# Programming model

## Parallelism

Fork join parallelism

master thread

W.T.

→ Parallele regionen

work sharing

↳ konstrukte

work = loops & task graphs

Finite sed. of tasks

## Threads

SPMD

ID

# Th. > # Processor/core

Par. Ausführung

$\boxed{\text{Shared}}$ &

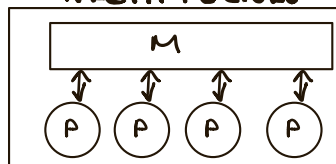
private vars

Ungeschützte par update

→ RACE Condition

Sync. Const um das zu vermeiden

## ARCHITECTURE




SIMD

Naive, Flach, S. m.

processor mem, No explicit cost mem

= Shared memory model

# Open MP

## Basic idea

Graduell parallelisieren

→ loops

locks nicht primär zur Synchronisation wie pThreads

## Task graph

# regionen + work

= längster Pfad

→ gute Kostenfunktion O(w/p) → guter speed up

## Ablauf von Program

seq {

master Th.

By default → var shared

Paralleles Konstrukt

Par. Region

→ private vars

können nicht von and. Th. beeinflusst werden

work sharing über Th.

Th. müssen warten

Implicit barrier

end konstrukt

} seq.

mem. konsistent

& alle Th. updates visible

Daten transfer ist transparent, implicit & kontrollierbar

Schachteln möglich

Fork join

# RACE CONDITIONS

| |
|---|
| When 2 threads update a variable at same time outcome may be write of either thread<br>↳ Result = NON-DETERMINISTIC |
| Special race condition = DATA RACE |
| 2 or more threads access shared memory<br>One of access is a write |

## DATA RACE

= Special kind of race condition

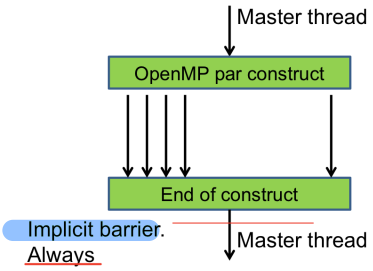
When 2 or more threads access a shared object & at least 1 of the accesses is an UPDATE

→ Undecidable to determine whether data race or not

# Pragmas

- **Parallel Region**

`# pragma omp parallel` → kann #Threads festlegen, d. par. r. ausführen




```
#pragma omp parallel
{
  // threads
}
```

Danach ist memory konsistent

! keine Breaks möglich!
↳ sonst nicht sicherstellen, ob Th. gestartet sind / barrier

DEADLOCK

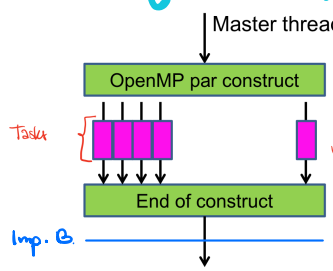
Synchronisierung

Aufgaben:
1. Aktiviert Th.
2. Kontrolliert Vars Sharing zw. Threads
3. Threads werden ASYNCHRON ausgeführt
4. work sharing
5. Endet IMMER mit Impicit barrier
   ↳ Konsistentes memory

- **For**

`#pragma omp for`

Teilung von Arbeit in Schleife
↳ Ebenmäßig (Schedule)



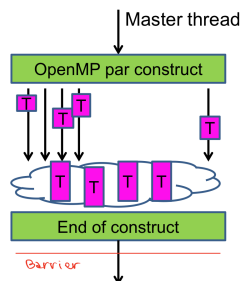

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<n; i++) {
    // iterations split
  }
}
```

Tasks aufgeteilt über Threads

work sharing

- **Task**

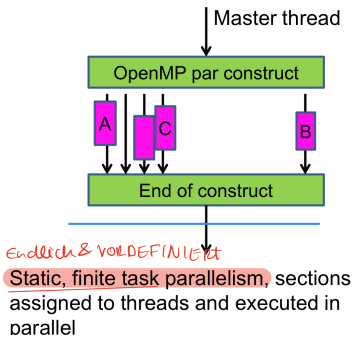
`#pragma parallel task`




Task graphen
Teile von Code = Tasks
→ wird verkapselt & später ausgeführt
→ kann von einem Thread ausgeführt werden
⇒ Alle tasks sind am Ende ausgeführt

- **Sections**

`# pragma omp sections` → `#pragma omp section`




wie Task → gekapselt → wenn section noch nicht ausgeführt → kann

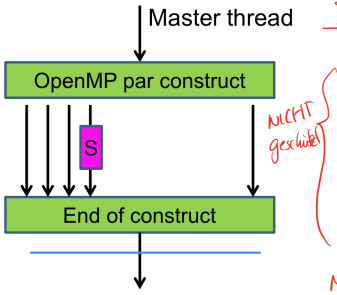
```
#pragma omp parallel
{
  #pragma omp sections
  #pragma omp section
  {
    // A
  }
  #pragma omp section
  {
    // B
  }
  // ...
}
```

jeder Thread in Region ausführen

wie tasks
⇒ Sections werden gekapselt & jeder Thread kann sie ausführen

## • Single

#pragma omp single




kann nur sequentiell ausgeführt werden → jeder Thread in Region

```
#pragma omp parallel
{
  #pragma omp single
  {
    // some thread
  }
}
```

NICHT geschützt

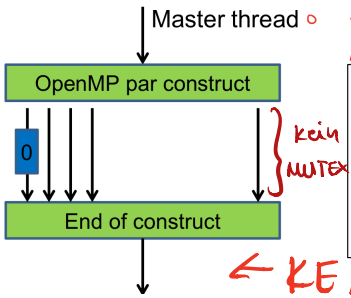
Master & Single ⇒ kein MUTEX

Kein MUTEX

muss gewährleisten, dass andere Threads nicht Single Th. beeinflussen } Th. nebenläufig

## • master

# pragma omp master




vor Region sicher

Manchmal priviligiert

```
#pragma omp parallel
{
  #pragma omp master
  {
    // master thread 0
  }
}
```

kein MUTEX

← KEINE BARRIER

Selbe wie single

→ Nur bestimmter Th: ID = 0

! KEIN MUTEX & KEINE BARRIER !

Kann priviligiert sein

Kann # von Thread festlegen

OMP_NUM_THREAD — überschreibt

omp_set_num_threads(t) — überschreibt

#pragma omp parallel num_threads(t)

→ Wer bin ich? get_thread_num( );

THREADSAFE

## Conditionals möglich

```
#pragma omp parallel if (n<=1000) num_threads(2)
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

  for (i=start; i<end; i++) {
    a[i] = f(i);
  }
}
```

Fixing number of threads for region

Conditional clause, scalar expression evaluated at runtime. If not fulfilled, number of threads in parallel region will be 1

Limit number of threads for small inputs… (weak scaling: little parallelism in small problems; but as written the clause is bad for large problems, only one thread)

Variablen

shared

private = lokale copy per thread

default (shared)

firstprivate

nicht initialisiert → wert wird im Master Thread genommen

Variables declared before parallel region are per default shared for all threads. Beware of race conditions (that will make the program erroneous)

```
int *a;
a = (int*)malloc(n*sizeof(a*));

#pragma omp parallel
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

  for (i=start; i<end; i++) {
    a[i] = f(i);
  }
}
```

Will be shared

Local per thread variables

kann nicht a private machen bei pragma → sonst wert leer/undefin.

Good practice: Disable default rule, explicitly declare sharing for all variables in region

```
int *a;
a = (int*)malloc(n*sizeof(a*));

#pragma omp parallel default(none) \
firstprivate(a)
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

  for (i=start; i<end; i++) {
    a[i] = f(i);
  }
}
```

Pointer is private

Better: initialized, pointer copied

## Klauseln

Summary of clauses for `#pragma omp parallel`

```
if (scalar_expression)
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction(operator: list)
copyin(list)
num_threads(integer-expression)
proc_bind(master|close|spread)
```

See later for reduction clause

See later on "pinning"

- Reduction clause: Important OpenMP capability, see later
- Binding threads to cores: proc_bind, see later lecture

## Work sharing Konstrukte

End. #Th. kann par. ausgeführt werden

- Sections

```
#pragma omp parallel
#pragma omp sections [clause ...]
{
#pragma omp section
  A(…);
#pragma omp section
  B(…);
#pragma omp section
  {
    … // explicit code for some task
  }
}
#pragma omp section
…
}
```

More sections than threads: Some threads execute more than one section

Scheduling (which thread executes which section) implementation dependent

Arbeitslast mögl. gleich verteilen

Summary of clauses for `#pragma omp sections`

```
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait
```

A bit more later

For `reduction` and `nowait` and `lastprivate`, see later…

- master / single

## Single construct, master construct

Egal welcher Thread  Thread mit id = 0 (= master thread)

Block inside parallel region that is to be executed by only one thread, either arbitrarily, or by master thread (recall: master thread is the thread with `omp_get_thread_num()==0`)

```
#pragma omp single [clause]
```

Some, but only one, thread executes block; implicit barrier after block

```
#pragma omp master
```

Master thread executes block, no barrier

```
#pragma omp parallel
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

#pragma omp master
  readarray(b,n);

  for (i=start; i<end; i++) {
    a[i] = b[i];
  }

#pragma omp barrier
#pragma omp single
  writearray(a,n); // all updates done!
}
```

Master thread reads; dangerous because no barrier

Explicit barrier

Implicit barrier here

Summary of clauses for `#pragma omp single`

```
private (list)
firstprivate (list)
nowait
```

`nowait`: Implicit barrier synchronization at end of construct will not take place

`nowait` also for `sections`, `for` constructs (but not for `parallel` construct)

Practice:
Use `nowait` with care for performance improvement, add `nowait`'s gradually, but correctly

- Barrier

## Barrier

Threads müssen auf andere Threads (compl.) warten

Inside parallel region construct, threads can be (semantically) synchronized by

```
#pragma omp barrier
```

An OpenMP thread reaching the barrier construct cannot proceed before all other threads have reached the barrier

By the semantics of barriers, all threads must do the same number of barriers, otherwise deadlock

single

```
#pragma omp parallel
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

#pragma omp single
  readarray(b,n);
  for (i=start; i<end; i++) {
    a[i] = b[i];
  }
#pragma omp single
  printf("now done?");
}
```

wait until all is read

Barrier

Implicit barrier, all threads will see their part of the array

1. Thread, der fertig ist, printet

Keine Barrier

Dangerous: No barrier before single

```
#pragma omp parallel
{
  int i;
  int block = n/omp_get_num_threads();
  int start = omp_get_thread_num()*block;
  int end = start+block;

#pragma omp single
  readarray(b,n);

  for (i=start; i<end; i++) {
    a[i] = b[i];
  }

#pragma omp barrier
#pragma omp single nowait
  printf(„now done?");
}
```

Remove implicit barrier

Entfernt impl. Barrier!

Implicit barrier here

Example: Merging by coranking

```
// global, read-only values A, B, n, m

int j[p+1], k[p+1]; // allocate in advance
j[p] = n; k[p] = m;
#pragma omp parallel
{
  int i = omp_get_thread_num();
  int p = omp_get_num_threads();

  corank(i*(n+m)/p,A,n,&j[i],B,m,&k[i]);
#pragma omp barrier
  merge(&A[j[i]],j[i+1]-j[i],
        &B[k[i]],k[i+1]-k[i],
        &C[i*(n+m)/p]);
}
```

Careful with last block

synchronisierung!

Explicit barrier

# For

## Parallel for

The basic work sharing construct: Iterations of a loop of independent iterations distributed among the available threads

```
#pragma omp for [clause]
```

Example: loop parallelization

```
for (i=0; i<n; i++) {
  a[i] = f(i);
}
```

Implicit barrier at end of block

Parallel loop shorthand

```
int i;

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

Probably most frequently used OpenMP construct

- Convenient shorthand for first opening parallel section, and then parallelizing the loop
- Implicit barrier after loop
- No break, or jump into/out of loop
- No `nowait` clause for shorthand (because no `nowait` for `parallel`)

ILLEGAL

Example: Some illegal loops

```
#pragma omp parallel for
for (;;) {
  // C open loop
}
```

Number of iterations unknown, not in canonical form

```
#pragma omp parallel for
for (i=0; i<n; i++) {
  if (exceptional(i)) break;
  if (i%2==0) continue;
}
```

Break out of loop illegal. Continue ok, does not change number of iterations

Also illegal

```
#pragma omp parallel for
for (i=0; i<n; i*=2) {
  a[i] = …;
}
```

ILLEGAL!

Number of iterations can be computed, floor($\log_2$ n), but loop itself not in canonical form

Reason:
Hard(?) for compiler to detect and transform into linear code

LASTPRIVATE

Example: Getting some last value

```
int t; // global

#pragma omp parallel for lastprivate(t)
for (i=0; i<n; i++) {
  if (event(i)) t = i; else t = -1;
}
// now t is index of last event or -1
```

Now private copies of t

`lastprivate(t)` clause causes the value of `t` from the last iteration to be copied to the global `t`

Bsp.:

Kanonische Form von Schleifen!

```
#pragma omp parallel
{
  int i;

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

Local variables in block per default private

Iteration variable is per default private

Implicit barrier

Implicit barrier, again

Loop iterations divided according to default schedule across threads

darf nicht while o. dowhile sein

Basic rule:
Total number of iterations must be known before loop, and all threads must compute same iteration bound

```
#pragma omp parallel
{
  int i;

#pragma omp for nowait
  for (i=0; i<n; i++) {
    a[i] = f(i);
  }
}
```

Remove implicit barrier

Implicit barrier, still

Loop iterations divided according to default schedule across threads

Basic rule:
Total number of iterations must be known before loop, and all threads must compute same iteration bound

Canonical form for parallel for loops

`i`: iteration variable
`i0`: lower bound
`n`: upper bound
`inc`: increment

```
for (i = i0; i<n | i<=n | i>=n | i>n ; i++ | ++i | i- | --i | i+=inc | i-=inc | i=i+inc | i=i-inc ) { <body> }
```

- No `break`, `goto` out of/into loop body; `continue` allowed
- Lower, upper, increment expressions must not change during loop iterations, must be known before loop starts, and same for all threads

Canonical form allows static scheduling: Assignment of loop iterations to threads

Summary of clauses for `#pragma omp for`

```
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait
```

Reduction, see later… For collapse, see later… For ordered, more later (don't use)

`lastprivate(a)` clause causes the value of a from the last iteration to be copied to the global `a`

Example: Initializing the matrix, collapsing loops

OpenMP can parallelize nested loops, certain to some constraints

```
#pragma omp parallel for collapse(2)
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    x[i][j] = f(i,j); // thread safe
                      // but time depends on i, j
  }
}
```

Any number of loops

Two loops are handled as one with iteration space of n*m iterations, proper schedule perhaps easier to find

Condition: Both loops must be in canonical form, both iteration spaces must be known prior to the outer loop (e.g., m cannot depend on n)

# Schedule

- `schedule(auto)`: schedule is chosen by compiler or runtime
- `schedule(runtime)`: schedule left to runtime & environment

## Parallel for schedules

p: number of threads, n: number of iterations

- Per default, iteration space divided into p blocks of approx. n/p consecutive iterations, one block is assigned to each thread (unspecified exactly what approx. means). Blocks of consecutive iterations are called chunks in OpenMP terminology
- Schedule (size of chunks, assignment of chunks to threads) can be influenced by schedule clause

Best schedule depends on many factors: Work per iteration, data structure access pattern, …

Chosen schedule can have a huge effect on performance (false sharing, e.g.)

Schedule clause, `chunksize` optional

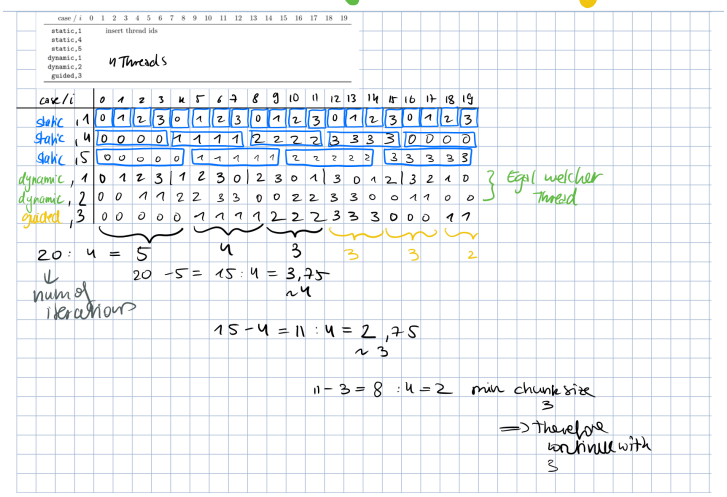
- `schedule(static[,<chunksize>])`: Iterations divided into chunks of size `chunksize` (default approx. n/p), chunks assigned to threads in a round robin fashion
- `schedule(dynamic[,<chunksize>])`: Chunks are distributed to threads as threads become free and request work (default `chunksize` 1)
- `schedule(guided[,<chunksize>])`: As dynamic, but the actual `chunksize` for each thread is the number of unassigned iterations divided by p, but no smaller than `chunksize` (default 1)

```
#pragma omp parallel for schedule(…,<chunksize>)
for (j=0; j<n; j++) {
  …
}
```

For static and dynamic schedules:
Each chunk covers `chunksize` iterations, number of chunks is ceil(`n/chunksize`), i'th chunk, 0≤i<ceil(`n/chunksize`), starts at index i*`chunksize`

For guided schedule:
The size of i'th chunk is max(`chunksize`,(n-∑(done and assigned iterations))/p)

static dynamic guided




```
#pragma parallel for <schedule>
(i=0; i<n; i++) {
  if (condition(i,b,c)) a[i] = f(a[i],b[i]);
}
```

- If `condition(i,…)` holds often, the loop has high spatial locality. In this case, `chunksize` could be a multiple of the cache line size to avoid false sharing
- If `condition(i,…)` holds infrequently, without predictable pattern, dynamic or guided schedule with small `chunksize` could be better

## Schedules, performance, some rules of thumb

Assume n large, p (number of threads) not so large:

- `chunksize` small: Scheduling overhead, each thread must handle several chunks. Static schedule has small scheduling overhead (div and mod operations), dynamic and guided schedules have larger overhead (atomic operation or worse)
- `chunksize` large: Risk of load imbalance, if some chunks have significantly more work than others, the threads handling the expensive chunks will do most of the work
- If known in advance that the work per chunk (for some `chunksize`) is more or less same, static schedule preferable; if not, dynamic or guided may do better
- If work per iteration differs significantly, small chunksize could be preferable

correctness

## Correctness, independence

**OpenMP principle:**
Parallel regions and all work sharing constructs assume that code regions executed by threads can be safely executed in parallel, i.e., no concurrent updates to shared variables

Code region executions must be independent: No update to a shared variable in one region can have an effect on other region (race conditions)

**OpenMP principle:**
It is the programmer's responsibility to ensure independence. Compiler & runtime are not required to check (and will not/cannot do everything)

Some loop carried dependencies (anti-dependencies)

```
for (i=k; i<n; i++) a[i] = a[i]+a[i+k];
```

can be eliminated with temporary variables

```
for (i=k; i<n; i++) aa[i] = a[i]+a[i+k];
// swap
tmp = a; a = aa; aa = tmp;
```

Simple rules for correct OpenMP parallelizable loops

1. Array updates only
2. Each array element updated in at most one iteration
3. No iteration reads element assigned by another iteration

Dependencies within same iteration allowed

```
for (i=0; i<n; i++) {
  a[i] = f(i);
  b[i] = g(i);
  c[i] = a[i]+b[i];
}
```

ok

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

Flow dependency within iteration i

Example: Parallel prefix-sums computation (1st loop)

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

No loop carried dependency; `x[i]` are every `kk`'th element and not read or updated in any other iteration

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
#pragma omp parallel for
  for (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
}
```

Implicit barrier after parallel for region

# Reduction

The prefix-sums pattern

```
for (i=k; i<n; i++) a[i] = a[i-1]+a[i];
```

differs from the reduction pattern

```
for (i=k; i<n; i++) sum = sum+a[i];
```

Reduction pattern:
Special kind of dependency that can be handled by OpenMP; one global reduction variable referenced in each iteration

Simple reduction in OpenMP:

More later

Options: + - * | & && || min max

```
sum = 0;
#pragma omp parallel for reduction(+: sum)
for (i=k; i<n; i++) sum = sum+a[i];
```

→ Abhängig!

Reduction operation and variables specified in reduction clause, reduction performed in index order, operation (here: +) assumed to be associative (why? See algorithms lecture)

OpenMP compiler/runtime applies suitable, parallel algorithm for the reduction; hopefully good, but OpenMP does not give any performance guarantees

Recall: O(n/p+log n) algorithms for reduction/prefix-sums

# • SIMD

Auf jedem Thread SIMD operationen!

SIMD

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

Sequential, "vectorizable" loop: Independent iterations that may be mapped to special instructions (vector operations)

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

Parallel, "vectorizable" loop

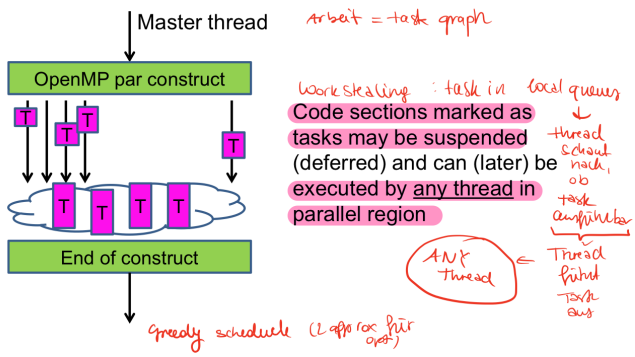
Summary of clauses for `#pragma omp [for] simd`

```
safelen(length)
simdlen(length)
linear(list[ : linear-step])
aligned(list[ : alignment])
private(list)
lastprivate(list)
reduction(reduction-identifier : list)
collapse(n)
```

Preferred length of vector-SIMD operations can be given as `simdlen`, `safelen` controls how compiler schedules vector instructions, concurrent iterations are no more than `safelen` iterations apart

# • Task

Work sharing construct: task




Quicksort calls as explicit tasks

```
int main(int argc, char *argv[]){
…
#pragma omp parallel
  {
#pragma omp single nowait
    QuickSort(x,n);
  }
  return 0;
}
```

Activate all default threads

BUT only one thread does Quicksort

An arbitrary task (single) executes QuickSort, parallelism comes from the tasks that may be executed by any available thread. Code in parallel region can have any number of task constructs

Quicksort calls as explicit tasks

```
void QuickSort(int x[],int n) {
  if (n<=1) return;

  pivot = choosepivot(x,n);
  ix = partition(x,n,pivot);
#pragma omp task shared(x)
  QuickSort(x,ix);
#pragma omp task shared(x)
  QuickSort(x+ix+1,n-1-ix);
//#taskwait
}
```

sort kleine Elems

sort große Elems

Mark independent `QuickSort` calls as tasks. Spawned at runtime

Wait for completion of immediately spawned tasks (not needed here)

Quicksort calls as explicit tasks

```
int main(int argc, char *argv[]){
…
#pragma omp parallel
  {
#pragma omp single nowait
    QuickSort(x,n);
  }
  return 0;
}
```

Activate all default threads

BUT only one thread does Quicksort

Implicit and explicit barriers wait for all tasks to complete (immediate tasks and descendants): For QuickSort, `taskwait` is not necessary.

Example: Counting occurrences in unordered array

```
int search(int x, int a[], int n)
{
  if (n==1) {
    return (a[0]==x) ? 1 : 0;
  } else {
    int s0, int s1;
#pragma omp task shared(s0,a)
    s0 = search(x,a,n/2);
#pragma omp task shared(s1,a)
    s1 = search(x,a+n/2,n-n/2);
#pragma omp taskwait
    return s0+s1;
  }
}
```

Since task can be executed by any thread in region, and updates result in s0, s0 must be shared

Tasks must be completed before sum of occurences can be computed

The `#pragma omp taskwait` construct waits for completion of immediately generated tasks (children), not transitively for all successor tasks

This gives rise to fully strict DAGs

Explicit `#pragma omp barrier` and implicit barriers (exit from parallel region and constructs) waits for completion of all generated tasks

Summary of clauses for `#pragma omp task`

```
if(scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
depend(dependence-type : list)
priority(priority-value)
```

Correct sharing important for correctness

Small tasks may have too much overhead, tasks can be marked as mergeable (at runtime)
Advanced features (?): Task can be prioritized, dependencies can be specified

Special for loops (II): Taskloops

```
#pragma omp parallel
#pragma omp single nowait
#pragma omp taskloop collapse(2)
for (i=0; i<m; i++) {
  for (j=0; j<i; j++) {
    a[i][j] = i+j;
  }
}
```

Try it!

Loops can be parallelized by generating tasks for chunks of iterations. Can give more flexibility to the runtime and result in better performance (sometimes)

Same rules as for standard loops (canonical form), but no schedule clause

Summary of clauses for `#pragma omp taskloop`

```
if([ taskloop :] scalar-expr)
shared(list)
private(list)
firstprivate(list)
lastprivate(list)
default(shared | none)
grainsize(grain-size)
num_tasks(num-tasks)
collapse(n)
final(scalar-expr)
priority(priority-value)
untied
mergeable
nogroup
```

Clauses for controlling the size and number of tasks

Cilk
→ Extension für task-parallel prg
↳ nicht von gcc unterstützt
→ Erweiterung von C

spawn
Spawn task
#pragma omp task

Sync
warten auf Kompeh. von child
#pragma omp taskwait

for
loops as tasks
# pragma omp task loop

kanonische For

## End of Cilk aside, back to OpenMP

- Loops with dependencies?
- Critical sections, atomic operations, reductions

× wären nicht data race – frei → keine gültige Programme für OpenMP

Solutions

Solution 1: Enforce sequential order

```
#pragma omp parallel for
for (i=2; i<n; i++) mark[i] = 1;

for (i=2; i*i<n; i++) {
  if (mark[i]) {
#pragma omp parallel for
    for (j=i*i; j<n; j+=i) mark[j] = 0;
  }
}
k = 0;
#pragma omp parallel for ordered
for (i=2; i<n; i++) if (mark[i]) {
#pragma omp ordered
  prime[k++] = i;
}
```

Sequential order will be enforced for ordered region

## Reductions

```
sum = 0;
for (i=0; i<n; i++) sum += i;
```

Standard loop structure with flow and output dependencies:

```
sum = 0;
for (i=0; i<n; i++) sum += expr(a[i]);
```

Such reduction patterns can be recognized by OpenMP compiler, and efficient algorithm & runtime support used

Summary: Four ways of handling simple reduction (no data races)

```
for (i=k; i<n; i++) sum = sum+a[i];
```

Canonical way, parallel thinking (parallel algorithm), best potential for good performance: Reduction clause

```
#pragma omp parallel for reduction(+,sum)
for (i=k; i<n; i++) sum = sum+a[i];
```

Home exercise:
Check quality of OpenMP implementation by comparing this to hand-written parallel reduction algorithm

Sequential thinking: Enforce sequential order with ordered

```
#pragma omp parallel for ordered
for (i=k; i<n; i++) {
#pragma omp ordered
  sum = sum+a[i];
}
```

Performance penalty:
Probably close to sequential loop (plus overhead?)

This loop carried dependency: Two possible solutions

0. Give up, drop `parallel for`
1. Ensure sequential order and hope for the best
2. Parallel solution, but ill supported by OpenMP

Recall:
It was required that the primes are stored in increasing order

Solution 2: Index computation in parallel

```
#pragma omp parallel for
for (i=2; i<n; i++) mark[i] = 1;

for (i=2; i*i<n; i++) {
  if (mark[i]) {
#pragma omp parallel for
    for (j=i*i; j<n; j+=i) mark[j] = 0;
  }
}
k = 0; kix[0] = 0; kix[1] = 0;
#pragma omp parallel for
for (i=2; i<n; i++) kix[i] = (mark[i]) ? 1 : 0;
Exscan(kix,n); // exclusive prefix-sums
#pragma omp parallel for
for (i=2; i<n; i++) if (mark[i]) prime[kix[i]] = i;
```

Array compaction with exclusive scan (prefix-sums)

Reduction pattern parallelization

```
sum = 0;
#pragma omp parallel reduction(+:sum)
for (i=0; i<n; i++) sum += i;
```

`reduction(<operator>:<variable list>)` clause specifies reduction with operator on list of variables

Operator: `+, *, -, &, |, ^, &&, ||` and min/max computations

Concurrent thinking: Critical section

```
#pragma omp parallel for
for (i=k; i<n; i++) {
#pragma omp critical
  sum = sum+a[i];
}
```

können Reihenfolge nicht enforcen ⇒ auch bei locks das Problem

Could perform reasonably if much other work in parallel region for the threads

Correct only if operator + is commutative! No guarantee in which order threads execute critical section

Variant: Dynamic locks, same problems

Delegate to hardware: Atomic operations

```
#pragma omp parallel for
for (i=k; i<n; i++) {
#pragma omp atomic update
  sum = sum+a[i];
}
```

Atomic update, expression in canonical form

Correct only if operator + is commutative! No guarantee in which order threads execute atomic operation (or how threads are serialized in case of simultaneous execution)

DATA RACE

## Other OpenMP ways of dealing with data races

OpenMP data race:
Parallel threads update an "unprotected" shared variable (scalar, array element) in such a way that the outcome is dependent on the relative order in which the threads perform their updates

Programs with such race conditions are illegal (note: cannot in general be checked by compiler). Shared variable updates must be protected by appropriate OpenMP construct

- Critical section, lock
- Atomic operation

Example: Counting accesses

```
int t;
t = 0;

#pragma omp parallel shared(t)
{
#pragma omp critical
  t++;
#pragma omp barrier
  print("Threads so far %d of %d by %d\n",
        t,omp_get_num_threads(),omp_get_thread_num());
}
```

Now in critical section, mutual exclusion on update of shared t guaranteed

Smart data race

```
#pragma omp atomic read
pvar = svar; // read shared variable atomically
```

```
#pragma omp atomic write
svar = pvar; // write to shared variable atomically
```

updates

```
#pragma omp atomic update
<expression statement>
```

```
#pragma omp atomic capture
<expression statement>
```

```
#pragma omp atomic capture
<structured block>
```

## Performance issues, critical sections, atomics

- Critical sections, locks: Slow, incur software/hardware locking/mutual exclusion overhead (some algorithm, OS interaction). If thread in critical section is delayed, other threads waiting for critical section must wait, can lead to serialization (Amdahl)

teuer ↓ 1 thread

- Atomic operations: Expensive, possibly slow, hardware must, usually, flush cache lines, expensive traffic on memory system. But usually (much) faster than critical sections, and if supported by hardware, threads cannot affect progress of other threads

## Critical sections, locks

A (named) critical section in parallel region enforces mutual exclusion in code region: At most one thread at a time executes critical section

```
#pragma omp critical [(<name>)]
```

Critical sections are statically designated (compile time)

Shared variables can be freely updated in a critical section, such updates are not race conditions (although there is no guarantee in which order threads can execute their critical section)

## Atomic operations

Concurrent read-modify-write on shared variable, supported by hardware, performed atomically: Operation cannot be interrupted by other threads, and completes in finite (bounded) time

- Read
- Write
- Update
- Capture (fetch-and-update)

Atomic updates on shared variables are not race conditions

in vielen Situationen sehr teuer

Optimistic performance assumption:
Hardware can complete atomic operation in O(1) time, independently of the number of other threads that may also perform the operation

Example: Counting accesses

```
int t;
t = 0;

#pragma omp parallel shared(t)
{
#pragma omp atomic update
  t++;
#pragma omp barrier
  print("Threads so far %d of %d by %d\n",
        t,omp_get_num_threads(),omp_get_thread_num());
}
```

schützen!

Atomic instead of critical section, probably better

## OpenMP Discussion

Easy to use language additions for parallel thread programming: Data parallel loops, tasks, access to possibly hardware-supported atomic operations, some low-level concurrent programming primitives

hauptsächlich schleifen parallelisiert + tasks + critical sections, ...

An OpenMP program should be a correct sequential program (with sequential versions of OpenMP library routines)

Stepwise parallelization idea?

- Program structure will need change (e.g. canonical loops, dependency elimination)
- Parallel algorithm still needed, parallelization of given sequential program may not go far (e.g., prime sieve example, √n non-parallelizable, dependent iterations)