UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

High-Performance Computing
– Shared Memory Programming with OpenMP –

Christian Plessl

High-Performance IT Systems Group
Paderborn University, Germany

version 1.0.2 2018-01-09

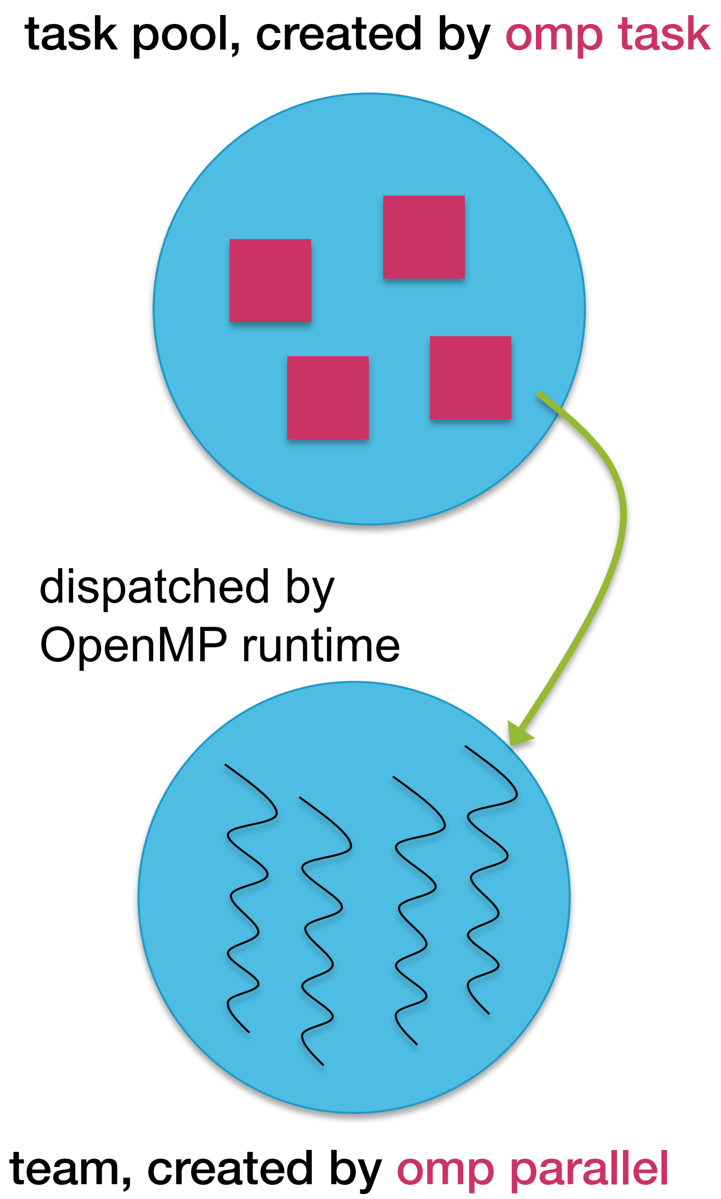
Paderborn Center for Parallel Computing

Outline

- Basic OpenMP (covered by Pacheco book, up to Section 5.7)
  - concepts
  - work sharing (loop parallelization)
  - variable scoping
- More OpenMP (covered now)
  - task parallelism
  - synchronization
  - ccNUMA-aware memory initialization
  - false sharing

OpenMP Tasks

- task construct defines independent units of work
- A task is composed of
  - code to execute
  - a data environment (initialized at creation time)
- A thread that encounters a task construct may
  - execute the task immediately, or
  - defer the execution and put the task into a task queue
- The threads in a team cooperate to complete the execution of tasks
- Tasks are more flexible than work sharing constructs (parallel for, section, etc.)
  - allow unstructured parallelism (while loops, recursion, …)
  - one task or many threads can spawn tasks
  - combination of work sharing and tasks


task pool, created by omp task
dispatched by OpenMP runtime
team, created by omp parallel

The task Directive

- Syntax

```
#pragma omp task [clauses]
  structured block
```

clauses: shared, private, firstprivate, default, if(expression)

- Example

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p = listhead;
    while (p) {
      #pragma omp task
         process(p);
      p=p->next;
    }
  }
}
```

spawn one process for each element in linked list, to be processed independently

The Data Environment

- Because tasks can be deferred, the data is captured at task creation
- The semantics of scopes are adapted to deferred execution scenario:
  - for a shared variable, the references to the shared variable within the task are to the memory location with that name at the time where the task was created
  - for a private variables the references to the variable inside the task are to new unitialized storage that is created when the task is executed
  - for a firstprivate variable, the reference to the variable inside the task is to new storage that is crated and initialized with the value of the existing memory of that name when the task is created
- Scopes are inherited from enclosing block

```
#pragma omp parallel shared(a) \
    private(b)
{
   ...
   #pragma omp task
   {
       int c;
       compute(a, b, c);
   }
}
```

a is shared
b is firstprivate
c is private

The Data Environment (2)

- Capturing data structures with firstprivate does capture only the pointer but not the pointed data.
- In other words, firstprivate capturing does not perform a deep copy of data structures. If required, you need to make a copy yourself.
- Make sure that origin data is available at the time of creating the deep copy

Typical Idiom for Creating Tasks

```
#pragma omp parallel {

  #pragma omp single {

      #pragma omp task
         foo ();
      #pragma omp task
         bar();
      #pragma omp task
         baz();
  }

}
```

create team of threads

let one thread package tasks

create three tasks, which are executed in any order

wait for completion of all tasks at implicit barrier

When are Tasks Completed?

- Barriers: tasks completion is ensured with at explicit or implicit thread barriers
  - #pragma omp barrier (explicit)
  - at end of worksharing construct, e.g. #omp parallel for
  - applies to all tasks generated in the current parallel region up to the barrier
- Taskwait directive
  - task completion can be enforced with #pragma omp taskwait directive
  - causes program to wait until all tasks defined in the current task have completed
- Beware: taskwait applies only to sibling tasks generated in the current task, not to “descendants” spawned by these tasks

Syntax:

```
#pragma omp taskwait
```

Task Synchronization with Taskwait (1)

```
#pragma omp parallel {

  #pragma omp single {

      #pragma omp task
         foo ();
      #pragma omp task
         bar();
      #pragma omp taskwait

      #pragma omp task
         baz();
  }

}
```

create team of threads

let one thread package tasks

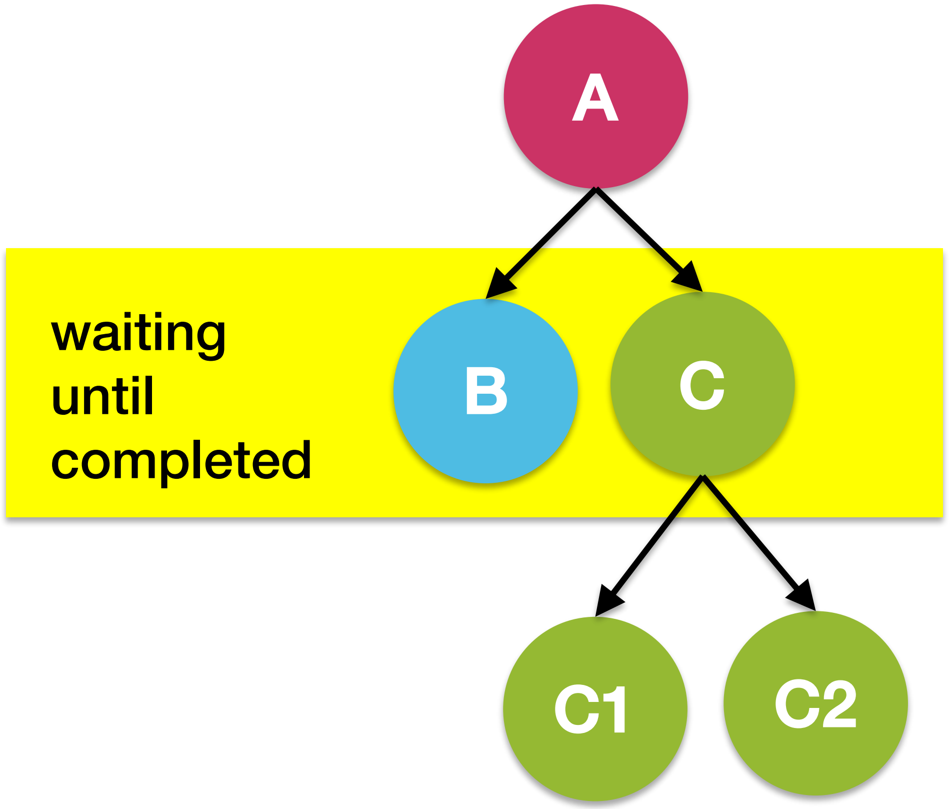
create two tasks, which are executed in any order

wait for completion of task foo and bar

wait for completion of baz at implicit barrier

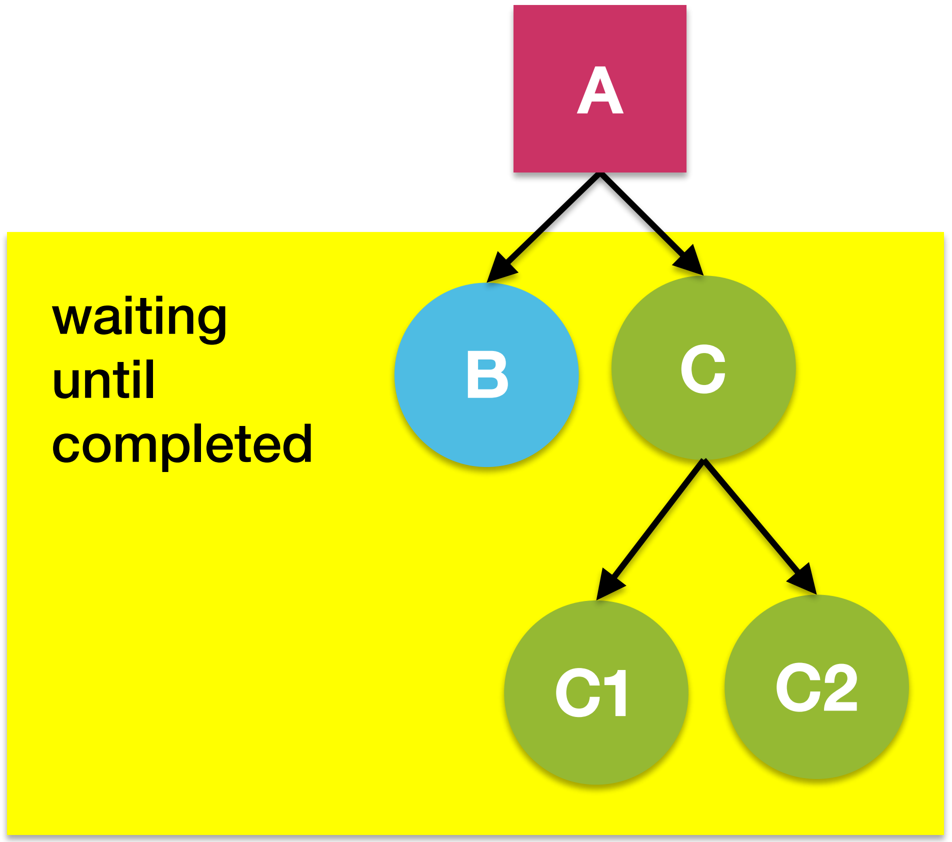
Task Synchronization with Taskwait (2)

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  {
    #pragma omp task
    { … }
    #pragma omp task
    { spawn subtasks C1, C2 …}
    #pragma omp taskwait
  }
}
```



Task Synchronization with Taskwait (3)

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup
  {
    #pragma omp task
    { … }
    #pragma omp task
    { spawn subtasks C1, C2 …}
    #pragma omp taskwait
  }
}
```



Example: Postorder Tree Traversal

- Process all elements in binary tree in postorder (left child, right child, root node)
- Traversal using recursive function postorder()

```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
    postorder(p->left);
  if (p->right)
    #pragma omp task
    postorder(p->right);
  #pragma omp taskwait
  process(p->data);
}
```

taskwait required to guarantees that processing is suspended until children are completed

Cutoff Mechanisms for Recursively Generated Tasks

- For recursively defined divide-and conquer strategies, a cutoff must be defined to prevent task/state explosion
- Mechanisms
  - explicitly programmed cutoff (explicitly call function that does not spawn further threads at cutoff)
  - if clause
  - mergeable clause
  - final clause
- Support in OpenMP implementations
  - explicitly programmed cutoff and if clause are widely supported
  - mergable and final clauses are currently not exploited by any implementation

Divide and Conquer with Explicit Cutoff (1)

- Add all elements in array with divide an conquer approach
- Create sufficient tasks to keep cores busy, prevent overheads be spawning too many tiny tasks

```
float sum(const float* a, size_t n)
{
  // base cases
  if(n==0) return 0;
  if(n==1) return*a;

  // recursive case
  size_t half=n/2;
  return sum(a,half) + sum(a+half,n-half);
}
```

serial version

Divide and Conquer with Explicit Cutoff (2)

```
float sum(const float* a, size_t n)
{
  // base cases
  if(n == 0) return 0;
  if(n == 1) return *a;

  // recursive case
  size_t half = n/2;
  float x, y;
  #pragma omp parallel
  #pragma omp single nowait
  {
    #pragma omp task shared(x)
    x = sum(a,half);
    #pragma omp task shared(y)
    y = sum(a+half,n-half);
    #pragma omp taskwait
    x += y;
  }
 return x;
}
```

- Create tasks for recursive calls
- Problem
  - creates too many tiny tasks
  - inefficient
- Solution
  - stop spawning new tasks at certain cutoff point

Divide and Conquer with Explicit Cutoff (3)

```
#define CUTOFF 100  // arbitrary

static float parallel_sum(const float *a,
  size_t n){
  // base case
  if (n <= CUTOFF){
    return serial_sum(a, n);
  }
  // recursive case
  float x, y;
  size_t half = n / 2;
  #pragma omp task shared(x)
  x = parallel_sum(a, half);
  #pragma omp task shared(y)
  y = parallel_sum(a + half, n - half);
  #pragma omp taskwait
  x += y;
  return x;
}
```

```
float sum(const float *a, size_t n){
  float r;
  #pragma omp parallel
  #pragma omp single nowait
  r = parallel_sum(a, n);
  return r;
}

static float serial_sum(const float *a,
  size_t n){
  float x = 0.0;
  for(int i=0; i<n; i++) {
    x += a[i];
  }
  return x;
}
```

If Clause

- Syntax

```
#pragma omp task if(expression)
```

- The if clause allows to decide at runtime whether a new task will be created
  - reduce parallelism
  - reduce competition for threads in team
- If the condition evaluate to false
  - parent task is suspended
  - new task is immediately executed (kind of ”inlining”)
  - parent task is resumed
- The generated “task” is denoted as undeferred

Mergeable Clause

- The mergeable clause allows to runtime to decide, whether a task is created as deferred or undeferred task
- Can be considered as an automatic if-clause controlled by OpenMP runtime

- Syntax

```
#pragma omp task mergeable
```

- Objectives
  - reduce overhead in OpenMP runtime
  - ‘inline’ data environment, efficient sharing

- Example (see final clause)

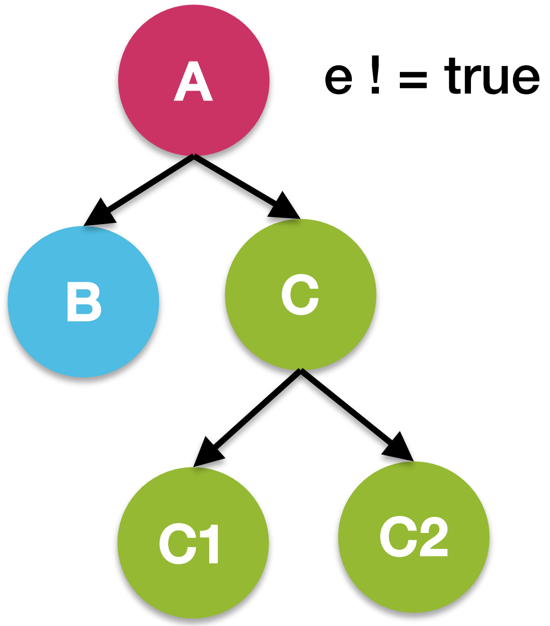
Final Clause

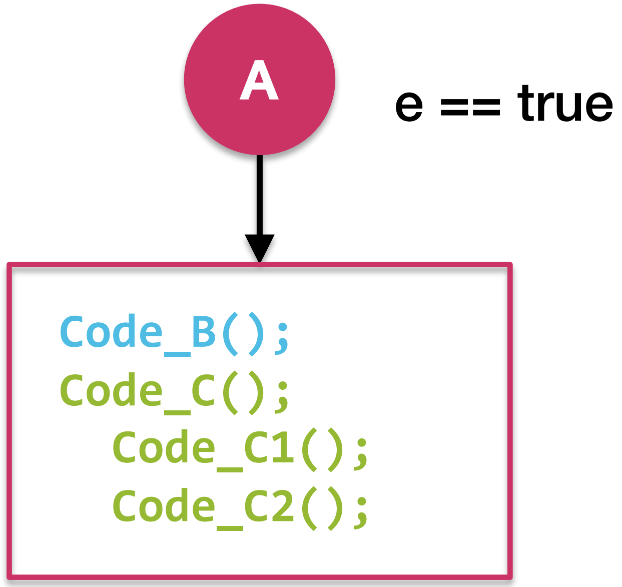
- The final clause prevents the creation of further tasks if condition evaluates to true

```
#pragma omp task final(e)
```

- Purpose
  - limits degree of parallelism, prevent too many tasks
  - new task is generated but this task cannot create further subtasks

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task final(e) {
    #pragma omp task
    { … }
    #pragma omp task
    { spawn subtasks C1, C2 …}
    #pragma omp taskwait
  }
}
```





```
Code_B();
Code_C();
  Code_C1();
  Code_C2();
```

Example: Cutoff with Final Clause

```
#include <string.h>
#include <omp.h>
#define LIMIT  3 /* limit on recursion depth */
void check_solution(char *);
void bin_search (int pos, int n, char *state)
{
  if ( pos == n ) {
    check_solution(state);
    return;
  }
  #pragma omp task final( pos > LIMIT ) mergeable
  {
    char new_state[n];
    if (!omp_in_final() ) {
      memcpy(new_state, state, pos );
      state = new_state;
    }
    state[pos] = 0;
    bin_search(pos+1, n, state );
  }
```

```
  #pragma omp task final( pos > LIMIT ) mergeable
  {
    char new_state[n];
    if (!omp_in_final() ) {
      memcpy(new_state, state, pos );
      state = new_state;
    }
    state[pos] = 1;
    bin_search(pos+1, n, state );
  }
  #pragma omp taskwait
}
```

Task Priorities

- At the creation of a task, a priority can be defined

```
#omp task priority(value) [further clauses]
structured block
```

- Priority is a non-negative value in the interval 0 ≤ priority ≤ omp_get_max_task_priority()
- When scheduling tasks to idle threads, tasks with higher priority get preferred
- Priority is only a hint to the OpenMP runtime scheduler, it is not allowed to rely on priorities for correct execution

```
#pragma omp parallel
#pragma omp single
{
  for(i=0;i<N;i++) {
    #pragma omp task priority(1)
    background_task(i);
  }
  for(j=0;j>M;j++) {
    #pragma omp task priority(100)
    foreground_task(j);
  }
}
```

Task Dependencies

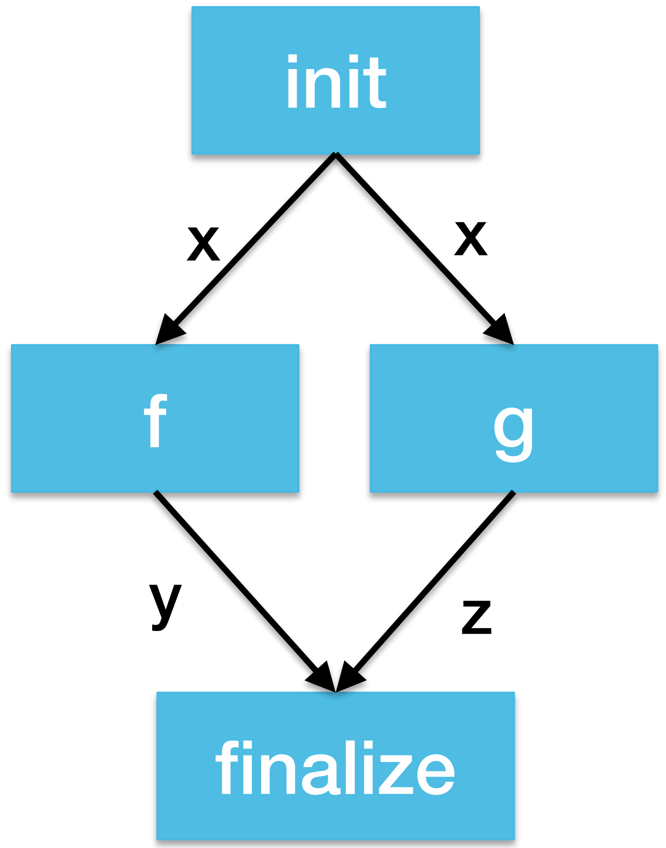
- In OpenMP 4.0 a task dependency clause has been introduce
- Purpose
  - provide more expressive synchronization than #pragma omp taskwait
  - allow to create real dataflow graph
- Syntax

```
#pragma omp task depend (dependency-type : list-items )
```

- Dependency types: `in`, `out`, `inout`
- List-item
  - variable, e.g. `depend(in: x)`
  - array section, e.g. `depend(in: a[0:9])`
- Tasks cannot complete until all predecessors have completed

Example: Data Flow Graph

```
#pragma omp parallel
#pragma omp single
{
  int x, y, z;

  #pragma omp task depend( out: x )
  x = init();

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

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

  #pragma omp task depend( in: y, z )
  finalize(y, z);
}
```



Philosophy of Task Dependencies

- Task dependencies are orthogonal to data sharing
  - dependencies define constraints on execution order
  - data sharing clauses describe how data is captured and accessible in task
- Dependencies and data sharing together allow for defining task data-flow model
  - data that is read in the task → input dependency
  - data that is written by the task → output dependency
- Task data-flow model improve code quality
  - better composability (explicit dependencies)
  - simplified extensibility to new regions of code

The taskloop Directive

- OpenMP worksharing with “parallel” for does not compose well, i.e. each thread may call additional worksharing constructs that span even more threads
- Problem
  - oversubscription of system (OS overheads, competition for caches, …)
  - OpenMP overheads (repeated creation/destruction of teams, scheduling)
  - difficult to write generic libraries that exploit all cores
- The taskloop directive works similar to a parallel for worksharing directive
  - divides loop iterations into chunks
  - adds a task for each junk
- Syntax:

```
#pragma omp taskloop [clauses]
for-loops
```

The taskloop Clauses

- Taskloop directive inherits clauses from task (T) and worksharing (WS) constructs
  - shared, private, default (T/WS)
  - firstprivate, lastprivate (WS)
  - reduction, in_reduction (WS, proposed for T in OpenMP 5.0)
  - collapse (WS)
  - final, untied, mergable (T)

- Additional clause: grainsize(grain-size)
  - number of logical loop iterations assigned to each chunk must be at least grain-size and at most 2*grain-size

- Additional clause: num_tasks(num-tasks)
  - create num-tasks tasks computing the iterations of the loop

Synchronization

- OpenMP supports many ways to synchronize tasks to
  - protect access to shared data
  - impose ordering constraints
- High-level constructs
  - critical
  - barrier
  - atomic
  - ordered (not discussed)
- Low-level constructs
  - flush (not discussed)
  - locks
  - nested locks

Critical Directive

- The critical directive protects critical sections
  - only one thread may enter a critical section at a time
- Syntax

```
#pragma omp critical [ (name) [hint(hint-expression)] ]
structured block
```

- Clauses
  - multiple critical sections are supported by giving them unique names
  - hint provides OpenMP runtime with information about the kind of expected conflicts, enables picking the most efficient mechanism (OS function, transactional memory, atomic operations, …)
    - omp_lock_hint_none: no specific lock requested, let OpenMP choose
    - omp_lock_hint_uncontended: conflicts are rare
    - omp_lock_hint_contended: conflicts are frequent
    - omp_lock_hint_nonspeculative: do not use speculative locking, high conflict potential
    - omp_lock_hint_speculative: use speculative locking
  - hint flags are bitmaks that can be combined

```
hint(omp_lock_contended | omp_lock_speculative)
```

Critical Directive Example

```
int dequeue(float*a);
void work(int i, float*a);

void critical_example(float*x, float*y){
  int ix_next, iy_next;

  #pragma omp parallel shared(x, y) private(ix_next, iy_next)
  {
    #pragma omp critical (xaxis)
      ix_next = dequeue(x);
    work(ix_next, x);

    #pragma omp critical (yaxis)
      iy_next = dequeue(y);
    work(iy_next, y);
  }
}
```

protect dequeing from queue xaxis

protect dequeing from queue yaxis

different threads can dequeue concurrently they work on different queues

example: OpenMP 4.5 official examples

Barrier Directive

- The barrier directive specifies a synchronization barrier
- All threads must reach the barrier, before the execution proceeds
- Syntax

```
#pragma omp barrier
```

```
double a[N], b[N];

#pragma omp parallel
{
  int t = omp_get_thread_it();
  a[t] = foo(t);

  #pragma omp barrier
  b[t] = bar(t,a);
}
```

ensure that all a[ ]'s have been computed

Atomic Directive

- The atomic directive provides mutual exclusion for a single memory location
  - very efficient by using hardware support
- Syntax

```
#pragma omp atomic [ read | write | update | capture ]
expression-stmt
```

- atomic operation expression-stmt must be in one of these forms:
  - if clause is read
    v=x
  - if clause is write
    x = expr
  - if clause is update or not present
    x++, x--, ++x, --x; x binop= expr; x = x binop expr; x= expr binop x
  - if atomic clause is capture
    v = x++; v=x--; v=++x; v=--x; v = x binop= expr;
  - binop is one of: +, *, -, /, &, ^, |, <<, or >>
- see OpenMP standard for details

OpenMP Locking

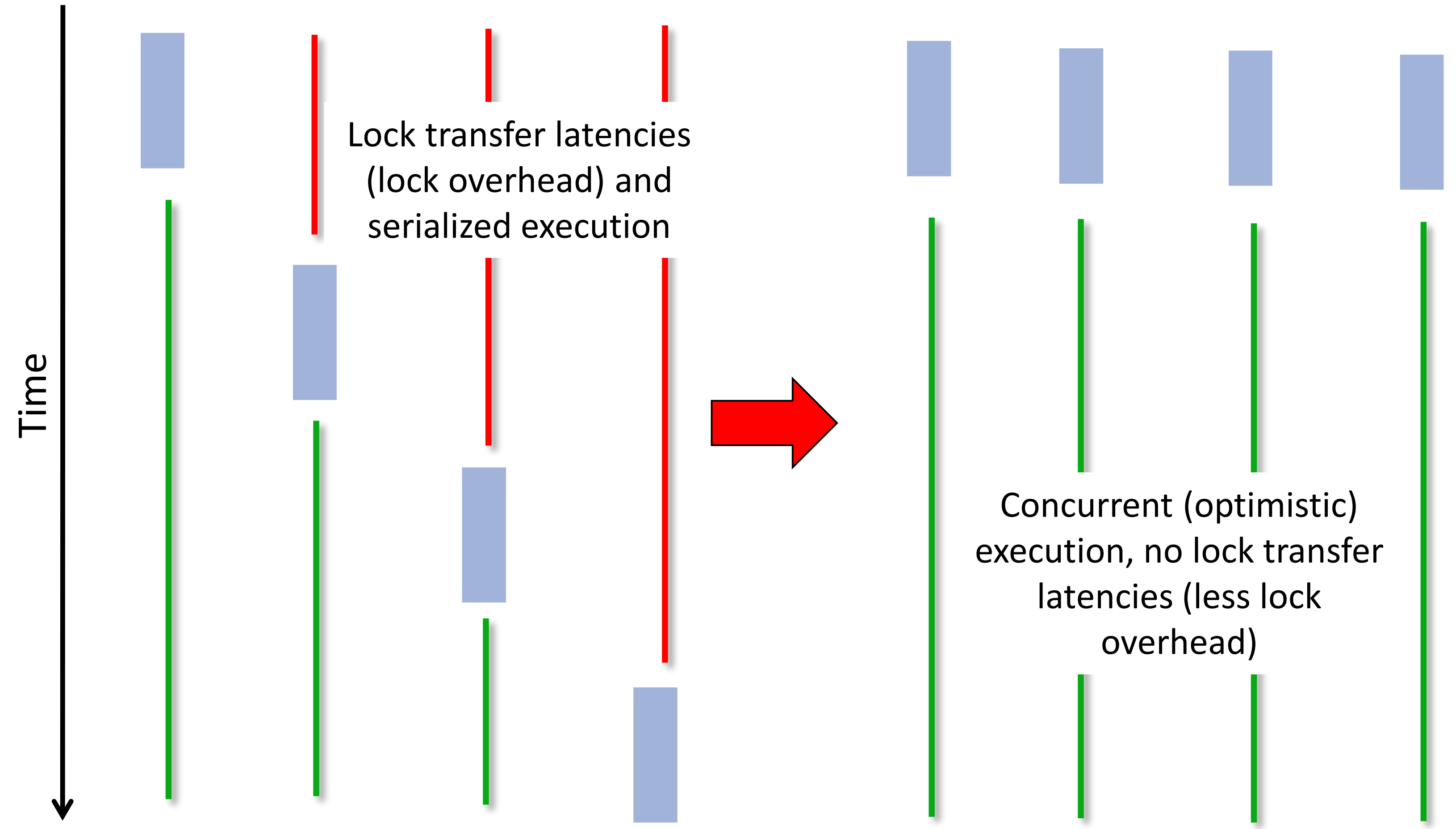
- OpenMP runtime provides general purpose locking functions for synchronization
- Two kinds of locks
  - simple locks
  - nestable locks
- Locks can be in three states: uninitialized, locked, unlocked
- The locking API is straightforward
  - simple locks are available if unlocked
  - nested locks are available if unlocked, or if it is already locked by the thread executing the lock function
  - nesting can be useful for writing functions that need to obtain lock on a data structure, but can be called from a context where the lock is already held and a context where the lock is not held

Locking API

- `omp_lock_t`
  - variable type for lock
- `omp_init_lock(omp_lock_t *lock)`
  - initializes the lock to unlocked, without hint
- `omp_init_lock_with_hint(omp_lock_t *lock, omp_lock_hint_t hint)`
  - initializes lock and attaches as hint to it, hint has same meaning as for critical directive
- `omp_set_lock(omp_lock_t *lock)`
  - waits until the lock is available, and then sets it
- `omp_unset_lock(omp_lock_t *lock)`
  - unsets the lock
- `omp_destroy_lock(omp_lock_t *lock)`
  - uninitializes the lock
- `omp_test_lock(omp_lock_t *lock)`
  - tests a lock and sets the lock if available

routines for nested locks are named analogously:
`omp_set_lock` → `omp_set_nest_lock`

Locking Hints Offer Opportunities for Lock Elision


Time
Lock transfer latencies (lock overhead) and serialized execution
Concurrent (optimistic) execution, no lock transfer latencies (less lock overhead)

Simple Lock Example

```
omp_lock_t lck;
int id;
omp_init_lock(&lck);

#pragma omp parallel shared(lck) private (id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lck);
  printf(”My thread id is %d.\n”, id);
  omp_unset_lock(&lck);

  while (! omp_test_lock(&lck)) {
    skip(id);
  }

  work(id);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

only one thread at a time can execute this printf

we don’t have the lock, do something else while waiting

now we have the lock, do the work

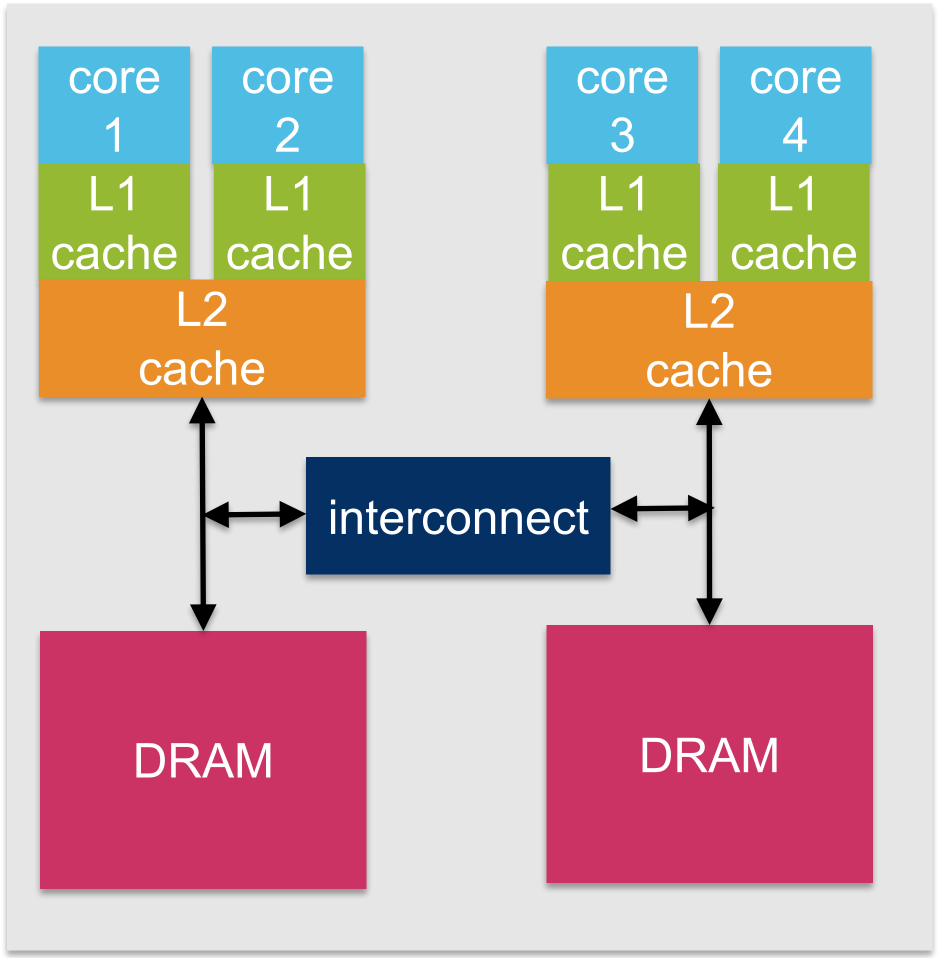
Efficient Use of Memory With OpenMP

- OpenMP makes writing shared memory programs simple
- But disregarding the underlying multi-core architecture can lead to poor performance
- Indirect interaction of threads via memory subsystem
  - memory allocators
  - caches and coherency protocols
  - non-uniform memory architectures

OpenMP and Non Uniform Memory Architectures

```
double *a;
a = (double *)malloc(N*sizeof(double));



for(i=0; i<N; i++) {
  a[i] = 0.0;
}
```

Questions:
- Where in the address space will a[] be allocated?
- In which DRAM will a[] be stored?
- What is the impact if we process loop with multiple threads?



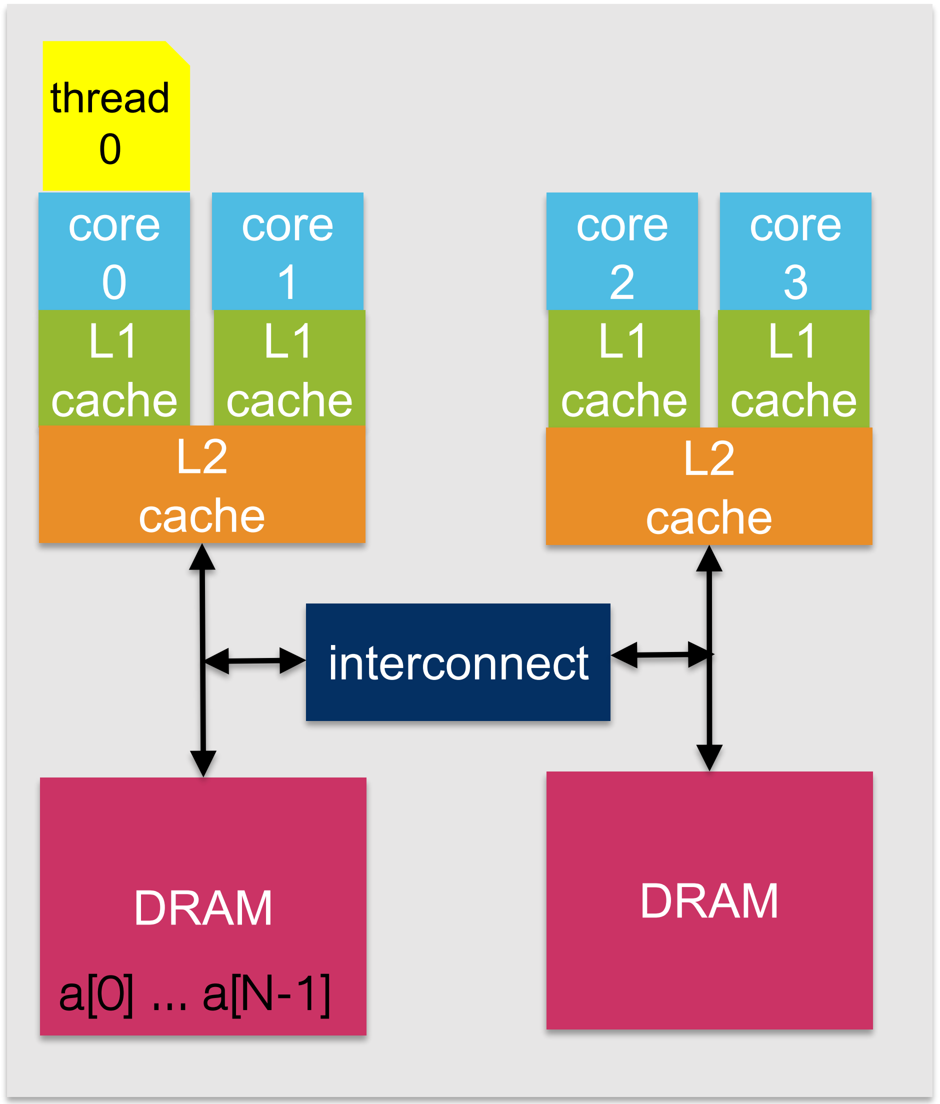
Data Allocation in ccNUMA Systems

- Modern multi-processor machines typically have cache-coherent non-unified memory access (ccNUMA) architecture
  - memory is transparently shared
  - cores have private and/or shared caches
  - latency and bandwidth of main memory access varies with placement of data in DRAM
  - shared memory paths subject to contention
- OpenMP has no knowledge or direct control over placement
- ‘First touch’ memory allocation policy (Linux, Windows, ...)
  - memory placement is not decided at malloc-time, but deferred until later
  - on first data access, the data is placed in DRAM attached to the processor (core) that performed the access

OpenMP and Non Uniform Memory Architectures (2)

- For serial code, all data will be placed in memory of processor that runs the thread

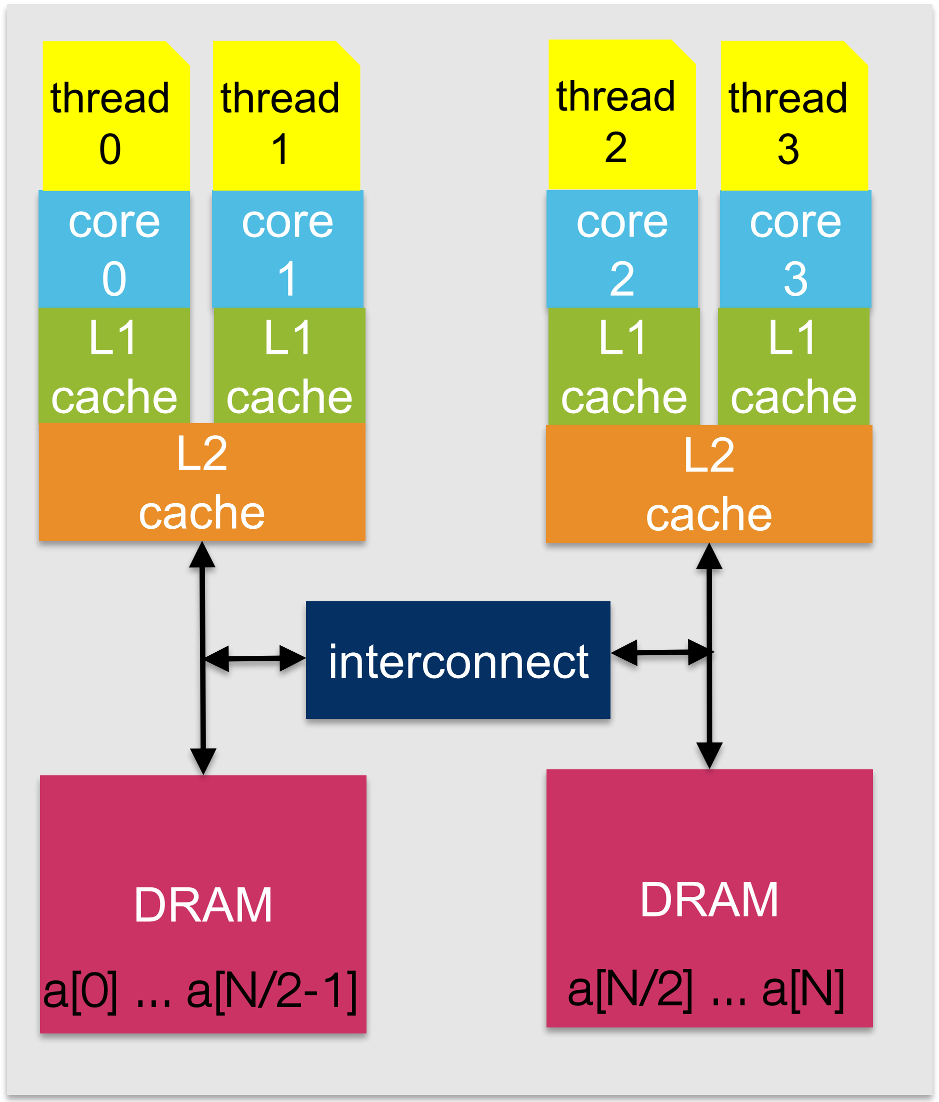
```
double *a;
a = (double *)malloc(N*sizeof(double));

for(i=0; i<N; i++) {
  a[i] = 0.0;
}
```



OpenMP and Non Uniform Memory Architectures (3)

- For parallel code, data will be placed in memory of processor that executes the thread performing the initialization
- Multi-threaded application can exploit the doubled memory bandwidth
- But we need to make sure that the data ends up in the right DRAM

```
double *a;
a = (double *)malloc(N*sizeof(double));

#pragma omp parallel for
for(i=0; i<N; i++) {
  a[i] = 0.0;
}
```



Example: ccNUMA Effect for Stream Benchmark

- Simplified version of stream benchmark
  - determine time for vector summation on long arrays (exceeding cache size)
  - compute achieved memory bandwidth
- Execution on dual-socket Intel SandyBridge server on Oculus

threads	serial init	parallel init
1	11003	10973
2	16643	22997
4	19810	36652
8	19191	50579
16	19004	48959

serial version does not run this loop in parallel

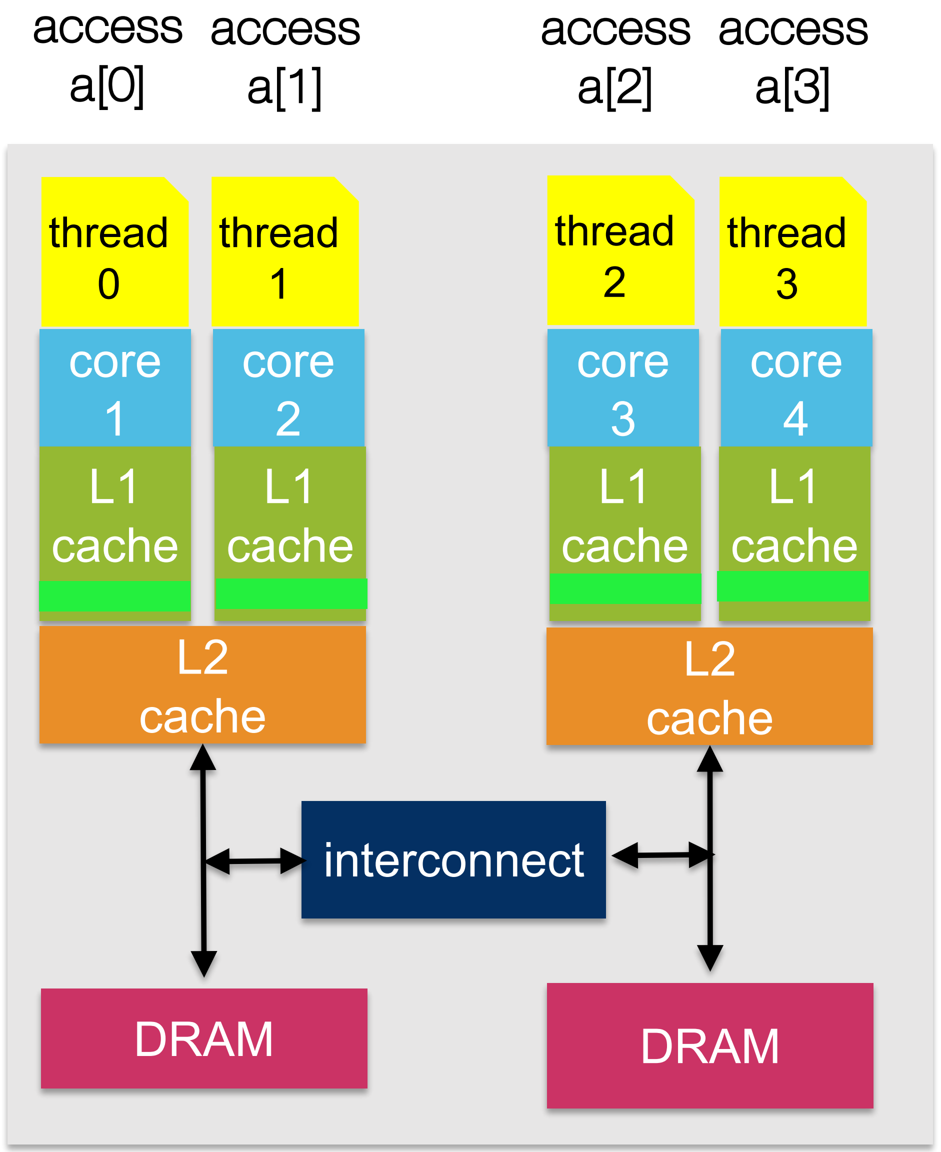
```
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
{
  a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;
}

for (k=0; k<NTIMES; k++)
{
  times[k] = mysecond();

  #pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE; j++)
    c[j] = a[j]+b[j];

  times[k] = mysecond() - times[k];
}
```

False Sharing

- False sharing not specific to OpenMP
  - can occur with any shared memory programming model
  - OpenMP's convenience for sharing arrays makes it easy to hit false sharing problem
- Symptoms
  - different threads access elements of a data structure that shares a cache line
  - different threads always access different structure elements (i.e. no sharing)
  - each write invalidates cached copy in other caches
  - lots of unneeded cache coherency traffic causes severe slow downs

cache line that is copied back and forth

a[0]a[1]a[2] .. a[7]



False Sharing Example

- Example
  - multiple array elements A[i] share one cache line
  - each thread i only writes to element A[i]
- Performance on Oculus for n = 100’000’000

	false sharing	
threads	time [s]	speedup
1	0.58	1.00
2	0.76	0.76
4	0.74	0.78
8	0.74	0.78
16	0.66	0.87

```
int main(int argc, char *argv[]) {

 const int NTHREAD = ...;
 int n = ...;

 int A[NTHREAD] __attribute__((aligned(64)));
 int sum=0;
 #pragma omp parallel num_threads(NTHREAD)
 {
   int thread =omp_get_thread_num();
   A[thread] = 0;

   #pragma omp for schedule(static,1)
   for(int i=0; i<=n; i++)
     A[thread] += i % 10;

   #pragma omp atomic
   sum += A[thread];
 }
 printf("The sum is %d\n",sum);

}
```

Avoiding False Sharing

- Techniques for avoiding false sharing
  - use private copy of data structure
  - pad data structures to ensure that elements are stored in different cache lines
- Performance on Oculus for n = 100’000’000

	false sharing		with padding	
threads	time [s]	speedup	time [s]	speedup
1	0.58	1.00	0.48	1.00
2	0.76	0.76	0.32	1.50
4	0.74	0.78	0.19	2.58
8	0.74	0.78	0.07	6.45
16	0.66	0.87	0.04	12.23

```
int main(int argc, char *argv[]) {
 const int NTHREAD = ...;
 int n = ...;
 const int CACHE_LINE_SIZE = 64;
 const int PAD = CACHE_LINE_SIZE/sizeof(int);

 int A[NTHREAD][PAD] __attribute__((aligned(64)));
 int sum=0;
 #pragma omp parallel num_threads(NTHREAD)
 {
   int thread =omp_get_thread_num();
   A[thread][0] = 0;

   #pragma omp for schedule(static,1)
   for(int i=0; i<=n; i++)
     A[thread][0] += i % 10;

   #pragma omp atomic
   sum += A[thread][0];
 }
 printf("The sum is %d\n",sum);

}
```

Determining Properties of the Memory Subsystem

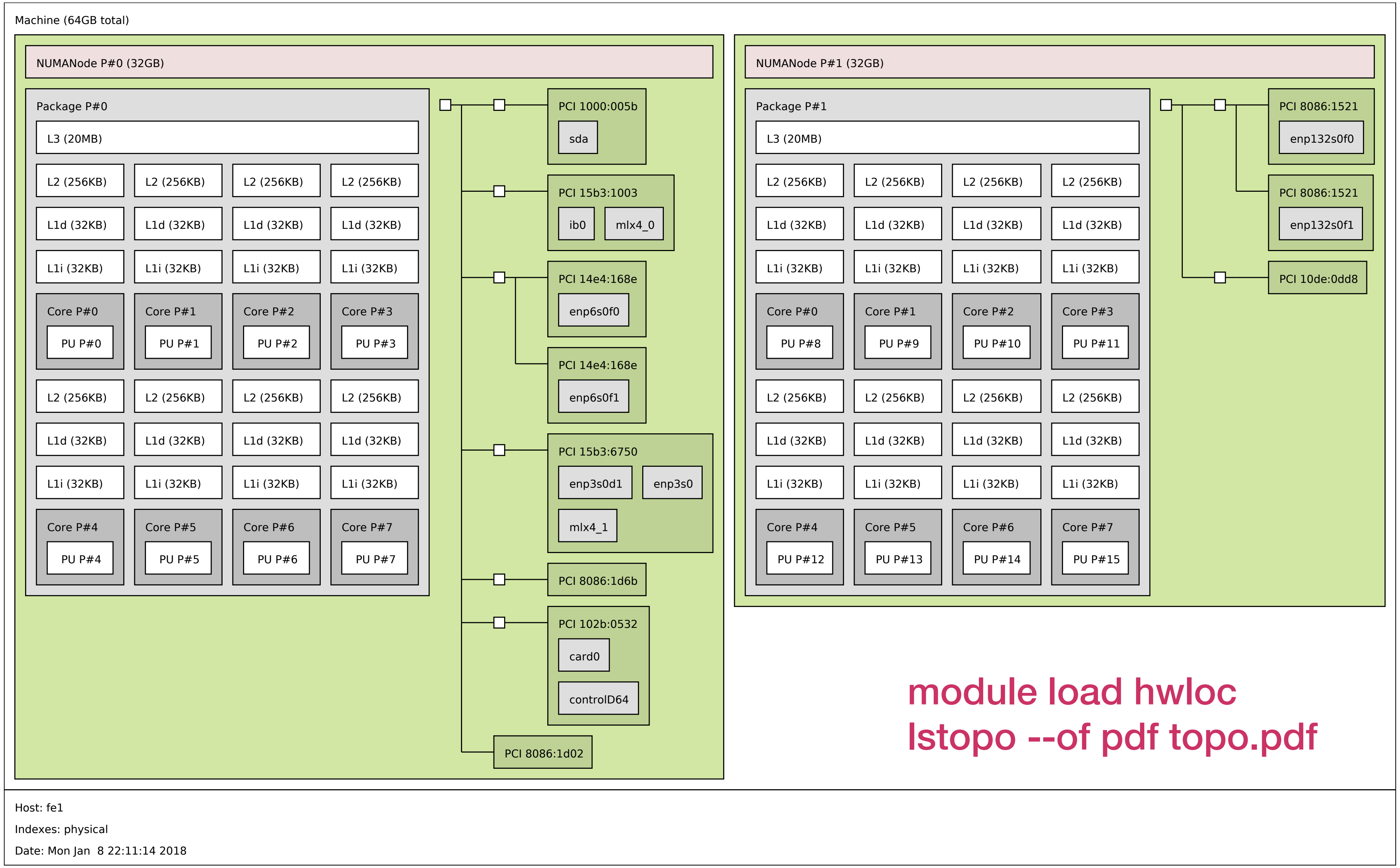
- Some useful tools for determining information about memory subsystem
- lstopo (part of hwloc tool)
  - NUMA architecture, cache architecture and I/O devices
  - textual or graphical output
- Intel cpuinfo (part of Intel MPI)
  - cache architecture, assignment of (hyper-)threads to cores
  - processor features, e.g. SIMD support, fused multiply add, …
- x86info (currently not installed on Oculus)
  - detailed cache architecture information
- Linux sysfs
  - read from sysfs directory /sys/devices/system/cpu/cpu0/cache/index1/..
  - detailed cache architecture, e.g. line size, number of sets, associativity, etc.

lstopo (Textual)

```
Machine (64GB total)
 NUMANode L#0 (P#0 32GB)
   Package L#0 + L3 L#0 (20MB)
     L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
     L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
     L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
     L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
     L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4 + PU L#4 (P#4)
     L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5 + PU L#5 (P#5)
     L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6 + PU L#6 (P#6)
     L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7 + PU L#7 (P#7)
   HostBridge L#0
     PCIBridge
       PCI 1000:005b
         Block(Disk) L#0 "sda"
     PCIBridge
       PCI 15b3:1003
         Net L#1 "ib0"
         OpenFabrics L#2 "mlx4_0"
     PCIBridge
       PCI 14e4:168e
         Net L#3 "enp6s0f0"
       PCI 14e4:168e
         Net L#4 "enp6s0f1"
    ....
```

```
module load hwloc
lstopo --of console
```

lstopo (Graphical)



```
module load hwloc
lstopo --of pdf topo.pdf
```

cpuinfo

```
===== Processor composition =====
Processor name    : Intel(R) Xeon(R) E5-2670 0
Packages(sockets) : 2
Cores             : 16
Processors(CPUs) : 16
Cores per package : 8
Threads per core : 1

===== Processor identification =====
ProcessorThread Id.Core Id.Package Id.
0      0 0 0
1      0 1 0
2      0 2 0
3      0 3 0
4      0 4 0
5      0 5 0
6      0 6 0
7      0 7 0
8      0 0 1
9      0 1 1
10      0 2 1
11      0 3 1
12      0 4 1
13      0 5 1
14      0 6 1
15      0 7 1
```

```
===== Placement on packages =====
Package Id.Core Id.Processors
0 0,1,2,3,4,5,6,70,1,2,3,4,5,6,7
1 0,1,2,3,4,5,6,78,9,10,11,12,13,14,15

===== Cache sharing =====
CacheSizeProcessors
L132 KBno sharing
L2256 KBno sharing
L320 MB(0,1,2,3,4,5,6,7)(8,9,10,11,12,13,14,15)
```

```
module load intel/mpi
cpuinfo
```

cpuinfo (2)

```
=====  Processor Feature Flags  =====
 ______ __________ ________ _________ ________ _____ _____ ______ _____ _______ _________ _____ ______ ______
| SSE3 | PCLMULDQ | DTES64 | MONITOR | DS-CPL | VMX | SMX | EIST | TM2 | SSSE3 | CNXT-ID | FMA | CX16 | xTPR |
|______|__________|________|_________|________|_____|_____|______|_____|_______|_________|_____|______|______|
| 1    | 1        | 1      | 1       | 1      | 1   | 1   | 1    | 1   | 1     | 0       | 0   | 1    | 1    |
|______|__________|________|_________|________|_____|_____|______|_____|_______|_________|_____|______|______|

 ______ ______ _____ ________ ________ ________ _______ ________ ______________ _____ _______ _________ _____ ______ ________
| PDCM | PCID | DCA | SSE4.1 | SSE4.2 | x2APIC | MOVBE | POPCNT | TSC-DEADLINE | AES | XSAVE | OSXSAVE | AVX | F16C | RDRAND |
|______|______|_____|________|________|________|_______|________|______________|_____|_______|_________|_____|______|________|
| 1    | 1    | 1   | 1      | 1      | 1      | 0     | 1      | 1            | 1   | 1     | 1       | 1   | 0    | 0      |
|______|______|_____|________|________|________|_______|________|______________|_____|_______|_________|_____|______|________|

 _____ _____ ____ _____ _____ _____ _____ _____ _____ ______ _____ ______ _____ _____ ______ _____ ________
| FPU | VME | DE | PSE | TSC | MSR | PAE | MCE | CX8 | APIC | SEP | MTRR | PGE | MCA | CMOV | PAT | PSE-36 |
|_____|_____|____|_____|_____|_____|_____|_____|_____|______|_____|______|_____|_____|______|_____|________|
| 1   | 1   | 1  | 1   | 1   | 1   | 1   | 1   | 1   | 1    | 1   | 1    | 1   | 1   | 1    | 1   | 1      |
|_____|_____|____|_____|_____|_____|_____|_____|_____|______|_____|______|_____|_____|______|_____|________|

 _____ _______ ____ ______ _____ ______ _____ ______ ____ _____ ____ _____
| PSN | CLFSH | DS | ACPI | MMX | FXSR | SSE | SSE2 | SS | HTT | TM | PBE |
|_____|_______|____|______|_____|______|_____|______|____|_____|____|_____|
| 0   | 1     | 1  | 1    | 1   | 1    | 1   | 1    | 1  | 1   | 1  | 1   |
|_____|_______|____|______|_____|______|_____|______|____|_____|____|_____|

 _________ ______ ______ ______ ______ ______ _________ _____ _____ _________ __________ ________ _____ ______
| FSGBASE | BMI1 | AVX2 | SMEP | BMI2 | ERMS | INVPCID | RTM | MPE | AVX512F | AVX512DQ | RDSEED | ADX | SMAP |
|_________|______|______|______|______|______|_________|_____|_____|_________|__________|________|_____|______|
| 0       | 0    | 0    | 0    | 0    | 0    | 0       | 0   | 0   | 0       | 0        | 0      | 0   | 0    |
|_________|______|______|______|______|______|_________|_____|_____|_________|__________|________|_____|______|

 ____________ _________ ____________ ______ _____ __________ __________ __________ _____ __________ __________
| AVX512IFMA | PCOMMIT | CLFLUSHOPT | CLWB | IPT | AVX512PF | AVX512ER | AVX512CD | SHA | AVX512BW | AVX512VL |
|____________|_________|____________|______|_____|__________|__________|__________|_____|__________|__________|
| 0          | 0       | 0          | 0    | 0   | 0        | 0        | 0        | 0   | 0        | 0        |
|____________|_________|____________|______|_____|__________|__________|__________|_____|__________|__________|

 _____________ ____________
| PREFETCHWT1 | AVX512VBMI |
|_____________|____________|
| 0           | 0          |
|_____________|____________|
```

module load intel/mpi
cpuinfo f

Acknowledgements

- These slides contain materials, examples and inspiration from numerous sources, which are hereby gratefully acknowledged:
  - Parallel Programming with OpenMP: an Introduction, Alejandro Duran, Barcelona Supercomputing Center (2009)
  - Introduction to OpenMP, EPCC
  - OpenMP Wikibooks
  - Tutorial: Mastering Tasking with OpenMP at SC'17
  - Tutorial: Advance OpenMP Programming at SC'17
  - OpenMP 4.5 official examples

Change Log

- 1.0.2 (2018-01-09)
  - fix typos: slide 33
  - clarification and cosmetics: slide 38-43
- 1.0.1 (2018-01-08)
  - finalized table of contents, added discussion of false sharing
  - improved discussion of ccNUMA
- 1.0.0 (2017-12-12)
  - initial version