Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 6: Advanced OpenMP and performance issues

High Performance Computing

Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Miscellaneous

Environment variables and API calls

High Performance Computing

Environment variables

- **`OMP_NUM_THREADS`**
  - (int) Number of threads to use in parallel regions if not set from code
- **`OMP_SCHEDULE`**
  - ([modifier:]type[,chunksize]) Parallel schedule to use for runtime-scheduled loops
  - Modifier: **`monotonic|nonmonotonic`** : are iterations fed to threads in original order? **`simd`**: make chunk size a multiple of SIMD width
- **`OMP_PLACES`**
  - Unit for placement of threads
- **`OMP_PROC_BIND`**
  - How threads should be put into places

Environment variables cont'd

- **OMP_STACKSIZE**
  - (int + B|K|M|G) Per-thread stack limit
- **OMP_DYNAMIC**
  - (TRUE|FALSE) Allow/disallow dynamic adjustment of thread count by runtime
- **OMP_WAIT_POLICY**
  - (ACTIVE|PASSIVE) What should threads do when waiting?
- **OMP_DISPLAY_AFFINITY**
  - (TRUE|FALSE) Display affinity info
- **OMP_AFFINITY_FORMAT**
  - Specify affinity output format (see standard)

Some API routines

- `omp_set_num_threads(int);`
  - Set no of threads is subsequent parallel regions without `num_threads` clause
- `int omp_get_num_threads();`
  - Number of threads in current team
- `int omp_get_thread_num();`
  - ID of calling thread
- `int omp_get_num_procs();`
  - Number of available processors
- `int omp_in_parallel();`
  - Determine if execution is within parallel region
- `omp_display_affinity();`
  - Print affinity info on stdout

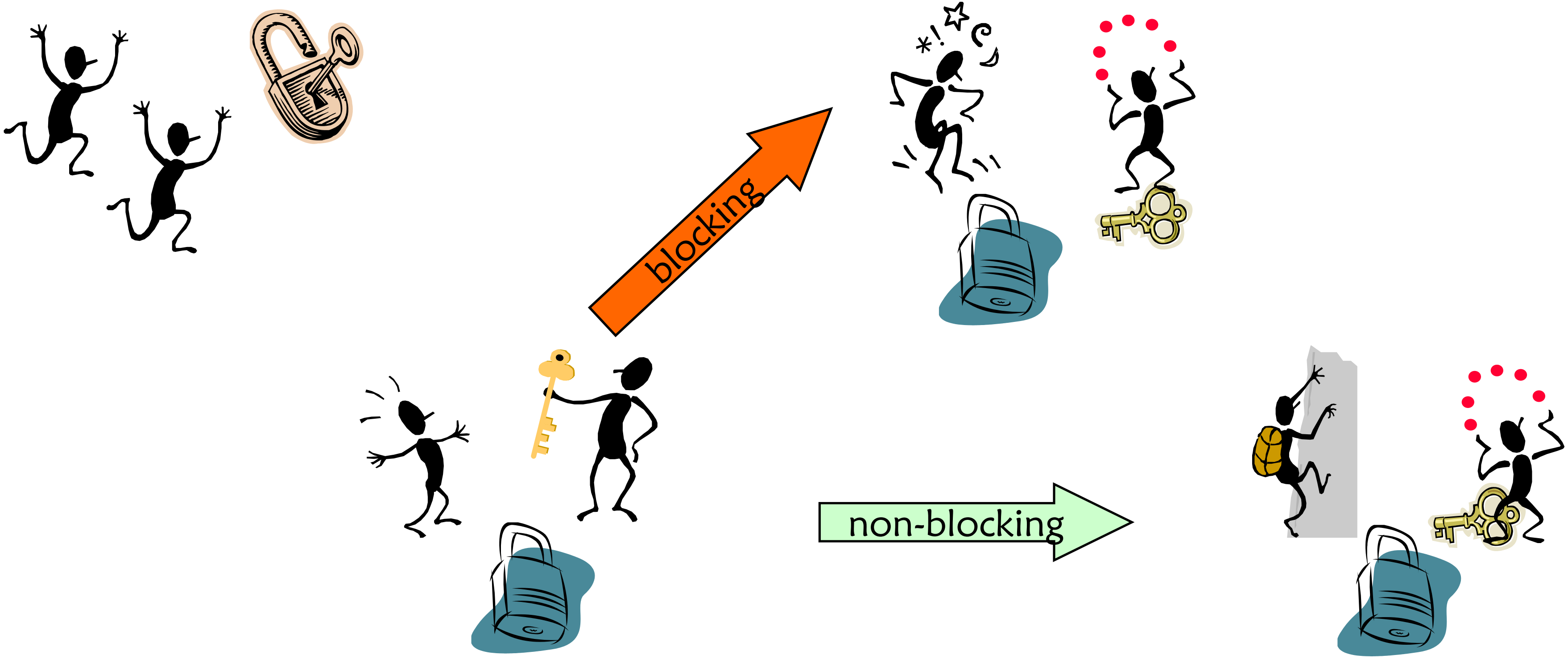
- `int omp_get_max_threads();`
  - # threads in next parallel region
- `double omp_get_wtime();`
  - Get time stamp
- `double omp_get_wtick();`
  - Seconds between successive time stamps

More OpenMP synchronization

OpenMP locks

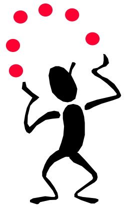
Lock synchronization

Shared lock variables allow fine-grained synchronization



Types of locks: simple and nestable

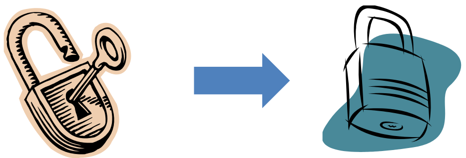
- C/C++ lock data types: **omp_lock_t**, **omp_nest_lock_t**
  - Fortran: **integer(omp_lock_kind)**, **integer(omp_nest_lock_kind)**
- Initialize a lock
  - **omp_init_lock(omp_lock_t *)**, **omp_init_nest_lock(omp_nest_lock_t *)**
- Object(s) protected by lock: defined by programmer
  - Lock must be initialized
  - Initial state: unlocked



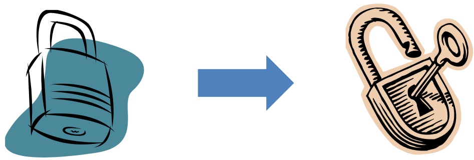
- Nested lock: may be locked/unlocked multiple times by same task/thread
- **omp_destroy_lock(omp_lock_t *)**, **omp_destroy_nest_lock(omp_lock_t *)**
  - Disassociate (initialized) lock variable from lock

Simple lock routines

- `void omp_set_lock(omp_lock_t *)`
  - Blocks if lock not available
  - Sets ownership and continues execution if lock available



- `void omp_unset_lock(omp_lock_t *)`
  - Release ownership of lock
  - Ownership must have been established before

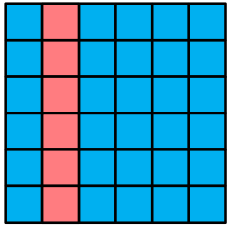


- `int omp_test_lock(omp_lock_t *)`
  - If lock set: return false
  - If lock free: set lock and return true

Lock example: column updates on a matrix

```
double m[N][N];
omp_lock_t locks[N];
#pragma omp parallel
{
#pragma omp for
  for(int i=0; i<N; ++i)
    omp_init_lock(&locks[i]);
...
#pragma omp for
  for(int i=0; i<K; ++i) {
    int c = col_calc(i);
    omp_set_lock(&locks[c]);
    for(int j=0; j<N; ++j)
      m[c][j] += f(c);
    omp_unset_lock(&locks[c]);
  }
}
```

Initialize all locks



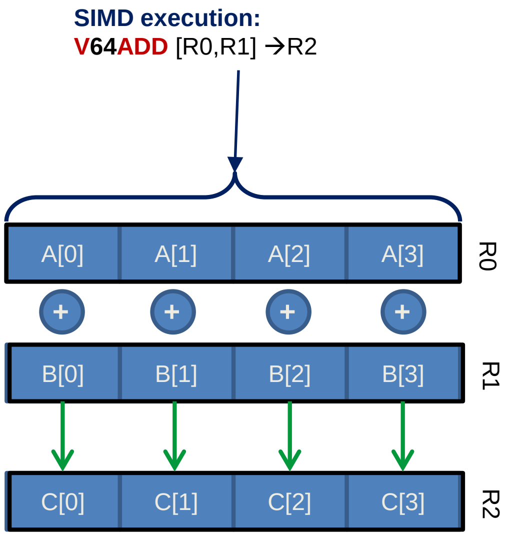
Protect update of column **c**

Is there an even better solution?

SIMD support in OpenMP

Why SIMD in OpenMP?

- Recurring challenges with SIMD
  - How to tell the compiler "it's OK – go ahead!"?
  - Interaction of loop chunk size and SIMD width
  - Variables whose relationship to the "SIMD direction" is unclear
  - Making SIMD available for function calls
- Reminder: SIMD has nothing to do with threading
  - … but is has everything to do with data parallelism
  - Special instructions work on vectors of operands
  - SIMD support in OpenMP since v. 4.0



SIMD construct

- User-mandated vectorization
- `#pragma omp simd` enables vectorization of a loop
  - Essentially a standardized “go ahead, no dependencies here!”
  - Do not lie to the compiler here!

```
// a[] and b[] do not
// overlap in a bad way
#pragma omp simd
for(int i=0; i<N; ++i)
  a[i] = s * b[i];
```

- Prerequisites
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs

- Clauses: `simdlen`, `linear`, `safelen`, `reduction`, `(first)private`, ...

SIMD construct clauses

- **simdlen(int)**
  - Preferred SIMD width in iterations (hint to the compiler)

- **safelen(int)**
  - No loop-carried dependencies for vectors of the specified size or below
  - Example:

```
#pragma omp simd safelen(8)
for (int i = k; i < n; ++i)
    b[i] = s * b[i-k];
```

    - This code is safe to vectorize with SIMD width up to 8 if $k \geq 8$

SIMD construct clauses

- **`linear(`*list*`[:`*step*`])`**
  - Linear relationship of induction variables (in list) to the loop counter

```
#pragma omp simd reduction(+:s) linear(p:2)
for(int i=0; i<N; ++i) {
  s += a[i] * b[i];
  q[p] += r[p];
  p += 2;
}
```

  - Enables the compiler to employ SIMD in presence of induction variables
  - After the loop: induction variable has the same value as in serial execution
  - Also applicable to workshared for loops

SIMD clause for workshared loops

- SIMD clause can be combined with OpenMP worksharing

```
#pragma omp for simd schedule(simd:static,c)
   for(int i=0; i<N; ++i)
     a[i] = exp(b[i]);
```

Compiler will use SIMD version of function if present

Extend chunk size to next SIMD width multiple

- Some compilers will automatically vectorize loops with calls to some intrinsic functions (e.g., Intel – SVML library)

SIMD functions

- Functions and subroutines can be declared as SIMD vectorizable and called from SIMD loops

```
#pragma omp declare simd
double hyp3d(double k, double l, double m) {
 return sqrt(k*k + l*l + m*m);
}

...

double a[N], b[N], c[N], hyp[N];
#pragma omp parallel for simd
  for(int i=0; i<N; ++i)
    hyp[i] = hyp3d(a[i],b[i],c[i]);
```

Makes compiler generate SIMD version(s) of the function

SIMD loop calls SIMD version of function

SIMD functions

- More flexible SIMD specifications for functions

```
#pragma omp declare simd linear(s:1) uniform(p,q,r) simdlen(4)
#pragma omp declare simd linear(s:1) uniform(p,q,r) simdlen(8)
double hyp3d_i(double *p, double *q, double *r, int s) {
  return sqrt(p[s]*p[s] + q[s]*q[s] + r[s]*r[s]);
}

...

double a[N], b[N], c[N], hyp[N];
#pragma omp parallel for simd
  for(int i=0; i<N; ++i)
    hyp[i] = hyp3d_i(a,b,c,i);
```

Declares linear relationship of variables to SIMD index

Declares variables to be invariant across SIMD index

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

OpenMP tasking

High Performance Computing

Tasks vs. threads

- Parallelism is not just about loops
- Data is not just about arrays: lists, trees, …
- OpenMP tasking constructs: **task**, **taskloop**

Single thread generates tasks

Task = code with data environment

Threads at task scheduling points are eligible for task execution

General pattern:

```
#pragma omp parallel
{
  #pragma omp single
  {
    ...
    #pragma omp task
    {
      ...
    }
  }
}
```

Basic tasking

- `#pragma omp task`
  `structured-block`
- Example: Execute function in loop only with some probability per iteration

`i` automatically firstprivate

`p[]` stays shared

  - **`private`** variables in enclosing context are automatically **`firstprivate`** per task

```
int i;
struct object p[N];
...
#pragma omp parallel private(r,i)
{
  #pragma omp single
  {
    for(i=0; i<N; ++i) {
      r = rand()/(double)RAND_MAX;
      if(p[i].weight > r) {
        #pragma omp task
          do_work_with(&p[i]);
      }
    }
  }
}
```

Flexibility of tasks

- Tasks do not all have to execute the same code
- Example: Overlapping communication and computation

Communication task

Computation tasks

```
int i;
struct object p[N], q[N];
...
#pragma omp parallel private(r,i)
{
  #pragma omp single
  {
    #pragma omp task
      communicate(q);
    for(i=0; i<N; ++i) {
      r = rand()/(double)RAND_MAX;
      if(p[i].weight > r) {
        #pragma omp task
          do_work_with(&p[i]);
} } } }
```

Tasks from loops: taskloops

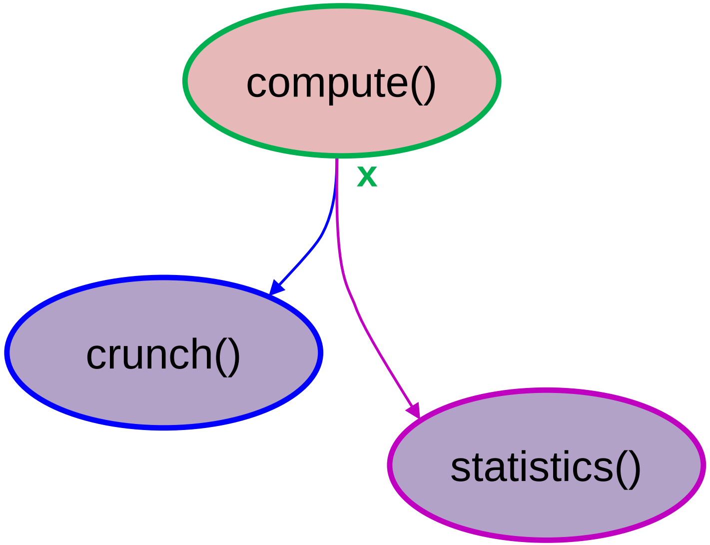
- Combining parallel loops with tasks is cumbersome if the **`task`** construct is all you have
- **`#pragma omp taskloop [clauses]`**
  `for-loop`

  breaks loop into chunks and makes them tasks
- By default implies a taskgroup construct: All tasks finish before loop is left
  - Disable with **`nogroup`** clause

```
struct object q[N];
double tmp, a[N], b[N], c[N];
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
      communicate(q);
    #pragma omp taskloop \
            grain_size(100)
      for(int i=0; i<N; ++i) {
        double tmp = func(c[i]);
        a[i] = b[i] + tmp;
      }
  }
}
```

Task dependencies

- Many problems require tasks to be executed only after other tasks are completed




```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task depend(out:x)
      x = compute();
    #pragma omp task depend(in:x)
      y += statistics(x);
    #pragma omp task depend(in:x)
      z = crunch(x);
  }
}
```

Task dependencies

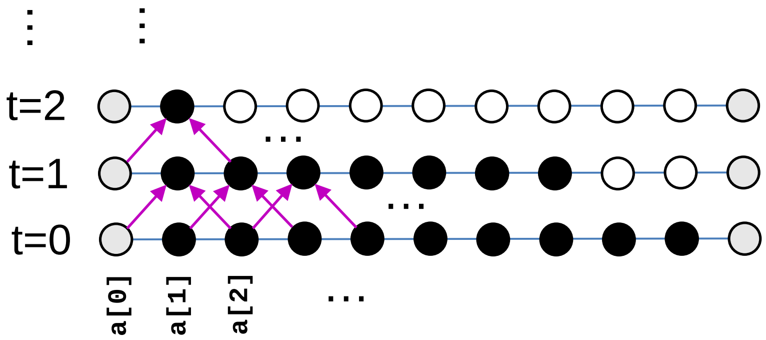
- **`#pragma omp task depend(`*type*`:`*list*`)`**
- The clause defines the currently generated task as dependent on a previously generated sibling task if at least one of the items in the list has the same storage location on both tasks

Dep. type	Creates dep. on types
IN	OUT, INOUT
OUT	IN, OUT, INOUT
INOUT	IN, OUT, INOUT

Task dependencies example

Finite-difference time stepping algorithm

```
double a[N];
#pragma omp parallel
{
  #pragma omp single
  {
    for(int t=0; t<100; ++t) {
      for(int i=1; i<N-1; ++i) {
        #pragma omp task \
          depend(in:a[i+1],a[i-1]) \
          depend(out:a[i])
          a[i] = func(a[i+1],a[i-1]);
} } } }
```



Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

OpenMP performance issues and remedies

High Performance Computing

Conditional parallelism

- Sometimes we want a flexible means to avoid parallelism
  - Barrier cost, cost of waking up the team of threads, scheduling cost
- **if** clause takes any valid condition in the base language
  - Can be applied to various constructs, including **task**

```
#pragma omp parallel if(n>8000)
{
#pragma omp for
  for(int i=0; i<n; ++i)
    a[i] = b[i] + c[i] * d[i];
}
```

Example: suppress nested parallelism in a library routine

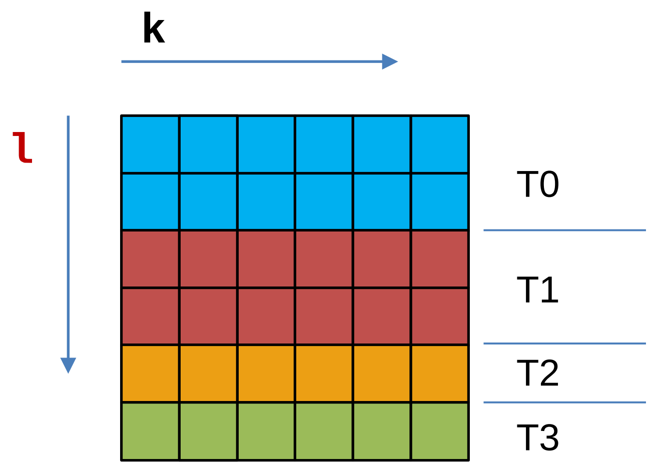
```
#pragma omp parallel \
  if(!omp_in_parallel())
{
  ... ! parallel region
}
```

- Less extreme option: **num_threads(n)** clause to reduce # of threads in region

Coarse granularity

- Even if there is enough work in a parallel loop, granularity may cause imbalance
- Example: load imbalance if M is “small,” i.e., comparable to number of threads

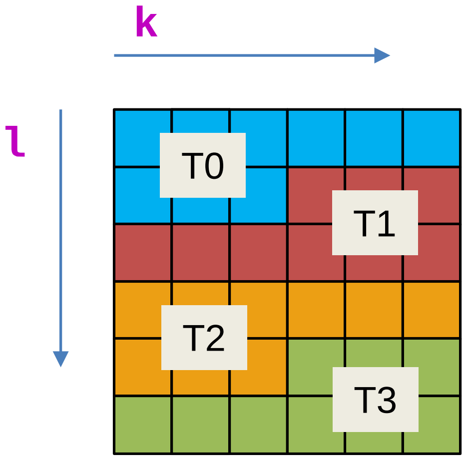
```
double a[M][N];
#pragma omp parallel for \
  schedule(static) reduction(+:res)
  for(int l=0; l<M; ++l)
    for(int k=0; k<N; ++k)
      res += a[l][k];
```



Coarse granularity

- **collapse(n)** clause coalesces perfect n-way loop nest

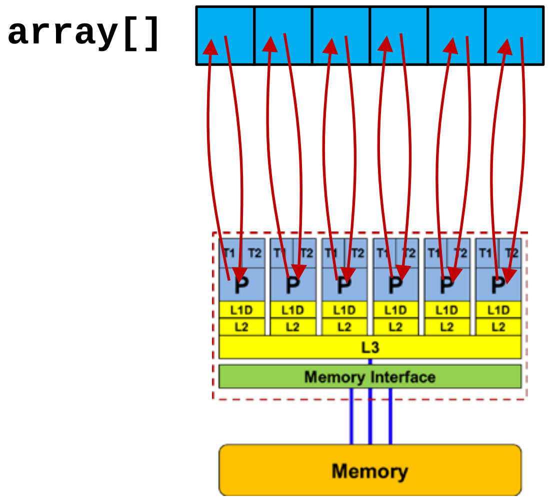
```
double a[M][N];
#pragma omp parallel for \
  schedule(static) reduction(+:res) \
  collapse(2)
  for(int l=0; l<M; ++l)
    for(int k=0; k<N; ++k)
      res += a[l][k];
```



False sharing

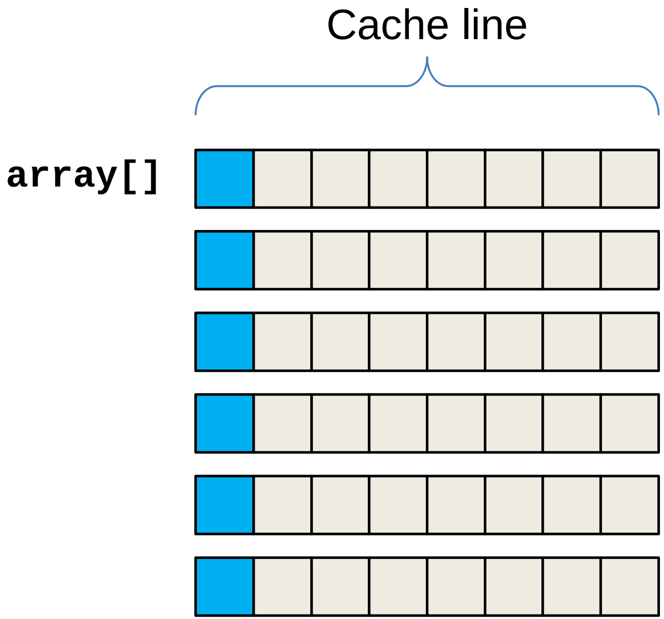
- If multiple threads frequently access the same cache line and at least one thread writes to it → false sharing

```
int a=0;
int array[omp_get_max_threads()];
#pragma omp parallel
{
  int id = omp_get_thread_num();
  array[id]=0;
  #pragma omp for
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      array[id] += x;
    }
  #pragma omp critical
    a += array[id];
}
```



False sharing: two solutions

1. Padding: leave ≥ 1 cache line of room between adjacent entries




```
int a=0;
const int CL=8;
int array[omp_get_max_threads()*CL];
#pragma omp parallel
{
  int id = omp_get_thread_num();
  array[id*CL]=0;
  #pragma omp for
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      array[id*CL] += x;
    }
  #pragma omp critical
    a += array[id*CL];
}
```

False sharing: two solutions

2. Privatization and reduction

```
int a=0;
#pragma omp parallel
{
  #pragma omp for reduction(+:a)
    for(int i=0; i<N; ++i) {
      int x = compute(i);
      a += x;
    }
}
```

If possible, prefer privatization over synchronization!

Wrap-up: advanced OpenMP and performance

- Locks
  - Fine(r)-grained synchronization, many locks possible
- SIMD
  - Loops (`simd`), parallel loops (`for simd`), functions (`declare simd`)
- Tasking
  - More flexible work distribution, parallelism beyond loops with `task`
  - `taskloop` for turning loop into a bag of tasks
- Performance issues
  - Overhead → `if`, `num_threads`
  - Granularity → `collapse`
  - False sharing → padding, privatization, reduction