

Universität Karlsruhe (TH)

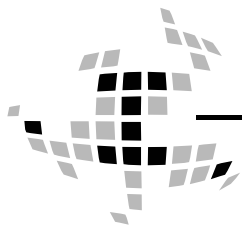
Forschungsuniversität · gegründet 1825

## Programmieren mit OpenMP

Prof. Dr. Walter F. Tichy

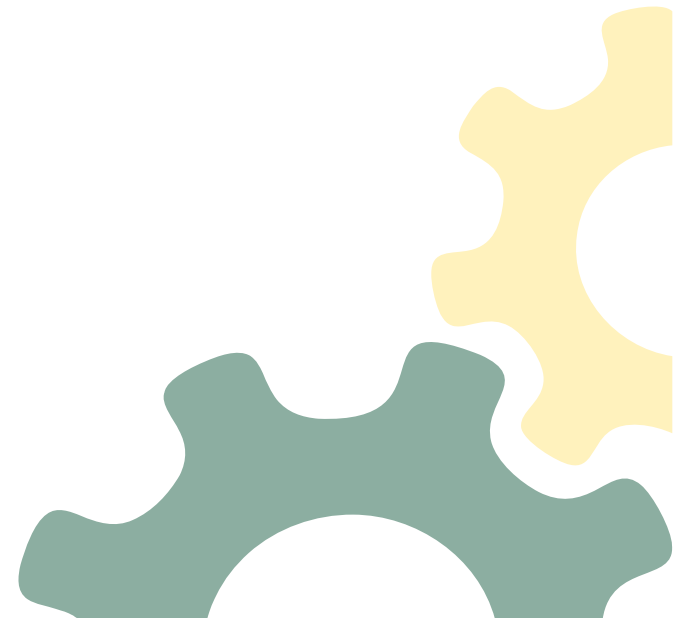
Dr. Victor Pankratius

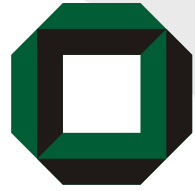
Ali Jannesari



Fakultät für **Informatik**

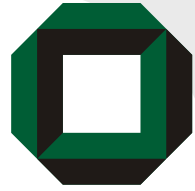
Lehrstuhl für Programmiersysteme





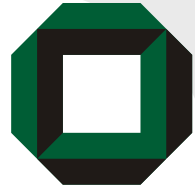
# Inhalt

- Was ist OpenMP?
  - Parallele Regionen
  - Konstrukte zur Arbeitsteilung
  - Sichtbarkeit / Schutz privater Daten
  - Explizite Synchronisation
  - Ablaufplanung bei Schleifen
  - Andere nützliche Konstrukte
- (übersetzte und erweiterte Präsentation von Intel)



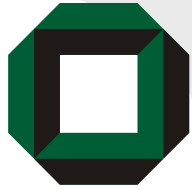
# Was ist OpenMP? (1)

- OpenMP ist eine Erweiterung sequenzieller Sprachen (derzeit C, C++, Fortran) für paralleles Programmieren bei gemeinsamem Speicher mittels:
  - Direktiven für den Übersetzer
  - Bibliotheksroutinen
  - Umgebungsvariablen
- Besonders geeignet für datenparalleles Programmieren
- Parallelität stufenweise veränderbar:  
Vereinigt sequenzielle und parallele Codeabschnitte in einer einzigen Quelldatei



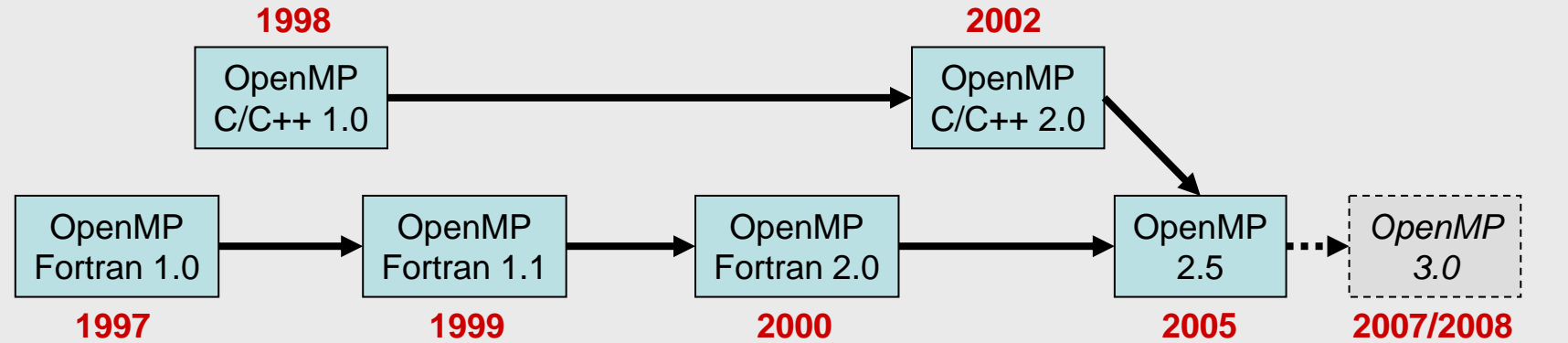
## Was ist OpenMP? (2)

- OpenMP ist portabel:
  - Unterstützung für C/C++ und Fortran
  - für viele Unix-artige Plattformen und Windows implementiert
- OpenMP ist standardisiert
  - Zusammenschluss mehrerer Hard- und Softwarehersteller:  
<http://www.openmp.org>
  - Standardisierung durch ANSI erwartet
- *OpenMP: Open specifications for Multi Processing via collaborative work with interested parties from the hardware and software industry, government and academia.*



# Was ist OpenMP? (4)

## Historie



<http://www.openmp.org>

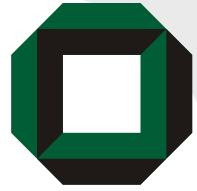
Aktuelle Spezifikation: OpenMP 2.5

250 Seiten, Mai 2005

(C/C++ und Fortran kombiniert)

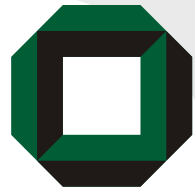
In Arbeit: OpenMP 3.0 (Draft 3.0 Public Comment)

324 Seiten, Oktober 2007



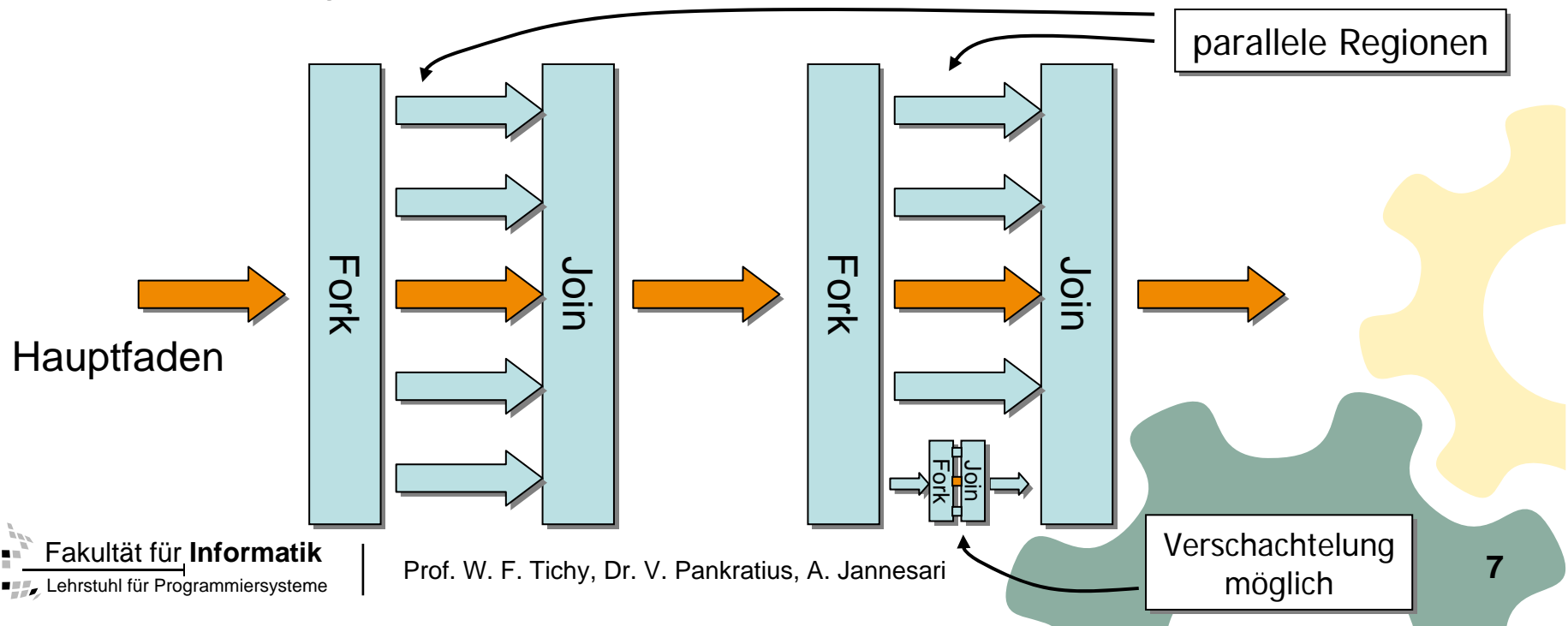
# OpenMP Architektur (1)

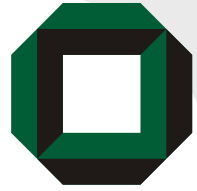
- Fork-join Modell: parallele Regionen wechseln sich mit sequenziellen Abschnitten ab
- Konstrukte
  - zur Arbeitsteilung
  - zur Beeinflussung der Datenumgebung
  - zur Synchronisation
- Umfangreiches API zur Feinsteuerung



# OpenMP Architektur (2)

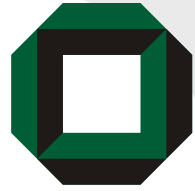
- Parallelität
  - entsteht **bei Bedarf** und **schrittweise** durch das Starten mehrerer Kontrollfäden, die auf dem selben gemeinsamen Speicher zugreifen.
  - Der Programmierer definiert die Stellen mit Parallelität selbst (es wird nicht ohne sein Wissen parallelisiert).
- Parallele Regionen wechseln sich mit sequenziellen Abschnitten ab.





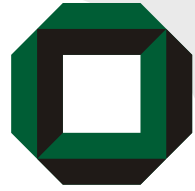
# OpenMP Programmiermodell

- Alle OpenMP Programme starten mit einem Hauptkontrollfaden (Master Thread). Dieser wird sequenziell ausgeführt bis zur ersten parallelen Region.
- **Fork:** Der Hauptkontrollfaden (Master Thread) erzeugt ein Bündel (Team) von parallelen Kontrollfäden. Die parallele Region wird von jedem Kontrollfaden im Bündel ausgeführt.
- **Join:** An einer Barriere endet die parallele Ausführung der Fäden, nur der Hauptkontrollfaden wird weiter ausgeführt.



# OpenMP Programmiermodell

- Anmerkungen:
  - Viele OpenMP-Implementierungen zerstören die Kontrollfäden nach einem Join aus Effizienzgründen nicht. Sie werden „geparkt“ und später wiederverwendet.
- Nicht von allen Implementierungen unterstützt:
  - Verschachtelung von parallelen Ausführungsbereichen
  - dynamische Veränderung der Anzahl der Kontrollfäden in den parallelen Ausführungsbereichen

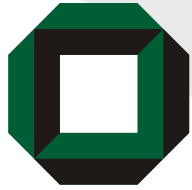


# OpenMP Direktiven

- Die meisten Konstrukte in OpenMP sind Direktiven an den Übersetzer.
- Für C und C++ haben diese die folgende Form:

```
#pragma omp construct [clause [clause]...]
```

- Idee: In einer Umgebung ohne OpenMP kann das Programm rein sequenziell ausgeführt werden. Es läuft langsamer, liefert aber dasselbe Ergebnis.



# OpenMP Programmstruktur

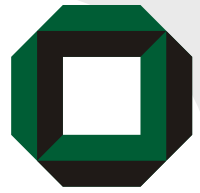
```
#include <omp.h>

main() {
    int var1, var2, var3;

    // serieller Code

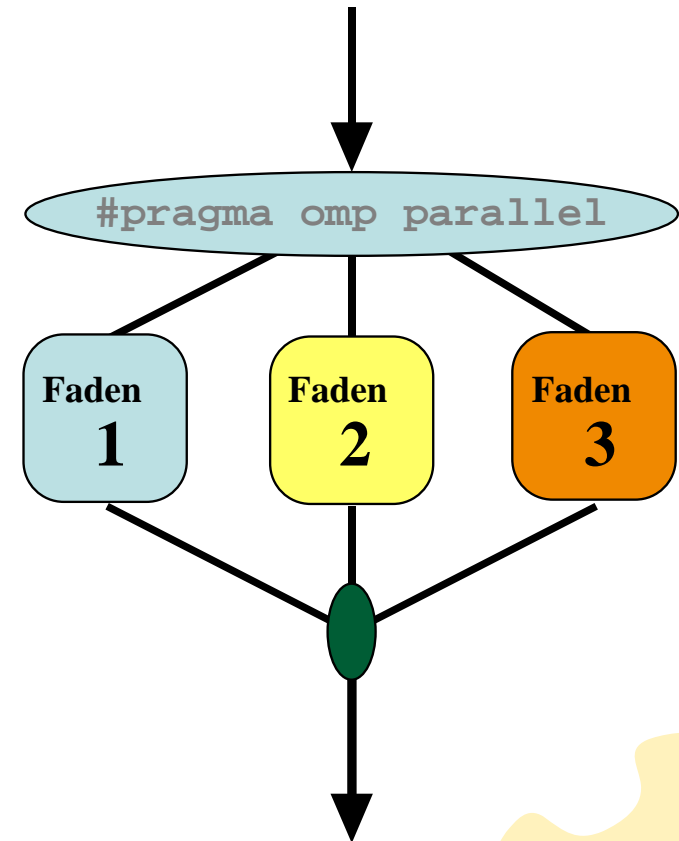
    // Beginn eines parallelen Abschnitts:
    // starte ein Bündel von Kontrollfäden.
    // Sichtbarkeit der Variablen wird spezifiziert.
    #pragma omp parallel private(var1, var2) shared (var3)
    {
        // paralleler Abschnitt, v. allen Fäden ausgeführt
        // ...
        // Barriere:
        // alle Fäden, bis auf den Master, enden
    }

    // weiterer serieller Code
}
```



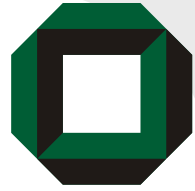
# Parallele Regionen

- Definiert eine parallele Region über einen Anweisungsblock
- Kontrollfäden
  - werden beim Passieren der „parallel“ Direktive erzeugt.
  - blockieren am Ende der Region.
- Daten werden von den Kontrollfäden gemeinsam genutzt, solange nichts anderes angegeben wird.



C/C++ :

```
#pragma omp parallel
{
    block
}
```

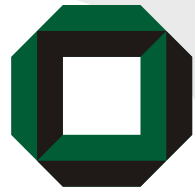


# Wie viele Kontrollfäden?

- Anzahl der Kontrollfäden kann über eine Umgebungsvariable festgelegt werden

```
set OMP_NUM_THREADS=4
```

- Es gibt keinen Standardwert für diese Variable.
- Auf vielen Systemen (z.B. Intel-Übersetzer):  
Anzahl Kontrollfäden = Anzahl der Prozessoren
- Auch über die Bibliotheksfunktion `omp_set_num_threads()` einstellbar.
- Normalerweise ist die Anzahl der Fäden konstant für alle Regionen, je nach Implementierung kann die Anzahl aber verändert werden, dafür ist aber in den dynamischen Modus umzuschalten.
  - Bibliotheksfunktion `omp_set_dynamic()`
  - Umgebungsvariable `OMP_DYNAMIC`

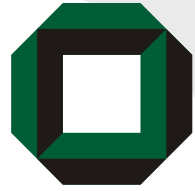


# OpenMP Beispiel: „parallel“

```
#include <omp.h>

main () {
  int nthreads, tid;
  #pragma omp parallel private(nthreads, tid)
  {
    /* Hole und zeige die Nummer des Fadens */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

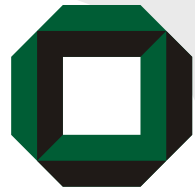
    /* Block wird nur von Faden 0 ausgeführt */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } /* Implizite Barriere. Alle Fäden, bis auf den Master,
    enden */
}
```



# Konstrukt zur Arbeitsteilung (1)

```
#pragma omp parallel
#pragma omp for
  for (I=0;I<N;I++){
    Do_Work(I);
  }
```

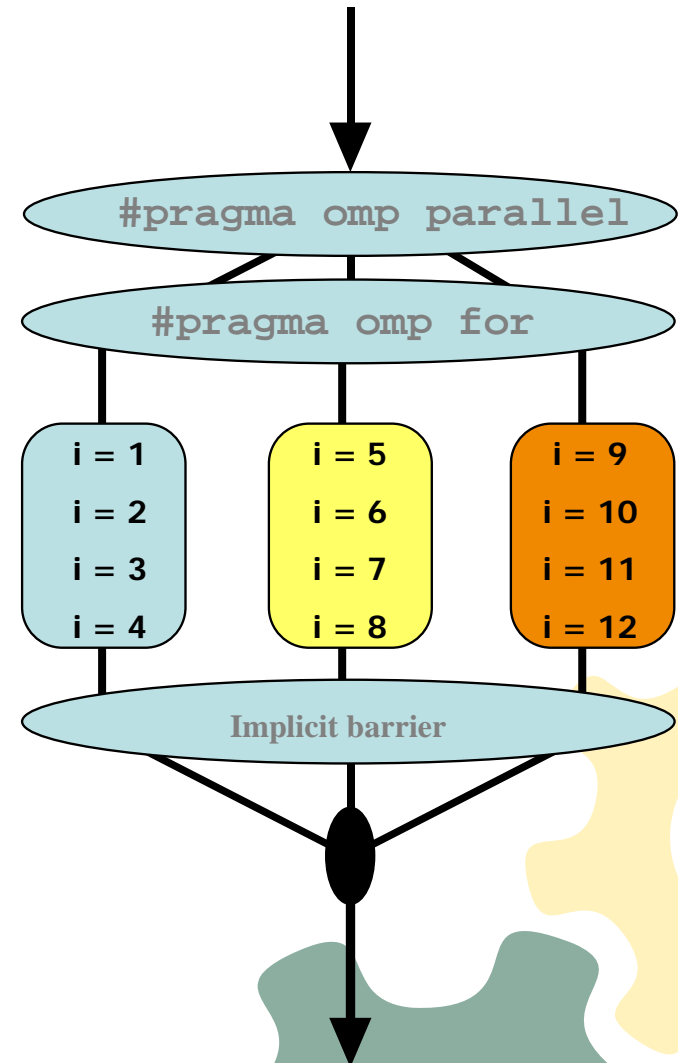
- Die „for“-Direktive
  - verteilt Schleifendurchläufe auf Kontrollfäden.
  - darf nur in einer parallelen Region verwendet werden.
  - muss der Schleife vorangehen.
  - Im Rumpf der Schleife darf es keine Datenabhängigkeiten zwischen den Iterationen geben.

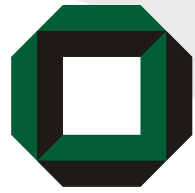


## Konstrukt zur Arbeitsteilung (2)

```
#pragma omp parallel
#pragma omp for
  for(i = 1, i < 13, i++)
    c[i] = a[i] + b[i]
```

- Jedem Kontrollfaden wird eine Menge von Schleifendurchläufen zugewiesen.
- Am Ende dieses Bereiches steht wieder eine implizite Barriere.



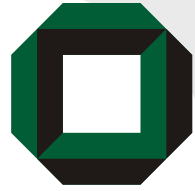


# Kombination der Direktiven

- Diese beiden Code-Fragmente sind äquivalent:

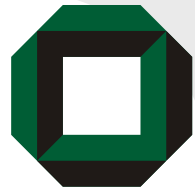
```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```



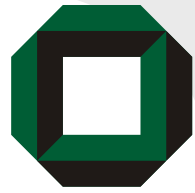
# Datenmodell (1)

- OpenMP verwendet das Programmiermodell „gemeinsamer Speicher“
  - Die meisten Variablen werden standardmäßig gemeinsam genutzt
  - Globale Variablen werden immer von den Kontrollfäden gemeinsam benutzt.
    - In C/C++ sind dies Variablen mit Datei-Sichtbarkeit sowie statische Variablen.



## Datenmodell (2)

- Aber: nicht alles wird gemeinsam genutzt...
  - Auf dem Stapel angelegte Variablen von Funktionen, die aus parallelen Regionen aufgerufen werden, sind **privat**.
  - Automatisch angelegte Variablen in einem Anweisungsblock sind **privat**.
  - Schleifenvariablen sind privat (mit Ausnahmen)
    - In C/C++: Die **erste** Schleifenvariable in geschachtelten Schleifen nach einem `#pragma omp for` ist **privat**.



# Sichtbarkeit (1)

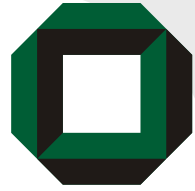
- Die Standard-Sichtbarkeit kann verändert werden durch

```
default (shared | none)
```

- Sichtbarkeitsattribute verändern die Sichtbarkeit einzelner Variablen.

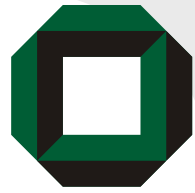
```
shared(varname,...)
```

```
private(varname,...)
```



## Sichtbarkeit (2)

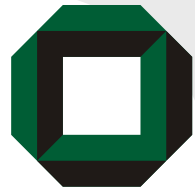
- „private“:
  - Deklariert Variablen, die privat für jeden Kontrollfaden deklariert sind.
  - Ein neues Objekt dieses Typs wird am Anfang der Region für jeden Thread angelegt.
  - Variante „threadprivate“: wie „private“, aber der Wert bleibt bis zum nächsten parallel ausgeführten Block bestehen.
- „shared“:
  - deklariert Variablen, auf die von allen Threads aus zugegriffen werden kann
  - Aufgabe des Programmierers, für Konsistenz zu sorgen (z.B. durch kritische Abschnitte)



# Die „private“-Direktive

- Erzeugt eine Instanz der Variablen für jeden Faden
  - Variablen werden nicht initialisiert; C++ Objekte mittels Default-Konstruktor angelegt.
  - Wert außerhalb der parallelen Region undefiniert.

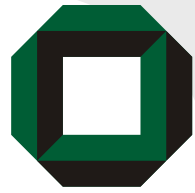
```
void* work(float* c, int N) {  
    float x, y; int i;  
    #pragma omp parallel for private(x,y)  
        for(i=0; i<N; i++) {  
            x = a[i]; y = b[i];  
            c[i] = x + y;  
        }  
}
```



# Beispiel: Skalarprodukt (1)

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

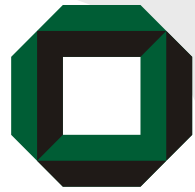
**Was ist falsch?**



## Beispiel: Skalarprodukt (2)

- Der Zugriff auf gemeinsam genutzte, veränderliche Daten muss geschützt werden:

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
        for(int i=0; i<N; i++) {
            #pragma omp critical
                sum += a[i] * b[i];
        }
    return sum;
}
```



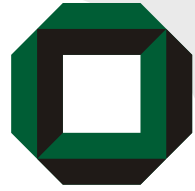
# kritische Abschnitte

`#pragma omp critical [(Name der Sperre)]`  
definiert einen kritischen Abschnitt.

Kontrollfäden müssen warten, bis sie an der Reihe sind. `consum()` wird immer nur von einem Faden gleichzeitig aufgerufen.

Die Angabe eines Namens für die Sperre des kritischen Abschnitts ist optional.

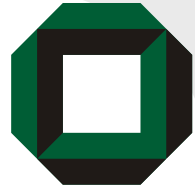
```
float RES;  
#pragma omp parallel  
{ float B;  
#pragma omp for  
  for(int i=0; i<niters; i++){  
    B = big_job(i);  
#pragma omp critical (RES_lock)  
    consum (B, RES);  
  }  
}
```



# Reduktion

reduction (Operator : Variablenliste)

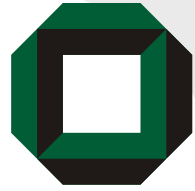
- Die Variablenliste nennt gemeinsam genutzte Variablen, die für die Reduktion genutzt werden sollen.
- **In** der parallelen Region bzw. Schleife:
  - Pro Kontrollfaden wird eine private Kopie jeder Variable angelegt und je nach Operator initialisiert (s. nächste Folie).
  - Die Kopien werden von den Kontrollfäden lokal verändert.
- Am Ende der parallelen Region werden die lokalen Kopien durch den angegebenen Operator zu einem einzigen Wert zusammengeführt und in der gemeinsamen Variable abgelegt.



# Reduktion: Beispiel

```
#pragma omp parallel for reduction(+:sum)
for(i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

- Jeder Faden hat seine eigene Kopie von „sum“.
- Alle Kopien werden am Schluss aufaddiert und in der globalen „sum“-Variable gespeichert.

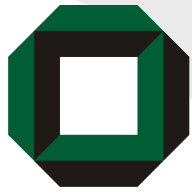


# C/C++ Reduktionsoperationen

- Eine Reihe von assoziativen Operationen kann für die Reduktion verwendet werden.
- Die Reduktionsvariable wird „mathematisch sinnvoll“ initialisiert.

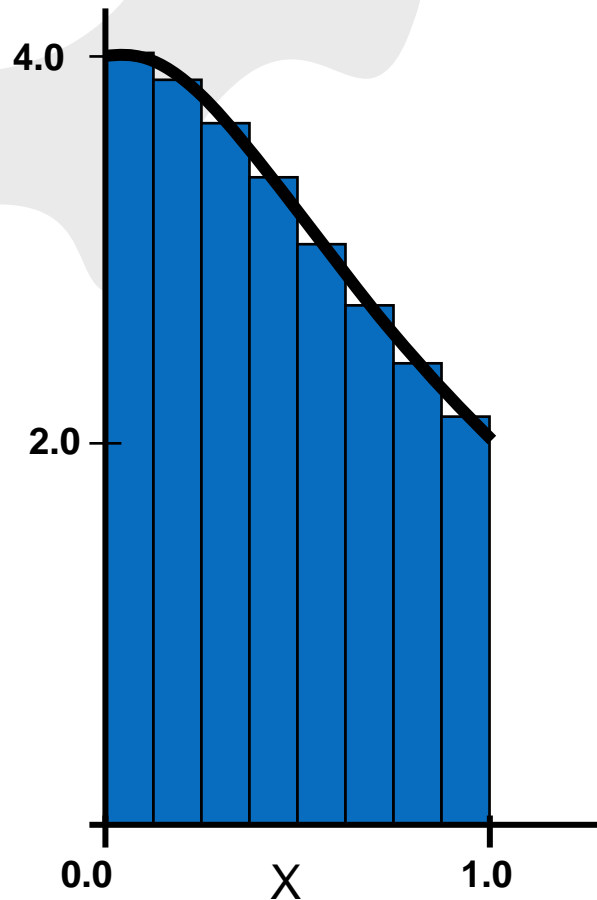
Operator	Initialer Wert
+	0
*	1
-	0
^	0

Operator	Initialer Wert
&	~0
	0
&&	1
	0



# Beispiel: Numerische Integration

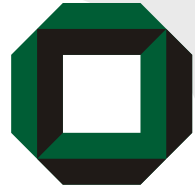
$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$



```
static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```



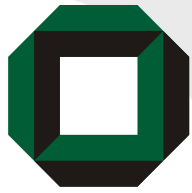
# Numerische Integration (2)

```
static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

- Parallelisierung mittels OpenMP
  - Welche Variablen können gemeinsam genutzt werden?
  - Welche Variablen müssen privat sein?
  - Wie kann Reduktion eingesetzt werden?



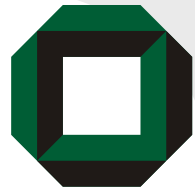
# Numerische Integration: Lösung

```
#include <stdio.h>

long long num_steps = 1000000000;
double step;

int main(int argc, char* argv[]) {
    double x, pi, sum=0.0; int i;
    step = 1./((double)num_steps);

    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++) {
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }
    pi = sum*step;
    printf("The value of PI is %f\n",pi);
    return 0;
}
```



# Aufteilung von Schleifendurchläufen

- Mit der „schedule“ Direktive kann festgelegt werden, wie die Iterationen auf die verfügbaren Kontrollfäden verteilt werden sollen.

```
schedule(static [,chunk])
```

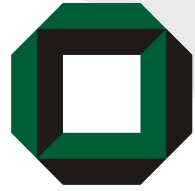
- Weist reihum Blöcke von Iterationen (der Größe „chunk“) zu.

```
schedule(dynamic[,chunk])
```

- Weist Blöcke der Größe „chunk“ zu.
- Fäden fordern einen neuen Block an, wenn sie mit dem alten Block fertig sind

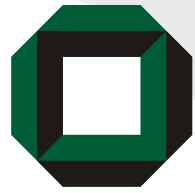
```
schedule(guided[,chunk])
```

- Dynamische Verteilung von Blöcken wie bei „dynamic“.
- Startet mit großen Blöcken, Blöcke werden immer kleiner, aber nicht kleiner als „chunk“



# Welche Strategie ist zu wählen?

Ablaufstrategie	Einsatz
STATIC	Vorhersagbare, gleich verteilte Menge an Arbeit pro Durchlauf
DYNAMIC	Unvorhersagbare, stark schwankend Menge an Arbeit pro Durchlauf
GUIDED	Spezialfall von „dynamic“ mit geringerem Overhead.

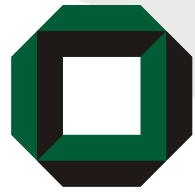


# Beispiel für die Ablaufplanung

```
#pragma omp parallel for schedule (static, 8)
  for( int i = start; i <= end; i += 2 )
  {
    if ( TestForPrime(i) ) gPrimesFound++;
  }
```

- Schleifendurchläufe werden in Blöcken zu je acht Durchläufen verteilt.

Wenn  $start = 3$  ist, besteht der erste Block aus den Durchläufen für  $i = \{3, 5, 7, 9, 11, 13, 15, 17\}$ .

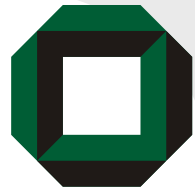


# Statische Verteilung („von Hand“)

- Vorgegeben:
  - Anzahl der Fäden (`Nthrds`)
  - Nummer des jeweiligen Fadens (`id`)
- Berechnung der Start- und Endwerte der Iteration:

```
#pragma omp parallel
{
    int i, istart, iend;
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart; i<iend; i++){
        c[i] = a[i] + b[i];
    }
}
```

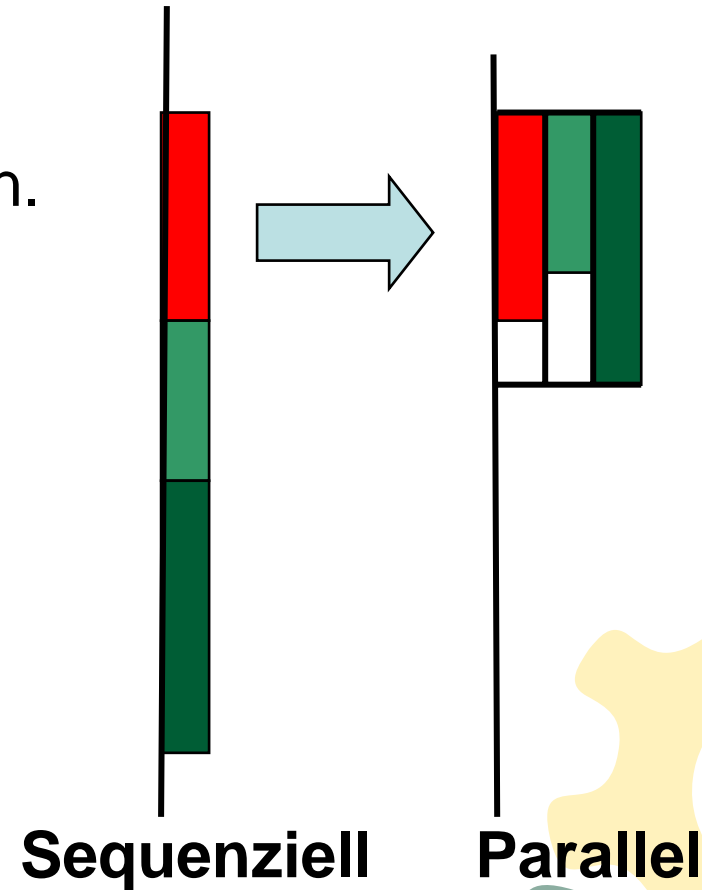
- Mit OpenMP ist eine solche „händische“ Aufteilung normalerweise nicht nötig, aber möglich.

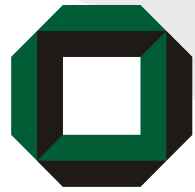


# Parallele Abschnitte

- Unabhängige Code-Abschnitte können parallel ausgeführt werden.

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```

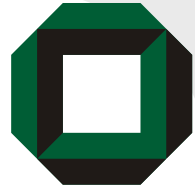




# „Single“ Direktive

- In einer parallelen Region kann Code vorkommen, der nur von einem Faden ausgeführt werden soll (z.B. für E/A-Operationen).
- Dieser Codebereich kann mit einer single-Region eingeklammert werden.
  - Es wird der erste Faden ausgewählt, der diese Stelle erreicht.
- Implizite Barriere am Ende (es sei, denn „nowait“ wurde angegeben).

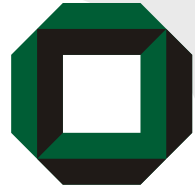
```
#pragma omp parallel
{
    DoManyThings();
    #pragma omp single
    {
        ExchangeBoundaries();
    } // andere Fäden warten hier
    DoManyMoreThings();
}
```



## „Master“ Direktive

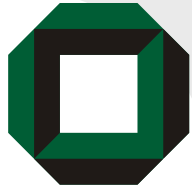
- Markiert einen Code-Block, der nur vom Hauptfaden ausgeführt werden soll.
- **Keine** implizite Barriere am Ende.

```
#pragma omp parallel
{
    DoManyThings();
    #pragma omp master
    {
        // springe weiter falls nicht Master
        ExchangeBoundaries();
    }
    DoManyMoreThings();
}
```



# Implizite Barrieren

- Einige OpenMP-Konstrukte beinhalten implizite Barrieren:
  - `parallel`
  - `for`
  - `single`
- Unnötige Barrieren beeinträchtigen die Leistung
  - Wartende Kontrollfäden erledigen keine Arbeit!
- Man kann unnötige Barrieren (auf eigene Gefahr!) mit der `nowait` Direktive unterdrücken.



# „Nowait“-Direktive

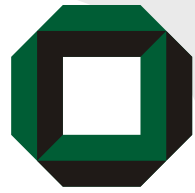
```
#pragma omp for nowait
for(...)
{...};
```

```
#pragma single nowait
{ [...] }
```

- Verwendung beispielsweise wenn die Fäden zwischen unabhängigen Berechnungen warten müssten:

```
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);

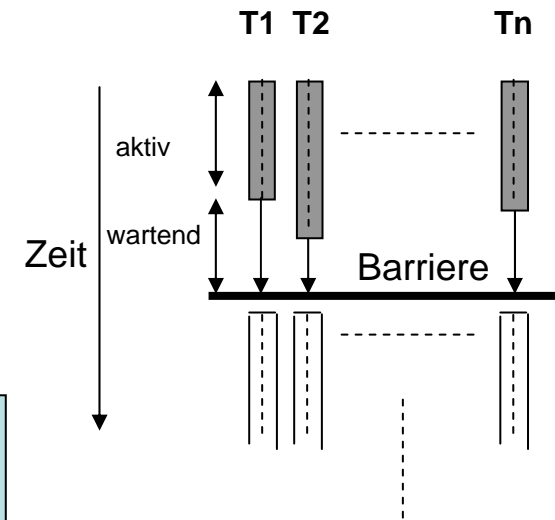
#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

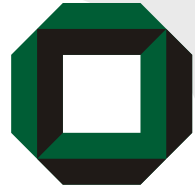


# Barriere

- Explizite Barrierensynchronisation.
- Jeder Faden wartet, bis alle anderen Fäden die Barriere erreichen.

```
#pragma omp parallel shared (A, B, C)
{
    DoSomeWork(A,B);
    printf("Processed A into B\n");
    #pragma omp barrier
    DoSomeWork(B,C);
    printf("Processed B into C\n");
}
```



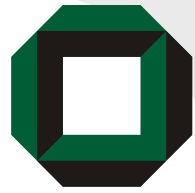


# „Atomic“-Direktive

- Spezialfall eines kritischen Abschnitts.
- Wirkt nur für die eine einfache Aktualisierung einer Speicherstelle (d.h. eine Zuweisung).

```
#pragma omp parallel for shared(x, y, index, n)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
      x[index[i]] += work1(i);
      y[i] += work2(i);
  }
```

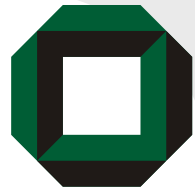
- nur die (schreibenden) Zugriffe auf **dasselbe** Element von x[] werden serialisiert, Zugriffe auf **unterschiedliche** Elemente von x[] können weiterhin parallel ausgeführt werden.



# „Firstprivate“-Direktive

- Kennzeichnet private Variable, aber im Gegensatz zu „private“ ist die Variable nicht uninitialized, sondern wird mit dem Wert der gemeinsamen Variable aus dem umgebenden Block initialisiert.
  - C++ Objects werden mit dem Copy-Konstruktor erzeugt.

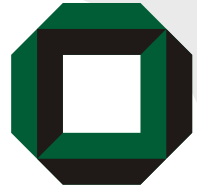
```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
    if ((I%2)==0) incr++;
    A[I]=incr;
}
```



## „Lastprivate“-Direktive

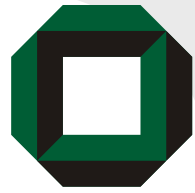
- Die gemeinsame (äußere) Variable wird mit dem Wert aus dem sequentiell letzten Schleifendurchlauf (letzter Iterationsindex) aktualisiert, wenn alle Fäden die Barriere erreicht haben.
- In C++ geschieht dies per Zuweisungsoperator.

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel
    #pragma omp for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    lastterm = x;
}
```



## Noch ein Beispiel zu „lastprivate“

```
void a30 (int n, float *a, float *b)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for lastprivate(i)
        for (i=0; i<n-1; i++)
            a[i] = b[i] + b[i+1];
    }
    a[i]=b[i]; /* hier i == n-1 */
}
```

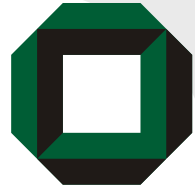


# „Threadprivate“-Direktive

- Globale Lebensdauer für fadenlokale Variablen.
- Nur erlaubt für Variablen mit Datei- oder Namensraum-Sichtbarkeit (Namespace Scope).
- Mittels „copyin“ kann private Kopie mit dem Wert der Variablen im Hauptfaden initialisiert werden.

```
struct Astruct A;  
#pragma omp threadprivate(A)  
...  
#pragma omp parallel copyin(A)  
    do_something_to(&A);  
...  
#pragma omp parallel  
    do_something_else_to(&A);
```

Private Kopien von “A”  
bleiben zwischen den  
parallelen Regionen  
erhalten.



# OpenMP Bibliotheksfunktionen

- Sinnvoll z.B. bei der Fehlersuche.

`int omp_get_num_threads(void):`

- Anzahl der parallelen Threads
- nur >1 in parallelen Abschnitten

`int omp_get_thread_num(void):`

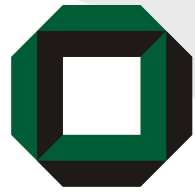
- gibt den Rang diese Threads zurück
- immer 0 für den Master Thread

`int omp_get_num_procs(void):`

- Anzahl der CPUs, die dem Programm zur Verfügung stehen

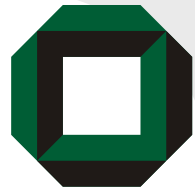
- Einbinden der OpenMP Deklarationsdatei erforderlich:

`#include <omp.h>`



# Überlegungen zur Performanz

- Wartende Kontrollfäden erledigen keine sinnvolle Arbeit.
- Die Arbeit sollte zwischen den Fäden so gleichmäßig wie möglich aufgeteilt werden.
  - Die Kontrollfäden sollten die parallelen Aufgaben alle zur gleichen Zeit beenden.
- Synchronisation kann erforderlich sein.
  - Jedoch: Die Zeit, in der ein Faden auf eine geschützte Ressource wartet, muss minimiert werden.

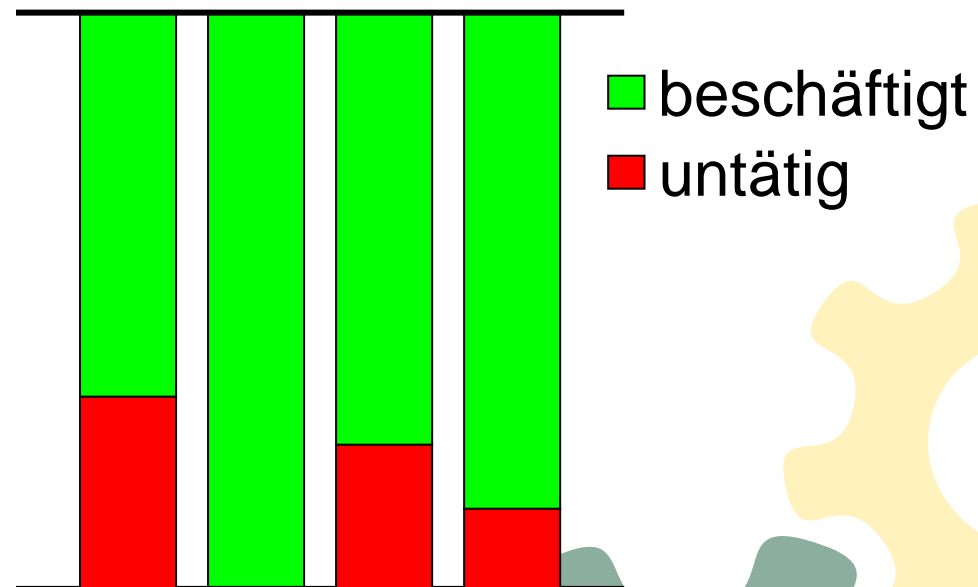


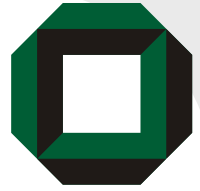
# Ungleiche Lastverteilung

- Ungleiche Verteilung der Aufgaben führt zu unausgelasteten Fäden und damit zu „verlorener“ Rechenzeit.

```
#pragma omp parallel  
{  
  
#pragma omp for  
  for( ; ; ){  
  
  }  
  
}
```

Zeit  
↓

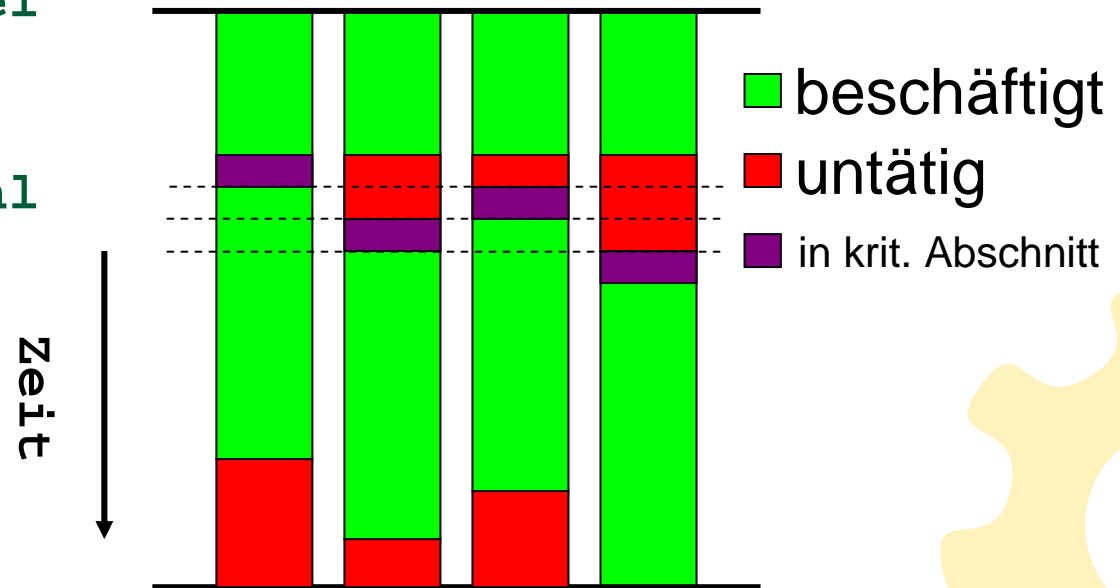


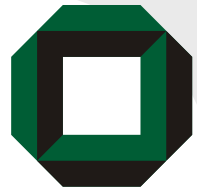


# Synchronisation

- „Verlorene“ Zeit beim Warten auf Ressourcen

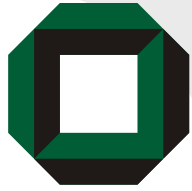
```
#pragma omp parallel  
{  
  
#pragma omp critical  
{  
  ...  
}  
  ...  
}
```





# Zusammenfassung: Programmieren mit OpenMP

- OpenMP ist:
  - Ein einfacher Ansatz für paralleles Programmieren für Systeme mit gemeinsamem Speicher.
- Grundlegende OpenMP-Konstrukte wurden vorgestellt:
  - Erzeuge parallele Code-Abschnitte (`omp parallel`)
  - Teile Arbeit (Schleifendurchläufe) auf mehrere Fäden auf (`omp for`)
  - Lege Sichtbarkeiten für Variablen fest (`omp private ...`)
  - Synchronisiere (`omp critical...`)



# Ausblick OpenMP 3.0 (Auszug aus Draft)

- Neu: Task-Konzept (`#pragma omp task`)
  - Ermöglicht Pointer-Verfolgung und dynamischen Kontrollfluss
  - Beispiel:

```
struct node {
    struct node *left;
    struct node *right;
};

void postorder_traverse( struct node *p ) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
            postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```