HPC Programming

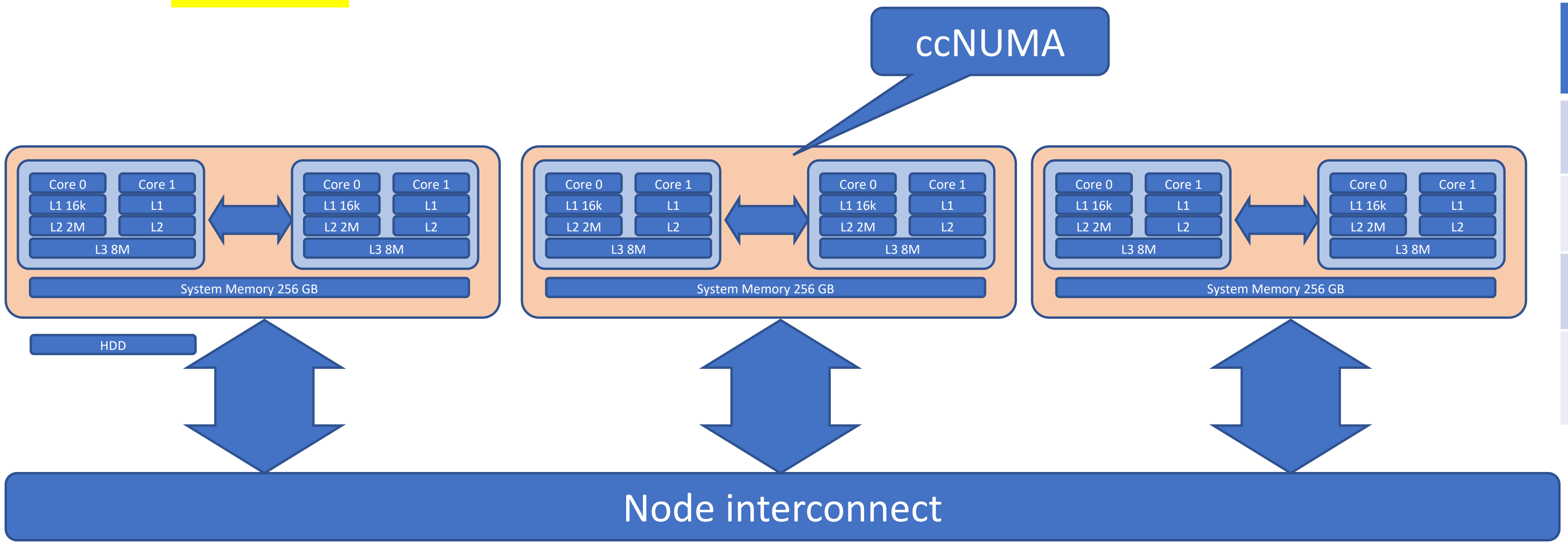
OpenMP, Part II

Peter-Bernd Otte, 29.10.2019

Recap

Anatomy of a cluster computer

- Latencies:



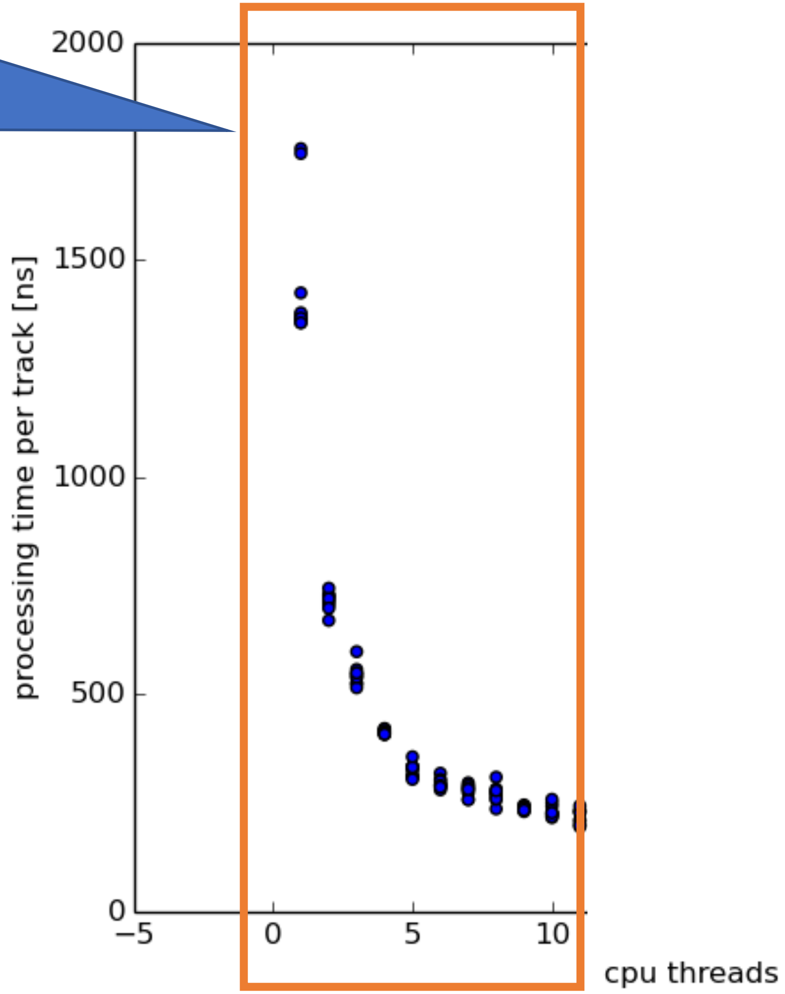
Operation	min overhead in cycles
Hit L1 cache	1-10
Miss all caches	100
Page miss	100.000
(Data via interconnect)	1000 (1µs)

(all numbers are platform dependent)

Motivation: Speed up

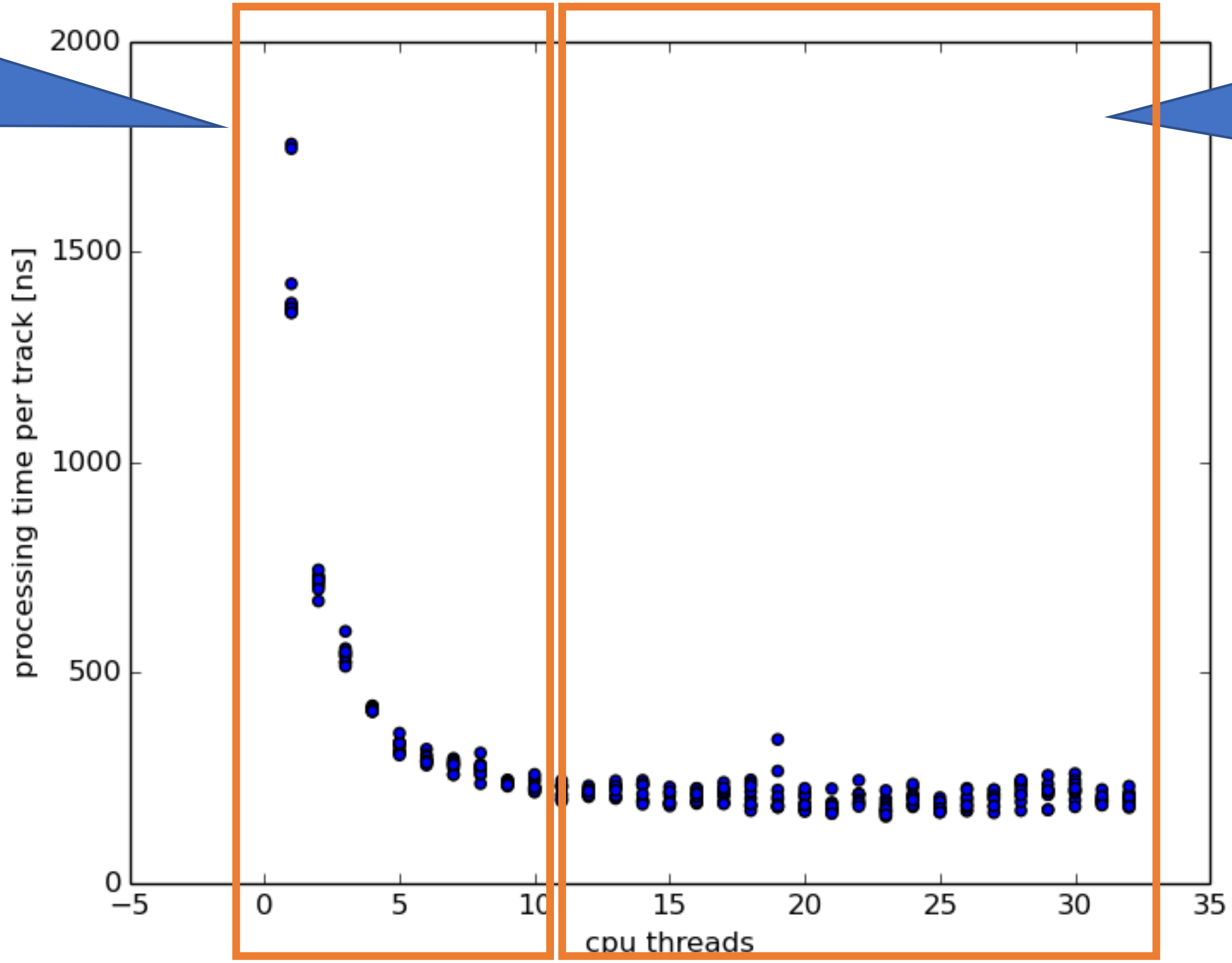
CPU bound
T = 1/N(cores)

- Ideal:
  some metric
  scales with
  1/N(cores)



Graph with curtesey to Stephan Maldaner

Comparision: CPU or RAM bound



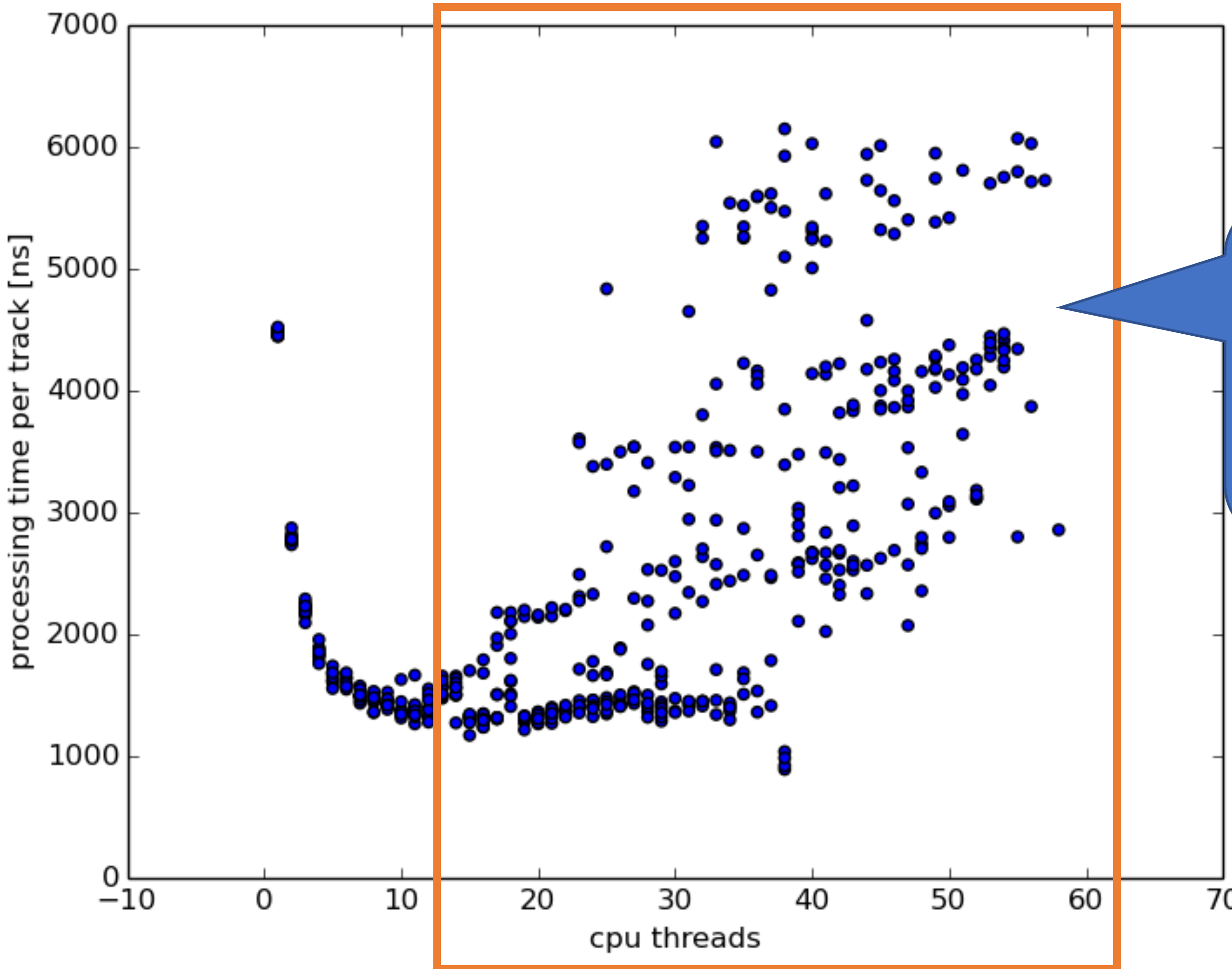
“CPU bound”
T = 1/N(cores)

“RAM bound”
RAM bandwidth and/or latency limit reached

Graph with curtesey to Stephan Maldaner

Comparision: CPU or RAM bound

- Same algorithm, but problem with thread non local memory



solution: fix threads to cores
→ see chapter: “common pitfalls”

Graph with curtesey to Stephan Maldaner

Comparison OpenMP / MPI

OpenMP

- shared memory directives (compile time)
    - to define work decomposition
    - no data decomposition (data in shared memory)
- synchronisation is implicit

Possible speedup:

- memory limited: Total bandwidth / single core bandwidth = 4 (hardware dependent)
- CPU limited: Number cores (+ possible cache effects)
- storage limited: do not use

MPI (Message Passing Interface, later this course)

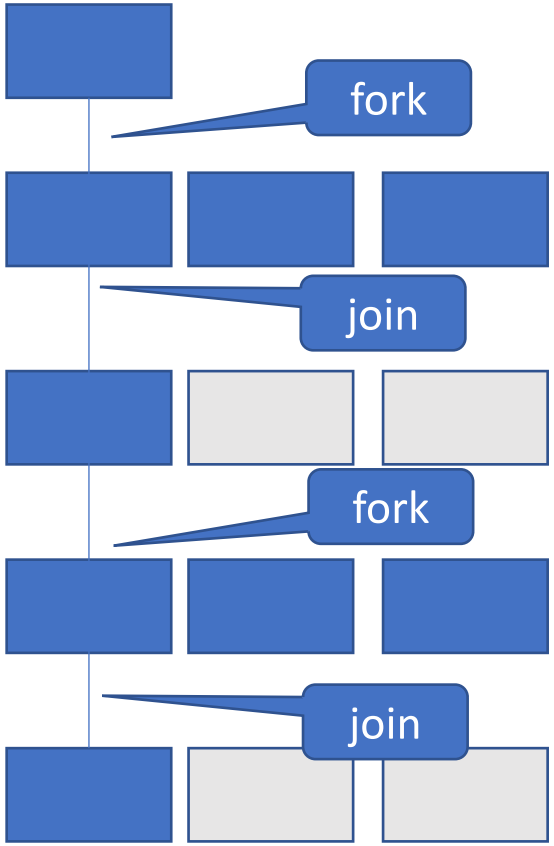
- software library (run time)
- user defines:
    - distribution of work & data
    - communication (when and how)

Possible speedup:

- Per node limits: see OpenMP
- RAM/CPU limited: utilisation of N nodes
- Storage limited: ? (use node local scratch)

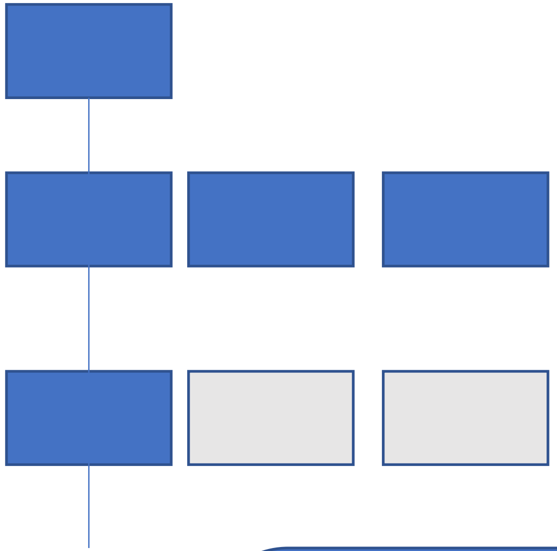
OpenMP: Execution Model (II)

- Begin execution as a single process (master thread)
- Fork-join of parallel execution
  1. Start of 1st parallel construct: Master thread creates N threads
  2. Completion of a parallel construct: threads synchronise (implicit barrier)
  3. Master thread continues execution
- At next parallel construct: work balancing with existing threads



OpenMP: Parallel Region Construct + Syntax

```
#pragma omp parallel [clause [, clause]]
      block
// emp end parallel
```



- *block* = to be executed by multiple threads in parallel. Each code executes the same code.
- Clause can be ("data scope"):
  - `private` (list) ← variables in list private to each thread & not initialised, standard for loop variables
  - shared (list) ← variables in list are shared among all thread, standard
  - firstprivate, lastprivate, threadprivate, copyin, reduction
  - set number of threads: num_threads(N)

Good practice:

- always declare all variable either in private or shared to avoid surprises (race conditions)
- or: default(none)
- Declare private var's inside parallel regions

Lessons learned from exercises:

- Python exercise 1 („Numba“):
  - installation of python packages in home directory („virtual environment“)
  - beneficial for quick results (JIT, parallelisation)

- C + OpenMP, exercise 1 (“Hello World”):
  - 1st OpenMP program, but no speedup
  - With OpenMP -> “no free lunch”

- C + OpenMP, exercise 2 (“Parallel Region”):
  - Multiple threads, output order undetermined
  - First race condition when having shared variables

Introduction OpenMP

1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives
5. Data environment and combined constructs
6. Common pitfalls and good practice (“need for speed”)

Control Structures - Overview

- Parallel region construct
  - `parallel`

- Worksharing constructs
  - `for`
  - `sections`
  - `task`
  - `single`
  - `master`

- Synchronisations constructs
  - `critical`

Comments:

- Defines **work load** among threads
- worksharing & sync constructs do not launch new threads
  - parallel construct creates a team of threads which execute in parallel
- worksharing comes with implicit barrier (threads wait until complete work finished):
  - none on entry
  - normally one at the end

OpenMP: for Directive (1)

- Parallelises the following for loop
  - in canonical form → see next slide.
  - loop iterations: all independent!

Allows the iteration count (of all associated loops) to be computed before the (outermost) loop is executed.

- Within parallel region
- `#pragma omp for [clause …] new-line`

```
#pragma omp for [clause …] new-line
     for-loop(s)
//end of for loop
```

- OR: Combined parallel worksharing constructs: “parallel for”

```
#pragma omp parallel for new-line
     for-loop(s)
//end of for loop + end of parallel region
```

OpenMP: for Directive (2)

- Canonical loop form (see 2.6 in “OpenMP Application Programming Interface”, Nov 2015)
    - Credo: number of iterations computable at start of loop
- for (initialize; test; increment) { ... }
    - initialize, test and increment: loop invariant expression
    - Initialize: var = lb, e.g. “int i = 0”
        - var = loop variable
    - Test: var operator b
        - operator is one of the following: <, <=, >, >=
    - Increment: (integer expression) e.g. i++, ++i, i=i+5, …
    - var
        - must not be modified in the loop body
        - integer (signed or unsigned)
- Examples:
    - wrong: for (int i = 0; i != n; i++)
    - canonical: for (int i = 0; i < n; i++)

OpenMP: for directive (3)

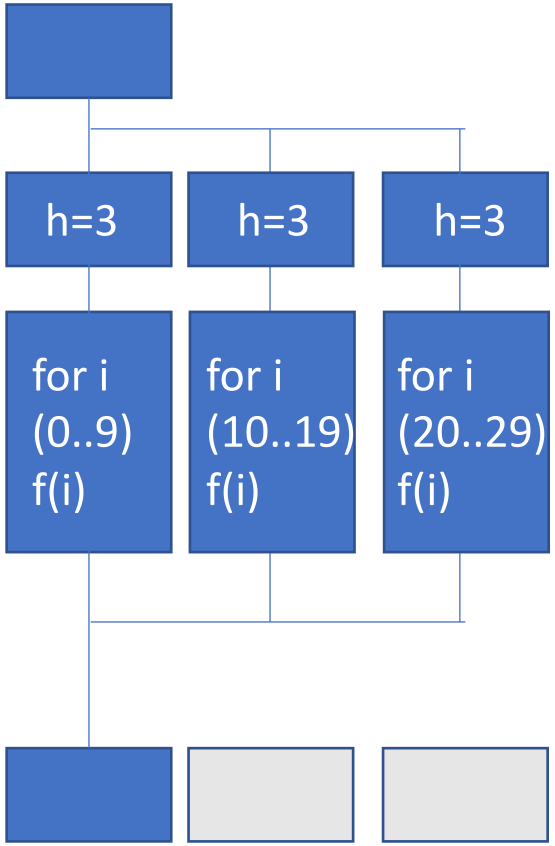
```
#pragma omp for [clause …] new-line
       for-loop(s)
//end of for loop
```

- Clause:
  - private (list)
  - reduction (op: list)
  - collapse (n)
    (n=const.: iterations of following n nested loops
    are collapsed into one larger iteration space)
  - schedule (type, chunk)
    (how the work is divided among the threads)
  - nowait
  - … (see API section 2.7.1)
- At the end of each for (unless nowait specified): implicit barrier
  (barrier? see next slide)

Will be discussed and used in next lecture

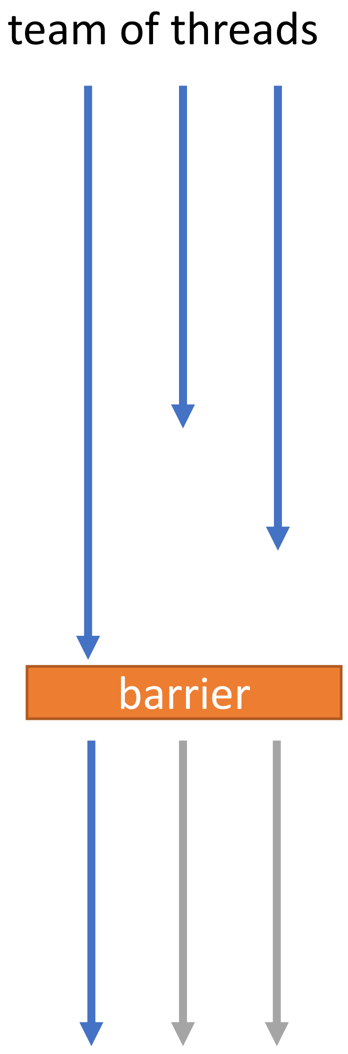
OpenMP: for Directive (4)

-
  ```
  double res[30];
  #pragma omp parallel private(i) shared(h)
  {
          h=3;

          #pragma omp for
          for (int i=0; i< 30; i++) {
                  res[i] = f(i);
          }


  } // OMP End parallel
  ```



OpenMP: Barrier

- barrier = all threads in a team wait until all threads reached barrier



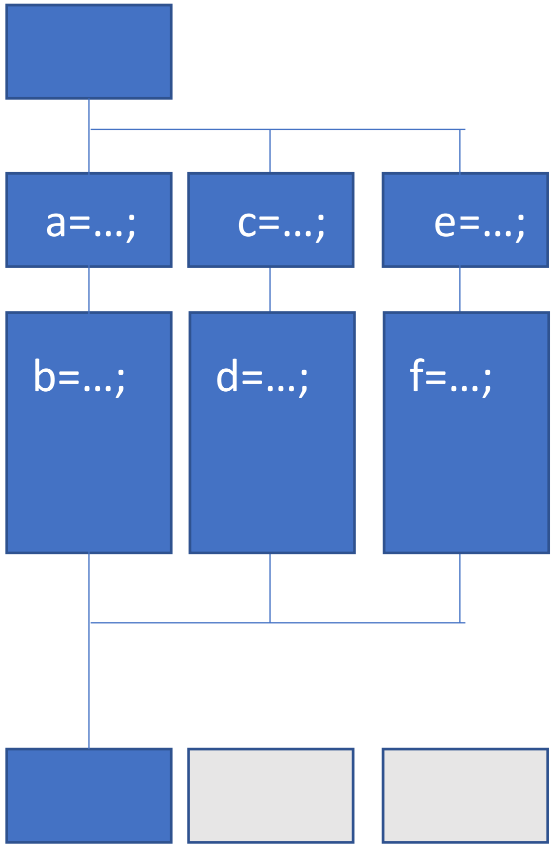
- Implicit barrier
    - entry and exit of parallel constructs
    - exit of all other control constructs (except: nowait clause)
- Explicit barrier
    - critical directive
    - single directive

→ see later

OpenMP: sections directive

- each block: independent
- each block is executed only once by one thread.
- order of execution is implementation dependent

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { /*block 1*/ a=…; b=…; }
    #pragma omp section
    { /*block 2*/ c=…; d=…; }
    #pragma omp section
    { /*block 3*/ e=…; f=…; }
    ...
  }  // end of omp sections
} // end of omp parallel
```



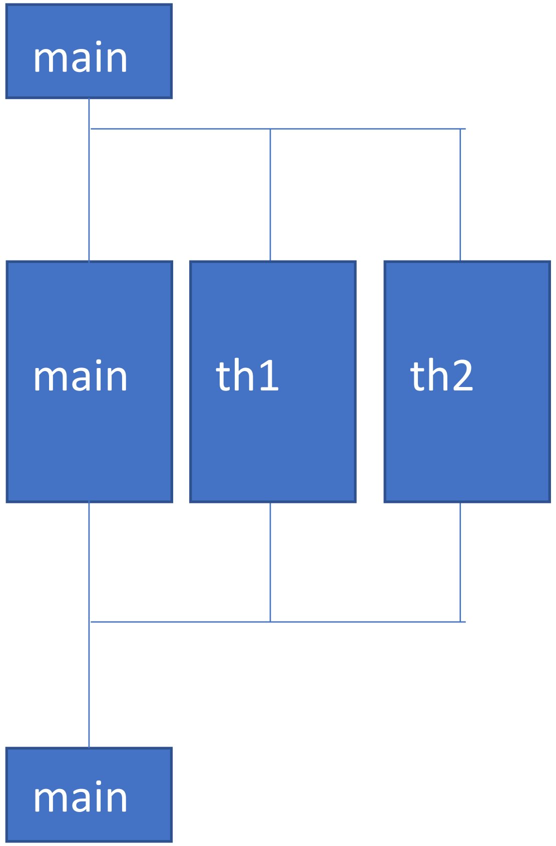
Comparison: sections directive ⇔ PThreads

- C++11 standard library: more flexibility, more things can go wrong.

```
#include <iostream>
#include <thread>

void function_1() {
  //Some work
}
void function_2() {
  //some work
}

int main() {
  std::thread thread_1(function_1); /*Thread constructor */
  std::thread thread_2(function_2);
  thread_1.join(); /*forces main thread to wait for thread1/2*/
  thread_2.join(); /*otherwise undefined behaviour */
  return 0;
}

g++ pthreadtest.cpp -pthread -std=c++11 -o pthreadtest
```

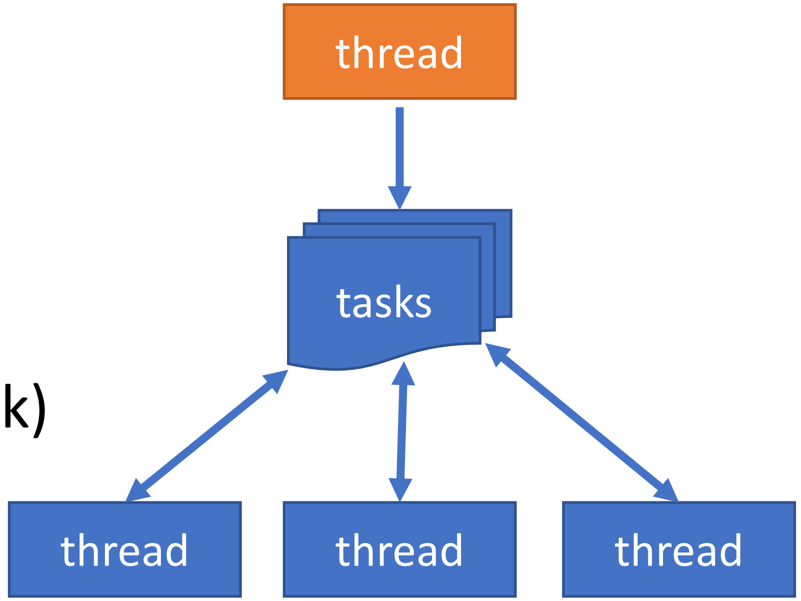
Pitfalls! Check for

- private and shared variables
- cache coherence effects



OpenMP: task directive concept

- parallelises several tasks
  - eg. traverse a linked list with a recursive algorithm, Fibonacci numbers
  - length not known at beginning (parallel for not possible)
- concept:
  1. thread generate tasks
  2. team of thread executes tasks
- Note:
  - tasks can be nested (task may generate a task)
  - all tasks can be executed independently
  - overhead(for) < overhead(tasks)



OpenMP: task directive syntax & example

- Defines a task within parallel region:

```
#pragma omp task [clauses] new-line
  block
```

- clauses:
    - untied
    - default (shared | none | private | firstprivate)
    - private (list)
    - firstprivate (list)
    - shared (list)
    - if (scalar expression)
- Optional: taskwait
  Specifies a wait on completion of all direct child tasks generated since beginning of current task (not to "descendants")

```
#pragma omp taskwait new-line
```

- OpenMP 4: Specifies to wait on completion of child tasks <u>and</u> their descendant tasks:
  #pragma omp taskgroup

Example:

```
#pragma omp parallel num_threads(2)
{
  #pragma omp single
  {
    printf(“E = “);
    #pragma omp task
      printf(” m ");
    #pragma omp task
      printf(” c^2 ");
    #pragma omp taskwait
    printf(“ Wow “);
  }
} // end of parallel region
```

only one thread packages tasks

Tasks are executed at task execution point, add: #pragma omp taskwait

Output (without and with taskwait):

1. “E = Wow m c^2” or ”E = WOW c^2 m”
2. ”E = m c^2 Wow” or ”E = c^2 m Wow”

OpenMP: master Directive

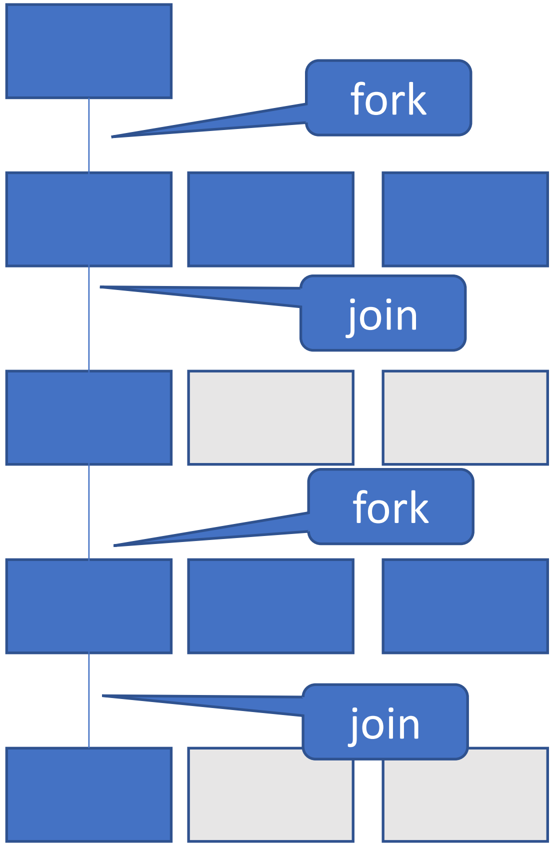
- master
  - section of code executed only by the master thread
  - no implicit barrier upon completion or entry
- Syntax:

```
#pragma omp master newline
  block
```

- `benefit? → next slide`

OpenMP: Single Directive

- single
  - section of code executed by single thread
  - not necessarily the master thread
  - implicit barrier upon completion
- Syntax:
  ```
  #pragma omp single [clauses] newline
    block
  ```
- Good practice:
  Reduce the fork-join overhead by combining
  - several parallel parts (for, task, sections)
  - sequential parts (single, master)
    in one parallel region (parallel)



Critical directive

Difference to single directive?

- Explicit barrier
- Enclosed code
  - executed by all threads
  - restricted to only one thread at a time
- Syntax:
  `#pragma omp critical [(name)] new-line`
  `  block`
- A thread waits at the entry of critical region until no other thread in the team is executing a region with the same name
  - If (name) is omitted: All regions belong to the same undefined region name.

- Example: count 0’s in matrix:

```
int matrix[rows][cols];
bool number_of_zeros = false;
#pragma omp parallel default(none)
shared(matrix, number_of_zeros)
{
  #pragma omp for
  for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
      if (matrix[row, col] == 0) {
        #pragma omp critical
        { number_of_zeros++; }
      }
    }
  }
}
printf(“The matrix has %d 0’s.”,
number_of_zeros);
```

OpenMP: single ⬄ critical

- single:
  - section executed by single thread
  - only once
- critical:
  - section executed by one thread at a time
  - num_threads() times

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
  #pragma omp single
  a++;
  #pragma omp critical
  b++;
}
printf("single: %d critical: %d", a, b);
```

```
result:
single: 1 critical: 4
```

OpenMP: cancel and cancelation point - directive

- Example: check matrix for 0 entry:

```
bool has_zero = false;
#pragma omp parallel default(none) shared(matrix, has_zero)
{
  #pragma omp for
  for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
      if (matrix[row, col] == 0) {
        #pragma omp critical
        { has_zero = true; }
        #pragma omp cancel for
      }
    }
    #pragma omp cancellation point for
  }
}
```

Set up your workbench

- Read the latest hints online:
  https://gitlab.rlp.net/pbotte/learnhpc/tree/master/openMP

Basic concept:

- Connect 2 times to Mogon2 / HIMster2 via SSH
  1) connection for your editor (gedit, vi, vim, nano, geany, …)
  2) second connection for compiling and running on compute node:
  srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs bash
  - (no analysis on the head node!)
  - Run with: OMP_NUM_THREADS=4 ./pi
- Download the files:
  1) first time: git clone https://gitlab.rlp.net/pbotte/learnhpc.git
  2) only update: git pull
  - Check for directory: openMP/exercise3/

Hints:

- "git pull" does not work? To reset your git repository to the master: "git reset –hard"
- Check compiler version: cc -v
- Run: OMP_NUM_THREADS=4 ./pi
  or export OMP_NUM_THREADS 4
- Possible to check reservation with: squeue -u $USER

Exercise 3: worksharing directives

Learning objectives:

- Use of “for”, “critical” and “single” directive

Steps:

1. Use the code from exercise 2 and compile as openmp program (-fopenmp with cc) and run with OMP_NUM_THREADS=4
2. Add (a) parallel region and (b) for directive and compile. Run with OMP_NUM_THREADS=1. Expected pi value: correct.
3. Run with OMP_NUM_THREADS=2. Expected pi value: wrong. Repeat also with different OMP_NUM_THREADS values. Why is it unpredictable? (Where is the race condition?)
4. Add private(x) clause, compile and run with OMP_NUM_THREADS=2 again. Repeat also with different OMP_NUM_THREADS values. Expected pi value: still unpredictable. Why?
5. Add critical directive around sum statement, compile and run. Test different OMP_NUM_THREADS several times in a row,
    1. how is the speedup with increasing OMP_NUM_THREADS? (why do e.g. 4 threads take longer than 2?)
    2. compare results. Are the results the same to the last digit? Why not?
6. Optimize: Move critical region outside loop. Run several times with different OMP_NUM_THREADS. How does
    1. speedup
    2. and precision evolve?
7. Modify exercise 1: Use a single construct to let only one thread print out the number of threads in the team.

Optional Exercise 4: Fibonacci Numbers

Write a parallel program that calculates a Fibonacci Number in a recursive implementation: F(n) = F(n-1) + F(n-2)

Comments:

- binary tree of tasks
  - F(n) = F(n-1) + F(n-2)
  - inefficient $O(n^2)$ recursive implementation (but excellent example)
- traversed using a recursive function
- taskwait: A task cannot complete until all tasks below in the tree are complete
- local variables: x, y → private to current task
  - declare as shared on child tasks to prevent firstprivate copies

```
int fibo (int n) {
  int x,y;
  if (n < 2) return n;
  #pragma omp task shared(x) if(n>30)
  x = fibo(n-1);
  #pragma omp task shared(y) if(n>30)
  y = fibo(n-2);
  #pragma omp taskwait
  return x+y;
}

int main() {
  int NN=30;
  #pragma omp parallel
  {
    #pragma omp master
    fibo(NN);
  }
}
```

Stop creating tasks at some level in the tree

OpenMP: References

- OpenMP Application Programming Interface, Examples
  Version 4.5.0 – November 2016
  https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf