Multicore Programming

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

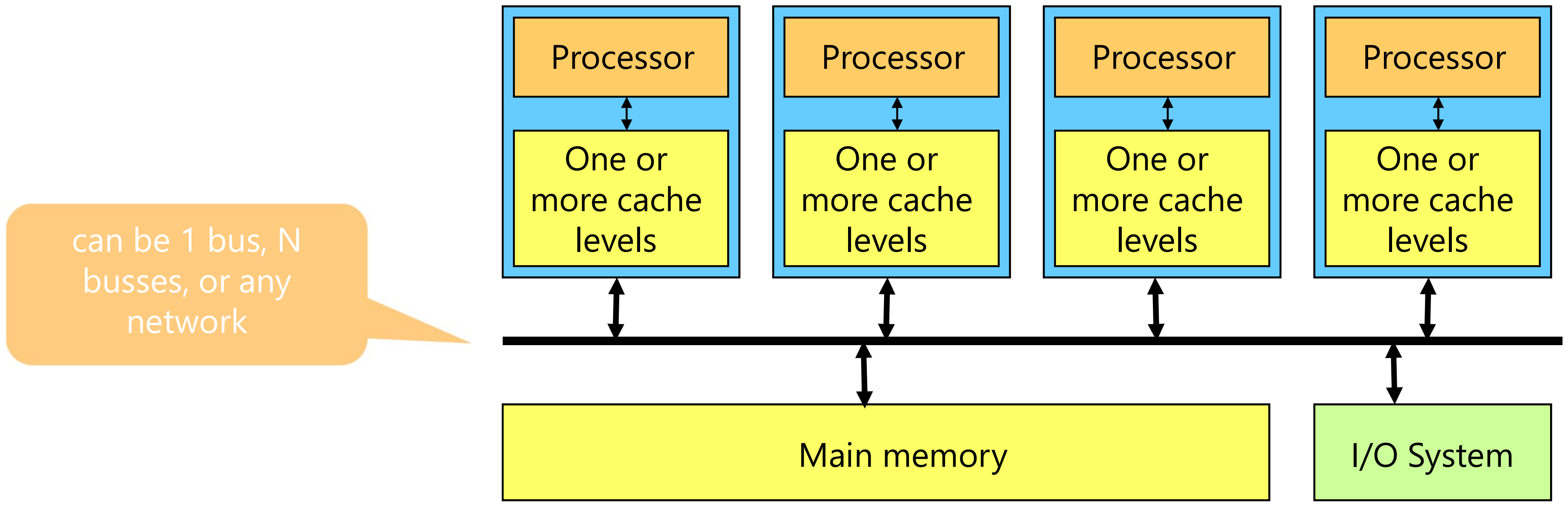
- Multiprocessors and multicore architectures
- Concurrency and parallelism
- OpenMP
  - Overview
  - Work sharing
  - Data scoping
- Parallel patterns
  - Map
  - Reduction

MIMD: Multiple Instruction Multiple Data

- Why is MIMD the choice for general-purpose multiprocessors?
  - Flexible
    - Can function as single-user machines focusing on high-performance for one application
    - Multi-programmed machine running many tasks simultaneously, or
    - Some combination of these two
  - Cost-effective: can use off-the-shelf processors
- Major MIMD Styles
  - Centralized shared memory
  - Distributed memory

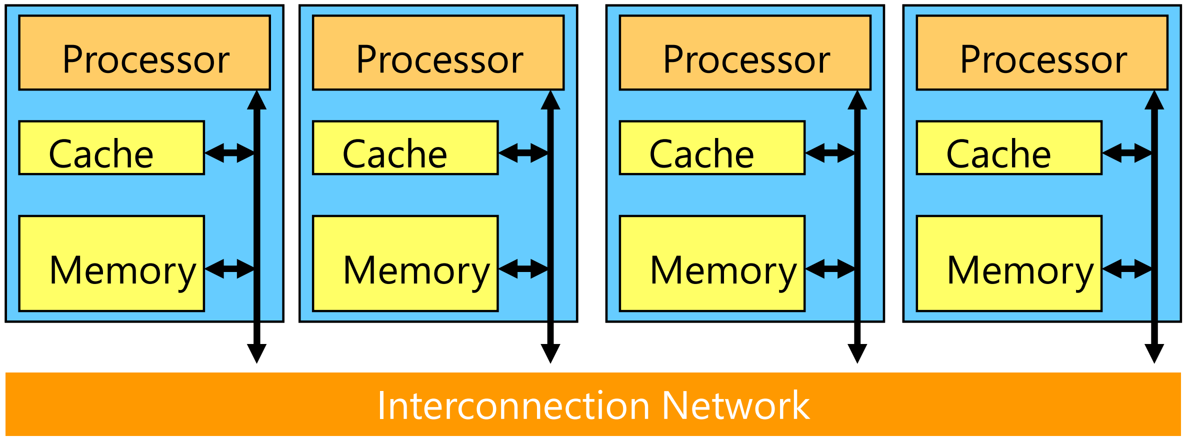
SMP: Symmetric Multi-Processor

- Memory: centralized with uniform access time (UMA) and bus interconnect, I/O
  - Examples: Sun Enterprise 6000, SGI Challenge, Intel



Distributed-Memory Multiprocessors

- Distributed-memory architectures
  - memory is physically distributed among the processors
  - typically have more processors than SMPs
  - believed to be more difficult to program than SMPs
  - require some kind of interconnect
    - direct (switches)
    - indirect (2- or higher dimensional meshes, hypercubes, fat trees, etc.)
  - also called Non-uniform Memory Access (NUMA) architectures



Communication Model

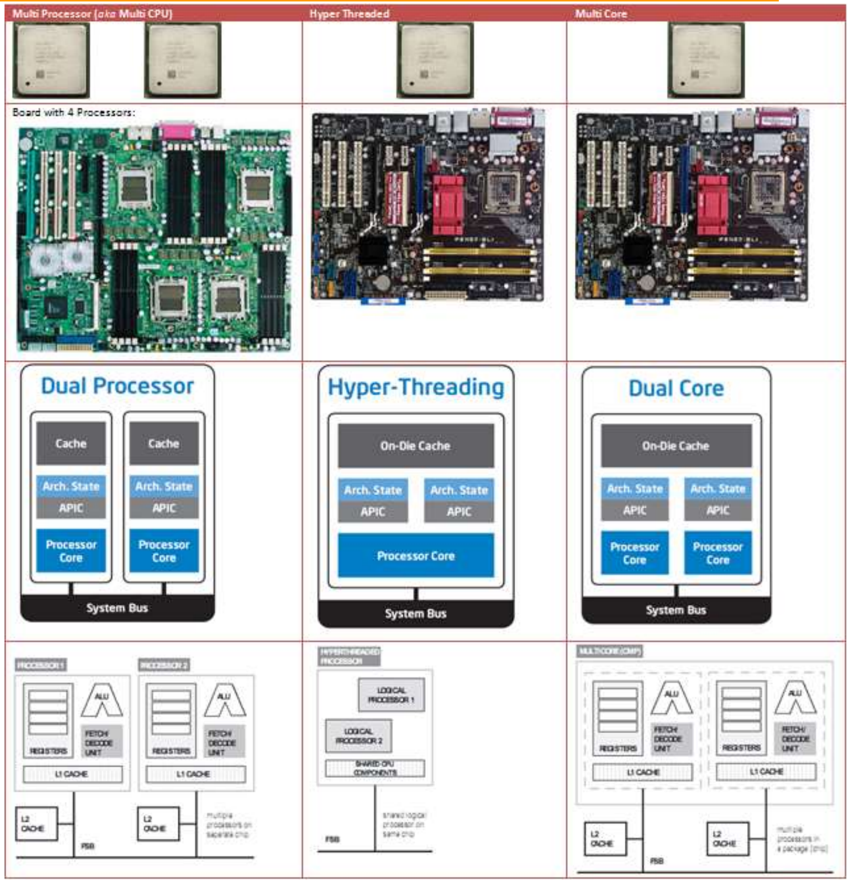
- Communication by explicitly passing messages among the processors: message-passing multiprocessors
  - e.g. clusters
- Communication through a shared address space (via loads and stores): shared memory multiprocessors either
  - UMA (Uniform Memory Access time) for shared address, centralized shared memory multiprocessor
  - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory multiprocessor

Thread-Level Parallelism

- To fully exploit an `n`-core MIMD multiprocessor, must have at least `n` threads or processes to execute
  - threads = processes running in the same address space
  - UNIX supports pthreads, Java also supports threads
- Independent threads identified by programmer or compiler
- Called thread-level parallelism (TLP), as opposed to ILP or DLP

Multiprocessor vs Multicore

- CPU is about I and D caches, decoders, execution units
- Multiprocessor
  - more than one such CPU, allowing them to work in parallel
  - Symmetric MultiProcessing (SMP)
- Multicore
  - subset of the CPU's components is duplicated, so that multiple cores can work in parallel on separate operations
  - Chip-level Multiprocessing (CMP)

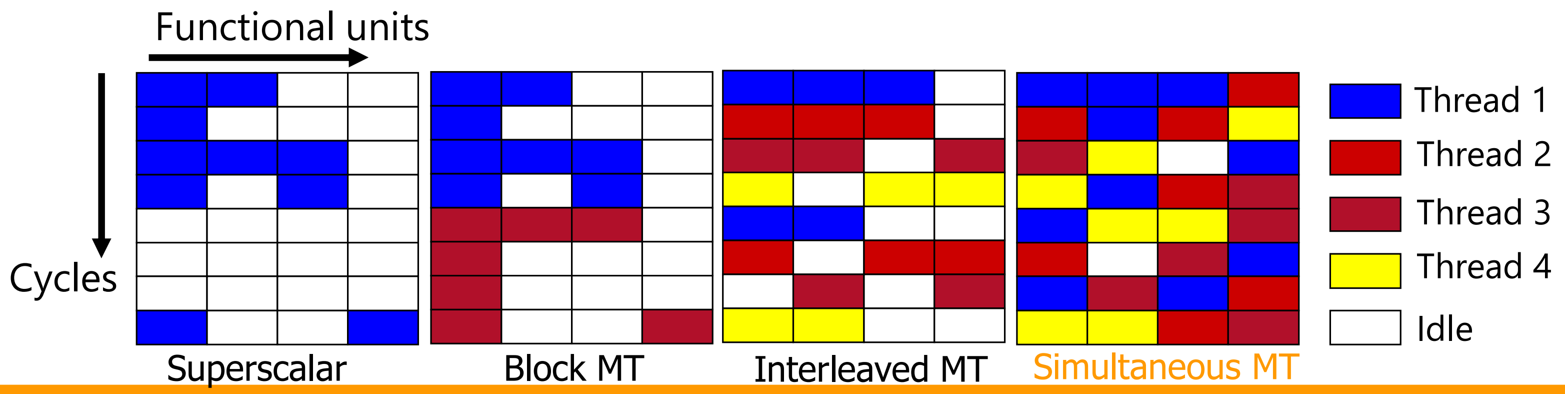


Hardware Multi-threading (1)

- When to switch (to which) thread?
- Cycle-by-cycle interleaving: fine-grained multi-threading
  - processor switches between software threads each cycle
  - round-robin among threads, skip when thread is stalled
- Blocking interleaving: course-grained multi-threading
  - processor switches to another thread when
    - a long latency operation stalls the current thread, e.g., L2 miss
    - max number of cycles/thread exceeded
- Simultaneous multi-threading (Intel's hyperthreading)
  - issue instructions from any thread every cycle
  - when one thread is stalled, other threads can continue

Hardware Multi-threading (2)

- Superscalar processor: high under-utilization
- Block MT (coarse-grain multithreading): switches threads only at e.g. L2 misses
  - Block MT doesn't perform well in OoO cores due to high cost of switching threads
- Interleaved MT (fine-grain multithreading): can only issue instructions from a single thread in a cycle
  - doesn't match OoO cores because instructions can stay in different pipeline stages for several cc
- Simultaneous multithreading: fetching, issuing and executing instructions from different threads in each cycle

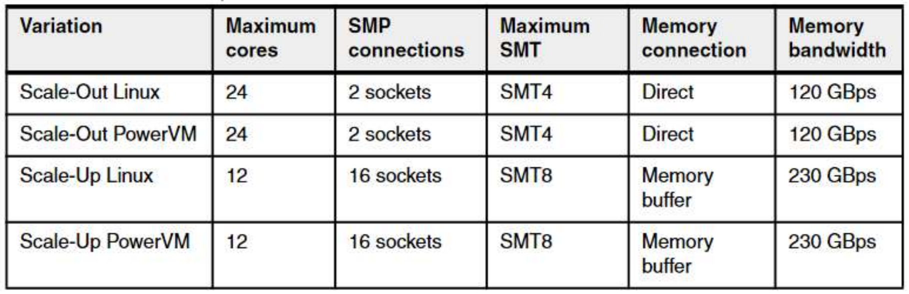


peak throughput = 4 IPC

Simultaneous Multithreading (SMT)

- with Simultaneous Multithreading (SMT)
  - smaller subset of a processor's or core's components is duplicated
    - duplicate thread scheduling resources
  - the core looks like two separate “virtual core" to the operating system
    - even though it only has one set of execution units
  - implementations
    - Intel's Hyperthreading is a 2-way SMT
    - Intel Xeon Phi has 4-way SMT (with time-multiplexed multithreading)
    - IBM's Blue Gene/Q has 4-way SMT
    - IBM POWER8 and POWER9 have up to 8 simultaneous threads per core (SMT8)

Example with POWER9 chip variations



Variation	Maximum cores	SMP connections	Maximum SMT	Memory connection	Memory bandwidth
Scale-Out Linux	24	2 sockets	SMT4	Direct	120 GBps
Scale-Out PowerVM	24	2 sockets	SMT4	Direct	120 GBps
Scale-Up Linux	12	16 sockets	SMT8	Memory buffer	230 GBps
Scale-Up PowerVM	12	16 sockets	SMT8	Memory buffer	230 GBps

Parallelism and Concurrency

- Concurrency: multiple tasks to be executed in a computer
  - tasks can execute at the same time (concurrent execution)
  - implies that there are no dependencies between the tasks
    - synchronizations is used to satisfy dependencies
  - concurrent is not the same as parallel!
- Parallel execution
  1. concurrent tasks actually execute at the same time
  2. multiple (processing) resources have to be available
- Parallelism = concurrency + "parallel" hardware
  - find concurrent execution opportunities
  - develop application to execute in parallel
  - run application on parallel hardware

Parallelism

- There are granularities of parallelism (parallel execution) in programs
  - processes, threads, routines, statements, instructions, …
  - think about what are the software elements that execute concurrently
- These must be supported by hardware resources
  - processors, cores, … (execution of instructions)
  - memory, DMA, networks, … (other associated operations)
  - all aspects of computer architecture offer opportunities for parallel hardware execution
- Concurrency is a necessary condition for parallelism
  - Where can you find concurrency?
  - How is concurrency expressed to exploit parallel systems?

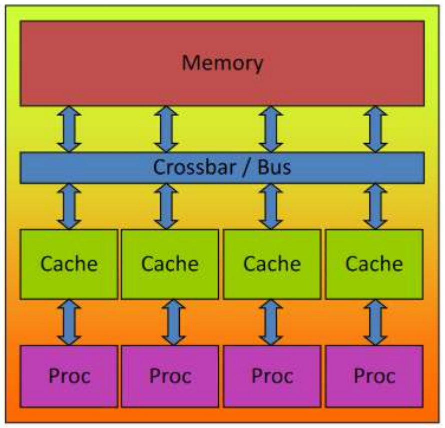
OpenMP Overview

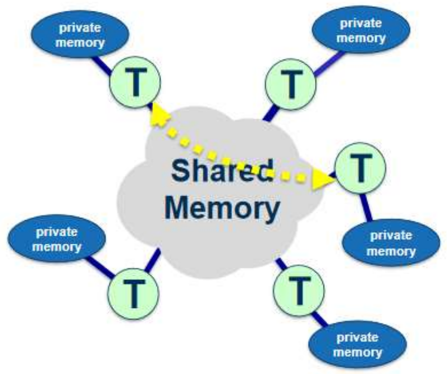
- De facto standard for shared-memory parallelization
  - development driven by the OpenMP Architecture Review Board (ARB)
  - specification, free to implement (“Open”)
- Compiler directives, runtime routines and environment variables
  - several ways to express parallelism
- Timeline

1997: OpenMP 1.0 for Fortran	2008: OpenMP 3.0 – first release with task parallelism
1998: OpenMP 1.0 for C and C++	2011: OpenMP 3.1
1999: OpenMP 1.1 for Fortran	2013: OpenMP 4.0 – support for accelerator programming
2000: OpenMP 2.0 for Fortran	2015: OpenMP 4.5
2002: OpenMP 2.0 for C and C++	2018: OpenMP 5.0 – support for tool interface
2005: OpenMP 2.5 for Fortran, C and C++	

OpenMP Machine and Memory Model

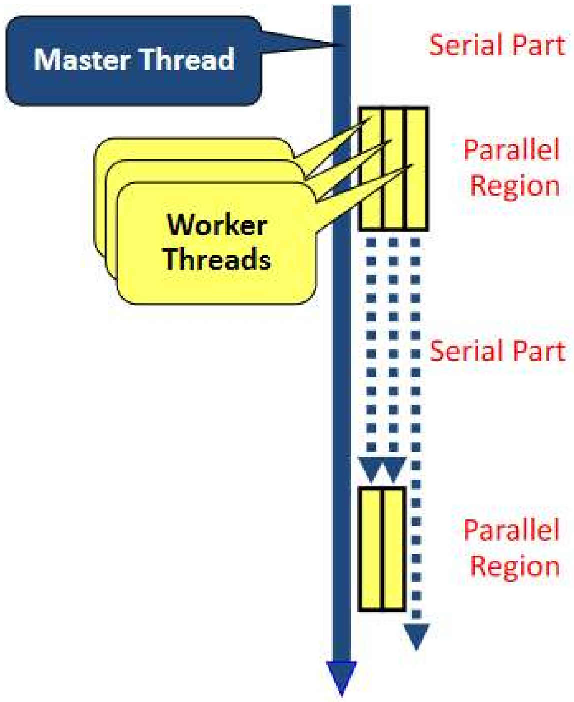
- Machine model
  - no explicit machine model in the standard
  - all processors/cores access a shared main memory
    - real architectures are more complex
  - parallelism expressed as multiple threads
- Memory model
  - all threads have access to the same, globally shared memory
  - data in private memory is only accessible by the thread owning this memory
  - no other thread sees the change(s) in private memory
  - data transfer is through shared memory and is 100% transparent to the application





OpenMP Execution Model

- Concept: Fork-Join
- OpenMP programs start with just one thread: the master thread or initial thread
- Worker threads are spawn at Parallel Regions
  - together with the Master they form the Team of threads
- In between Parallel Regions the Worker threads are put to sleep
  - the OpenMP Runtime takes care of all thread management work
- Designed for allowing incremental parallelization



OpenMP Parallel Region and Structured Blocks

- Parallelism is explicit
- Structured Block
  - exactly one entry point at the top
  - exactly one exit point at the bottom
  - branching in or out is not allowed
  - terminating the program is allowed (abort / exit)

```
#pragma omp parallel
{
   ...
   structured block
   ...
}
```

OpenMP Number of Threads

- Specification of number of threads
  - environment variable: OMP_NUM_THREADS=…
  - by adding `num_threads(num)` to the parallel construct
- In Linux: from within a shell, global setting of the number of threads:
  - export OMP_NUM_THREADS=4
  - ./program
- In Linux: from within a shell, one-time setting of the number of threads:
  - OMP_NUM_THREADS=4 ./program

Lab Exercise

- First OpenMP Hello World program

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  #pragma omp parallel
  {
    printf("Hello World... from thread = %d\n", omp_get_thread_num());
  }
  // end of parallel region
}
```

- Compile with:

```
> g++ HelloWorld.cpp -fopenmp
```

Runtime Library

- C and C++
  - if OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined
    - to use the OpenMP runtime library, the header `omp.h` has to be included
  - `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next
  - `int omp_get_num_threads`: Returns the number of threads in the current team
  - `int omp_get_thread_num()`: Returns the number of the calling thread in the team
    - the master has always the id `0`
- Additional functions are available, e.g., to provide locking functionality

OpenMP Work Sharing on Loops

- OpenMP's most common work sharing construct: for
  - loops often account for most of a program's runtime!
- If only the parallel construct is used, each thread executes the structured block
  - distribution of loop iterations over all threads in a team
  - scheduling of the distribution can be influenced

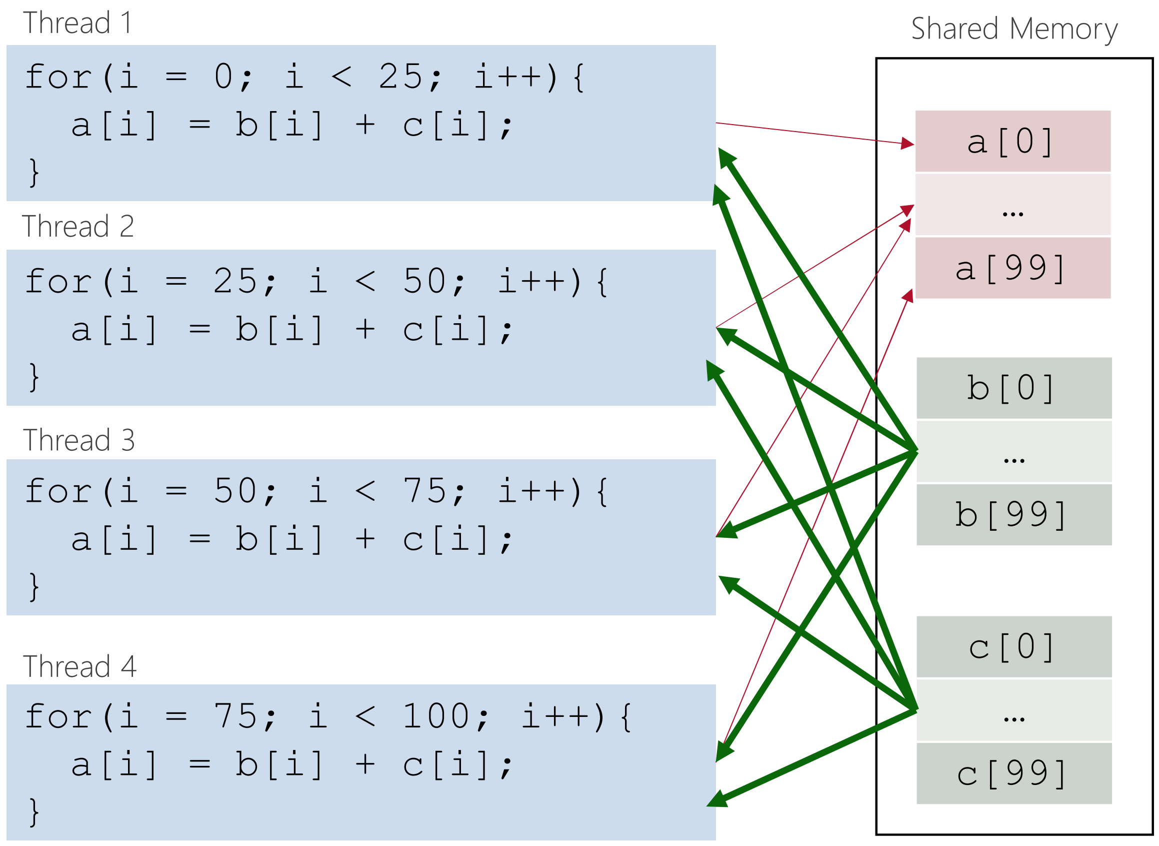
```
int i;
#pragma omp for
for(i = 0; i < 100; i++){
  a[i] = b[i] + c[i];
}
```

Work Sharing Example with 4 Threads

- Simplified example

Serial

```
for(i = 0; i < 100; i++){
  a[i] = b[i] + c[i];
}
```



Thread 1

```
for(i = 0; i < 25; i++){
  a[i] = b[i] + c[i];
}
```

Thread 2

```
for(i = 25; i < 50; i++){
  a[i] = b[i] + c[i];
}
```

Thread 3

```
for(i = 50; i < 75; i++){
  a[i] = b[i] + c[i];
}
```

Thread 4

```
for(i = 75; i < 100; i++){
  a[i] = b[i] + c[i];
}
```

Vector Addition

```
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[]) {
  const int size = 512;
  double A[size];
  double B[size];
  double C[size];

#pragma omp parallel for
  for (int i = 0; i < size; i++) {
    A[i] = 1.0;
    B[i] = 2.0;
    C[i] = 3.0;
  }

  double t_init = omp_get_wtime();
```

```
#pragma omp parallel
  {
#pragma omp single
    {
      cout << "Th.num: "<<omp_get_num_threads()<<endl;
      fflush(stdout);
    }
#pragma omp for
    for (int i = 0; i < size; i++) {
      A[i] = B[i] + C[i];
    }
  }

  // print results
  for (int i = 0; i < size; i++) {
    cout << C[i] << ' ';
  }
  cout << "Time:" << omp_get_wtime()-t_init << endl;
  return 0;
}
```

What is a `parallel for`?

- OpenMP parallel for are a combined construct, which is semantically equivalent to

```
int i, int s = 0;
#pragma omp parallel for
for(i = 0; i < 100; i++){
  a[i] = b[i] + c[i];
}
```

```
int i, int s = 0;
#pragma omp parallel
#pragma omp for
for(i = 0; i < 100; i++){
  a[i] = b[i] + c[i];
}
```

- **`#pragma omp parallel`**
  - Creates a parallel region
- **`#pragma omp for`**
  - Thread scheduling

OpenMP For Loop Scheduling

- For loop scheduling: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the schedule clause
  - `schedule(static[, chunk])`
    - iteration space divided into blocks of chunk size; blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threadsblocks
  - `schedule(dynamic[, chunk])`
    - iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks
  - `schedule(guided[, chunk])`
    - similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk
- Default on most implementations is `schedule(static)`

Synchronization Overview

- Can we parallelize all loop with for?

```
int s = 0;
#pragma omp parallel for
for(int i = 0; i < 100; i++){
  s = s + a[i];
}
```

- Race condition
  - if between two synchronization points at least one thread writes to a memory location from which at least one other thread reads

Critical Region

- A critical region is executed by all threads, but by only one thread simultaneously
  - mutual exclusion: no two threads can execute a critical construct of the same name at the same time
  - may optionally contain a global name that identifies the critical construct
  - if the name is omitted, a default name is assumed
  - critical section names are global to the entire program (regardless of module boundaries)

```
#pragma omp critical (name)
{
   …structured block…
}
```

```
#pragma omp critical(dataupdate)
{
  datastructure.reorganize();
}
 ...
#pragma omp critical(dataupdate)
{
  datastructure.reorganize_again();
}
```

Critical Region

- What about this example?

```
int i, int s = 0;
#pragma omp parallel for
for(i = 0; i < 100; i++)
{
  #pragma omp critical
  {  s = s + a[i];   }
}
```

Barrier

- Barrier
  - threads wait until all threads of the current team have reached the barrier
- In OpenMP, barrier are either implicit or explicit
  - Explicit:

```
#pragma omp barrier
```

  - Implicit: all work sharing constructs contain an implicit barrier at the end

```
#pragma omp for
{
   …
}
```

```
#pragma omp sections
{
   …
}
```

```
#pragma omp single
{
   …
}
```

Single Construct

- Only one thread in the team executes the block
  - the single construct specifies that the enclosed structured block is executed by only one thread of the team
  - it is up to the runtime which thread that is
  - implicit barrier at the end, unless `nowait` is used

```
#pragma omp single [clause]
{
   …
}
```

- Examples
  - I/O
  - memory allocation and deallocation
  - implementation of the single-creator parallel-executor patterns

Master Construct

- The master construct specifies that the enclosed structured block is executed only by the master thread of a team
- Note: The master construct is not a work sharing construct
  - does not contain an implicit barrier at the end

```
#pragma omp master [clause]
{
    …
}
```

Lab exercise

- Parallelize the following code in OpenMP, report on running time

```
int i, int s = 0;
for (i = 1; i < 100; i++){
  s = a[i-1] + a[i];
}
```

Scoping

- Shared variable
  - one instance of a variable is shared between multiple threads
  - OpenMP does not put any restriction to prevent data races between shared variables
    - this is a responsibility of a programmer
  - shared variables may introduce an overhead, often is best to minimize the number of shared variables when a good performance is desired
- Private variable
  - when a variable is declared private, OpenMP replicates this variable and assigns its local copy to each thread

Scoping Rules

- Scoping in OpenMP: Dividing variables in shared and private
  - `private-list` and `shared-list` on parallel region
  - `private-list` and `shared-list` on work sharing constructs
  - general default is `shared` for parallel region, `firstprivate` for tasks
  - loop control variables on for-constructs are `private`
  - non-static variables local to parallel regions are `private`
  - `private`: a new uninitialized instance is created for the task or each thread executing the construct
    - `firstprivate`: initialization with the value before encountering the construct
    - `lastprivate`: value of last loop iteration is written back to master
  - static variables are `shared`

Privatization of Global/Static Variables

- Global / static variables can be privatized with the `threadprivate` directive
- One instance is created for each thread
  - before the first parallel region is encountered
  - instance exists until the program ends
  - does not work (well) with nested parallel region

```
static int i;
#pragma omp threadprivate(i)
```

- Based on thread-local storage (TLS)
  - `TlsAlloc` (Win32-Threads)
  - `pthread_key_create` (Posix-Threads)
  - keyword `__thread` (GNU extension)

Back to our example

- Correct, but does not scale
  - Why?
  - What can we do?

```
int i, int s = 0;
#pragma omp parallel for
for(i = 0; i < 100; i++)
{
  #pragma omp critical
  {  s = s + a[i];  }
}
```

- Exercise: Implement a scalable version of this code using partial sums

Improving the Scalability using Partial Sums

- What about this?

```
int i, int s = 0;
#pragma omp parallel
{
  int ps = 0;
  #pragma omp for
  for(i = 0; i < 100; i++){
    ps = ps + a[i];
  }
  s = s + ps;
}
```

Reduction Clause

- In a reduction operation the operator is applied to all variables in the list
  - the variables have to be `shared`.
  - `reduction (operator:list)`
  - the result is provided in the associated reduction variable

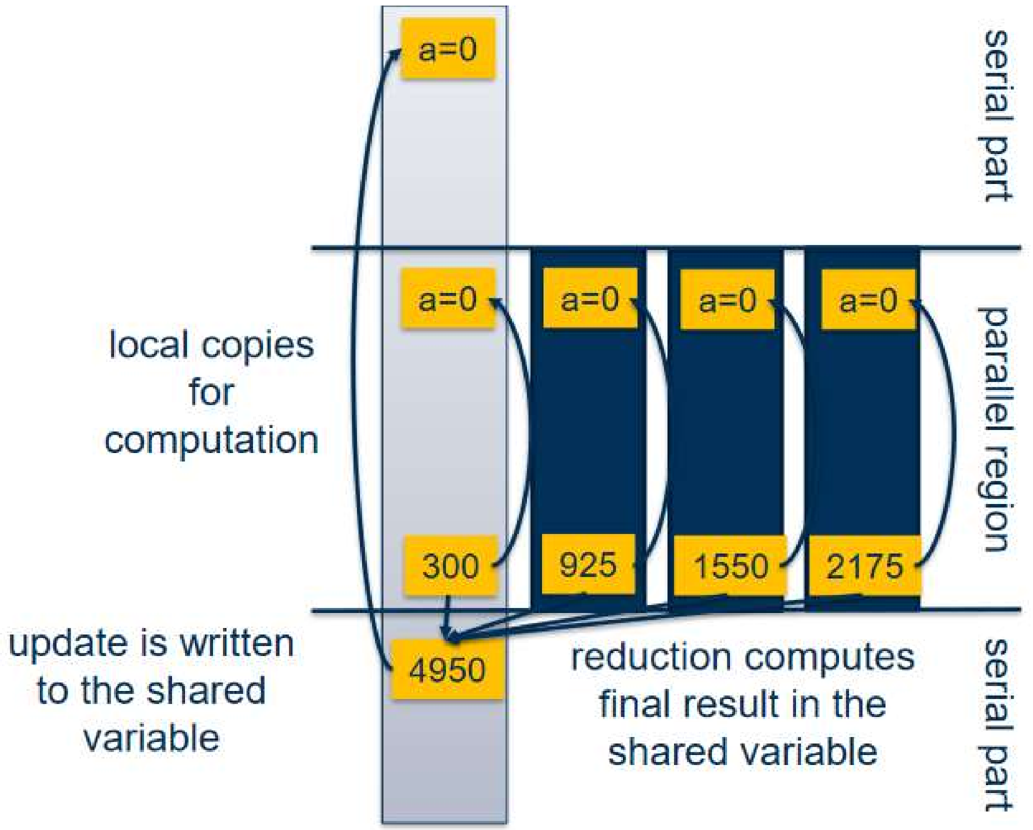
```
int i, int s = 0;
#pragma omp parallel for reduction (+:s)
for(i = 0; i < 100; i++){
  s = s + a[i];
}
```

  - Possible reduction operators with initialization value:

`+ (0), * (1), -(0), & (~0), | (0), && (1), || (0), ^ (0),`
`min (largest number), max (least number)`

Reduction Operations

```
int a = 0;
#pragma omp parallel
#pragma omp for reduction (+:a)
for(i = 0; i < 100; i++){
  a+=i;
}
```


a=0
serial part
local copies for computation
a=0
a=0
a=0
a=0
parallel region
300
925
1550
2175
update is written to the shared variable
4950
reduction computes final result in the shared variable
serial part

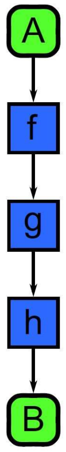
Parallel Patterns

- Parallel patterns: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design
  - patterns provide us with a “vocabulary” for algorithm design
  - it can be useful to compare parallel patterns with serial patterns
  - patterns are universal and recognized by several parallel programming models

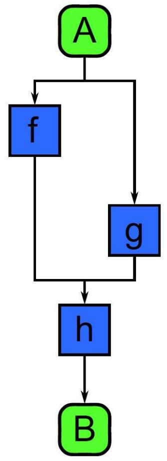
Sequential Patterns

- Sequence: ordered list of tasks that are executed in a specific order
  - assumption – program text ordering will be followed (obvious, but this will be important when parallelized)

```
T = f(A);
S = g(T);
B = h(S);
```

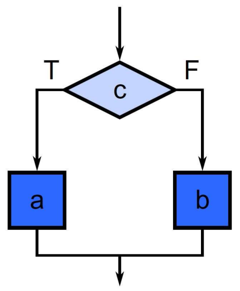


```
T = f(A);
S = g(A);
B = h(S,T
```



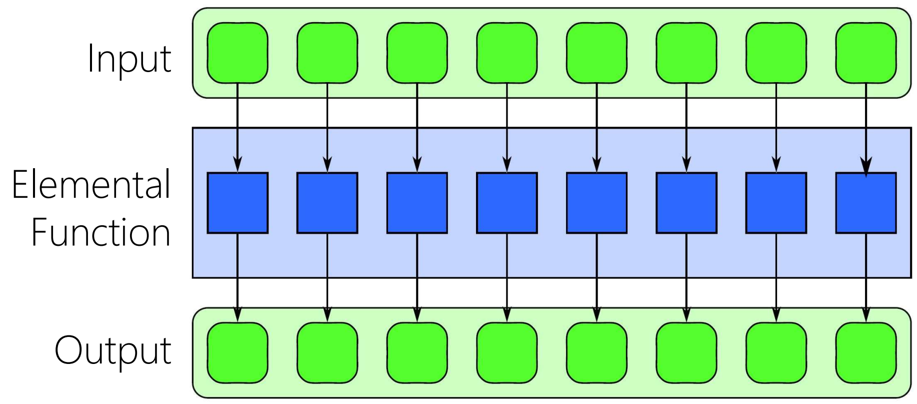
- Selection: condition `c` is first evaluated. Either task `a` or `b` is executed depending on the true or false result of `c`
  - assumptions – `a` and `b` are never executed before `c`, and only `a` or `b` is executed - never both

```
if (c) {
  a;
} else {
  b;
}
```



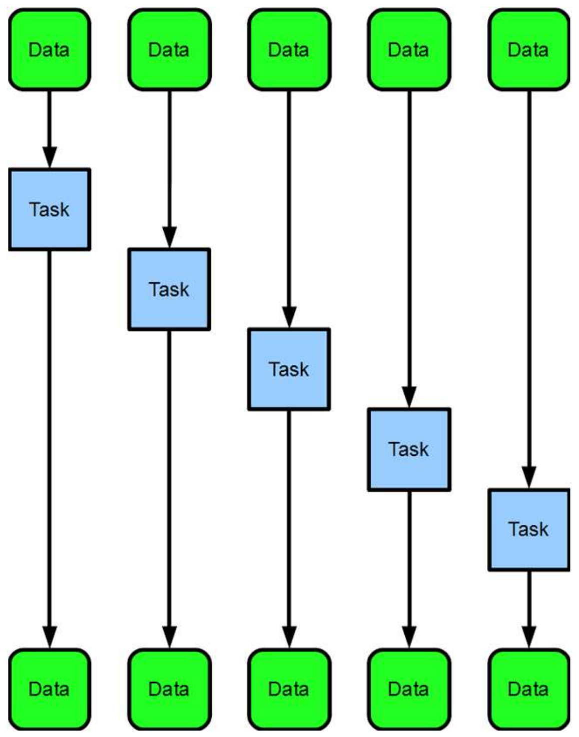
Parallel Patterns: Map

- Map: performs a function over every element of a collection
    - Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
        - an operation is a map if it can be applied to each element without knowledge of neighbors.
    - The replicated function is referred to as an "elemental function"
- Map is a "foreach loop"
    - where each iteration is independent
- Embarrassingly parallel computations

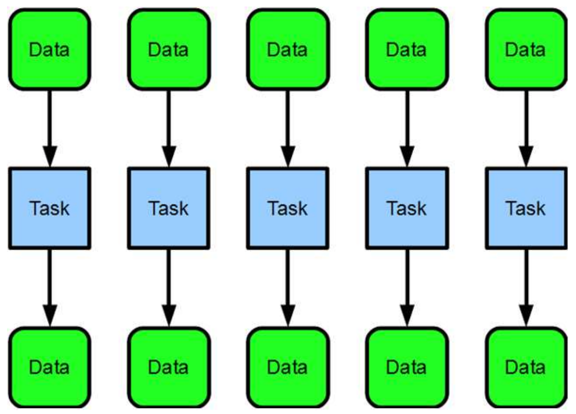


Parallel Patterns: Map

Serial Map



Parallel Map

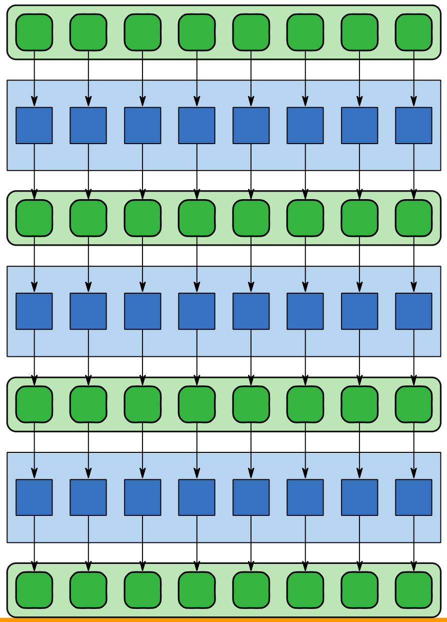


```
for(i = 0; i < 100; i++)
{
  compute(A[i]);
}
```

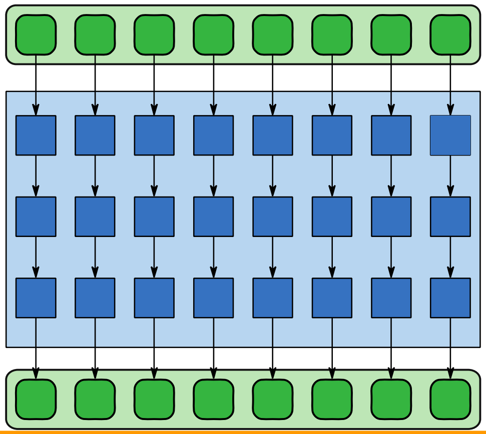
```
#pragma omp parallel for
for(i = 0; i < 100; i++)
{
  compute(A[i]);
}
```

Parallel Patterns: Map Optimizations

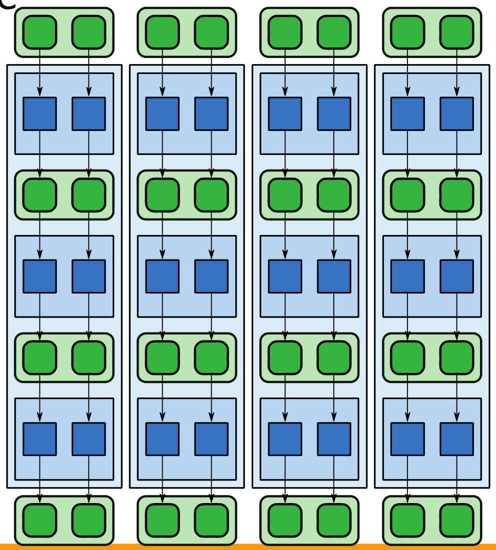
- Sequence of Maps
  - non optimized



- Code Fusion
  - adds arithmetic intensity
  - reduces memory/cache usage



- Cache Fusion
  - break the work into blocks, give each CPU one block at a time
  - hopefully, operations use cache alone



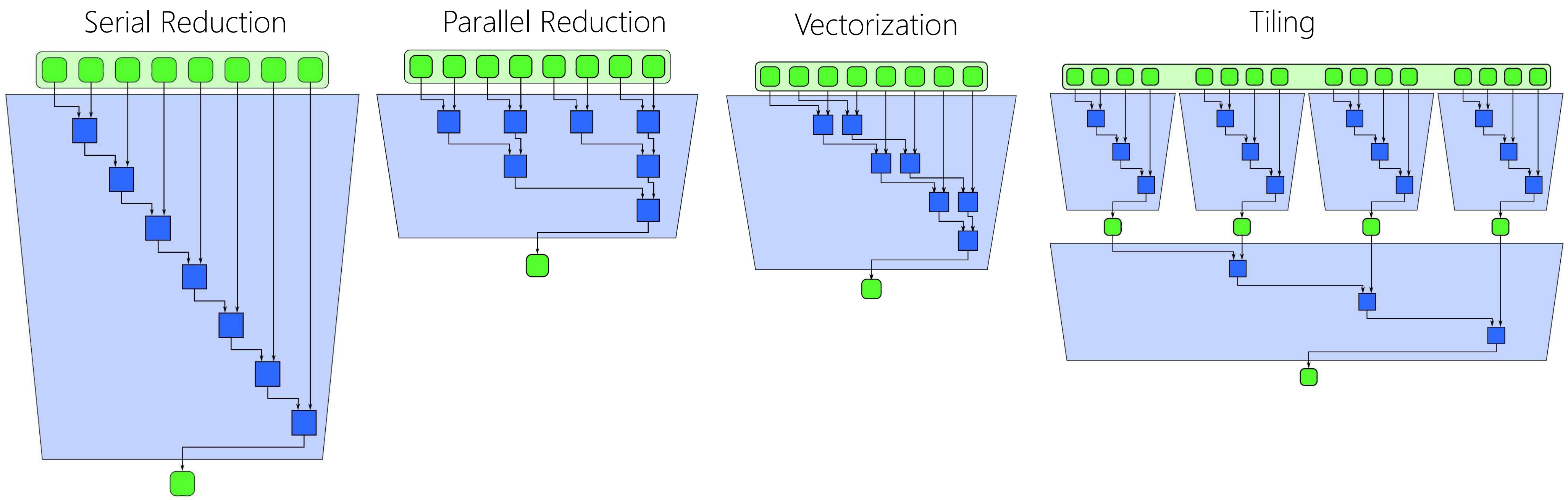
Parallel Patterns: Collective Patterns

- Collective operations deal with a collection of data as a whole, rather than as separate elements
- Collective patterns include
  - Reduce
  - Scan
  - Partition
  - Scatter
  - Gather

Parallel Patterns: Reduction

- Reduction: Combines every element in a collection using an associative "combiner function"
  - combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be associative to be parallelizable
  - because of the associativity of the combiner function, different orderings of the reduction are possible
- Example of combiner functions: addition, multiplication, max, min, and, or, xor
- Warning: float point addition and multiplication are approximately associative
  - different orderings of floating-point data can change the reduction value

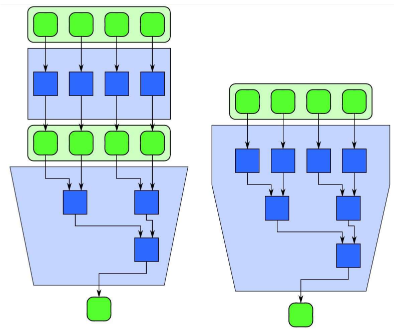
Reduce Pattern and Optimizations



Lab Exercises

1. Parallelize a dot product
   - Input of same length
   - map (*) and reduce (+)
   - can potentially fuse the two patterns

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i.$$



2. Parallelize the following loop
   - with a partial summations
   - with a reduction
   - apply the optimizations seen in the previous lecture

```
int s = 0;
for(int i = 0; i < 100; i++){
  s = s + a[i];
}
```

Note: Report on running time and multithreading scaling for all versions

Caches and False Sharing

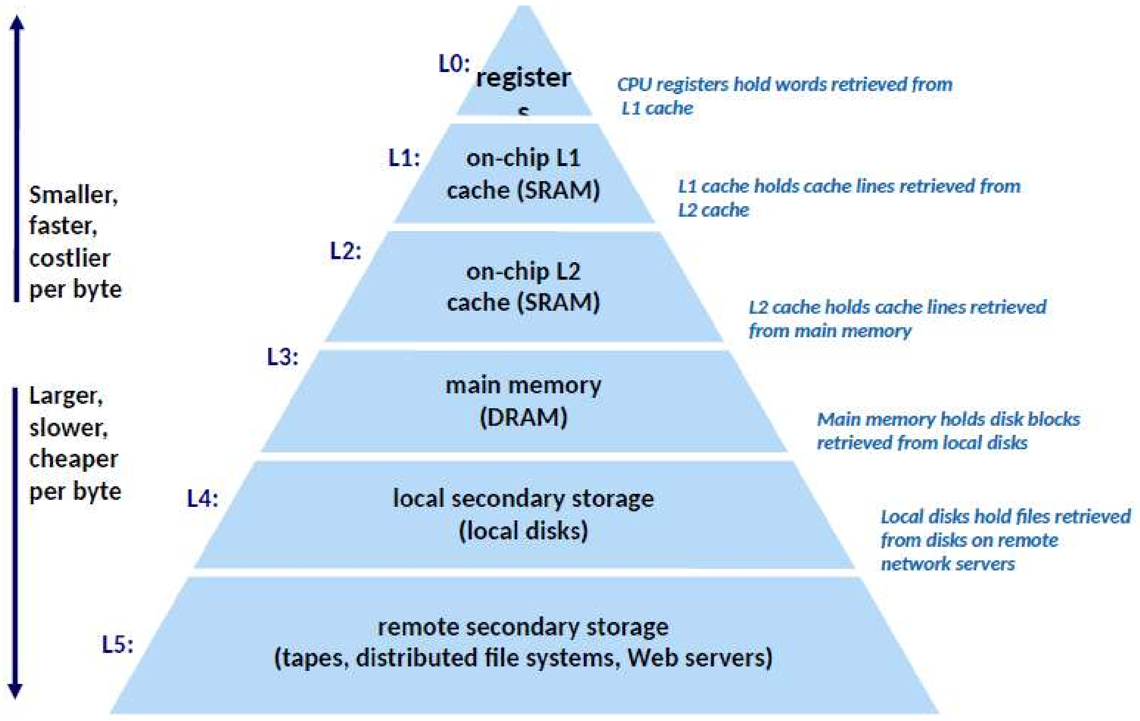
High Performance Computing, Summer 2021

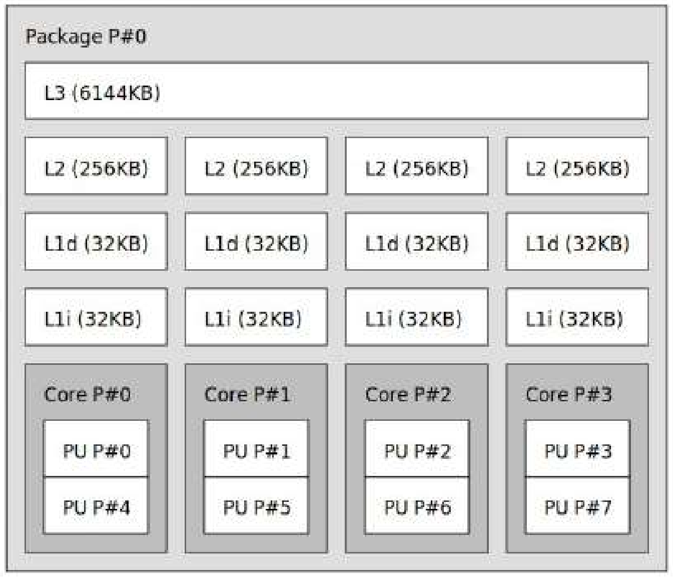
Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Cache
  - Cache coherence
  - Snooping protocols
- OpenMP: False sharing

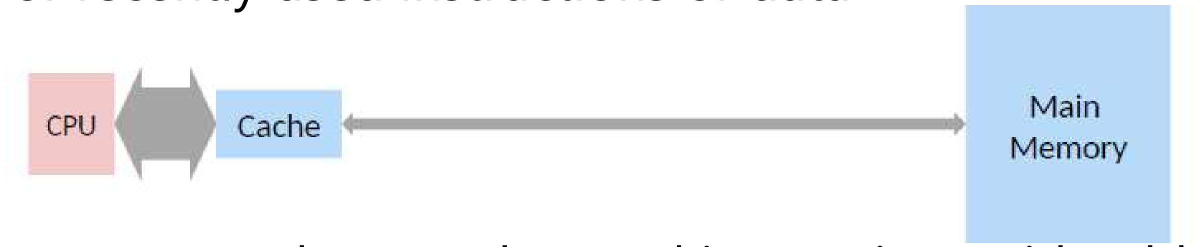
Memory Hierarchy


Smaller,
faster,
costlier
per byte
Larger,
slower,
cheaper
per byte
L0: registers
CPU registers hold words retrieved from L1 cache
L1: on-chip L1 cache (SRAM)
L1 cache holds cache lines retrieved from L2 cache
L2: on-chip L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory
L3: main memory (DRAM)
Main memory holds disk blocks retrieved from local disks
L4: local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers
L5: remote secondary storage (tapes, distributed file systems, Web servers)



Cache & Locality

- Cache: computer memory with short access time used for the storage of frequently or recently used instructions or data



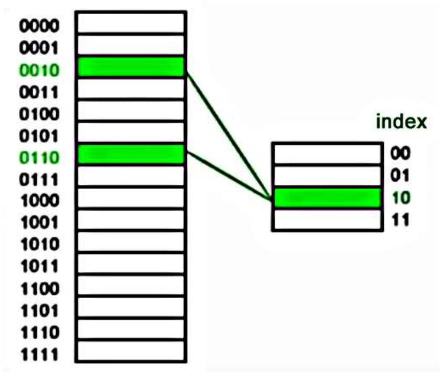
- Locality: programs tend to use data and instructions with addresses near or equal to those they have used recently

1. Temporal locality
   - recently referenced items are likely to be referenced again in the near future
2. Spatial locality
   - items with nearby addresses tend to be referenced close together in time

Cache Mapping and Associativity

- Map memory locations to cache

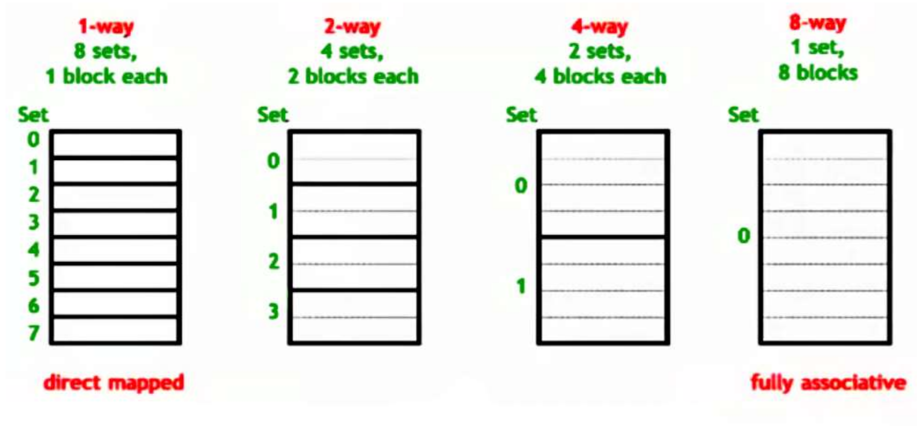
1. Direct mapped cache
   - means every block from memory has a unique location in cache
     - multiple sets with a single cache line per set
   - efficient, but lower cache hit rate
2. Fully associative cache
   - a memory block can occupy any of the cache lines
     - single cache set with multiple cache lines
   - one can view the register file as a fully associative cache
   - better cache hit rate , but too expensive to build, placement policy is slow



Cache Mapping and Associativity

3. Set associative cache

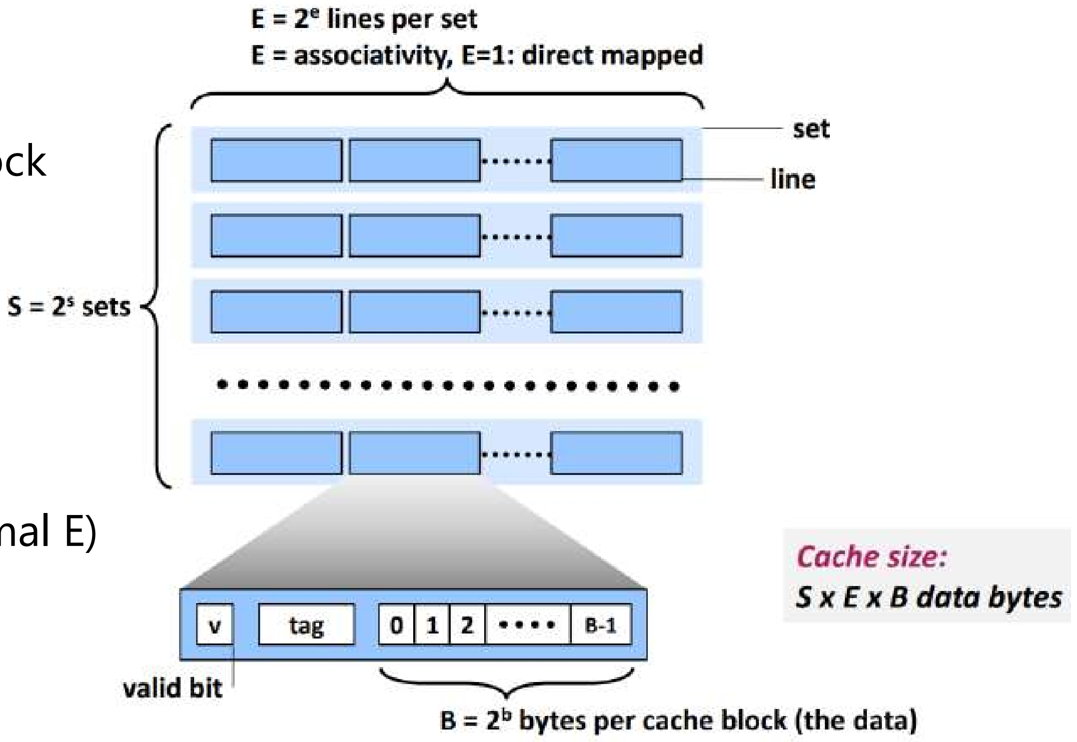
- the cache is divided into `S` sets and each set contains `E` cache lines
  - a memory block is first mapped onto a set and then placed into any cache line of the set
- trade-off between direct-mapped and fully associative cache
- four-way set-associative very common in modern processors



- Block replacement
  - LRU (least recently used) when selecting which block should be replaced

Cache Organization

- Three important values: S, E, B
  - B: cache block size
  - E: number of place in cache where a block can be mapped
  - S: number of set
- Cache size: S x E x B data bytes
  - Direct mapped cache: E = 1
  - Fully associative cache: S = 1 (i.e., maximal E)



Types of Cache Misses (The 3 Cs)

- Compulsory (cold) miss
  - occurs on first access to a block
- Capacity miss
  - occurs when working set is larger than the cache
- Conflict miss
  - conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

Caching with Writes

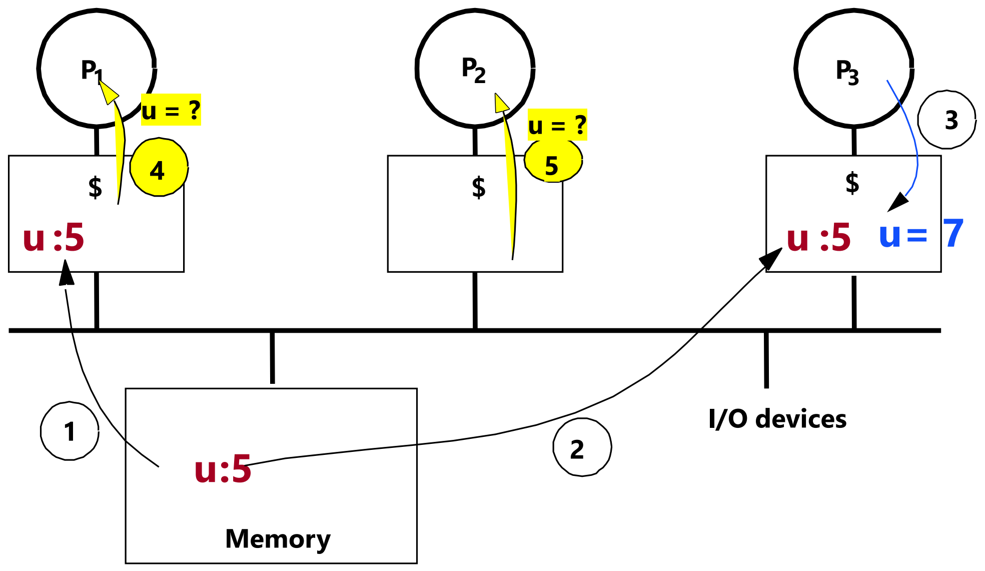
- What to do on a write-hit?
  - Write-through: write immediately to memory
    - data is written to the cache and memory, and the write finishes when both have finished
  - Write-back: defer write to memory until replacement of line
    - data is written to a block in the cache
    - cache block is only written to memory when it is replaced (in effect, a lazy write)
      - dirty bit for each block
- What to do on a write-miss?
  - Write-allocate (fetch on write): load into cache, update line in cache
  - No-write-allocate (write-no-allocate, write around): writes immediately to memory the backing store
    - only system reads are being cached

Cache and Parallelism

- From multiple boards on a shared bus to multiple core inside a single chip
- Caches both
  - private data used by a single processor
  - shared data used by multiple processors
- Caching shared data
  - reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  - introduces cache coherence problem

Cache Coherence Problem

- Processors see different values for `u` after event 3
- With write back caches
  - value written back to memory depends on which cache accidentally flushes or writes back value and when
  - threads accessing main memory may see very stale value



Example

Assume initial value of A and flag is 0

P_1

```
A = 1
flag = 1;
```

P_2

```
while (flag == 0); /* spin idly */
print A;
```

- If P_2 loads `flag` in its cache before P_1 sets it, P_2 spins forever (without cache coherence)
- Coherence not sufficient to guarantee intuition
  - pertains only to single location
  - expect memory to respect order between accesses to different locations issued by a given process
    - all reads and writes complete in order
    - in example above, write to `flag` may occur before write to `A`

Memory Model Issues

- Reading an address should return last value written to that address
  - easy in uniprocessors, except for I/O
- Too strict and too difficult to enforce. Two issues:

1. Coherence defines which values can be returned by a read
   - Coherence defines behavior to same location
2. Consistency determines when a written value will be returned by a read
   - Consistency defines behavior to other locations

Cache Coherent Protocols

- Snooping
  - before writing, send address to all processors
  - if read miss, also send address to all processors
  - processors snoop the bus to see if they have copy of the data and respond accordingly
    - send the cache tags
  - requires broadcast → works well with bus (natural broadcast medium)
  - dominates for small scale machines (most of the market)
- Directory-based
  - keep track of what is being shared in one centralized place
  - physically distributed memory → distributed directory for scalability
    - avoids bottlenecks, hot spots

Snooping Protocols

- Write invalidate
  - get exclusive access to cache block (invalidate all other copies) before writing

Processor	Bus activity	Cache A	Cache B	Memory
				X = 0
A reads X	Cache miss X	X = 0		X = 0
B reads X	Cache miss X	X = 0	X = 0	X = 0
A writes 1 to X	Invalidate X	X = 1		X = 0
B reads X	Cache miss X	X = 1	X = 1	X = 1

- Write update/broadcast
  - update all cached copies
  - to keep bandwidth requirements under control, need to track whether words are shared or private

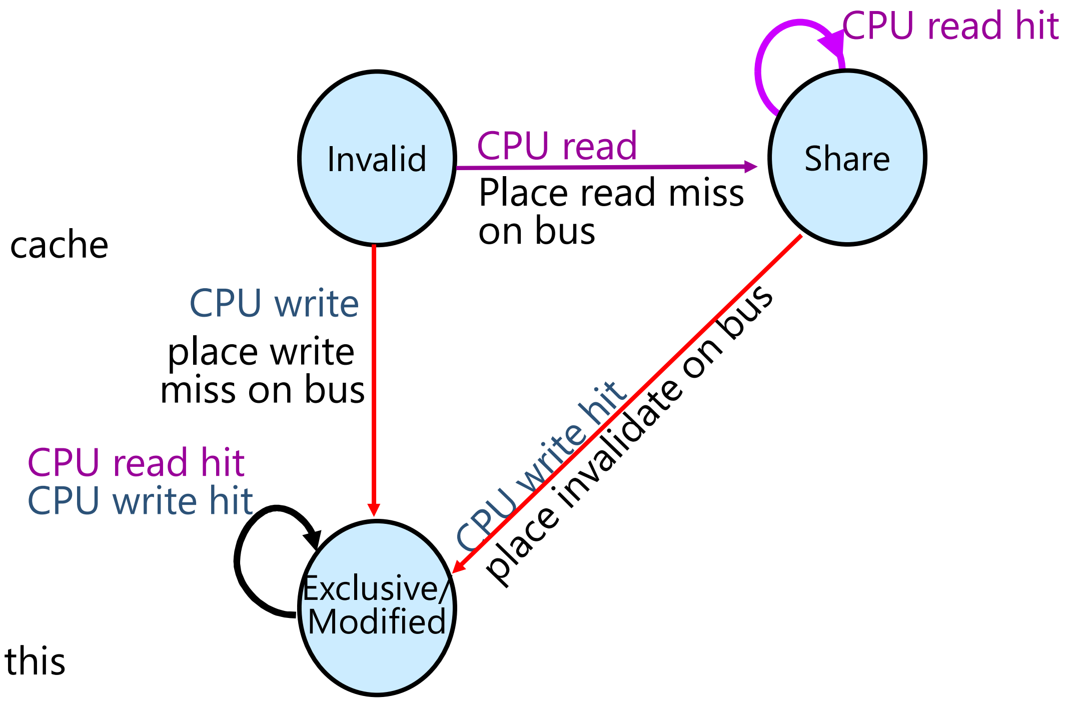
Processor	Bus activity	Cache A	Cache B	Memory
				X=0
A reads X	Cache miss X	X=0		X=0
B reads X	Cache miss X	X=0	X=0	X=0
A writes 1 to X	Update X	X=1	X=1	X=1
B reads X		X=1	X=1	X=1

Qualitative Performance Differences

- Write invalidate versus write update:
  - Multiple writes to same word require
    - multiple write broadcasts in write update protocol
    - one invalidation in write invalidate
  - When cache block contains multiple words, each word written to a cache block requires
    - multiple write broadcasts in write update protocol
    - one invalidation in write invalidate
      - write invalidate works on cache blocks, write update on words/bytes
  - Delay between writing a word in one processor and reading the new value in another is less in write update
- Write invalidate is often preferred when bus bandwidth is most precious

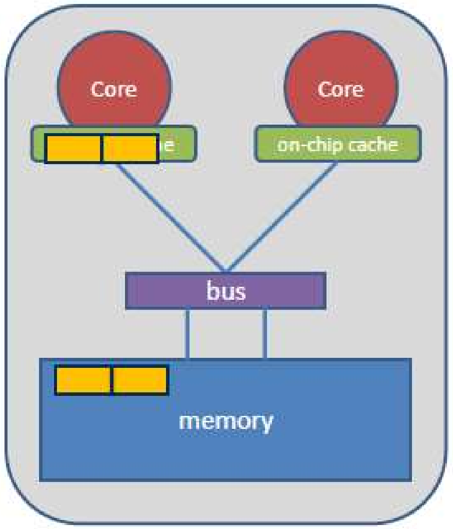
Example of Write-Invalidate Protocol: MSI

- MSI protocol; 3 cache block states
  - Invalid
    - (as if) block is not present
    - need to fetch it from memory or other cache
  - Shared
    - in > 1 caches and in memory
    - can read it without communication
  - Modified
    - in 1 cache
    - can write it without communication
    - others that need it need to get it from this cache
    - textbook calls modified exclusive



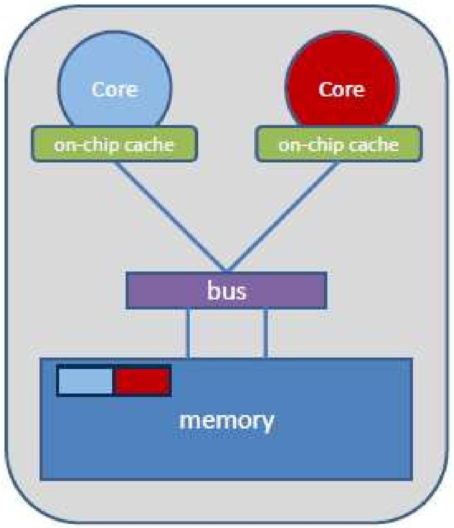
False Sharing

- When data is used, it is copied into caches
- The hardware always copies chunks into the cache, so called cache-lines
- This is useful, when
  - the data is used frequently (temporal locality)
  - consecutive data is used which is on the same cache-line (spatial locality)



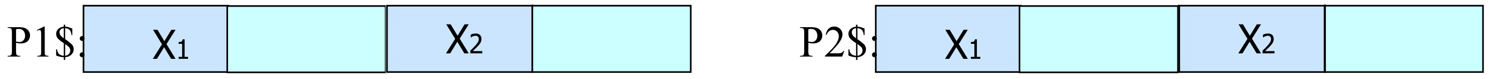
False Sharing

- False sharing occurs when
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line
- As a result, the cache line is moved “between the threads”
  - although there is no real data dependency
- False sharing is a performance problem, not a correctness issue



True vs False Sharing

- True sharing: word being read same as word being written
- False sharing: word being read different from word being written, but they are in same cache block



Time	P1	P2	Comment
1	Write X1		True sharing miss; invalidation required in P2
2		Read X2	False sharing miss, since X2 is invalidated by the write of X1 by P1; now cache block in shared state
3	Write X1		False sharing miss, since X2 is shared again after P2 read it
4		Write X2	False sharing miss since writing to X2 while invalid for the X1 write
5	Read X2		True sharing miss since it involves a read of X2 which was invalidated

False Sharing in OpenMP

- Let’s try to improve the scalability of this code by using partial sums stored on a shared array

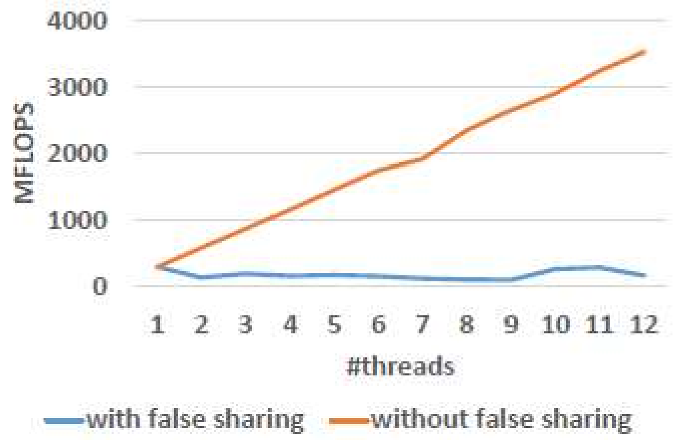
```
#pragma omp parallel
{
  #pragma omp for
  for(i = 0; i < 100; i++)
  {
    s = s + a[i];
  }
} // end parallel
```

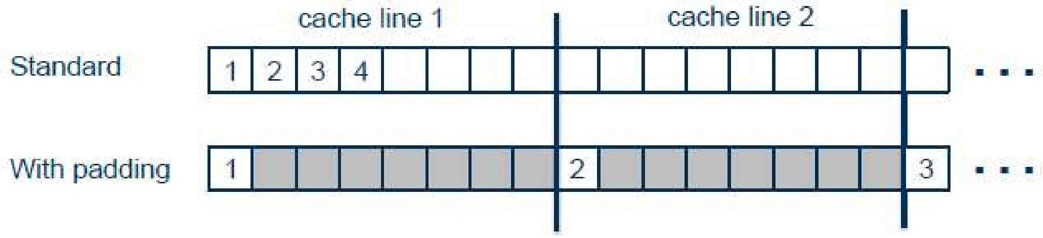
False Sharing

```
double s_priv[nthreads];
#pragma omp parallel num_threads(nthreads)
{
  int t = omp_get_thread_num();
  #pragma omp for
  for(i = 0; i < 100; i++){
    s_priv[t] += a[i];
  }
} // end parallel

for(i = 0; i < nthreads; i++){
  s += s_priv[i];
}
```

False Sharing

- No performance benefit for more threads!
- Reason: false sharing of `s_priv`
- Solution: padding so that only one variable per cache line is used





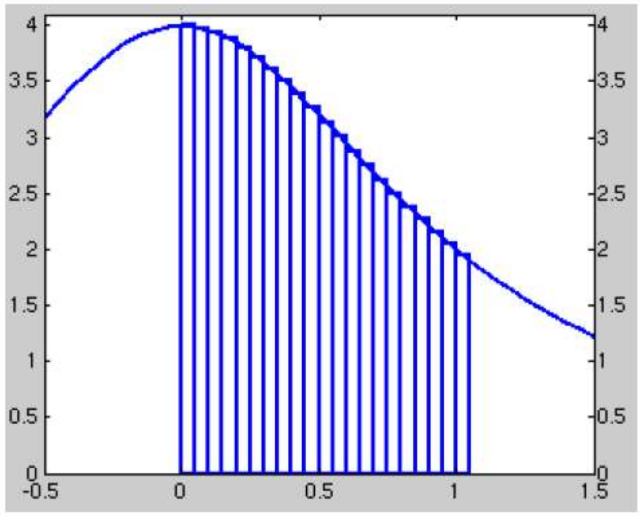
Avoiding False Sharing with Padding

```
double s_priv[nthreads * 8];
#pragma omp parallel num_threads(nthreads)
{
  int t = omp_get_thread_num();
  #pragma omp for
  for(i = 0; i < 100; i++){
    s_priv[t * 8] += a[i];
  }
} // end parallel

for(i = 0; i < nthreads; i++) {
  s += s_priv[i * 8];
}
```

Example: PI Calculation

```
double f(double x) {
  return(4.0 / (1.0 + x*x));
}

double CalcPi(int n) {
  const double fH = 1.0 / (double) n;
  double fSum = 0.0;
  double fX;
  int i;
  for(i = 0; i < n; i++){
    fX = fH* ((double)i + 0.5);
    fSum += f(fX);
  }
  return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Parallelization Attempt 1

- Is it correct?
- How do we fix it?

```
double f(double x) {
  return(4.0 / (1.0 + x*x));
}
double CalcPi(int n) {
  const double fH = 1.0 / (double) n;
  double fSum = 0.0;
  double fX;
  int i;
#pragma omp parallel for
  for(i = 0; i < n; i++){
    fX = fH* ((double)i + 0.5);
    fSum += f(fX);
  }
  return fH* fSum;
}
```

Parallelization Attempt 2

- Is it correct now?

```
double f(double x) {
  return(4.0 / (1.0 + x*x));
}
double CalcPi(int n) {
  const double fH = 1.0 / (double) n;
  double fSum = 0.0;
  double fX;
  int i;
#pragma omp parallel for private(fX,i)
  for(i = 0; i < n; i++){
    fX = fH* ((double)i + 0.5);
    fSum += f(fX);
  }
  return fH* fSum;
}
```

Parallelization Attempt 3

- Is it correct now?
- YES!

```
double f(double x) {
  return(4.0 / (1.0 + x*x));
}
double CalcPi(int n) {
  const double fH = 1.0 / (double) n;
  double fSum = 0.0;
  double fX;
  int i;
#pragma omp parallel for private(fX,i) reduction(+:fSum)
  for(i = 0; i < n; i++){
    fX = fH* ((double)i + 0.5);
    fSum += f(fX);
  }
  return fH* fSum;
}
```

Race Condition

- Data Race: the typical OpenMP programming error, when:
  - two or more threads access the same memory location, and
  - At least one of these accesses is a write, and
  - the accesses are not protected by locks or critical regions, and
  - the accesses are not synchronized, e.g. by a barrier
- Non-deterministic occurrence
  - e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run
- In many cases private clauses, barriers or critical regions are missing
- Data races are hard to find using a traditional debugger
  - use tools like Intel Inspector XE, ThreadSanitizer, Archer

Lab Exercise

1. Implement the false sharing example with padding
   - Try with different pad size and report on runtime
2. Implement the PI computation code
   - Plot the scalability graph with increasing number of threads

Task Parallelism

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Data vs task parallelism
- Fork-join
  - Divide and conquer
- Tasking
- OpenMP Tasking
  - Tasking and Data Scoping
  - Tasking and Synchronization

Data Parallelism vs Task Parallelism

- Data Parallelism
  - concurrent execution of the same task on each multiple computing core
    - same task are performed on different subsets of same data
  - synchronous computation is performed
  - amount of parallelization is proportional to the input size
    - often higher speedup
- Task Parallelism
  - different task are performed on the same or different data
  - asynchronous computation is performed
  - amount of parallelization is proportional to the number of independent tasks
    - often lower speedup

Fork-Join

- Fork-Join is a fundamental way (primitive) of expressing concurrency within a computation
- Fork is called by a (logical) thread (parent) to create a new (logical) thread (child) of concurrency
  - Parent continues after the Fork operation
  - Child begins operation separate from the parent
  - Fork creates concurrency
- Join is called by both the parent and child
  - Child calls Join after it finishes (implicitly on exit)
  - Parent waits until child joins (continues afterwards)
  - Join removes concurrency because child exits

Fork-Join Concurrency Semantics

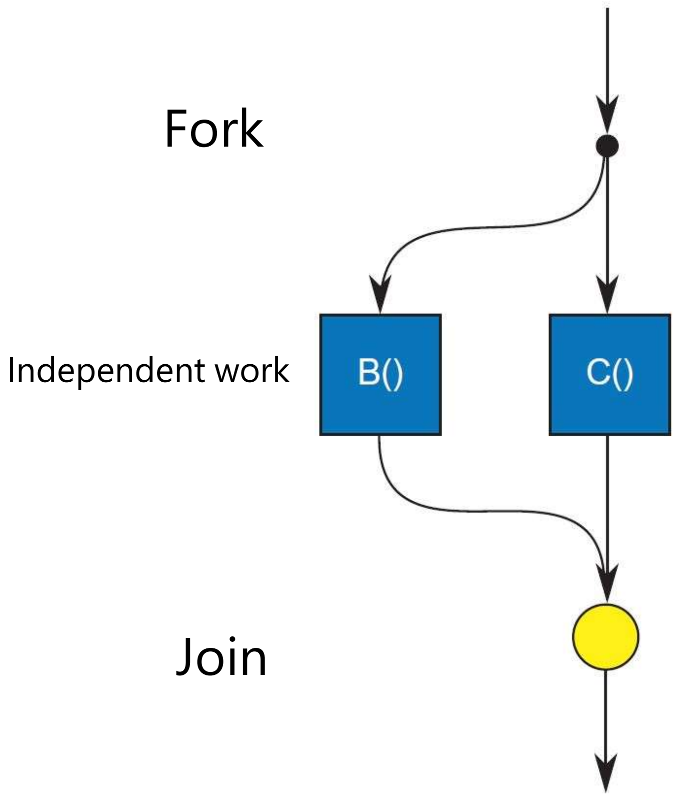
- Fork-Join is a concurrency control mechanism
  - fork increases concurrency
  - join decreases concurrency
- Fork-Join dependency rules
  - a parent must join with its forked children
  - forked children with the same parent can join with the parent in any order
  - a child can not join with its parent until it has joined with all of its children
- Fork-Join creates a special type of DAG
- Fork-Join heritage in Unix

Fork-Join

- Fork
  - child thread performs the piece of work and then exits by calling join with the parent
  - child work is usually specified by providing the child with a function to call on startup
- Join
  - informs the parent that the child has finished
  - child thread notifies the parent and then exits
  - parent thread waits for the child thread to join
  - two scenarios
    1. child joins first, then parent joins with no waiting
    2. parent joins first and waits, child joins and parent then continues

Fork-Join as a Pattern

- Control flow divides (forks) into multiple flows, then combines (joins) later
- During a fork, one flow of control becomes two
- Separate flows are "independent"
  - Does "independent" mean "not dependent" ?
  - No, it just means that the 2 flows of control "are not constrained to do similar computation"
- During a join, two flows become one, and only this one flow continues



Fork-Join Pattern for Divide-and-Conquer

- Typical divide-and-conquer algorithm implemented with fork-join

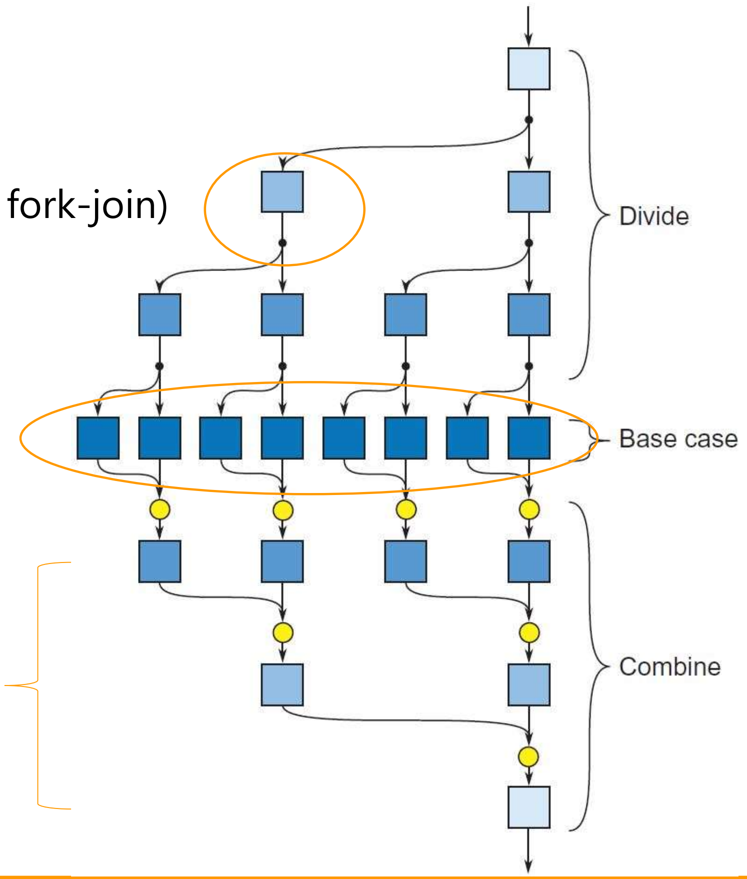
```
void DivideAndConquer( Problem $P$ ) {
    if( $P$ is base case ) {
        Solve $P$;
    } else {
        Divide $P$ into $K$ subproblems;
        Fork to conquer each subproblem in parallel;
        Join;
        Combine subsolutions into final solution;
    }
}
```

Fork-Join Pattern for Divide-Conquer

K = 2   (2-way fork-join)

2^3=8-way parallelism

N = 3   (3 levels of fork-join)



Example: Fibonacci Numbers Computation with Tasking

- Fibonacci numbers
  - a sequence such that each number is the sum of the two preceding
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2), for n>1
- Note: for educational purpose, we are using a recursive implementation of Fibonacci
  - a most efficient implementation would replace the recursion with a loop

```
int main(int argc, char * argv[]){
  …
  fib(input);
  …
}

int fib(int n) {
  if(n < 2) return n;
  int x = fib(n-1);
  int y = fib(n-2);
  return x+y;
}
```

OpenMP `task`

- Deferring (or not) a unit of work (executable for any member of the team)
  - always attached to a structured block

```
#pragma omptask [clause[[,] clause]...]
{ … structured-block … }
```

- Clause can be:
- Data environment
  - `private(list), firstprivate(list), shared(list), default(shared | none)`
  - `in_reduction (r id: list) ≥ 5.0, untied`
- Cutoff strategy
  - `if(scalar expression), mergeable, final(scalar expression)`
- Dependency
  - `depend(dep type: list), priority(priority value)`

Tasks in OpenMP: Data Scoping

- Some rules from parallel regions apply
  - static and global variables are `shared`
  - automatic storage (local) variables are `private`
  - task variables are `firstprivate` unless `shared` in the enclosing context
- Only `shared` attribute is inherited
- Exception: orphaned task variables are `firstprivate` by default

Fibonacci parallelized with Tasking - First version

- Only one task / thread enters `fib()` from `main()`
  - which is responsible for creating the two initial work tasks
- `taskwait` is required, otherwise `x` and `y` would be lost

```
int main(int argc, char * argv[]){
  …
#pragma omp parallel
  {
#pragma omp single
    { fib(input); }
  }
…
}
```

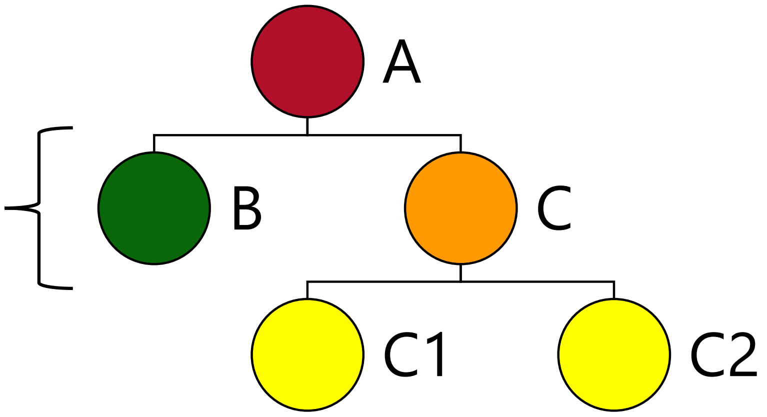
```
int fib(int n) {
  if(n < 2) return n;
  int x, y;
#pragma omp task shared (x)
  { x = fib(n-1); }
#pragma omp task shared (y)
  { y = fib(n-2); }
#pragma omp taskwait
  return x+y;
}
```

The `taskwait` Directive

- `taskwait` is a shallow task synchronization
  - stand-alone directive, wait on the completion of child tasks of the current task
  - just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

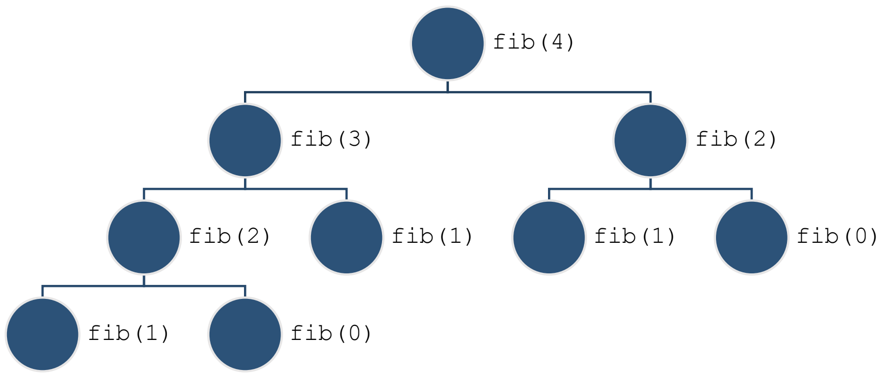
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task                task A
  {
      #pragma omp task task B
      { … }
      #pragma omp task            task C
      { … #pragma omp task … C1, C2 … }
      #pragma omp taskwait
  }
}
```

wait for B and C

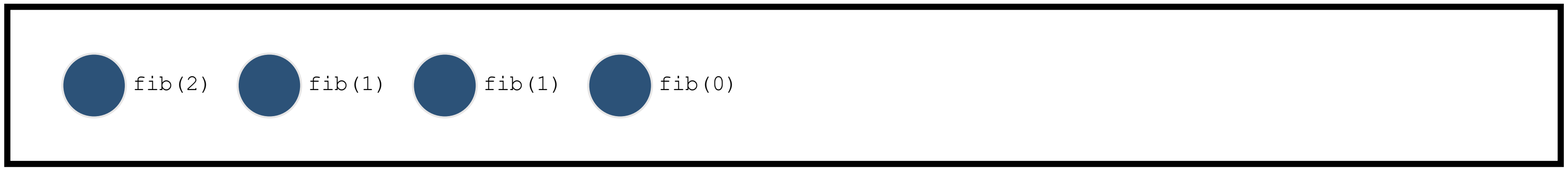


Example of Fibonacci Illustration

- T1 enters `fib(4)`
- T1 creates tasks for `fib(3)` and `fib(2)`
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1-T4 execute tasks

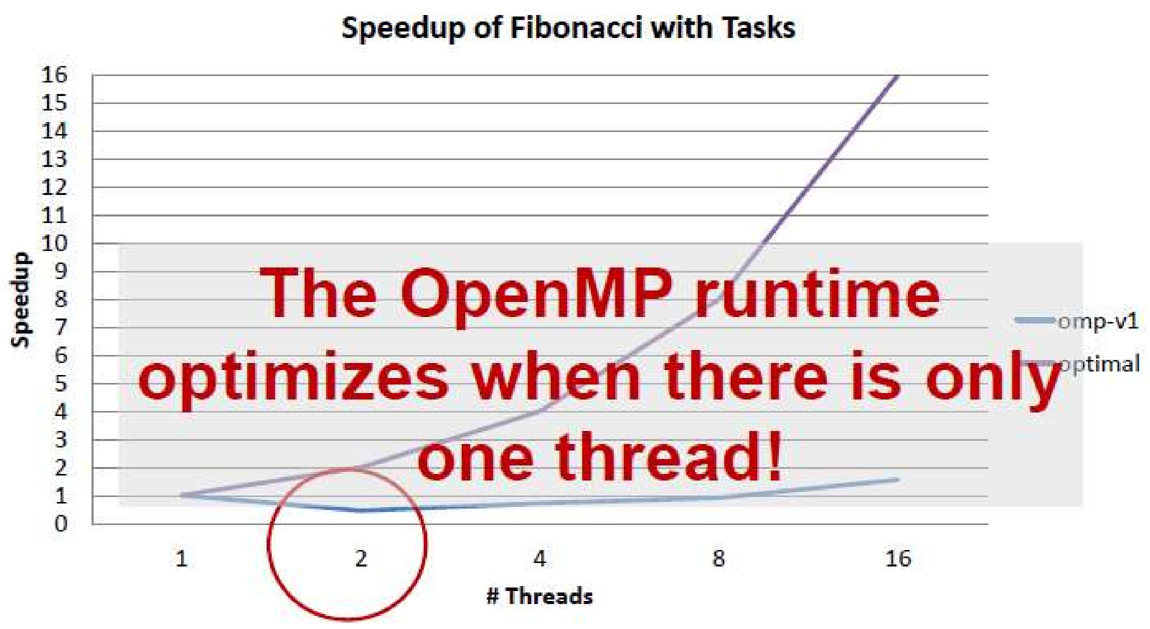


Task queue instance example



Scalability Issues

- Overhead of task creation prevents scalability



`if` Clause

- The `if` clause of a task construct
  - allows to optimize task creation/execution
  - reduces parallelism but also reduces the pressure in the runtime's task pool
  - for "very" fine grain tasks you may need to do your own (manual) if
- If the if expression of the clause evaluates to false
  - the encountering task is suspended
  - the new task is executed immediately
  - the parent task resumes when the task finishes
- This is known as undeferred task

```
#pragma omp task if(expression)
{ .. Structured block… }
```

Fibonacci parallelized with Tasking - 2nd version with `if` clause

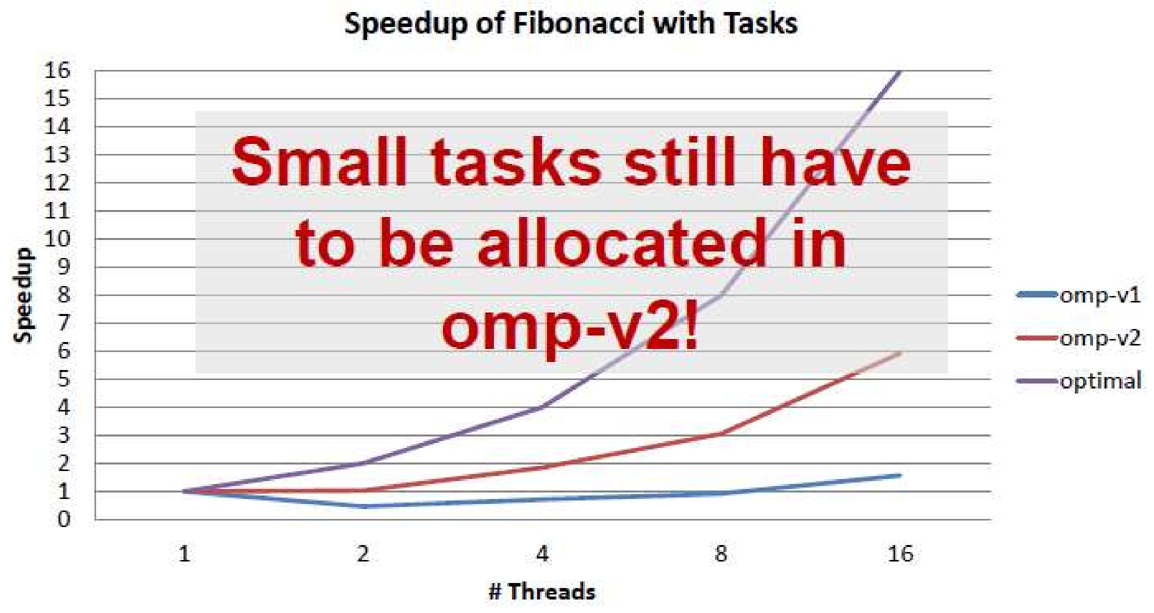
- Don‘t create yet another task once a certain small enough `n` is reached

```
int main(int argc, char * argv[]){
  …
#pragma omp parallel
  {
#pragma omp single
    { fib(input); }
  }
…
}
```

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

Scalability Issues

- Speedup is better, but still not great



Fibonacci parallelized with Tasking – 3rd version with sequential `if` clause

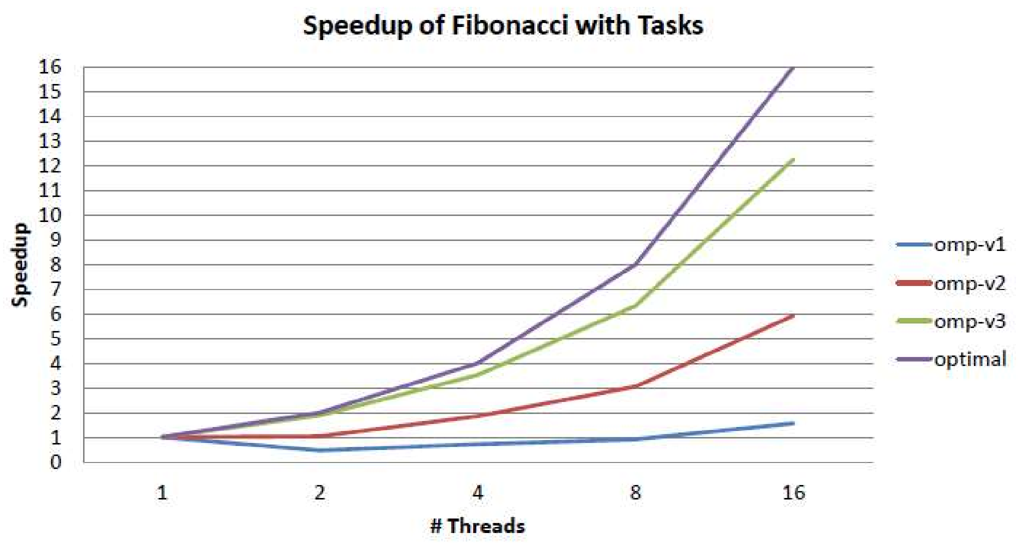
- Don't create yet another task once a certain small enough `n` is reached

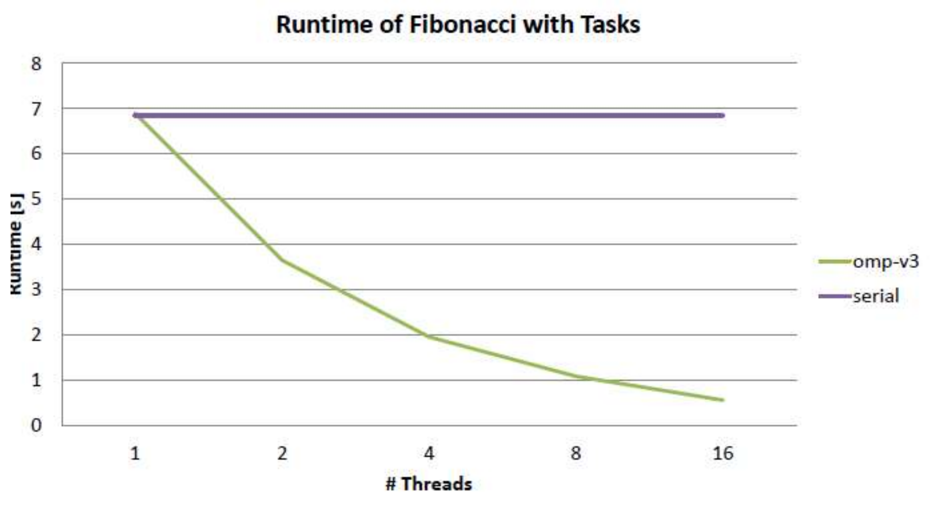
```
int main(int argc, char * argv[]){
  …
#pragma omp parallel
  {
#pragma omp single
    { fib(input); }
  }
…
}
```

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

Scalability Results

- Good scaling, first two versions were slow because of overhead





Tasking Overhead

- Tasking overheads
  - task creation: populate task data structure, add task to task queue
  - task execution: retrieve a task from the queue (may including work stealing)
- If tasks become too fine grained, overhead becomes noticeable
  - execution spends a higher relative amount of time in the runtime
  - task execution contributing to runtime becomes significantly smaller
- A rough rule of thumb to avoid (visible) tasking overhead
  - OpenMP tasks: 80 100k instructions executed per task
  - TBB tasks: 30 50k instructions executed per task
  - other programming models may have another ideal granularity

Threads vs Tasks

- Threads do not compose well
  - example: multi-threaded plugin in a multi threaded application
  - composition usually leads to oversubscription and load imbalance
- Task models are inherently composable
  - a pool of threads executes all created tasks
  - tasks from different modules can freely mix
- Task models make complex algorithms easier to parallelize
  - programmers can think in concurrent pieces of work
  - mapping of concurrent execution to threads handled elsewhere
  - task creation can be irregular, e.g., recursion, graph traversal

Threads vs Tasks

- Some scenarios are more amenable for traditional threads
  - Granularity too coarse for tasking
  - Isolation of autonomous agents
- Static allocation of parallel work is typically easier with threads
  - Controlling allocation of work to cache hierarchy
- Graphical User Interfaces (event thread + worker threads)
- Request/response processing, e.g.,
  - Web servers
  - Database servers

Task and Scoping

- Parallel region rules
  - automatic storage (local) variables are `private`
  - static and global variables are `shared`
- Tasking
  - variables are `firstprivate` unless `shared` in the enclosing context
    - only `shared` attribute is inherited
    - exception: orphaned task variables are `firstprivate` by default

Data Scoping Example: Guess the Scope

```
int a = 1;
void foo() {
  int b = 2, c = 3;
#pragma omp parallel private(b)
{
  int d = 4;
  #pragma omp task
  {
    int e = 5;
    // Scopeof a:
    // Scopeof b:
    // Scopeof c:
    // Scopeof d:
    // Scopeof e:
  }
}
}// foo
```

Data Scoping Example: Guess the Scope

```
int a = 1;
void foo() {
  int b = 2, c = 3;
#pragma omp parallel private(b)
{
  int d = 4;
  #pragma omp task
  {
    int e = 5;
    // Scopeof a: shared
    // Scopeof b: firstprivate
    // Scopeof c: shared
    // Scopeof d: firstprivate
    // Scopeof e: private
  }
}
}// foo
```

Hint: Use default(none) to be forced to think about every variable if you do not see clear.

Data Scoping Example: Guess the Value

- What values do they have?

```
int a = 1;
void foo() {
  int b = 2, c = 3;
#pragma omp parallel private(b)
{
  int d = 4;
  #pragma omp task
  {
    int e = 5;
    // Scopeof a: shared       a=1
    // Scopeof b: firstprivate b= 0 / undefined
    // Scopeof c: shared       c=3
    // Scopeof d: firstprivate d=4
    // Scopeof e: private      e=5
  }
}
}// foo
```

Scoping and Lifetime

- How long do `private` / `firstprivate` instances exist?
  - Alive until the end of assigned structured block or consturct

```
int i = 5;
#pragma omp parallel firstprivate(i)
{
  // private copy per thread
  // initialized with 5 alive
  // until end of parallel region

} <- here
```

```
#pragma omp parallel
#pragma omp single
{
  int i =5;
  #pragma omp task
  {
    // firstprivate copy of i for task
    // alive until end of task
  } <- here
}
```

Tied vs Untied Tasks

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time
    - if deferred, the task is placed in a pool of tasks associated with the current parallel region
    - all team threads will take tasks out of the pool and execute them until the pool is empty
    - a thread that executes a task might be different from the thread that originally encountered it
- The code associated with a task construct will be executed only once
    - a task is tied if the code is executed by the same thread from beginning to end
    - otherwise, the task is untied (the code can be executed by more than one thread)

```
#pragma omp task untied
```

Lab Exercise: Parallel divide-and-conquer

1. Matmul optimization with a recursive tiling strategy
   - multiplications of n/2 × n/2 matrices, 1 addition of n × n matrices
   - optimize memory access beyond L1 (L2, L3)

- Hints:
  - stop the recursion when too small
  - use the `matmul` used in the introduction lecture for smaller sizes

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

Nested Parallelism and NUMA

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Nested parallelism
- NUMA architecture
- Thread affinity
- Atomics

Loop Parallelism

- Assume compute f is expensive
  - but variable in execution time

```
N=10;
for (i=0; i< N; i++)
   for (j=0; j< N; j++)
      for (k=0; k< N; k++)
         A[i][j][k] = f(i, j, k);
```

- Your target machine has 30 cores
  - all the loops are parallel, but none is adequately large enough to employ all the threads
  - further, the variability in execution time means the need to use dynamic schedule
- How do we parallelize this loop?

More Parallelism by “Collapsing” Loops

```
N=10;
#pragma omp parallel for schedule(dynamic)
for (m=0; i< N*N*N; i++)   {
  i =  m /(N*N);
  j = (m % (N*N))/N;
  k =  m % N;
  A[i][j][k] = f(i, j, k);
}
```

OpenMP Collapse Clause

```
N=10;
#pragma omp parallel for collapse(3) schedule(dynamic)
for (i=0; i< N; i++)
  for (j=0; j< N; j++)
    for (k=0; k< N; k++)
      A[i][j][k] = f(i, j, k);
```

- 3 nested loops are combined to make a single loop with 1000 iterations
  - `i, j, k` calculated automatically
  - no need to declare `j` and `k` private, they are implicitly private
- the collapse clause is useful even when we don't have an expensive yet variable function `f`, requiring dynamic balancing

Nested Parallelism

- OpenMP uses a fork-join model of parallel execution
  - when a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads
  - when a thread finishes its work within the parallel construct, it waits at the implicit barrier at the end of the parallel construct
- OpenMP parallel regions can be nested inside each other
  - if nested parallelism is disabled, then the new team created by a thread encountering a parallel construct inside a parallel region consists only of the encountering thread
  - if nested parallelism is enabled, then the new team may consist of more than one thread
  - `omp_set_max_active_levels(int)`: limits the number of nested active parallel regions on the device,

Nested Parallelism Example

- Parallel region within parallel region

```
omp_set_max_active_levels(2);
omp_set_dynamic(0); // make thread number adjustment explicit
#pragma omp parallel num_threads(2)
{
  int t1 = omp_get_thread_num();
#pragma omp parallel num_threads(3)
  {
    int t2 = omp_get_thread_num();
#pragma omp critical
    { cout << "[" << t1 << "," << t2 << "]" << endl; }
  }
}
```

```
[0,0]
[0,1]
[1,0]
[0,2]
[1,1]
[1,2]
```

Nested Parallelism Example

- Parallel region within parallel region

```
omp_set_max_active_levels(1);
omp_set_dynamic(0); // make thread number adjustment explicit
#pragma omp parallel num_threads(2)
{
  int t1 = omp_get_thread_num();
#pragma omp parallel num_threads(3)
  {
    int t2 = omp_get_thread_num();
#pragma omp critical
    { cout << "[" << t1 << "," << t2 << "]" << endl; }
  }
}
```

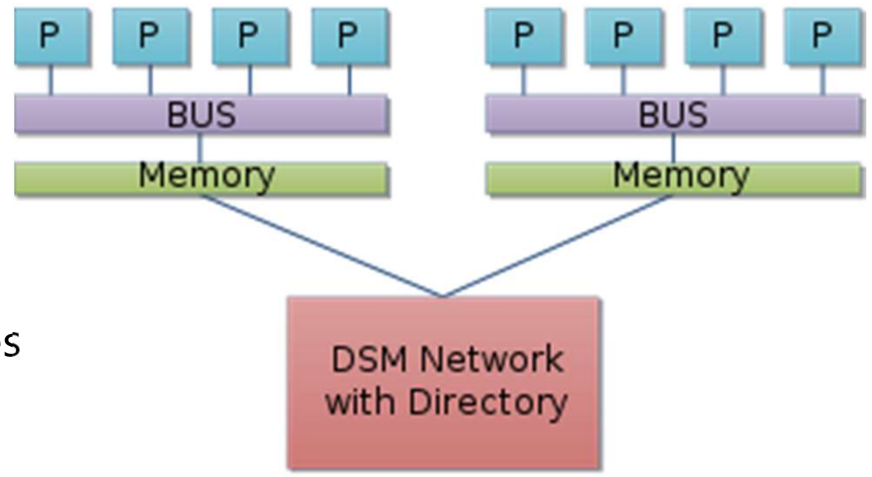
```
[0,0]
[1,0]
```

Notes on Nested Parallelism

- Nesting parallel regions can easily create too many threads and oversubscribe the system
- Creating nested parallel regions adds overhead
  - if there is enough parallelism at the outer level and the load is balanced, generally it will be more efficient to use all the threads at the outer level of the computation than to create nested parallel regions at the inner levels

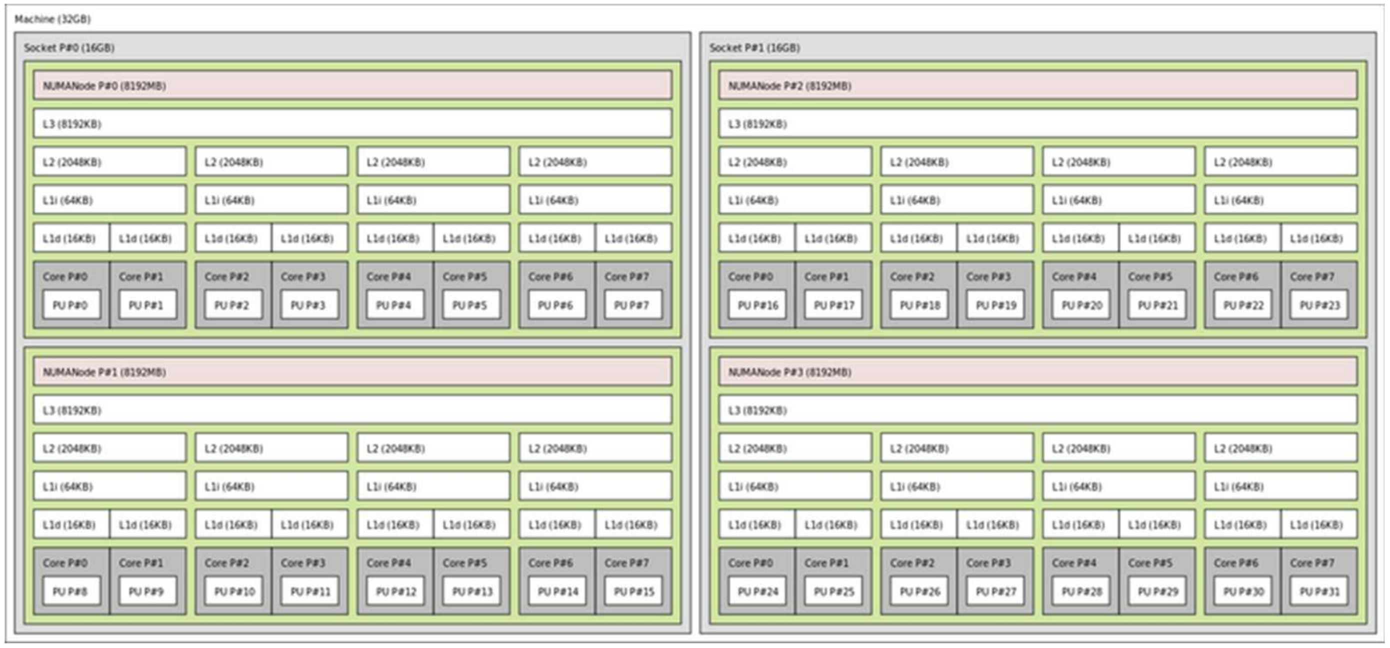
NUMA

- Non-uniform memory access (NUMA)
  - memory access time depends on the memory location relative to the processor
- Cache coherent NUMA (ccNUMA)
  - with NUMA, maintaining cache coherence across shared memory has a significant overhead
  - ccNUMA uses inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location
    - e.g., directory-based cache coherency protocol



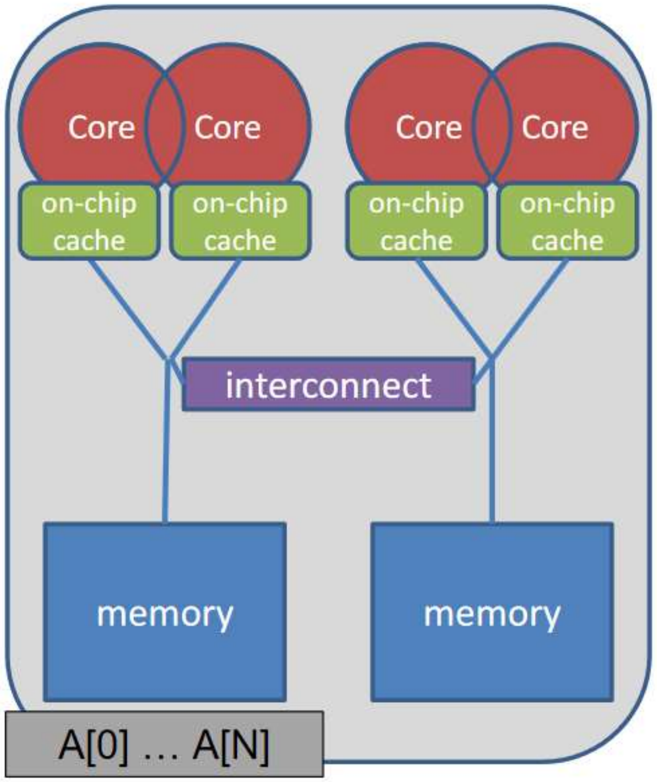
Example: AMD Bulldozer Model FX-8100 (2011)

- 8 cores, 4 modules, two socket
- Four NUMA domains



Data Distribution on NUMA

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

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



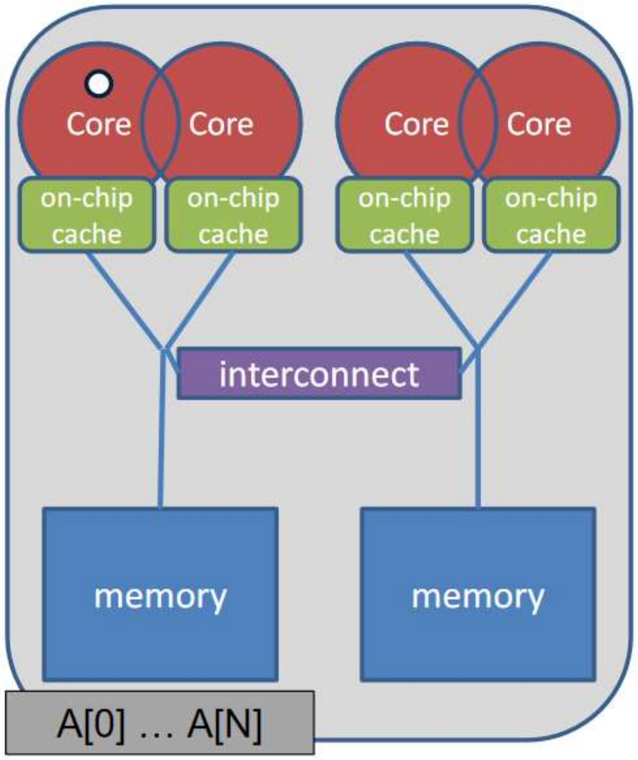
About Data Placement

- Data distribution is an important aspect of cc-NUMA
  - if not optimal, longer memory access times and hotspots
- OpenMP does not provide explicit support for cc-NUMA on first sight
- Data placement comes from the Operating System
  - thus, is OS dependent
  - OpenMP 5.0 introduce feature for fine-grained control of memory management
- “First touch” placement policy
  - in use by Windows, Linux, Solaris
  - maybe possible to change

Non-Uniform Memory Architecture

- Serial code: all array elements are allocated in the memory of the NUMA node containing the code executing this thread

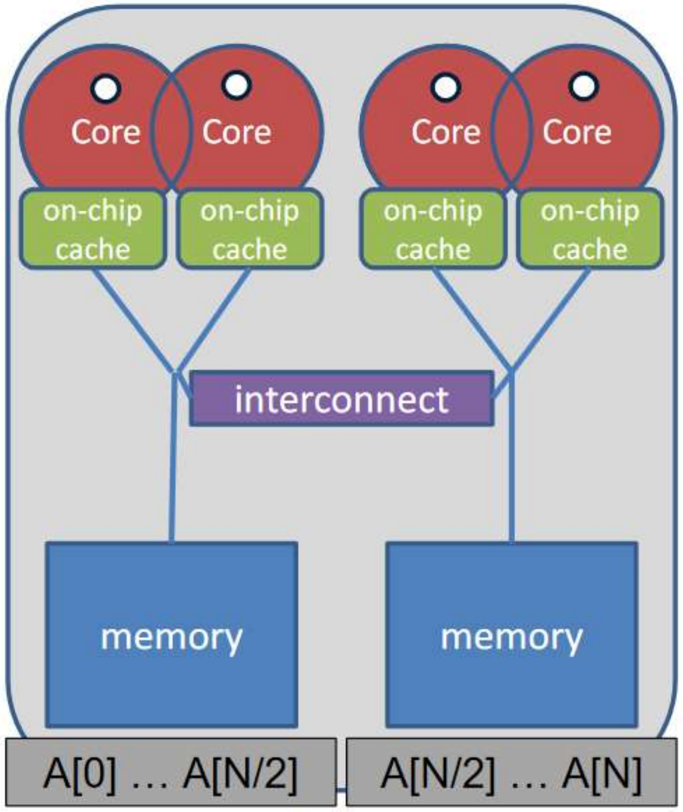
```
double *A;
A = (double*) malloc(N * sizeof(double));

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



Non-Uniform Memory Architecture

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

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

omp_set_num_threads(4);

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

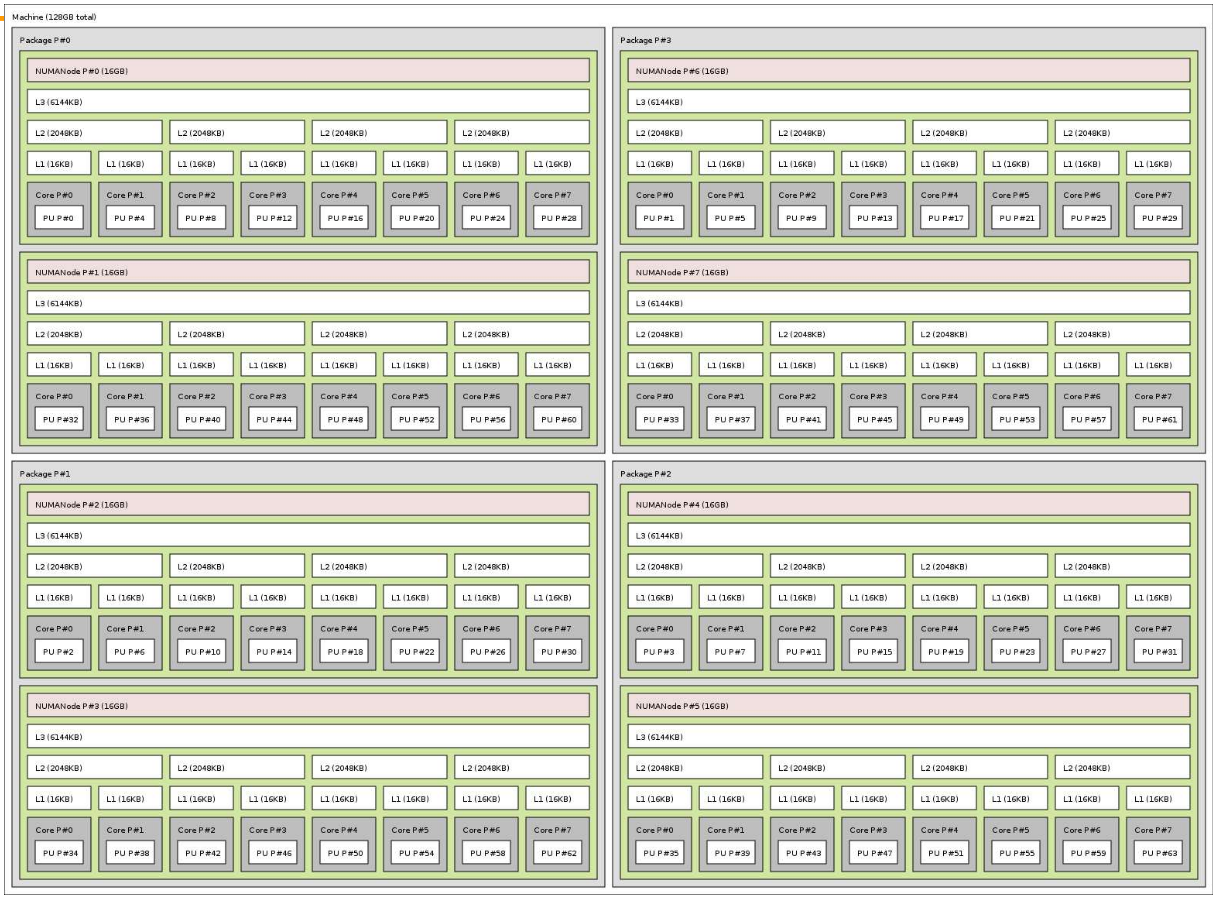


Understanding System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology
- Tools to discover system topology
  - `cpuinfo`
    - delivers information about the number of sockets and the mapping of processor ids used by the operating system to CPU cores
  - `hwloc` and `lstopo`
    - displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to CPU cores and additional info on caches

Topology Example: AMD

- 4x Opteron Magny-Cours 6272 (from 2012, with hwloc v1.11)
  - each package has 2 NUMA domains

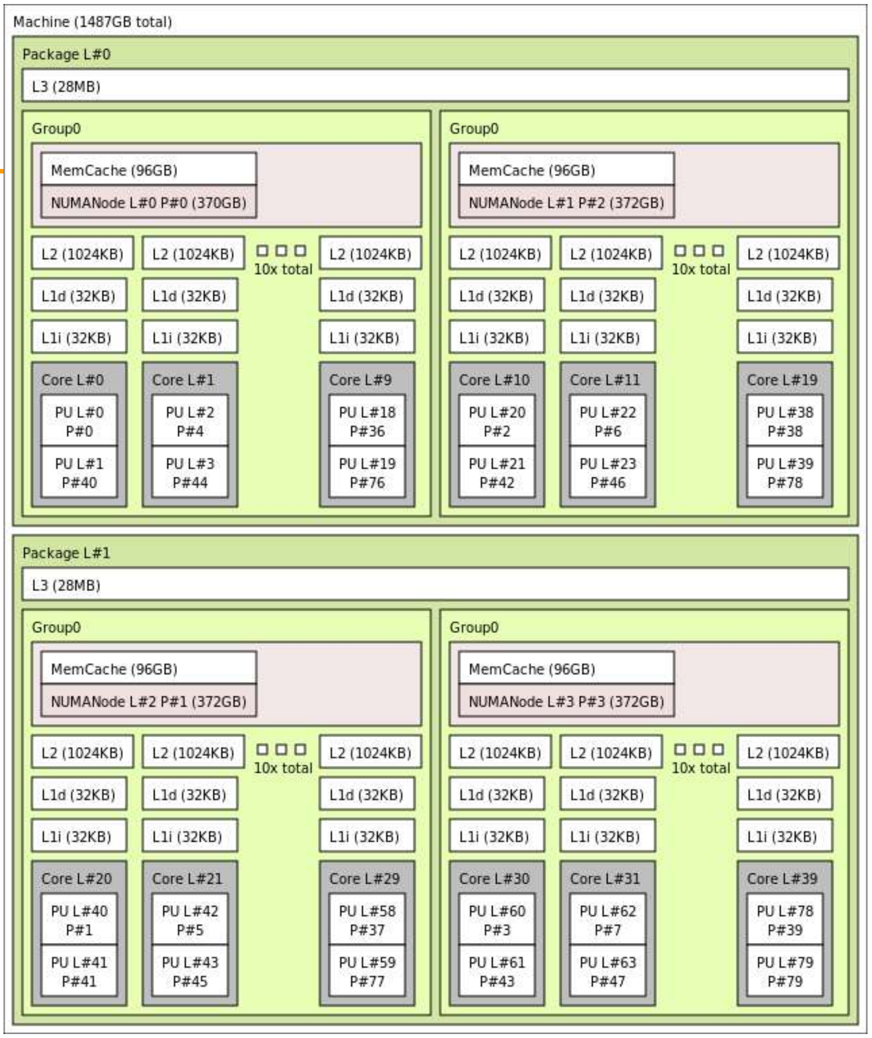

Machine (128GB total)
Package P#0
NUMANode P#0 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#0 PU P#4 PU P#8 PU P#12 PU P#16 PU P#20 PU P#24 PU P#28
NUMANode P#1 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#32 PU P#36 PU P#40 PU P#44 PU P#48 PU P#52 PU P#56 PU P#60
Package P#3
NUMANode P#6 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#1 PU P#5 PU P#9 PU P#13 PU P#17 PU P#21 PU P#25 PU P#29
NUMANode P#7 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#33 PU P#37 PU P#41 PU P#45 PU P#49 PU P#53 PU P#57 PU P#61
Package P#1
NUMANode P#2 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#2 PU P#6 PU P#10 PU P#14 PU P#18 PU P#22 PU P#26 PU P#30
NUMANode P#3 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#34 PU P#38 PU P#42 PU P#46 PU P#50 PU P#54 PU P#58 PU P#62
Package P#2
NUMANode P#4 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#3 PU P#7 PU P#11 PU P#15 PU P#19 PU P#23 PU P#27 PU P#31
NUMANode P#5 (16GB)
L3 (6144KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB) L1 (16KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7
PU P#35 PU P#39 PU P#43 PU P#47 PU P#51 PU P#55 PU P#59 PU P#63

SubNUMA clustering

- Sub-NUMA Clustering divides the cores, cache, and memory of the processor into multiple NUMA domains
  - enabling it can increase performance for workloads that are NUMA aware and optimized

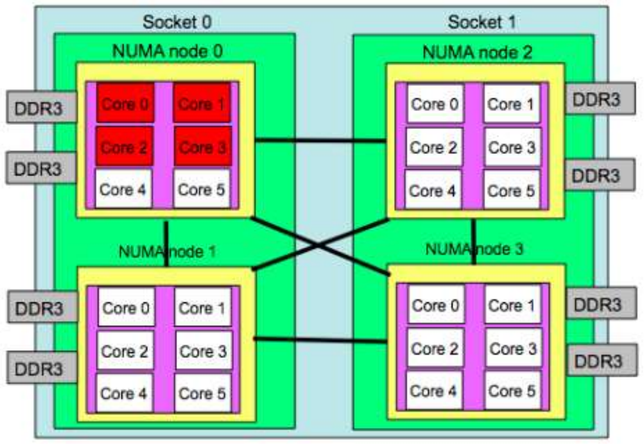
Topology Example:

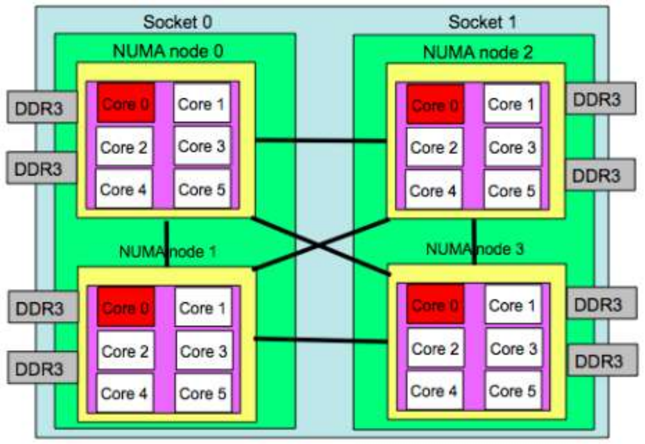
- 2x Xeon CascadeLake 6230 with DDR as a cache in front of NVDIMMs (from 2019, with hwloc v2.1)
  - processors are configured in SubNUMA-Cluster mode which shows 2 NUMA nodes per package



Thread Affinity

- Thread affinity: forces each process or thread to run on a specific subset of processors, to take advantage of local process state
  - also: MPI thread affinity
- Thread locality is important since it impacts both memory and intra-node performance





Binding Strategies

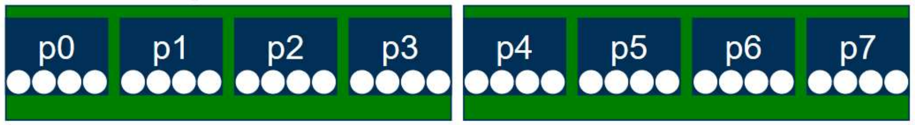
- The best binding strategy depends on the topology and the characteristics of your application

Binding strategies:

- Putting threads far apart, i.e., on different sockets
  - may improve the aggregate memory bandwidth available to your application
  - may improve the combine cache size available to your application
  - may decrease performance of synchronization constructs
- Putting threads close together, i.e., on two adjacent cores, which possibly share some caches
  - may improve performance of synchronization constructs
  - may decrease the available memory bandwidth and cache size

OpenMP Places

- OpenMP places: a list of places that threads can be pinned on
  - set of OpenMP threads running on one or more processors
  - can be defined by the user, i.e., `OMP_PLACES=cores`
- Example: 2 sockets, 4 cores per socket, 4 hyper-threads per core



- Place values
  - `threads`: each place corresponds to a single hardware thread
  - `cores`: each place corresponds to a single core (having one or more hardware threads)
  - `sockets`: each place corresponds to a single socket (consisting of one or more cores)
  - in OpenMP 5: `ll_caches` and `numa_domains`
  - a list with explicit place values, such as
    - `"{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"`
    - `"{0:4},{4:4},{8:4},{12:4}"`

OpenMP Binding Policies

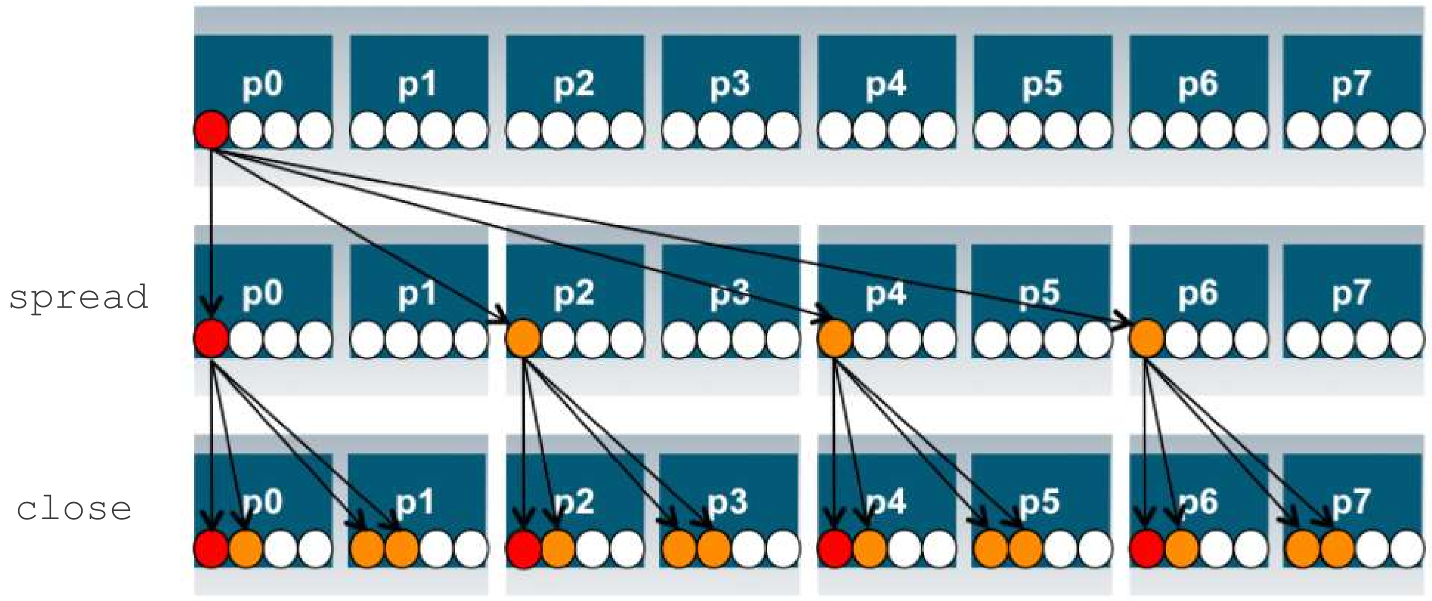
- OpenMP thread affinity places
  - `master`: assign every thread in the team to the same place as the master thread
  - `close`: assign the threads in the team to places close to the place of the parent thread
  - `spread`: spread threads evenly among the places
- Goals
  - user has a way to specific where to execute OpenMP threads for
  - locality between OpenMP thread / avoid false sharing / memory bandwidth

OpenMP Bindings

- When a thread encounters a parallel directive without a `proc_bind` clause, the binding value is used to determine the policy for assigning OpenMP threads to places within the current place partition
  - within the places listed in the place variable for the implicit task of the encountering thread
- If the `parallel` directive has a `proc_bind` clause then the binding policy specified by the `proc_bind` clause overrides the policy specified by the first element of the binding variable
  - once a thread in the team is assigned to a place, the OpenMP implementation should not move it to another place

Example of OpenMP Thread Affinity

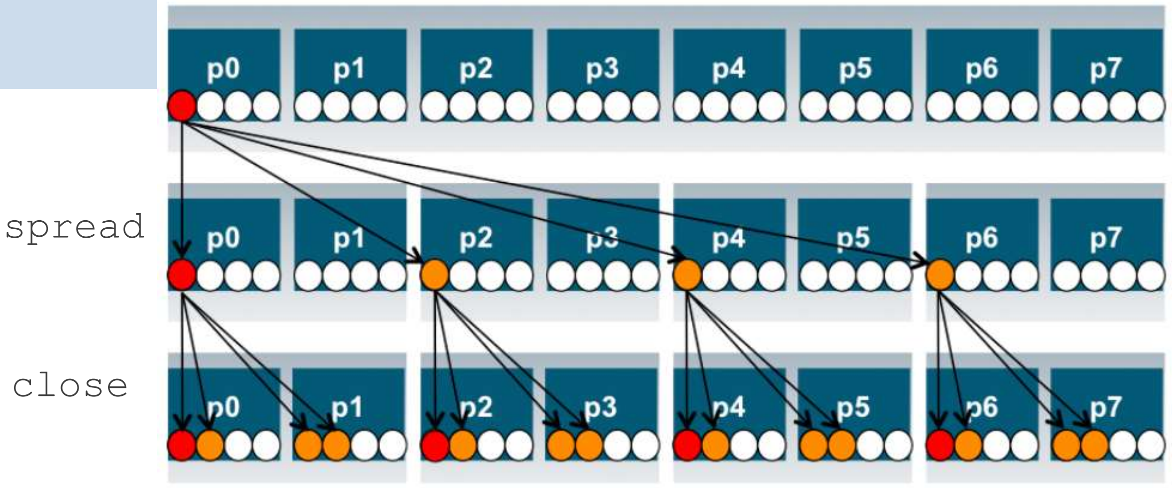
```
setenv OMP_PLACES threads
setenv OMP_NUM_THREADS 4,4
setenv OMP_PROC_BIND spread,close
```



Example: parallel for with `proc_bind`

- Assume: same `PLACES` as in previous example

```
#pragma omp parallel proc_bind(spread) num_threads(4)
for (int i=0; i<N; i++){
#pragma omp parallel proc_bind(close) num_threads(4)
  for (int j=0; j<M; j++){
        /* do some work */
  }
}
```



Example: Socket Init

- Run a init function on each socket
- Assume OMP_PLACES = "{0:8},{8:8}“ in a two socket with 8 cores in each socket

```
int n_sockets, socket_num;
omp_set_nested(1);
omp_set_max_active_levels(2);
n_sockets = omp_get_num_places();
#pragma omp parallel num_threads(n_sockets) private(socket_num) proc_bind(spread)
{
  socket_num = omp_get_place_num();
  socket_init(socket_num);
}
```

OpenMP Atomic

- The atomic construct ensures that a specific storage location is accessed atomically
  - rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values

```
int count;
void Tick() {
#pragma omp atomic
  count = count+1;
}
```

- Alternative: OpenMP critical section
  - critical section is completely general, it can surround any arbitrary block of code
  - significant overhead every time a thread enters and exits the critical section (on top of the inherent cost of serialization)
    - atomic operation has much lower overhead

Notes on OpenMP Performance

- Minimize synchronization
  - avoid or minimize the use of `barrier`, `critical`, `sections`, `ordered` regions, and locks
  - there is always an implied barrier at the end of a parallel region, use the `nowait` clause where possible to eliminate redundant or unnecessary barriers
  - use named `critical` sections for fine-grained locking
  - use explicit `flush` with care, can cause data cache restores to memory, and subsequent data accesses may require reloads from memory, all of which decrease efficiency
- By default, idle threads will be put to sleep after a certain time out period
  - it could be that the default time out period is not sufficient for your application, causing the threads to go to sleep too soon or too late
  - implementation may allow to change that value

Notes on OpenMP Performance

- Parallelize at the highest level possible, such as `outer` for loops
  - enclose multiple loops in one parallel region
  - in general, make parallel regions as large as possible to reduce parallelization overhead
- Use `master` instead of `single` wherever possible
  - the `master` directive is implemented as an `if`-statement with no implicit barrier:

```
if(omp_get_thread_num()==0){…}
```

  - the `single` directive is implemented similar to other work sharing constructs, keeping track of which thread reached `single` first adds additional runtime overhead, there is an implicit synchronization if `nowait` is not specified
  - but load unbalance may favor `master`

Notes on OpenMP Performance

- Choose the appropriate loop scheduling
  - `static` causes no synchronization overhead and can maintain data locality when data fits in cache, but may lead to load imbalance
  - `dynamic`, `guided` incurs a synchronization overhead to keep track of which chunks have been assigned; these schedules could lead to poor data locality, but can improve load balancing; experiment with different chunk sizes
- Use `lastprivate` with care, as it has the potential of high overhead
  - data needs to be copied from private to shared storage upon return from the parallel construct
  - checks which thread executes the logically last iteration: extra work at the end of each chunk in a parallel `for`
- Use efficient thread-safe memory management
  - applications could be using `malloc()` and `free()` explicitly, or implicitly
  - the thread-safe `malloc()` and `free()` in libc have a high synchronization overhead caused by internal locking
- Small data cases may cause OpenMP parallel loops to underperform
  - use the `if` clause on `parallel` constructs to indicate that a loop should run parallel only in those cases where some performance gain can be expected

Lab Exercise

1. Parallelize a Mandelbrot code and experiment with nested parallelism and affinity
   - Typically implemented with 3 nested loops: `for x, for y, for iterations`
     - E.g., https://www.geeksforgeeks.org/fractals-in-cc

2. Histogram counting: Given an array of 10000 random integers in `[1,100]`, implement a parallel algorithm that counts how many elements fall in each bucket
   - `[1,10]`
   - `[20,30]`
   - …
   - `[90,100]`