe

# EINFÜHRUNG IN PARALLELITÄT

Folien z.T. aus VL „Programmierung von Hardwarebeschleunigern“ von Alexander Reinefeld und Thomas Steinke, WS09

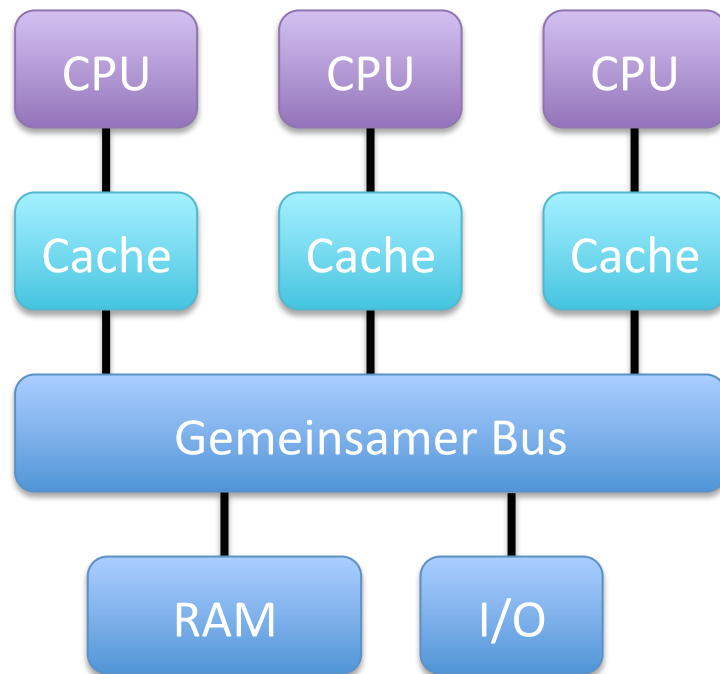
# Was ist Parallelität?

- Parallelität ist eine Optimierungstechnik um die Laufzeit zu reduzieren
- Dazu wurden benutzt mehrere Prozessoren (CPUs oder Cores)

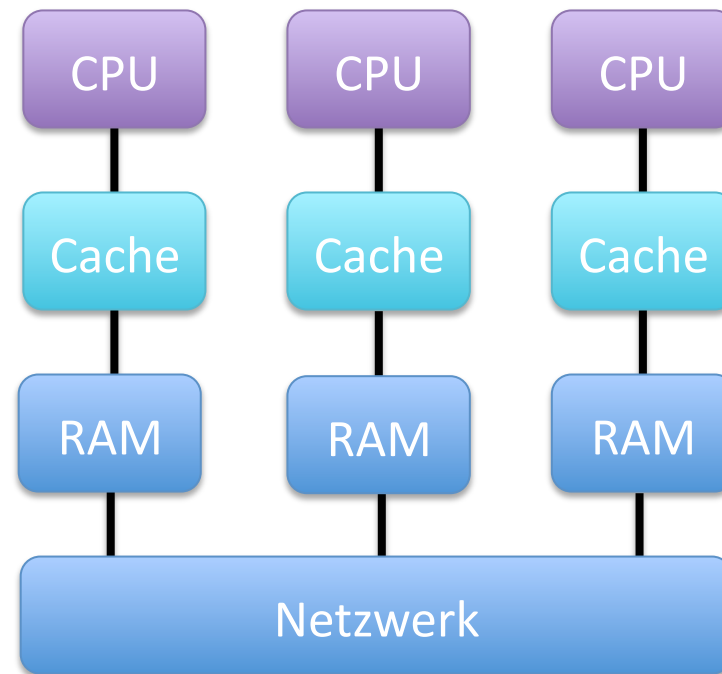


# Parallele Plattformen

Gemeinsamer Speicher  
(shared memory)



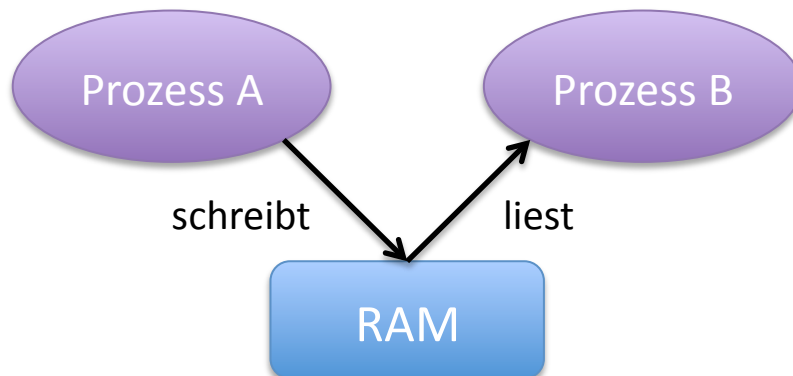
Verteilter Speicher  
(distributed memory)



# Prozesskommunikation ...

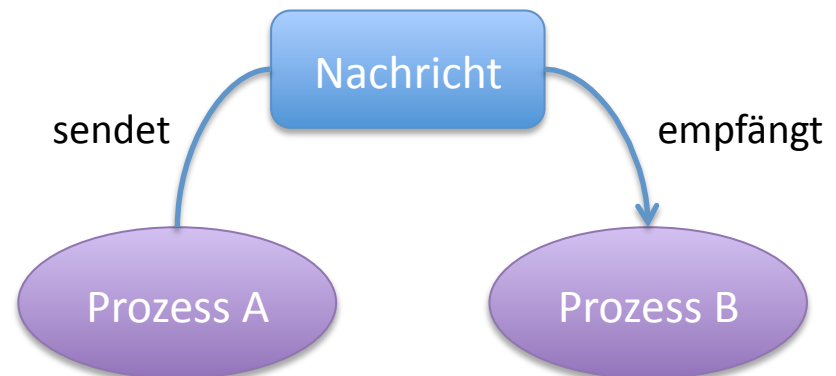
## Gemeinsamer Speicher (shared memory)

- ... über gemeinsame Speicherbereich
  - Schreiben/Lesen im RAM
  - Synchronisation über gegenseitigen Ausschluss



## Verteilter Speicher (distributed memory)

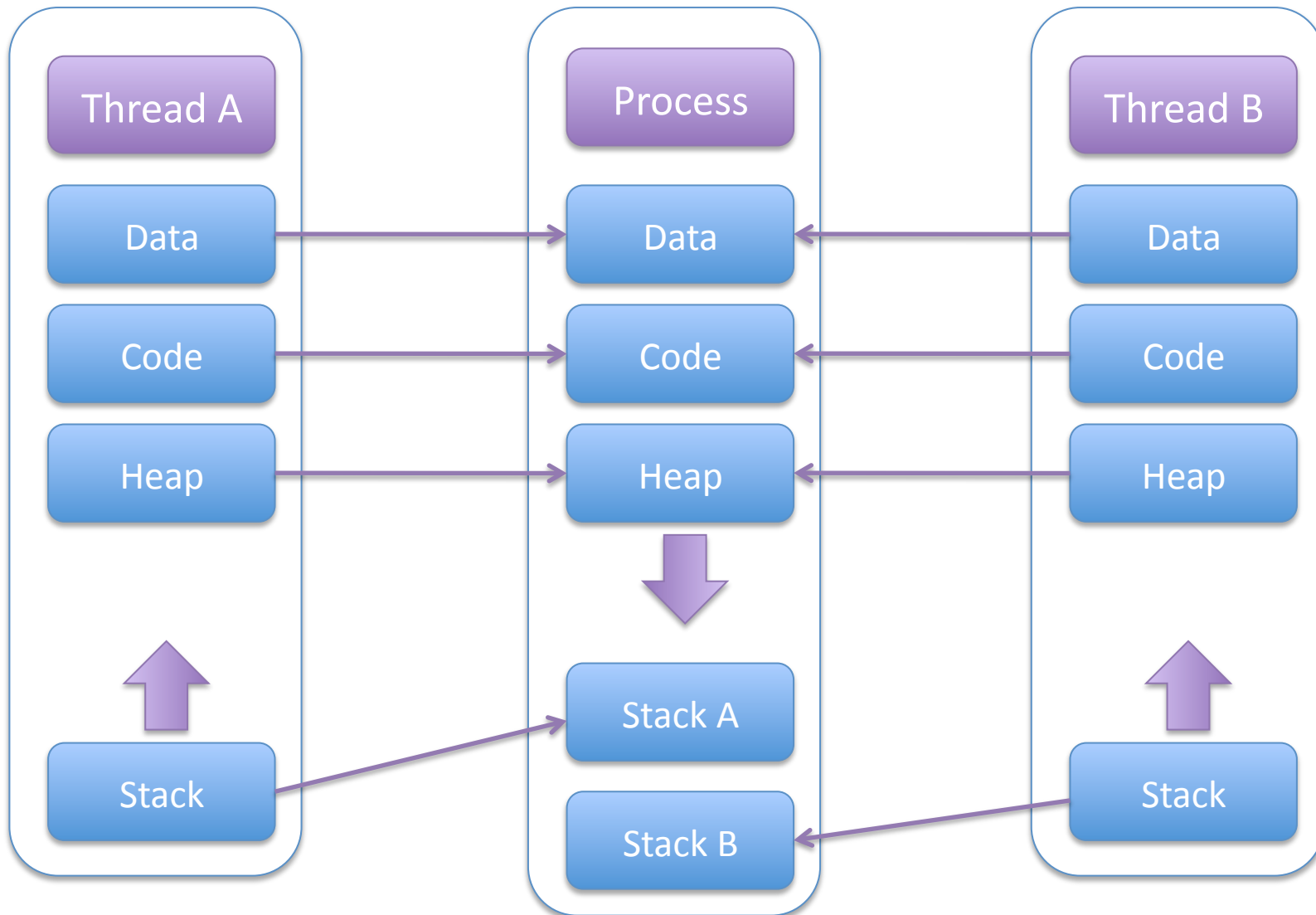
- ... über Nachrichtenaustausch (message passing)
  - Senden/Empfangen von Nachrichten
  - Nur lokale (private) Variablen



# Process VS Thread

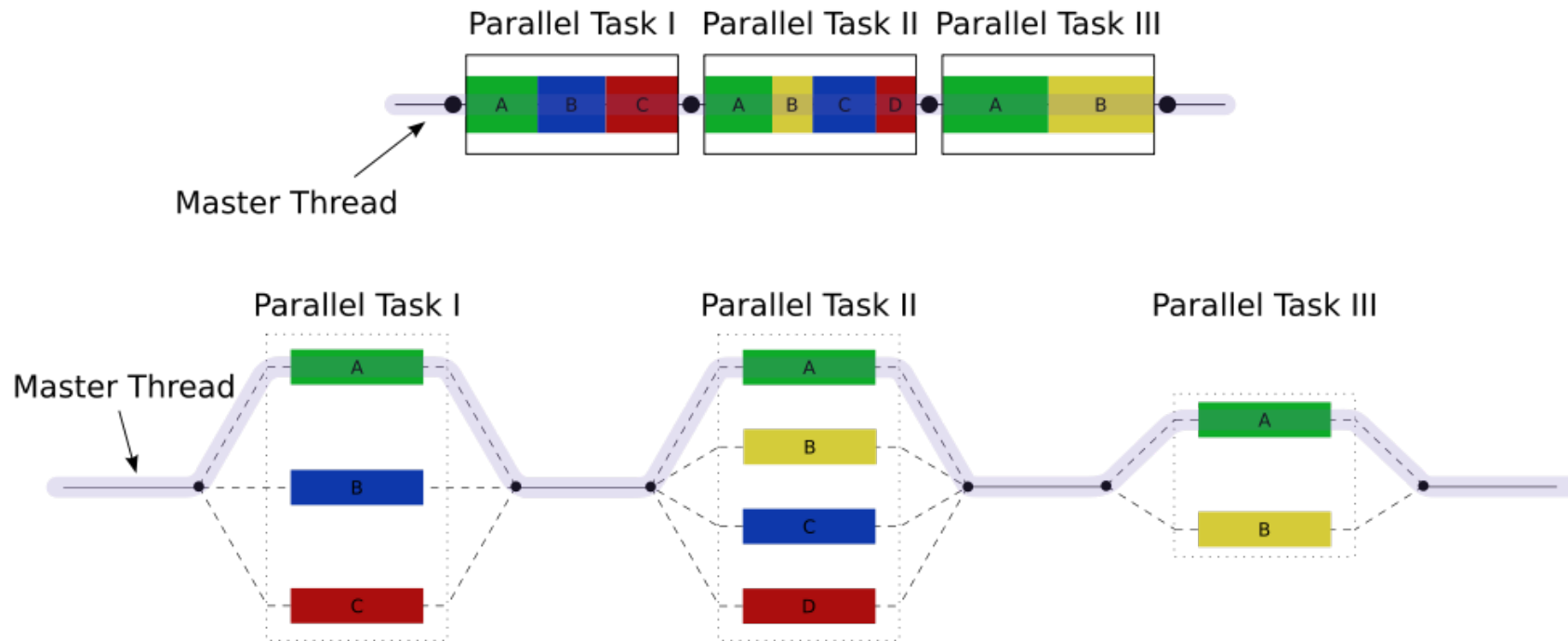
- Prozess
  - Laufende Instanz eines Programms
  - Besteht aus eigenem Adressraum
    - Für Stack, Heap, Programmcode, ...
    - Ist dadurch vor anderen Prozessen geschützt
  - Kann Ressourcen reservieren
  - Hat einen oder mehrere Threads, die den Code ausführen
- Thread
  - Ausführungseinheit des Prozess
  - Bestehen aus eigenem Stack und CPU-Registerzustand
  - Haben den Adressraum des zugehörigen Prozesses
    - Threads desselben Prozesses sind nicht voreinander geschützt
  - Beispiel: Auf mehrere Threads verteilte for-Schleife eines Programms

# Prozesslayout





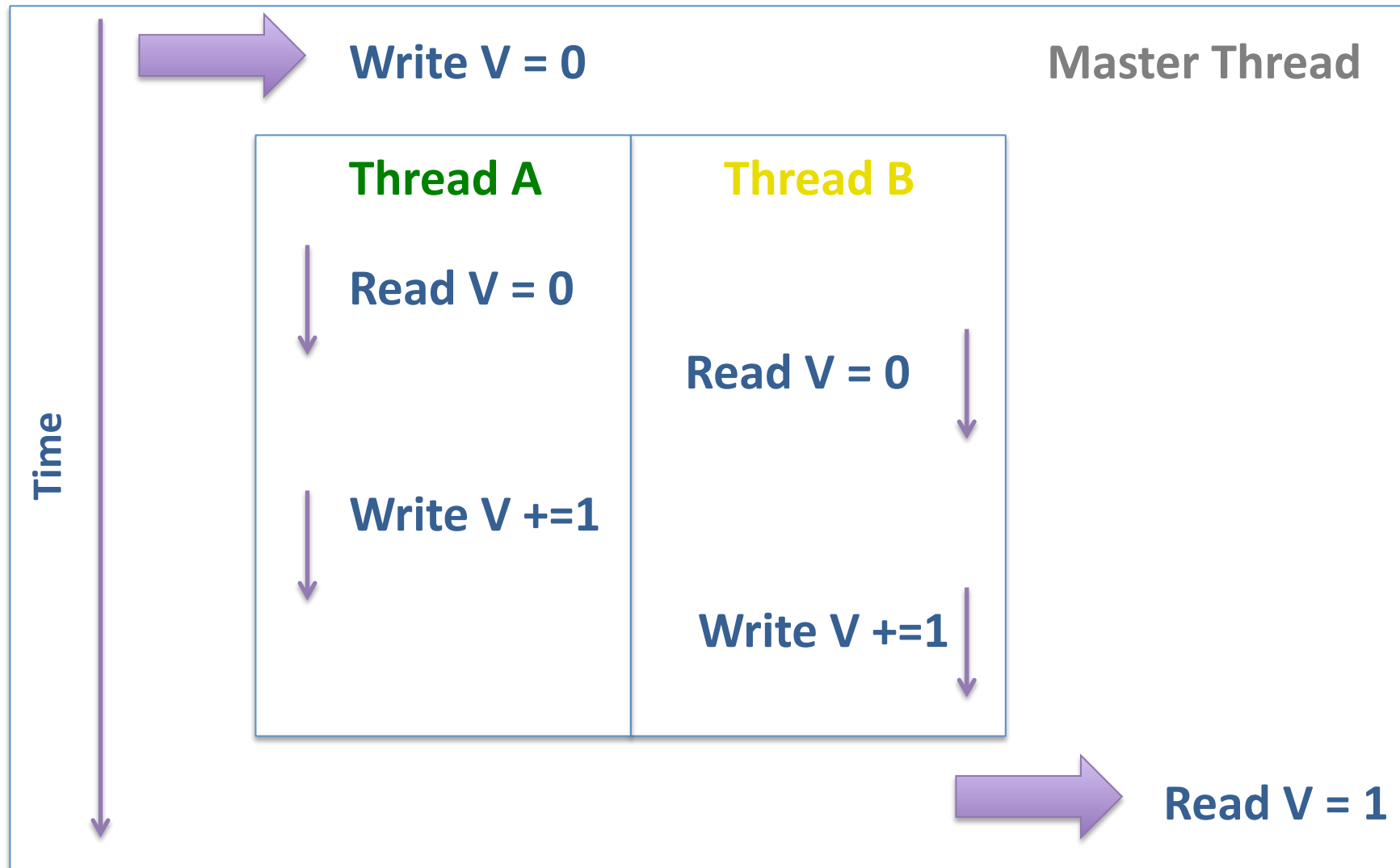
# Fork & Join Modell



# Fallstricke

- Kritischer Wettlauf (Race Condition)
  - Situationen, in denen das „Wettrennen“ (race) der Prozesse beim Zugriff auf gemeinsame Ressourcen Auswirkungen auf das Ergebnis eines Programmlaufs hat
  - Beispiel: 2 Threads schreiben in dieselbe Speicherstelle
- Verklemmung (Deadlock)
  - Threads warten gegenseitig auf die Ressourcen der anderen
- Aushungern (Starvation)
  - Ressource wechselt unfairerweise nur innerhalb einer Gruppe von Threads
  - Threads außerhalb der Gruppe erhalten die Ressource nie

# Kritischer Wettlauf

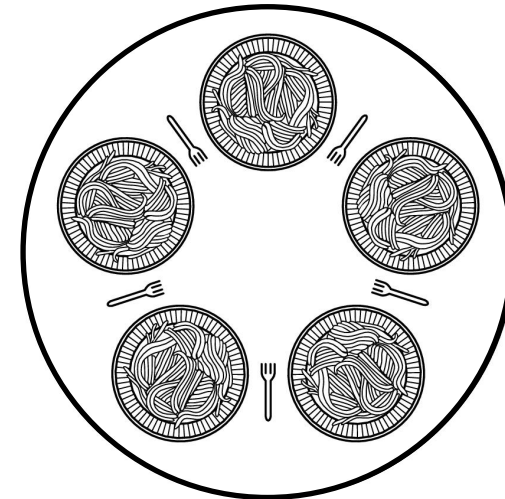


# Lösung

- Synchronisation
  - **Mutex** (Mutual Exclusion)
    - Kann von mehreren Threads verlangt werden (lock), aber nur einer besitzt sie bis er sie freigibt (unlock)
  - **Semaphore**
    - Kann von mehreren Threads verlangt werden, ist aber immer nur in Besitz von höchstens k Threads
    - Mutex ist Semaphore mit  $k=1$
  - **Barrier**
    - Eine Gruppe von Threads hält solange an einer Barriere an, bis alle angekommen sind, danach laufen alle weiter

# Verklemmung

- Philosophenproblem Algorithmus:
  - Denk bis die linke Gabel ist verfügbar
  - Denk bis die rechte Gabel ist verfügbar
  - Iss
  - Lieg die rechte Gabel ab
  - Lieg die linke Gabel ab
  - Wiederhole
- Verklemmung
  - Jeder nimmt die linke Gabel auf und wartet auf die rechte
- Lösung
  - Eine Mutex für den ganzen Tisch zum Prüfen und Aufnehmen zweier Gabeln
  - Semaphore mit  $k=4$



# OPEN MULTI-PROCESSING

Folien z.T. aus VL „Parallele Programmierung“ von Nicolas Maillard und Marcus Ritt, TU Berlin

# Einführung

- OpenMP stellt eine Programmierschnittstelle (API) für C/C++ (und Fortran) zur Verfügung
  - Erweiterung der Programmiersprache durch #pragma-Direktiven
    - Eine Direktive besteht aus einem Namen und einer Klauselliste:  
#pragma omp directive [clause list]
  - Bibliotheksfunktionen
    - #include <omp.h>
- Nur für Plattformen mit gemeinsamen Speicher (shared memory)
- Nützliche Links
  - OpenMP Homepage - <http://www.openmp.org/>
  - OpenMP Tutorial - <http://computing.llnl.gov/tutorials/openMP/>
  - Kurzübersicht - <http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf>

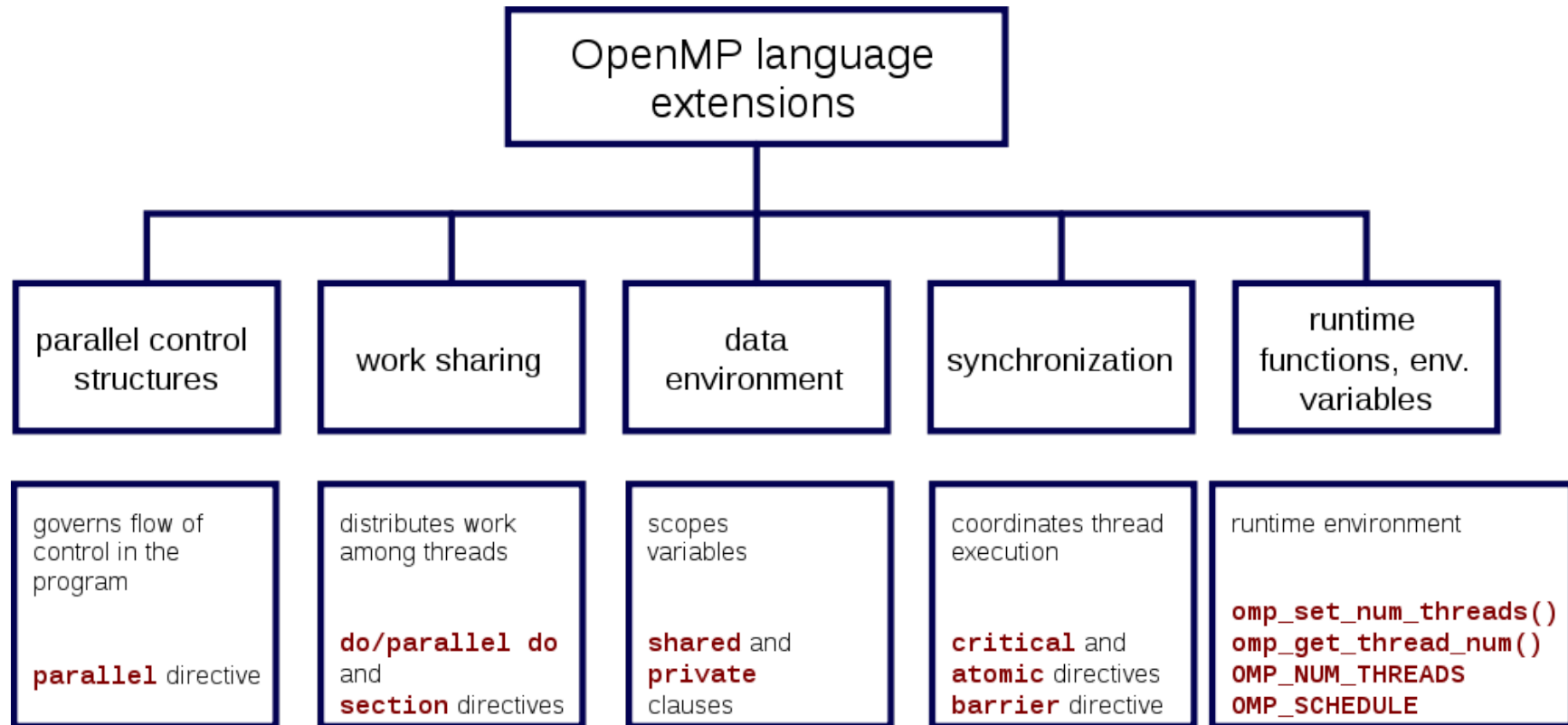
# Verfügbarkeit

Compiler	OpenMP Unterstützung	Compiler Schalter
g++	GCC 4.2: OpenMP 2.5 GCC 4.4: OpenMP 3.0	<b>-fopenmp</b>
Visual C++	VS 2005-2010: OpenMP 2.0	<b>/openmp</b> In der IDE*
Intel C++	V9: OpenMP 2.5 V11: OpenMP 3.0	Windows: <b>/Qopenmp</b> Linux: <b>-openmp</b>
Sun Studio	V12: OpenMP 2.5 V12 U1: OpenMP 3.0	<b>-xopenmp</b>

- OpenMP aktivieren in Visual Studio 2005 oder später:
  - Öffne Properties des Projektes und gehe zu:  
Configuration Properties -> C/C++ -> Language
  - Setze OpenMP Support auf yes
- Achtung:
  - Visual Studio 2005 **Express Edition** unterstützt kein OpenMP, obwohl der Schalter existiert



# Überblick



# OpenMP Beispiel

- Einfaches Beispielprogramm in OpenMP:

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

int main(int, char*[ ])
{
    #pragma omp parallel
    printf("Hello, world.\n");

    return 0;
}
```

- Ausgabe auf Intel Core 2 Duo:

```
Hello, world.
Hello, world.
```

# OpenMP Beispiel (II)

- Unterscheidung der Threads:

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

int main(int, char*[])
{
    #pragma omp parallel
    printf("I'm thread %i of %i.\n",
           omp_get_thread_num(),
           omp_get_num_threads());

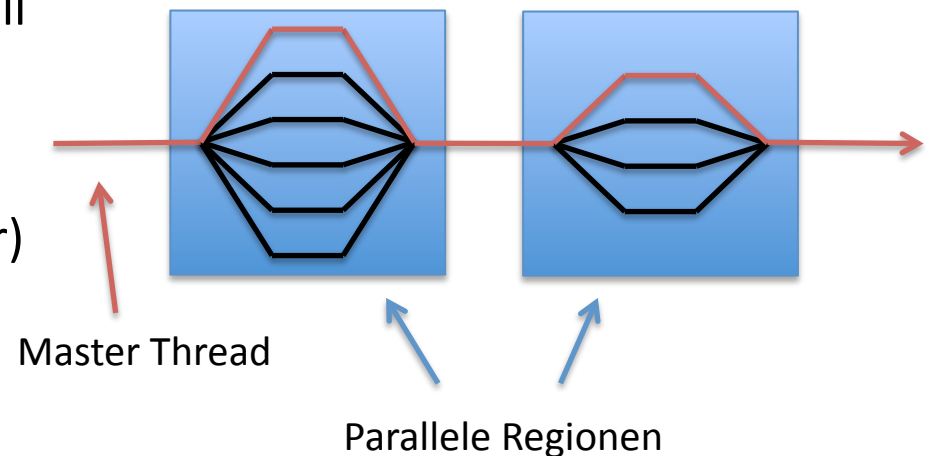
    return 0;
}
```

- Ausgabe:

```
I'm thread 0 of 2.
I'm thread 1 of 2.
```

# Programmiermodell

- Parallelität lässt sich schrittweise erzeugen
  - Ein serielles Programm wird zu einem parallelen
  - Serielle Version weiterhin nutzbar
- Direktive **parallel** startet ein Team
  - Aktueller Thread ist Master
  - Alle Threads bearbeiten den folgenden Abschnitt
- Direktive **for** teilt Iterationen auf Team-Mitglieder auf
  - Jeder Thread bearbeitet einen Teil
- Ende des parallelen Blocks
  - Implizite Synchronisation (barrier)
  - Nur Master läuft weiter



# Von seriell zu parallel

- Serielles Programm:

```
double A[10000];  
for (int i = 0; i < 10000; ++i)  
    A[i] = langwierige_berechnung(i);
```

- Die berechnungen sind unabhängig
  - langwierige\_berechnung braucht nur *i*
- Die Schleife ist sehr einfach parallelisierbar

# parallel

- Parallele Ausführung mit **parallel**
  - Folgender Block wird von allen Threads parallel ausgeführt
  - Programmierer verteilt die Arbeit
    - Entweder manuell
    - Oder mit weiteren Direktiven, bspw. for, sections, ...

```
double A[10000];
int cnt = omp_get_num_threads();
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int i_start = 10000 * id / cnt;
    int i_end = 10000 * (id+1) / cnt;
    langwierige_berechnung(A, i_start, i_end);
}
```

# for

- Aufteilung von Schleifeniteration mit **for**
  - OpenMP teilt Iterationen den einzelnen Threads zu
  - Nur for-Schleifen mit bestimmter Syntax

```
double A[elemente];  
#pragma omp parallel for  
for (int i = 0; i < 10000; ++i)  
    A[i] = langwierige_berechnung(i);
```

- Seit OpenMP 3.0 sind auch Iterator-Schleifen parallelisierbar:

```
vector<int> v(10000);  
typedef vector<int>::iterator iter;  
#pragma omp parallel for  
for (iter i = v.begin(); i < v.end(); ++i)  
    langwierige_berechnung(*i);
```

# for (II)

- Einschränkungen
  - Nur ganzzahlige Schleifenvariablen (mit oder ohne Vorzeichen)
  - Test nur mit  $<$ ,  $<=$ ,  $>$ ,  $>=$
  - Schleifenvariable verändern: nur einfache Ausdrücke
    - Operatoren  $++$ ,  $--$ ,  $+$ ,  $+=$ ,  $-=$
  - Obere und untere Grenze unabhängig von Schleifendurchlauf
    - Ausdrücke sind möglich
- Automatische Synchronisation nach **for** mit Barriere
  - kann mit **nowait** unterdrückt werden



# for (III)

- Verschiedene Arten, den Indexraum auf Threads zu verteilen
- Auswahl erfolgt mit **schedule**
  - `schedule(static[,k])`
    - Indexraum wird in Blöcke der Größe k zerlegt und den Threads **reihum** zugewiesen
    - k=1: 012012012.....0
    - k=5: 000001111122222000001111122222...00000
  - `schedule(dynamic[,k])`
    - Indexraum wird in Blöcke der Größe k zerlegt und den Threads **nach Bedarf** zugewiesen
  - `schedule(guided[,k])`
    - Indexraum wird in Blöcke **proportional zur Restarbeit** auf Threads aufgeteilt und nach Bedarf zugewiesen; k = minimale Iterationszahl
  - `schedule(auto)`
    - Implementierung-spezifisch (Standard, wenn keine Angabe)
  - `schedule(runtime)`
    - Entscheidung zur Laufzeit (`omp_set_schedule`, `omp_get_schedule`, `OMP_SCHEDULE`)

# Klauseln

- Festlegen der Anzahl der Threads:

```
double A[10000];  
#pragma omp parallel for num_threads(4)  
for (int i = 0; i < 10000; ++i)  
    A[i] = langwierige_berechnung(i);
```

- Alternativ ginge auch `omp_set_num_threads(4);`
- Oder über Umgebungsvariable `export OMP_NUM_THREADS=4`

- Bedingte Parallelisierung:

```
double A[elemente];  
#pragma omp parallel for if (elemente > 100)  
for (int i = 0; i < elemente; ++i)  
    A[i] = langwierige_berechnung(i);
```

# Mehrdimensionale Schleifen

- Zusammenfassung von Schleifen
  - for-Direktive wirkt nur auf nächste Schleife
  - **collapse(k)** kombiniert Indexraum der folgenden k Schleifen
- Beispiel: Matrixmultiplikation

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < dim1; ++i)
    for (int j = 0; j < dim2; ++j)
        for (int k = 0; k < dim3; ++k)
            C[i][j] = A[i][k]*B[k][j];
```

# sections

- Parallelisierung ohne Schleifen mit Abschnitten
  - Jedem Abschnitt wird ein Thread zugewiesen
  - Nur statische Parallelität möglich: Abschnitt gilt für ganzes Team

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    arbeit1();
    #pragma omp section
    arbeit2();
    #pragma omp section
    arbeit3();
}
```

# sections (II)

- Rekursionen sind mit **sections** sehr einfach parallelisierbar
  - Beispiel: Quicksort mit verschachtelten Teams:

```
void quicksort(int a[], int l, int r)
{
    if (l < r)
    {
        int i = partition(a, l, r);
        #pragma omp parallel sections
        {
            #pragma omp section
            quicksort(a, l, i - 1);
            #pragma omp section
            quicksort(a, i + 1, r);
        }
    }
}
```

# task

- Parallelisierung ohne Schleifen mit Tasks (OpenMP 3.0)
  - Jeder Task wird (reihum) einem Thread zugewiesen
  - Unterschied zu sections
    - Definition eines Tasks an beliebiger Stelle von beliebigem Thread im Team
    - Ein Thread im Team führt den Task aus

```
void quicksort(int a[], int l, int r)
{
    if (l < r)
    {
        int i = partition(a, l, r);
        #pragma omp task
        quicksort(a, l, i - 1);
        quicksort(a, i + 1, r);
    }
}
```

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        quicksort(a, 1, 99);
    }
}
```

# Synchronisation

- Block der nur von einem / dem Master Thread im Team ausgeführt werden soll
  - `#pragma omp single / #pragma omp master`
  - Beispiel: Datei einlesen, Textausgabe auf der Konsole
- Block der von jedem Thread einzeln (seriell) ausgeführt werden soll
  - `#pragma omp critical`
  - Analog zur Mutex, entspricht dem Bereich zwischen lock und unlock
  - Beispiel: Zugriff auf gemeinsame Datenstruktur / Resource
- Atomares Schreiben (“kleiner kritischer Abschnitt”)
  - `#pragma omp atomic`
  - Beispiel: Inkrementieren eines globalen Zählers
- Synchronisation aller Threads mit Barriere
  - `#pragma omp barrier`

# master, barrier

```
int main()
{
    int a[5];
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 5; i++) a[i] = i * i;

        #pragma omp master
        for (int i = 0; i < 5; i++)
            std::cout << a[i] << std::endl;

        #pragma omp barrier

        #pragma omp for
        for (int i = 0; i < 5; i++) a[i] += i;
    }
}
```



# atomic, critical

```
void max_zeros(int *a, int size)
{
    int max = a[0]; int zeros = 0;

    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < size; i++)
    {
        if (a[i] == 0)
            #pragma omp atomic
            zeros++;

        if (a[i] > max)
            #pragma omp critical
            {
                if (a[i] > max) max = a[i];
            }
    }

    std::cout << max << zeros << std::endl;
}
```

# SpeicherklauseIn für Variablen

- **shared**
  - Daten sind für alle Threads sichtbar/änderbar. Standard in Schleifen.
- **private**
  - Jeder Thread hat eigene Kopie. Daten werden nicht initialisiert. Sind außerhalb des parallelen Abschnitts nicht bekannt.
- **firstprivate**
  - private Daten. Initialisierung mit letztem Wert vor parallelem Abschnitt
- **lastprivate**
  - private Daten. Der Thread, der die letzte Iteration ausführt, übergibt den Wert aus dem parallelen Abschnitt an das Hauptprogramm.
- **threadprivate**
  - globale Daten, die im parallelen Programmabschnitt jedoch als privat behandelt werden. Der globale Wert wird über den parallelen Abschnitt hinweg bewahrt.
- **reduction**
  - private Daten, werden am Ende auf einen globalen Wert zusammengefasst.

# **BEMERKUNGEN ZUR P-AUFGABE**

# Bemerkungen zu Aufgabe 3

- Typischer Fehler:
  - Variable-length Arrays sind nicht Teil des C++ Standards

```
int rows = 5;  
int cols = 10;  
  
int matrix [rows][cols];
```



```
int rows = 5;  
int cols = 10;  
  
vector<int> matrix(rows*cols);
```

- delete anstatt delete[]



```
int *matrix = new int[rows*cols];  
  
// ...  
delete matrix;
```



```
int *matrix = new int[rows*cols];  
  
// ...  
delete[] matrix;
```

# Bemerkungen zu Aufgabe 3 (II)

- Mögliche Verbesserungen:
  - Auf die Traceback-Matrix kann auch verzichtet werden
    - Traceback-Matrix speichert Richtung, aus der das Maximum kam
    - Alternativ kann man die Richtung des Traceback während des Zurückverfolgens berechnet werden
  - Funktionen wie score oder max als inline definieren
  - Eine Matrix-Klasse programmieren
    - <https://svn.mi.fu-berlin.de/agbio/aldabi/ws11/documents/aufgabe3/matrix.h>

# Hinweise zu Aufgabe 4

- Was passiert, wenn der Text einfach nur in nichtüberlappende Teile zerlegt wird?

- Matches, die halb in beiden Teilen liegen, werden nicht gefunden

what	ever	whateve	r
------	------	---------	---

- Wie muss der Text aufgeteilt werden, damit

- Alle Matches gefunden werden
- Keine Matches doppelt ausgegeben werden

- Wie verhindert man, dass 2 Threads gleichzeitig den `vector` mit Matches beschreiben