703650 VO Parallel Systems WS2020/2021
OpenMP Advanced

Philipp Gschwandtner

Overview

- task-based parallelism
- modern OpenMP features
  - affinity
  - vectorization
  - accelerators
- common OpenMP pitfalls

Motivation

- OpenMP offers constructs for parallelism and work sharing
  - `parallel`
  - `for`, `section`, `critical`, ...
- but they lack flexibility, e.g. because they
  - do not support nested work sharing (e.g. traversing branches of a tree in parallel) without explicit increase in parallelism
  - restrict data structures (e.g. try to process all elements of a linked list with a `for`...)

Motivation

- example scenario on the right
  - `for` directive will distribute work across all threads, including the one loading data from storage ➔ leads to load imbalance

- How to improve this? Bad choices include:
  - `nowait` removes implicit barrier, but does not change load distribution
  - dynamic or guided loop scheduling reduces amount of work for the thread loading data from storage, but incurs overhead and doesn't fully solve the issue
  - `for` should be inside another `section`, but work share directives cannot be nested
  - nest multiple `parallel` directives and carefully control degree of parallelism

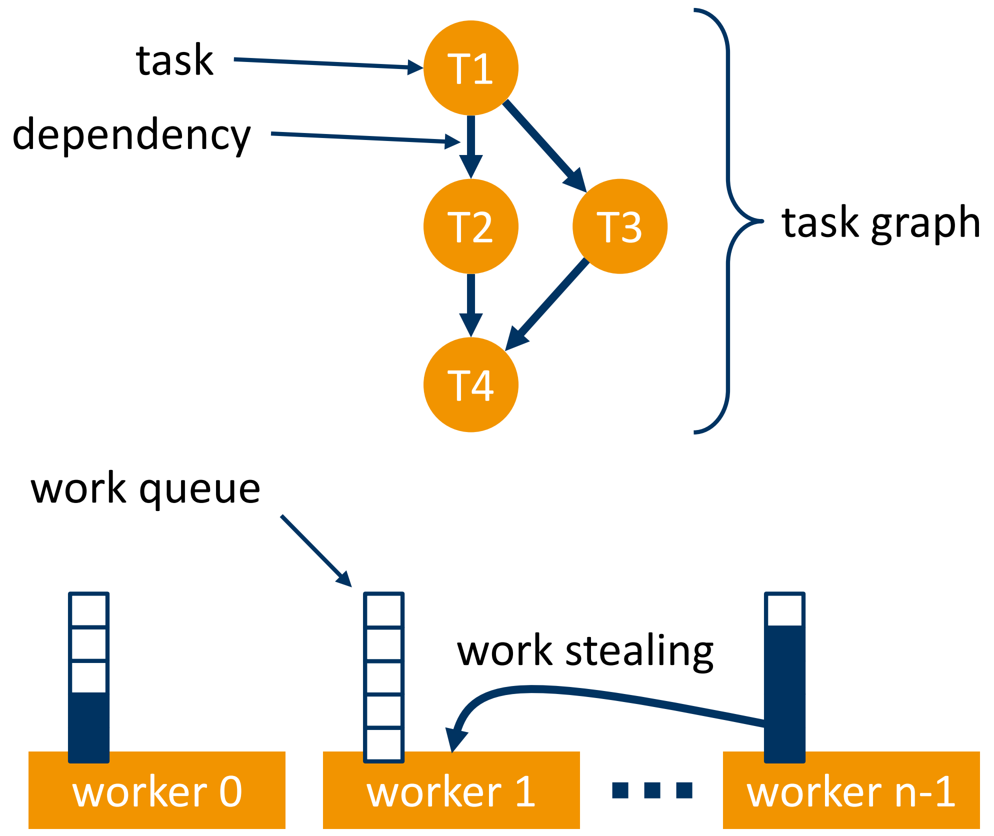
```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {/*load part of data from storage*/}
  }
  #pragma omp for
  { /*compute part of data on-the-fly*/ }
  // use both parts for processing
}
```

Task-based Parallelism

- offered by many programming models, including OpenMP
  - also used under the hood for `for`, `sections`, …
- different from data parallelism
  - focuses on decomposing work (*“tasks”*) rather than data
  - allows to evaluate dependencies between tasks and run parts of work in parallel
- offers many advantages, e.g.
  - supports easy nesting of parallel work
  - enables efficient load balancing strategies

Task-based Parallelism cont’d

- decompose work into tasks and put in *work queues*
  - worker threads can take tasks from queue and execute them
  - if a worker runs empty, steal tasks from another worker
  - if queue is full, execute task immediately without further task generation



`task` Directive

- allows explicit specification of tasks
  - careful, `firstprivate` is the default
- whenever a thread encounters a `task` directive, a task is generated
  - task may be immediately executed
  - or execution may be deferred
- wait for completion using `taskwait`
  - waits for child tasks spawned by the current task

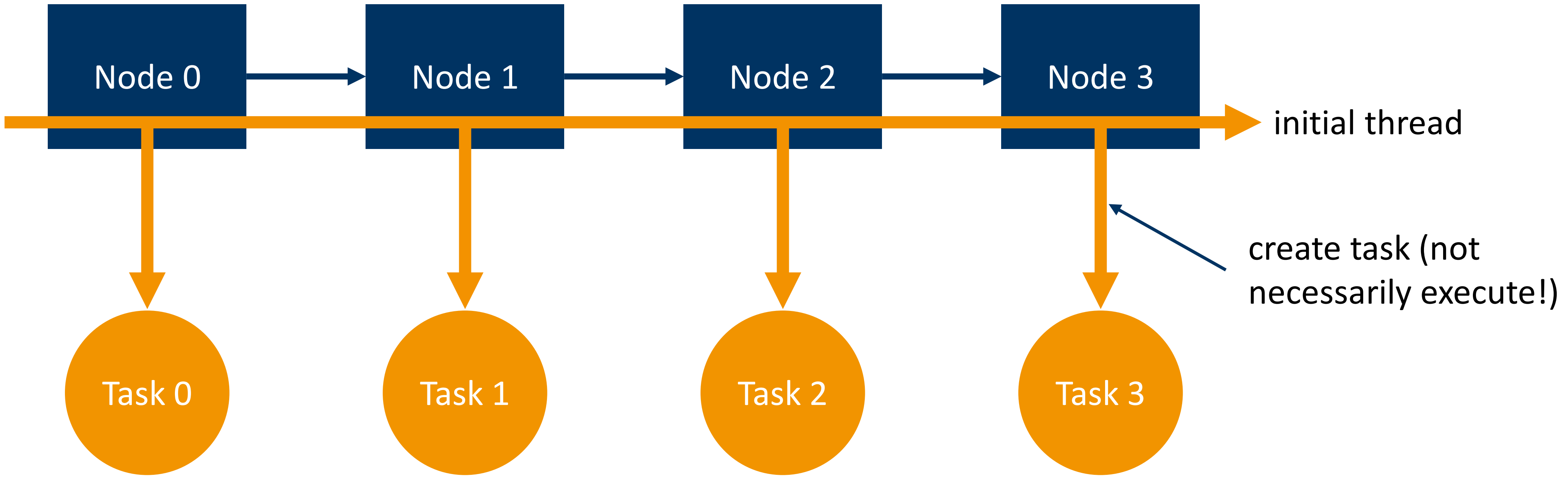
```
int fib(int n) {
  int i, j;
  if (n < 2)
    return n;

  #pragma omp task shared(i)
  i = fib(n-1);

  #pragma omp task shared(j)
  j = fib(n-2);

  #pragma omp taskwait
  return i + j;
}
```

Example: Traversing a Linked List

```c
struct Node {
   struct Node *next;
   struct Data *data;
};

void traverse(struct Node *p) {
  if (p->next) {
    #pragma omp task
    traverse(p->next);
  }
  process(p); // do work
}
```

```c
int main(int argc, char **argv) {
  struct Node *head;
  head = ... // produce list
  #pragma omp parallel
  {
    #pragma omp single
    {
      traverse(head);
    }
  }
}
```

Example: Traversing a Linked List cont’d



Task Clauses

- `final`
  - if expression evaluates to true: no more task generation but plain function calls
  - useful for specifying *“cut-off”* and preventing excessive oversubscription
  - e.g. `final(n < THRESHOLD)` for Fibonacci
- `untied`
  - if an untied task is suspended it can be resumed by any thread in the team (otherwise only by the same thread)
  - useful if tasks hold no per-thread state information (e.g. allocated resources, MPI, …)
- `depend`
  - allows to specify data dependencies and establish a partial order on tasks
  - useful for establishing task graphs

Task Dependencies

- OpenMP allows to establish partial task order using depends clause
  - allows to establish Read-after-Write, Write-after-Read, Write-after-Write, and Read-after-Read relationships for task pairs
  - only applies to already generated sibling tasks ➔ cannot re-order tasks
  - rather tells compiler how data is accessed to prevent race conditions and enable optimizations

```
void foo() {
  #pragma omp task depend(out:x)
  x = f();

  #pragma omp task depend(in:x)
  y = g(x);


  ...
}
```

Task Scheduling Points

- tasks can be suspended or resumed at task scheduling points
  - in the current task, immediately after generating a new task
  - at the end of a task region
  - in implicit and explicit barriers
  - at `taskwait`
  - when using `untied` clause: everywhere inside the untied task

- once commenced execution, a task will remain with the same thread unless it is an `untied` task

`taskwait` vs. `taskgroup`

- `taskwait` will wait for all children directly spawned by the current task
- `taskgroup` will wait at its end for all children spawned by the current task and their descendants

```
#pragma omp taskwait
// wait for all directly
// spawned children


#pragma omp taskgroup
{
// wait for all descendants
// at the end of this region
}
```

`taskloop` Directive

- allows to parallelize `for` loops using task parallelism
  - not a work sharing construct, should only be executed by one thread (like all other task constructs!)
  - loop must have canonical form (like with `for` directive)
  - can take all clauses that `task` or `for` can, except `schedule`, `linear`, `ordered`, `nowait`
  - `reduction` available with OpenMP 5.0

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskloop
    for(i=0; i<30; i++) {
      a[i] = b[i] + f * (i+1);
    }
  }
}
```

`taskloop` Clauses

- `grainsize`
  - limits task size $(\text{grainsize} \le \text{iterationsPerTask} \le 2 \times \text{grainsize})$
- `num_tasks`
  - specify number of tasks to create
- `nogroup`
  - optionally disable task group creation

Solving Motivation Example with `taskloop`

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {/*load part of data from storage*/}
  }
  #pragma omp for
  { /*compute part of data on-the-fly*/ }
  // use both parts for processing
}
```

```
#pragma omp parallel
{
  #pragma omp single
  #pragma omp taskgroup
  {
    #pragma omp task
    { /*load from storage*/ }
    #pragma omp taskloop nogroup
    for (i=0; i<N; i++) { /*compute*/ }
  }
  // use both parts for processing
}
```

Affinity

OpenMP and Affinity

- knowing and controlling affinity is important
  - communication latencies, locality in data and instruction caches, etc.

- little guaranteed support before OpenMP 4.0
  - implementation dependent solutions, mostly environment variables, e.g.
    - GNU runtime: GOMP_CPU_AFFINITY=0-7
    - Intel runtime: KMP_CPU_AFFINITY=0-7
  - mostly enumerations of (possibly strided) core ranges, e.g.
    - `0-7`
    - `0,1,18,19`
    - `8-15:2`
  - no explicit support for nested parallelism

OpenMP and Affinity cont’d

- OpenMP defines *places* consisting of one or more *processors*
  - pinning can be done on basis of places, threads are free to be migrated within a place

- controlled by env var OMP_PLACES
  - either using abstract names such as `threads`, `cores`, `sockets`
  - or using explicit but OS-dependent enumeration
    (`start : [number of entries] : [stride]`)
  - e.g. 4 places with 4 cores each
    - `{0,1,2,3}, {4,5,6,7}, {8,9,10,11}, {12,13,14,15}`
    - `{0:4}, {4:4}, {8:4}, {12:4}`
    - `{0:4}:4:4`

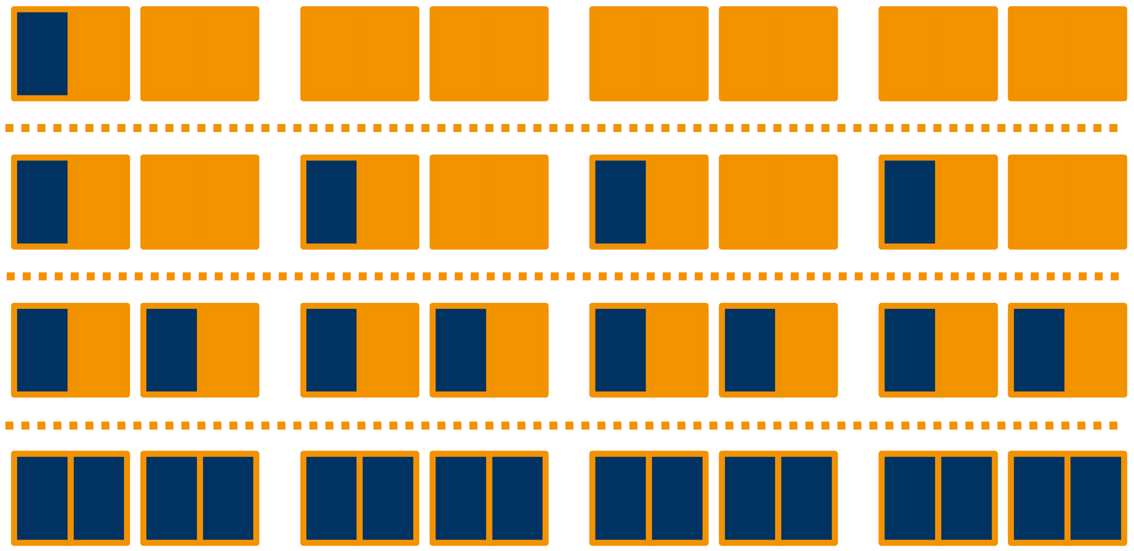
OpenMP and Affinity cont'd

- binding is controlled with OMP_PROC_BIND
  - `true:` don't migrate threads at all
  - `false:` free to migrate
  - `spread:` distribute across places first
  - `close:` fill places first
  - `master:` as close to the master thread as possible

- nested degree of parallelism controlled with OMP_NUM_THREADS
  - e.g. `4,2,2`
  - first `#pragma omp parallel` spawns 4 threads, the next nested ones 2 and 2

OpenMP Nested Affinity Example

```
// OMP_PLACES=threads
// OMP_NUM_THREADS=4,2,2
// OMP_PROC_BIND=spread,spread,close

#pragma omp parallel
{
  #pragma omp parallel
  {
    #pragma omp parallel
    {
      // work ...
}}}
```

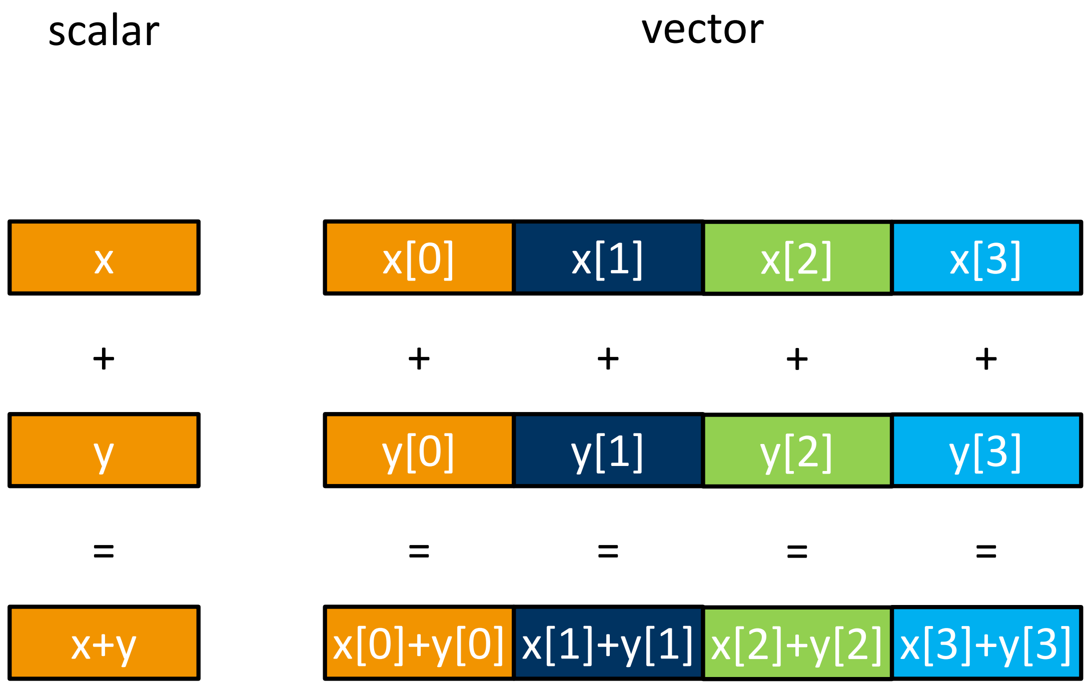


Modern Features of OpenMP ≥ 4.0

Vectorization

- modern CPUs have vector units
  - allow multiple operands per operation
  - performance gains of up to e.g. 4x without any thread- or process-based parallelism
  - Intel/AMD MMX/SSE/AVX, ARM NEON, IBM AltiVec, …

- available operations and number of operands (“vector width”) depend on your software/hardware stack
  - hard to code manually (compiler intrinsics or assembler)



What Could go Wrong With Automatic Vectorization?

- compiler-based auto-vectorization (e.g. GCC’s `-ftree-vectorize`)
  - requires analysis and heuristics
  - has lots of points of failure
    - loop-carried dependencies
    - pointer aliasing
    - memory alignment
    - data type mixing
    - overly conservative heuristics
    - numerical stability issues

```
void foo(double* a, double* b) {
  for(int i=0; i<32; ++i) {
    a[i] = a[i+1] + b[i];
  }
}
```

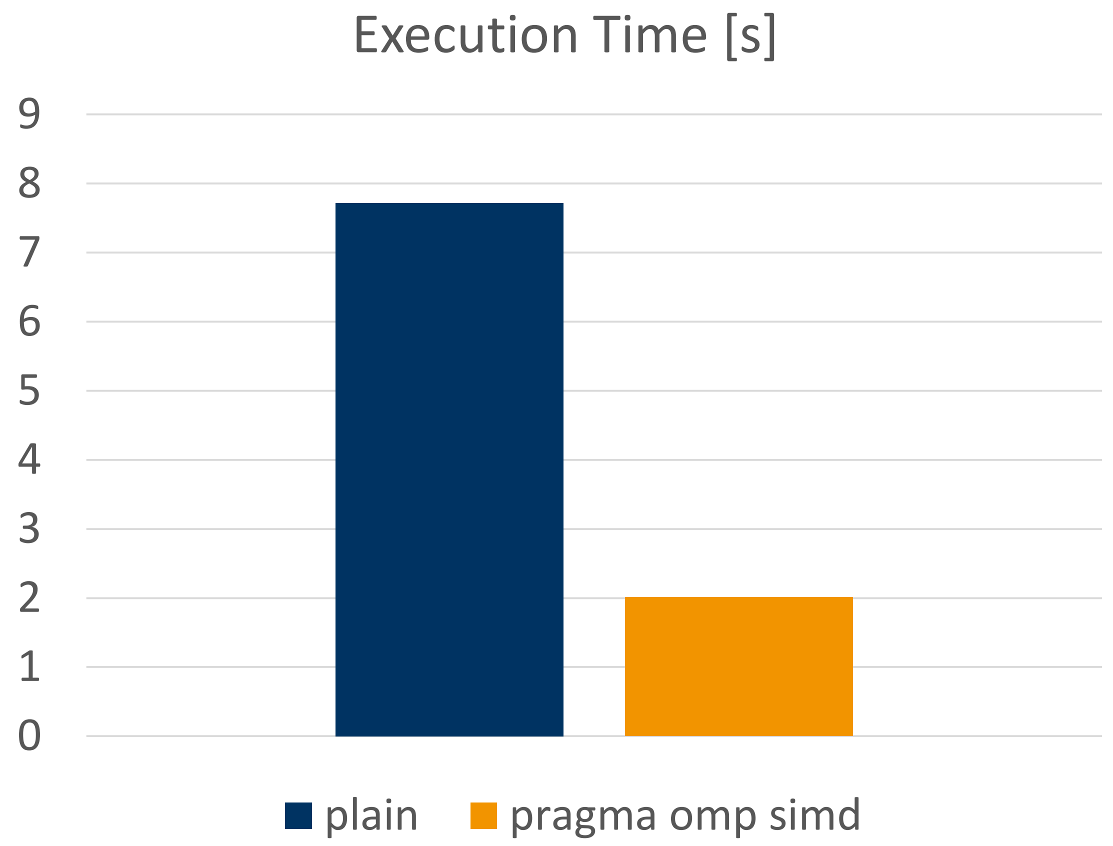
`simd` directive

- portable vectorization without compiler- or hardware-specific intrinsics
  - no need to know about GCC/LLVM/Intel/ARM…
- not the be-all end-all solution to vectorization but helps a lot
- can be combined with `for` directive
  - distributes vectorized loop iteration chunks among threads

```
// note: aligned_alloc requires -std=c11
int* a = aligned_alloc(32,
                       sizeof(int)*SIZE);
int* b = aligned_alloc(32,
                       sizeof(int)*SIZE);

// initialize a, b, and f...

#pragma omp simd aligned(a,b:32)
for(int i = 0; i < SIZE; ++i) {
  b[i] += a[i] * f;
}
```

Vectorization Performance Comparison

- LCC2, gcc/8.2.0, single-threaded,
  - `-march=native`
    `-mtune=native`
    `-O2`
- 10^8 vector sums on integer arrays of length 64
  - code example of previous slide
- execution time reduced by 3.84x
  - 4 integers per operation incl. some overhead



`simd` clauses

- `safelen`
  - maximum number of iterations with no dependencies to be vectorized
  - increases loop coverage that can be vectorized
- `aligned`
  - specify memory alignment in bytes after aligned allocation with e.g. `aligned_alloc()`
  - required for additional compiler optimizations

```
#pragma omp simd safelen(4)
for(int i=0; i<N; i++) {
  a[i] = a[i+4] * f;
}

#pragma omp simd aligned(a,b:32)
for(int i = 0; i < SIZE; ++i) {
  b[i] += a[i] * f;
}
```

Vectorization and Functions

- function calls are problematic
  - function definition could be too complex for auto-vectorization or hidden in another object file only visible during linking
  - solution: specify SIMD-capable functions with `declare simd`

```
#pragma omp declare simd
int max(int a, int b) {
  return a > b ? a : b;
}
```

`declare simd` Clauses

- `aligned`
  - specify memory alignment of arguments
- `inbranch/notinbranch`
  - specifies that function will always/never be called inside a conditional branch of an SIMD loop
  - required due to masking (conditionally loading arguments in vector registers)
- `simdlen`
  - specify preferred SIMD-length, i.e. the number of loop iterations per SIMD invocation

Accelerator Support in OpenMP

- programming accelerators is difficult
  - usually completely different hardware architecture and ISA (e.g. GPUs)
  - new/additional software stack
  - often requires copying data from/to device memory
  - often lack of debugging tools (I'm looking at you, OpenCL!)
- OpenMP tries to add abstraction layer to improve usability
  - c.f. SIMD support
- already present in 4.0 (2013!), some clarifications in 4.5, usability improvements in 5.0
  - but check compiler/runtime implementation support!

`target` directive

- creates a target task to be executed on device and maps variables to data environment on device
  - map clause: specify mapping of original data on host to data on device (`to`), vice versa (`from`), or both (`tofrom`)

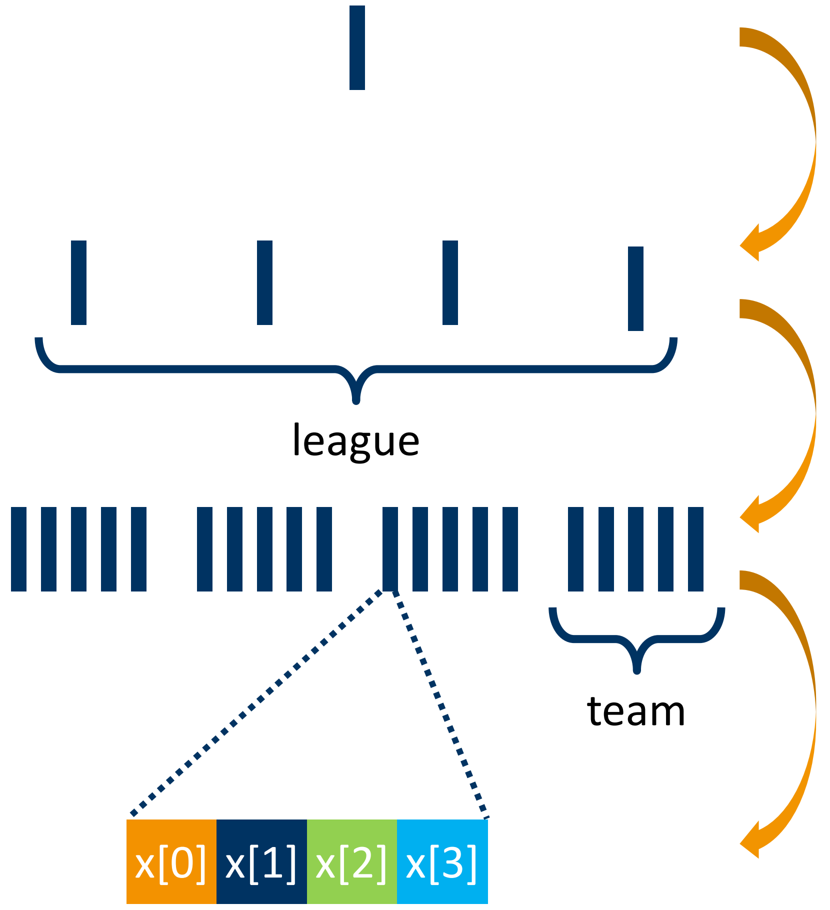
```
void add(float *a, float *b,
         float *c, int size)
{
  #pragma omp target \
    map(to:a[0:size],b[0:size],size) \
    map(from:c[0:size])
  {
    #pragma omp parallel for
    for (int i = 0; i < size; i++)
      c[i] = a[i] + b[i];

  }
}
```

Additional Accelerator Directives and Clauses

- `teams`
  - creates a collection of teams (a single team is always implicitly created if `teams` is not specified)
  - relevant for performance: barriers only done among threads of the same team
- `parallel for`
  - distribute loop iterations to threads of a team
- `distribute`
  - similar to work share constructs but distributes iterations to different teams
- additional, known clauses can be used, e.g. SIMD
  - leads to e.g. `#pragma omp teams distribute parallel for SIMD`

OpenMP `target` Thread Hierarchy

- `target` creates a single, initial thread on target
- `teams` creates a *league* of several teams, each with a single, initial thread
- `parallel` creates several threads within each team
- `SIMD` vectorizes code executed by each thread of each team in the league



Other Neat OpenMP 5.0 Features and Fixes

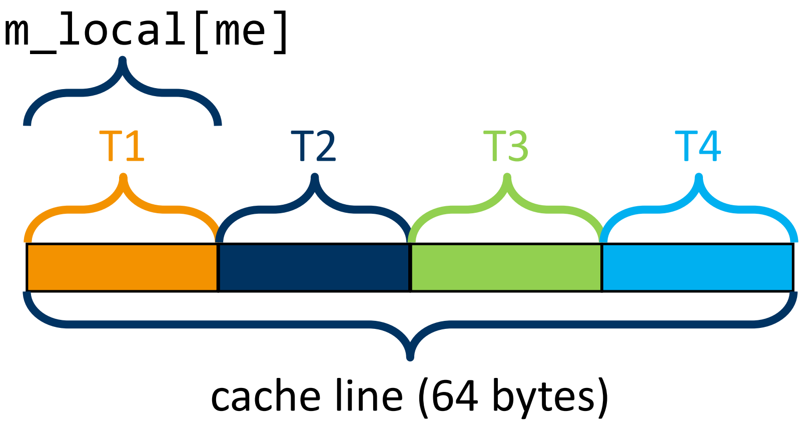
- reductions in `taskloop`
  - also reductions in `task_group`
- range-based `for` loops in C++ and `!=` as loop condition
- many new combined directives
- tool support
  - e.g. callback functions

```
#pragma omp taskloop reduction(+:sum)
  for(int i = 0; i < N; i++)
    sum += ..;

#pragma omp parallel for
for(const auto& x : vec)
  ...

#pragma omp parallel master taskloop
...
```

Common OpenMP Pitfalls

False Sharing

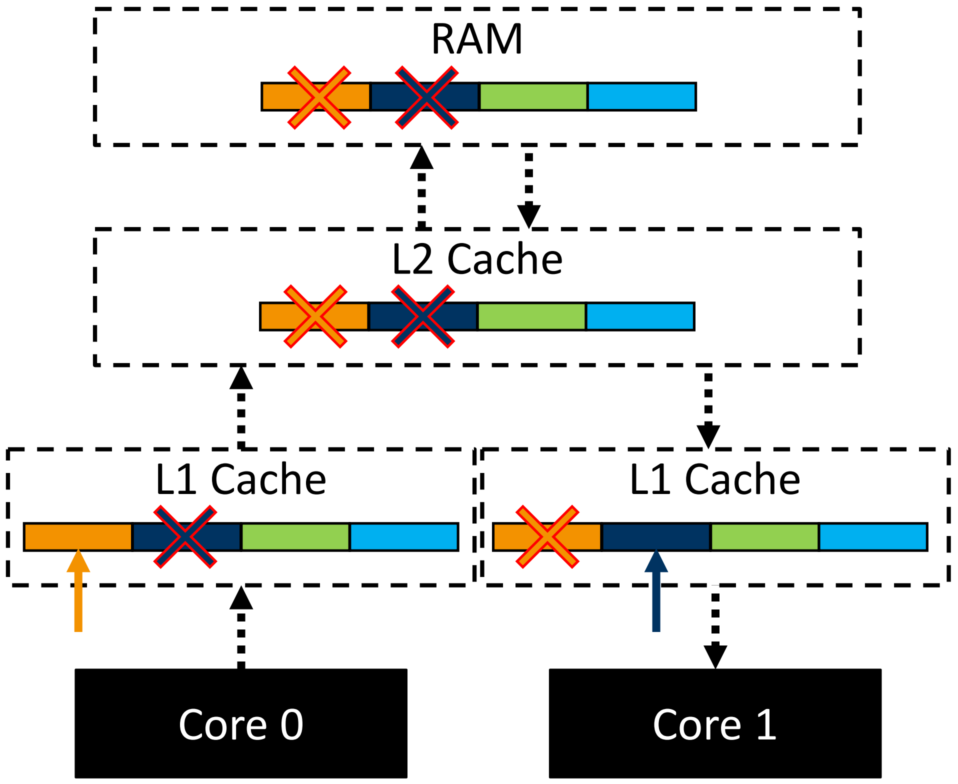
- common performance pitfall in OpenMP
  - cache coherence tries to keep all data up-to-date and valid for all threads
  - causes unnecessary coherence traffic and cache misses



```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;

  #pragma omp for
  for (i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me];
}
```

False Sharing cont'd

- thread 1
  - reads first 8 bytes
    - causes entire cache line to be fetched
  - writes first 8 bytes
    - entire cache line invalidated for thread 2
- thread 2
  - reads second 8 bytes
    - causes entire cache line to be fetched
  - writes second 8 bytes
    - entire cache line invalidated for thread 1



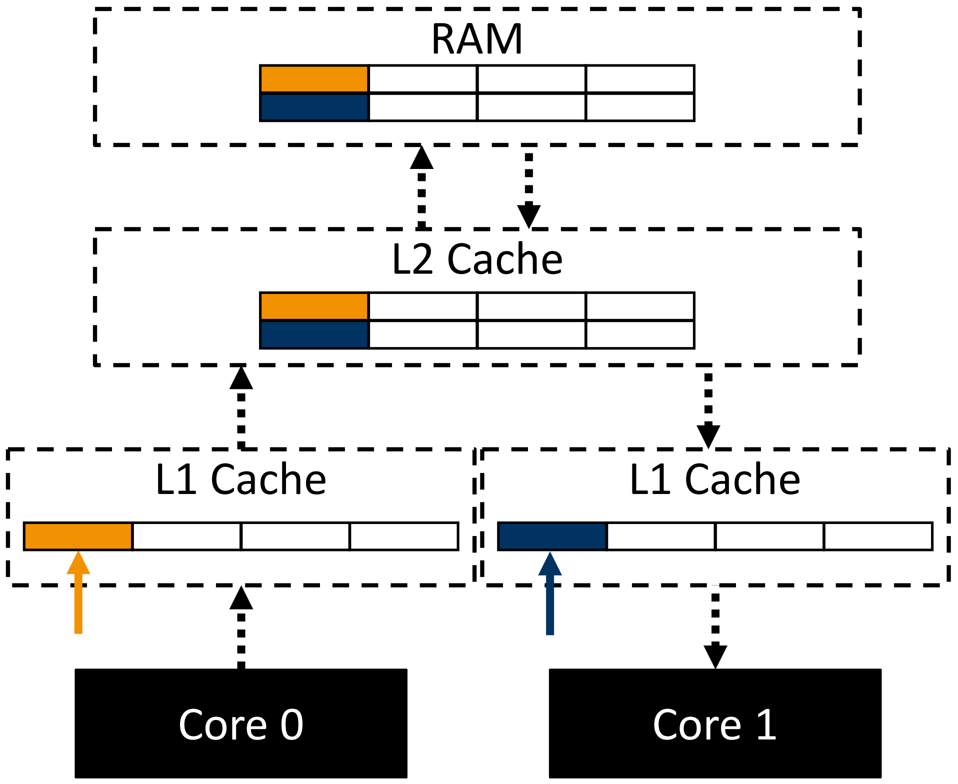
False Sharing Solution

```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;

  #pragma omp for
  for (i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me];
}
```

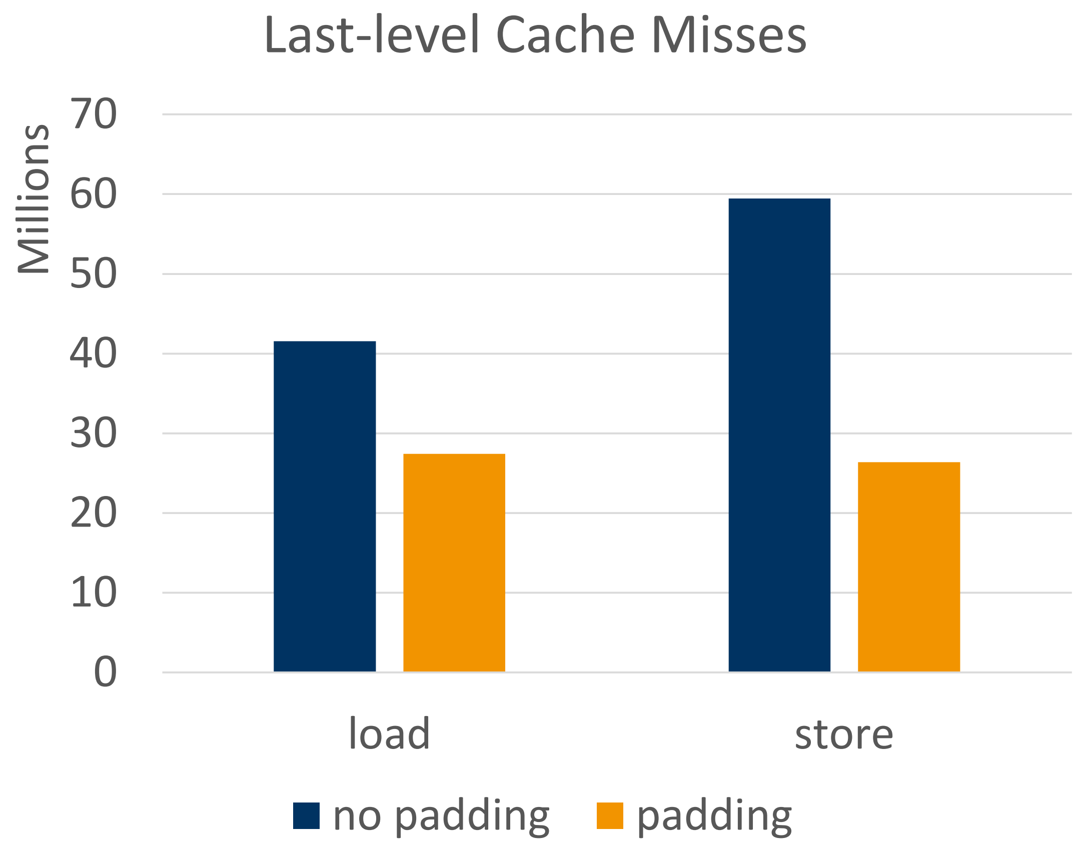
```
double sum = 0.0;
double sum_local[MAX_NUM_THREADS][8];

#pragma omp parallel
{
  int me = omp_get_thread_num();
  sum_local[me][0] = 0.0;

  #pragma omp for
  for (i = 0; i < N; i++)
    sum_local[me][0] += x[i] * y[i];

  #pragma omp atomic
  sum += sum_local[me][0];
}
```

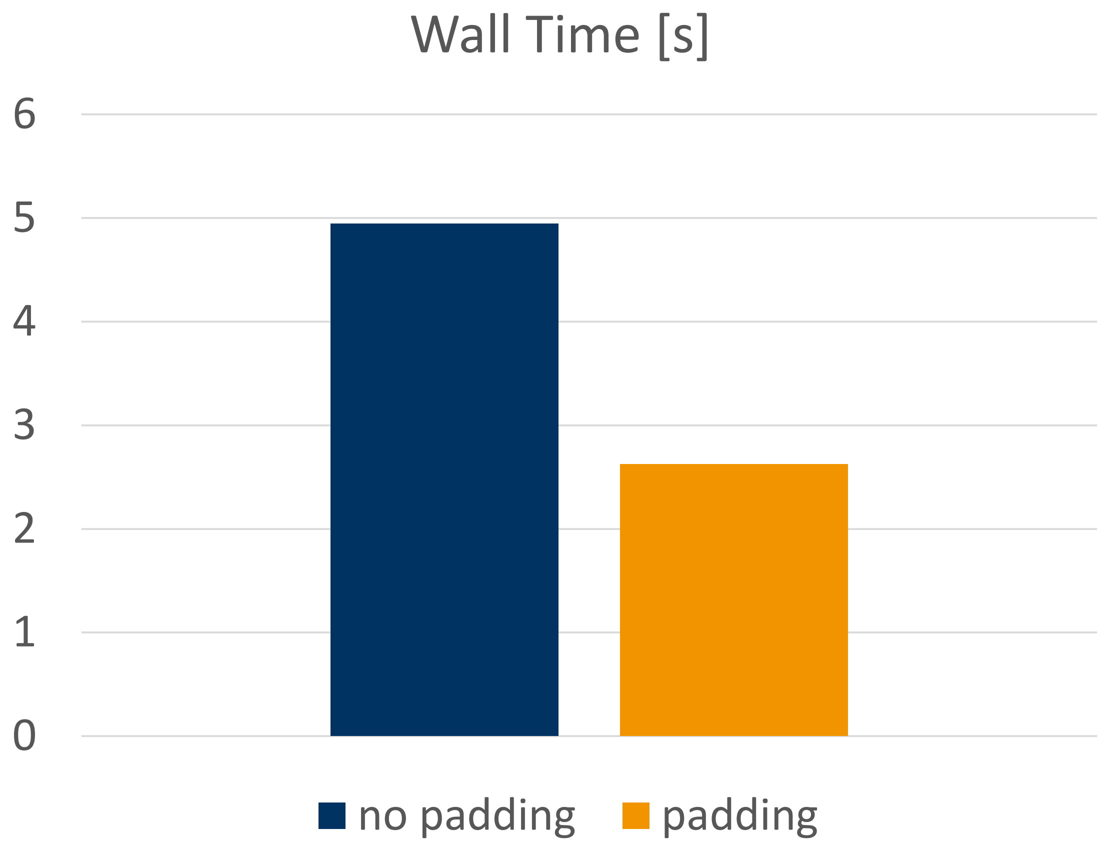
False Sharing Solution cont'd

- thread 1
  - reads first 8 bytes of first cache line
    - causes first cache line to be fetched
  - writes first 8 bytes
- thread 2
  - reads first 8 bytes of second cache line
    - causes second cache line to be fetched
  - writes second 8 bytes



False Sharing Performance Comparison (LCC2, 10^9 iterations)





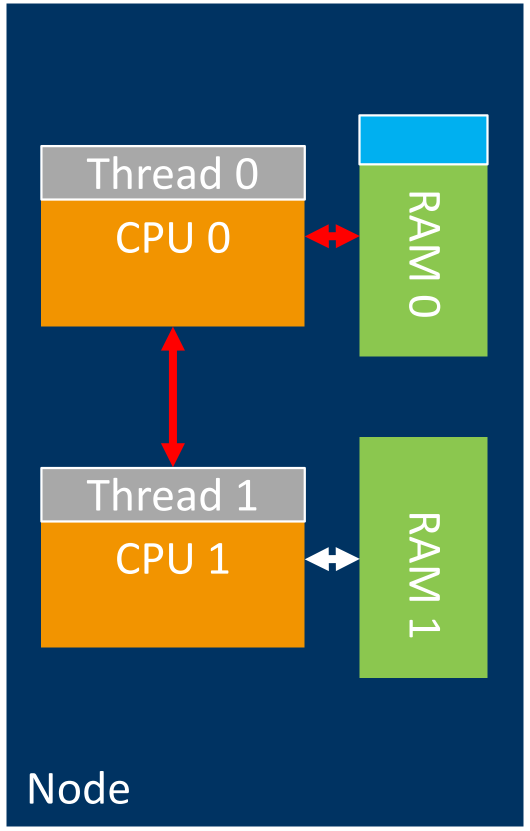
First Touch & NUMA – How to Initialize Your Data?

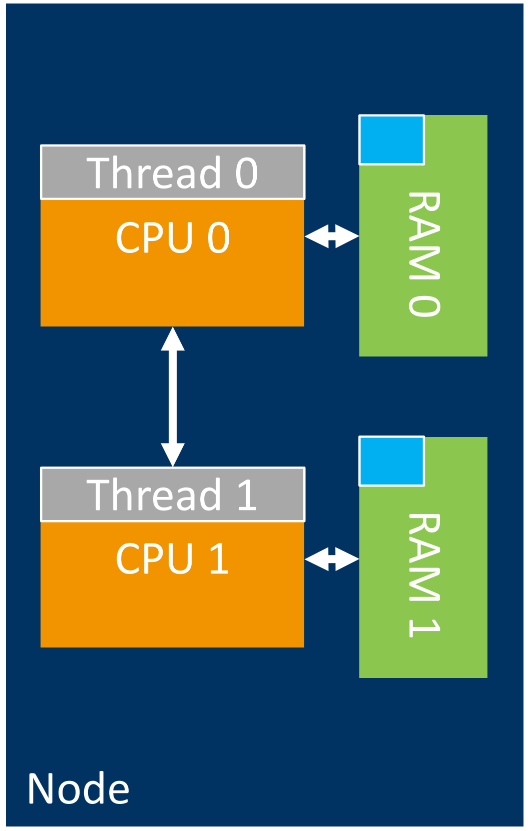
```
double* x = malloc(sizeof(double)*SIZE);
double* y = malloc(sizeof(double)*SIZE);

for(int i = 0; i < SIZE; ++i) {
  x[i] = 0.0; y[i] = 1.0;
}

#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] += y[i];
  }
}
```

```
double* x = malloc(sizeof(double)*SIZE);
double* y = malloc(sizeof(double)*SIZE);

#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] = 0.0; y[i] = 1.0;
  }
  #pragma omp for schedule(static)
  for(int i = 0; i < SIZE; ++i) {
    x[i] += y[i];
  }
}
```

Sequential vs. Parallel Initialization on NUMA

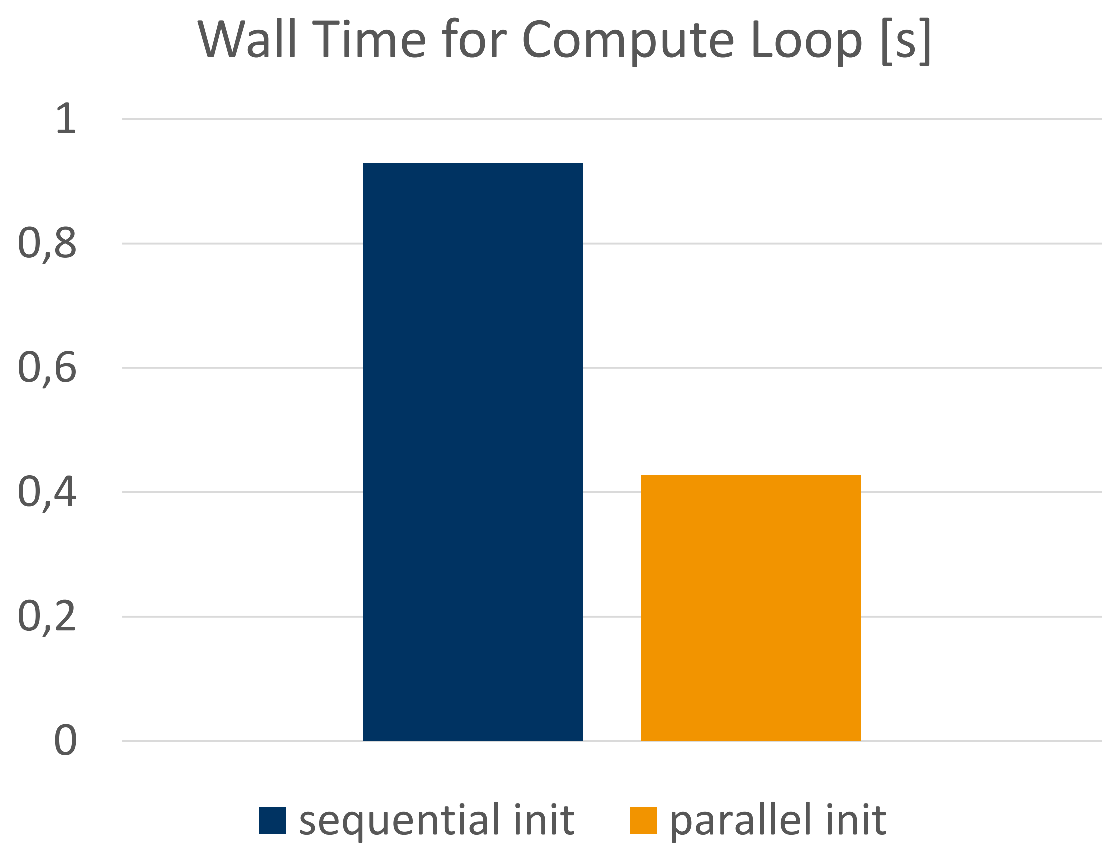
- data is not allocated upon allocation but upon first access (“*first touch*”)
  - happens when you initialize data in the memory of the initializing thread
- sequential initialization
  - all data resides with RAM modules of core processing initial thread
  - causes bottleneck on single memory bus, additional inter-CPU traffic and higher latency for core 1
- parallel initialization
  - data resides with RAM of the threads initializing the respective chunk of data
  - only downside: one more pragma





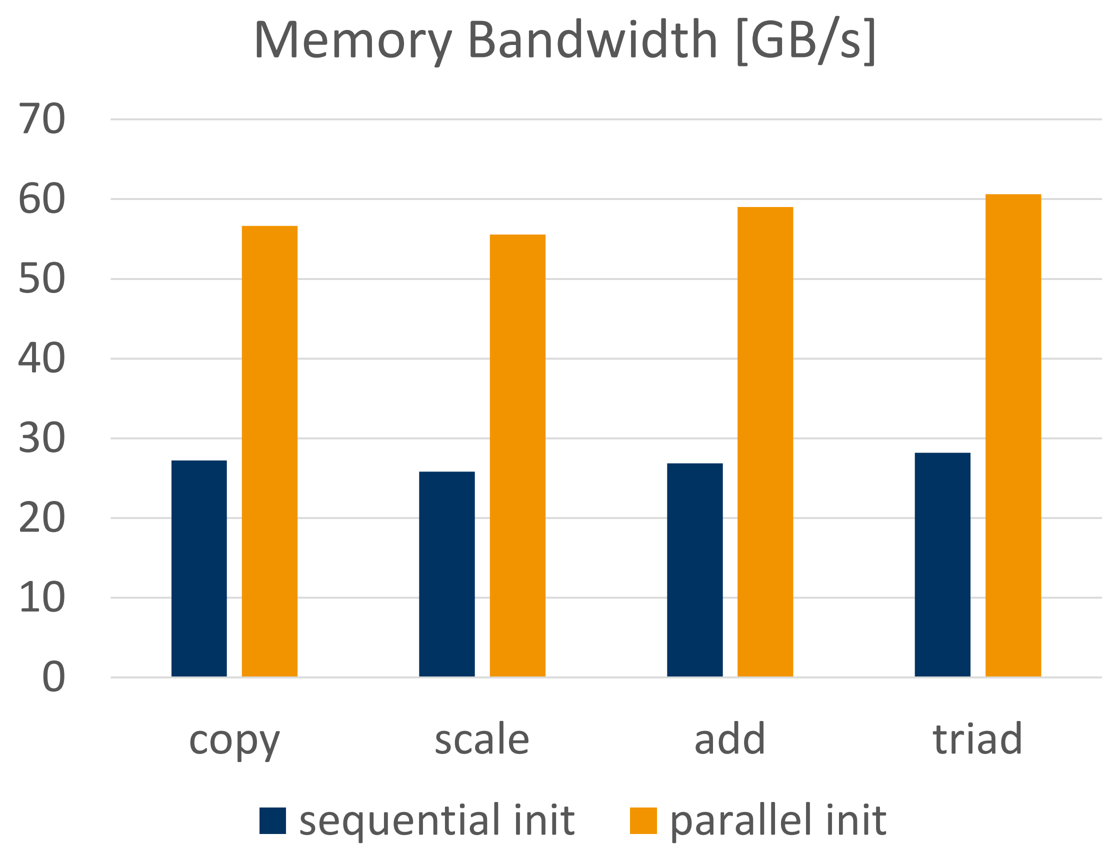
Performance Impact of First Touch and NUMA

- hudson server (2x Intel Xeon E5-2699 v3 18-core), gcc 6.3.0, 10^8 double elements, 10 repetitions
- performance improvement of compute loop (not initialization!) of 2.17x



Performance Impact of First Touch and NUMA cont’d

- same platform, stream memory benchmark, 3 threads per CPU
  - https://www.cs.virginia.edu/stream/
- between 2.08x and 2.20x higher bandwidth



Summary

- task-based parallelism
- modern OpenMP features
  - affinity
  - vectorization
  - accelerators
- common OpenMP pitfalls