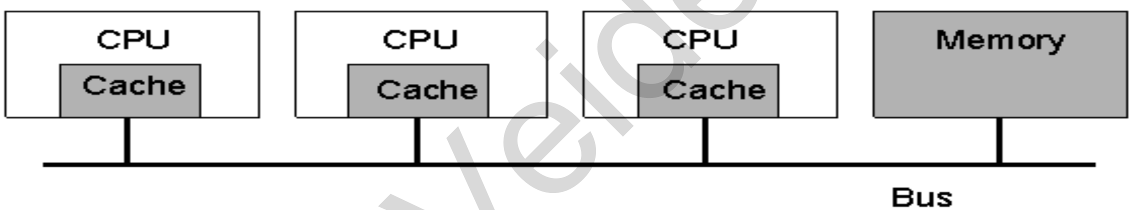
CompSci 131

Parallel and Distributed Systems

Prof. A. Veidenbaum

Parallel Programming on SMPs

- **SMP = Symmetric Multi-Processor**
  - **Shared Memory, “tightly coupled”**
    - » **Processors are connected to memory via a bus or switch**
      - **A bus-based multiprocessor**



- **Caches are a (partially) replicated memory**
  - **Improve performance by lowering average memory latency**
  - **But they create a memory coherence problem**
    - » **We’ll look at how this can happen**

Multiprocessors (1)

- Multi-core vs multiprocessor:
  - A core is a processor in a single-chip microprocessor
    - » Intel Xeon Gold is a multi-core processor w/ up to 28 cores
  - A multiprocessor today is built from multiple multi-cores
    - » Intel terminology: a processor = socket = 1 Xeon Gold
- Multi-cores require additional hardware support for
    - » Inter-core communication and interrupts
    - » Mutual exclusion
    - » Indivisibility of memory RMW
    - » Cache coherence
- May use processor or core interchangeably
  - Should be clear from the context

Multiprocessors (1)

- **Multi-cores require additional *OS* support for**
  - **Process isolation and management**
  - **Mutual exclusion**
  - **VM management**

- **Mutual exclusion is tied to process management**
  - **A process/thread blocked on a semaphore is suspended**
  -

- **Let's look at hardware support for mutual exclusion**

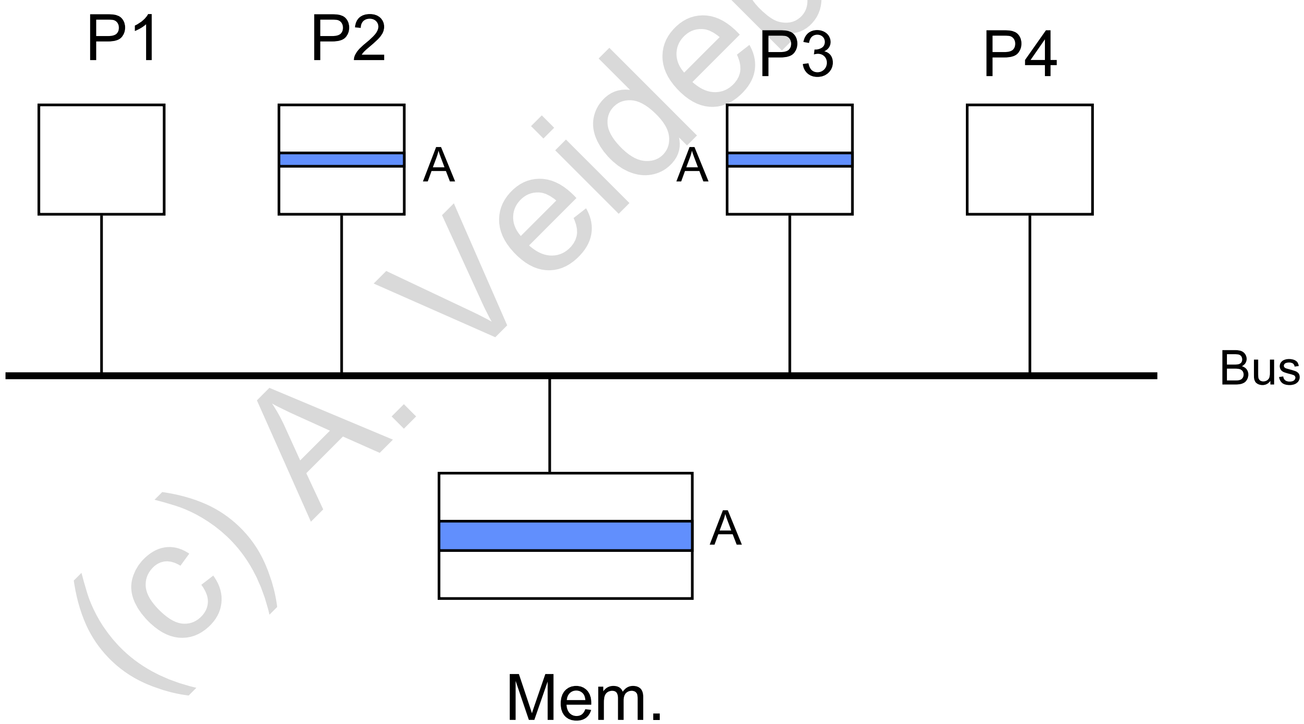
Multiprocessors (2)

- **Intel instructions for locking, indivisibility**
  - Aka atomic processor instructions
    - » atomically update a memory location
  - An atomic instruction uses a lock prefix on the instruction and has its destination operand in memory.
  - These instructions can use a lock prefix
    - » ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG

- **Inter-processor interrupt (IPI)**
  - Each core has an APIC
  - A core puts the IV, destination ID into its local APIC
    - » Its local APIC delivers it to the remote APIC

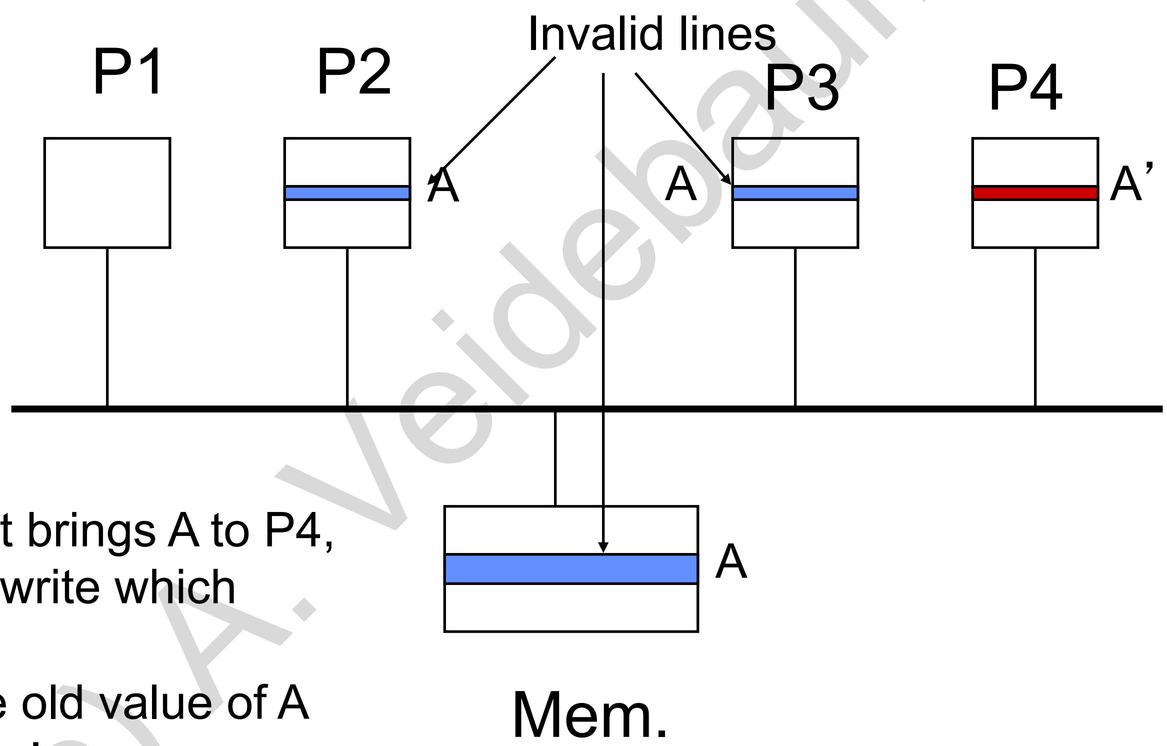
Cache Coherence: The problem

- P2 reads A, it is automatically stored in its cache
- P3 reads A, same deal



Now a Write occurs

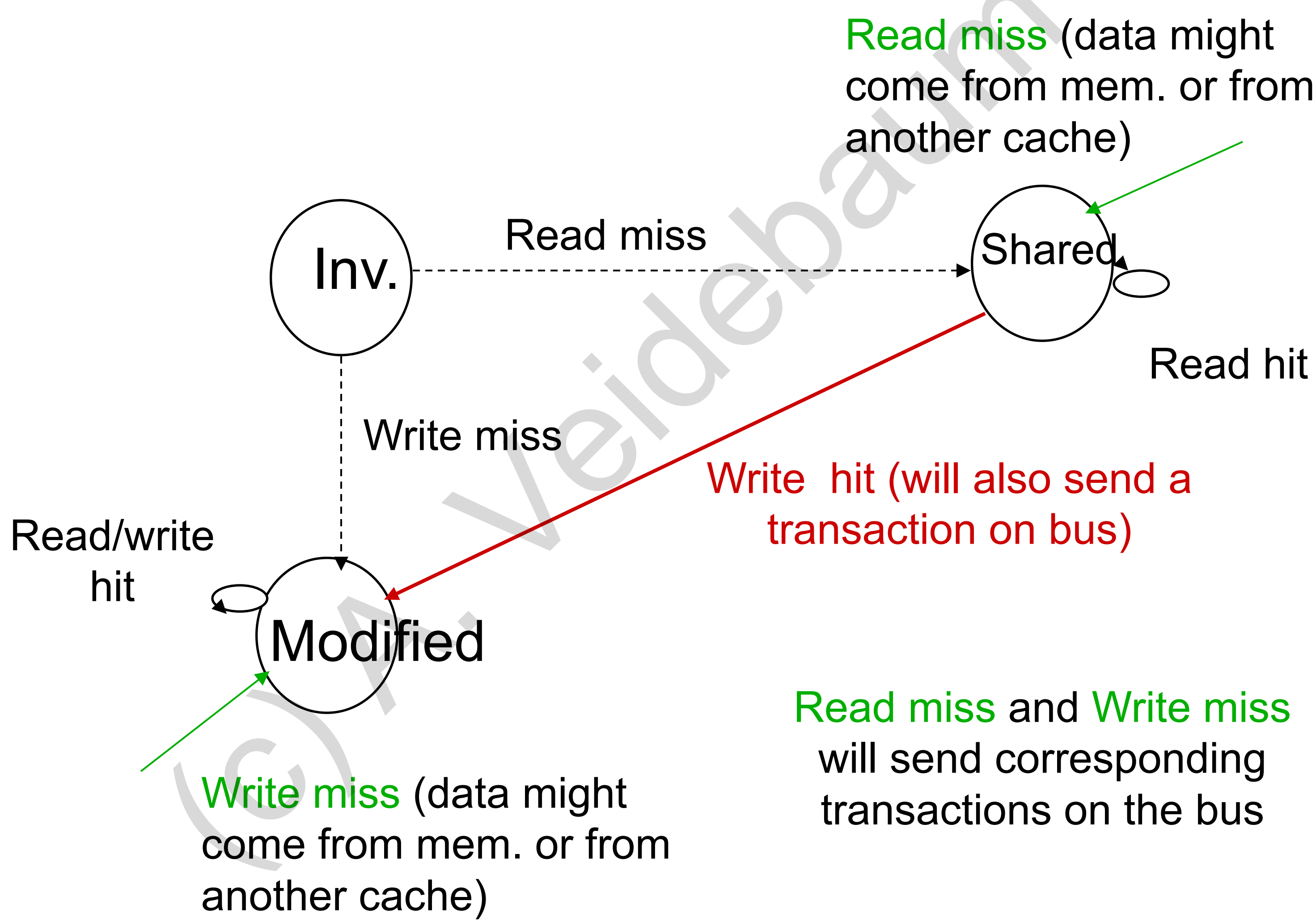
- P4 has a write miss on line A



- A write miss first brings A to P4,
- followed by the write which creates A’
- P2, P3 have the old value of A
- So does memory!

- Multiple copies are no longer identical, this is the coherence problem
- Solved by a hardware cache coherence protocol

A 3-State Protocol: Processor Actions



SMP Programming Models

- The abstract model is parallel RAM (PRAM)
  - Concurrent reads, writes *to the same address* are OK
    - » What does the latter mean?
  - Good for theoretical modeling, not for programming

- All SMP models are based on multiple threads
  - Each thread is largely a von Neumann model
    - » But how do they interact, access memory?

- All models define and enforce a memory model
  - Access ordering to *same* and *different* RAM locations
    - » From same and different threads

Time ordering of memory accesses

- **Time ordering of accesses to *a DRAM address***
  - **Accesses from a thread/core are (program) ordered**
  - **Accesses from different threads/cores are not ordered**
  - **The only guarantee of ordering is via synchronization**

- **Time ordering of accesses to different RAM addresses (same or different threads)**
  - ***Cannot be observed in general***
    - » ***Why for the same thread?***
  - **Can only be guaranteed via synchronization**

- **The "sequentially consistent" model**
  - **Memory accesses within a thread are strictly sequential**
  - **Accesses to memory are an interleaving of sequential accesses from each thread**
    - » **Access order for a thread remains the same in all interleavings**
    - » **All threads see the same ordering**

- **This is said to be preferred by programmers**

Parallel programming on SMPs

- Goal – find parallelism in a program to speed it up
- Approach - define concurrent tasks to run on each core
  - Divide data or program into independent "chunks"
    - » Data parallelism or thread parallelism
    - » A chunk = unit of work to be done sequentially
  - Assign each chunk to a different core to run in parallel
- Data parallelism is easiest to grasp
  - Independent operations on multiple data elements
    - » find all occurrences of x in an array, add two arrays, etc.
  - Sometime operations are dependent, e.g. array sum
    - » x = x + A[i]
      - Can something be done in parallel here?
    - » Update all elements of a linked list
      - Is there parallelism here?

Data Parallel Programming

- The program remains quite similar to sequential
  - Each processor runs the same code on a subset of data
    - Hence data parallelism
  - A sequential example

```
for (i=1; i< N; i++)
    A[i] = B[i] * C[i]
```

  - A data parallel version

```
for (i=1; i< N; i+Chunksize)
    spawn sum( i, ChunkSize, A[i], B[i], C[i]);
```

    - Assume “spawn” means “create a new thread”
- What is a good chunk size?
  - Application and system dependent
  - Large chunks – less parallel work, less contention
  - Small chunks – more parallelism, larger overhead, contention

Thread-Parallel Programming

- This is more general than Data-parallel
  – Threads may execute different code on different data

- A sequential version of main

```
…
call func1(…)
…
call func2(…)
```

- A thread parallel version

```
…
rc1 = pthread_create( tid1,  NULL, func1, (void *) arg1)
…
rc2 = pthread_create( tid2,  NULL, func2, (void *) arg2)
…
pthread_join(tid1,NULL)
pthread_join(tid2,NULL)
```

- **A pthread exits when it completes func execution**
  - **Or executes pthread_exit**
- **A program is finished when all threads finished**

Another way to look at parallel programming

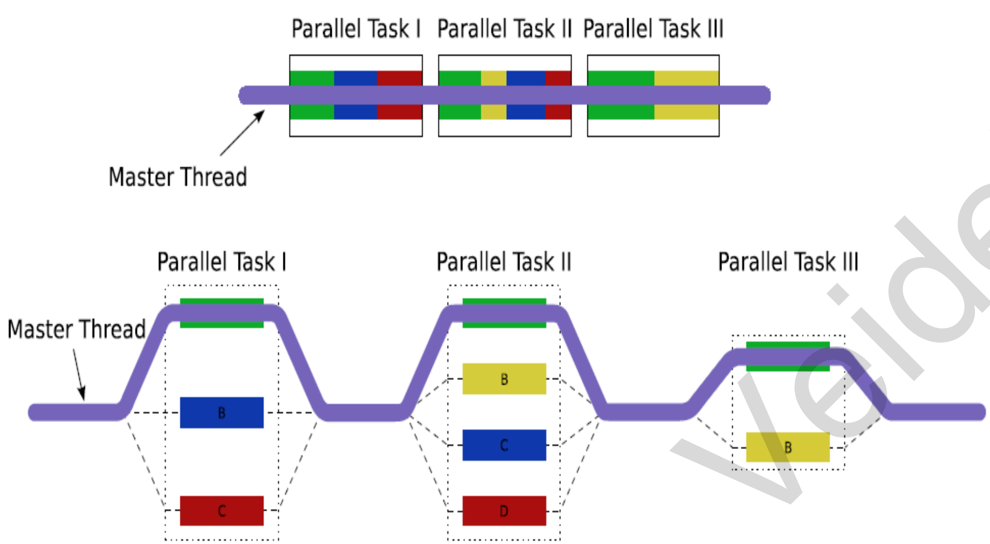
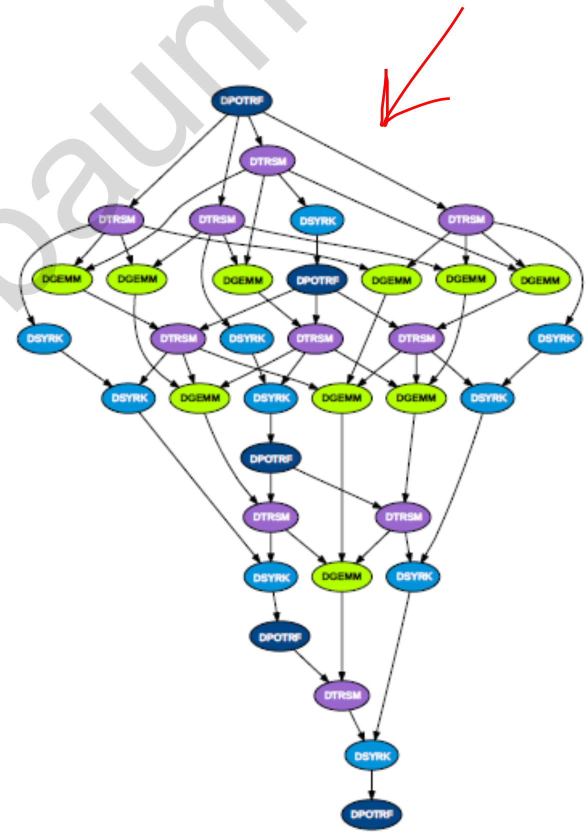


figure from Wikipedia: "OpenMP"

fork-join



dag

OpenMP

- Thread programming is difficult and low-level
- Another approach is to use OpenMP
  - Another standard, simplifies loop-based parallelization
    - » Can also do tasks
  - Based on Prof. Kuck's work on parallel loops at UIUC
- A user inserts directives (pragmas) in her code

```
#pragma omp parallel for schedule(static, ChunkSize)
for (i=1; i< N; i++)
      A[i]  = B[i] * C[i]
```

  - Compiler does the rest

- A "parallel for" must have independent iterations
- Has an implied barrier at the end (unless "nowait")

Data Dependence

- **Independent = no data dependencies across iterations**
  - A programmer needs to understand and satisfy/resolve data dependencies between program or data chunks

- **Data dependence: output, data, and anti- dependencies**
  - Data (flow) dependence (between iterations i and i+3)

```
for (i=1; i< N; i++)
    A[i] = …
    …     = A[i-3]
```

  - Output dependence (between all iterations)

```
for (i=1; i< N; i++)
    x = x + A[i]
```

  - Anti-dependence (use A[i+3] before iteration i+3 overwrites it)

```
for (i=1; i< N; i++)
    …   = A[i+3]
    A[i] = …
```

a data race

- **OpenMP *parallel for* cannot have any dependencies**
  - A compiler does not check!

- **Programming approach:**
  - Select compute intensive loops in the code
  - Make sure there are no dependencies between loop iterations
  - Apply omp constructs

- **An "omp parallel for" loop has an implicit barrier at the end**
  - Waits for all tasks to finish

OpenMP

- **C/C++/Fortran compilers support OpenMP**
  - e.g. Intel ICC, GNU GCC, MS Visual Studio, LLVM
    - » A compiler cannot check if the resulting parallel code is correct
- **A programmer defines number of tasks**
  - Env. variable, dynamically in code
- **OpenMP run-time schedules tasks**
  - relies on pthreads and kernel scheduler
  - thread to core mapping can be forced to stay fixed (*affinity)*
  - Implements multiple scheduling algorithms
    - » Static, dynamic, guided

OpenMP

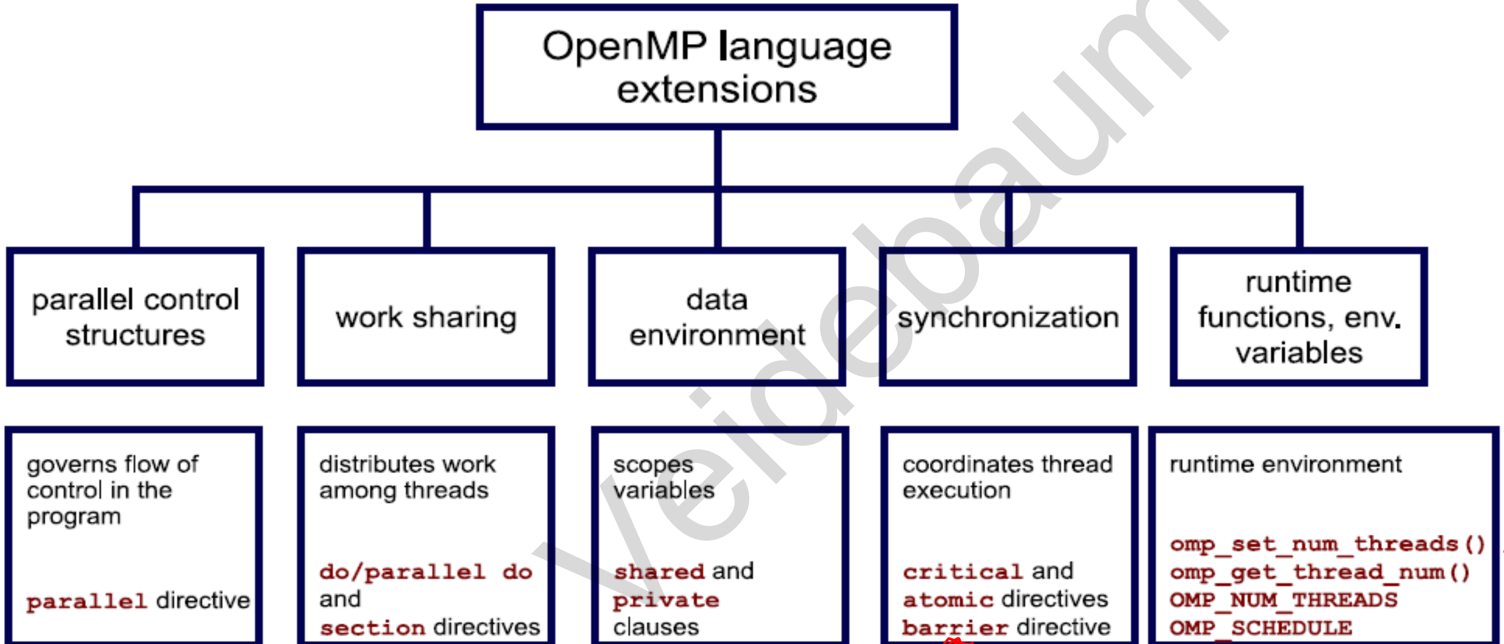


figure from Wikipedia: "OpenMP"

threads launched

Pragma

lib func

OpenMP 2.5+

Current OpenMP version is 5

OMP matrix transpose and scale

```
void transpose(int *M, int X, int n) {
    #pragma omp parallel
    {
      int i, j;
      #pragma omp for schedule(dynamic,20)
      for (i=0; i<n; i++) {
          for (j=i+1; j<n; j++) {
              M[i,j] =  M[j,i] * X
             }
           }
    }
 }
```

- Tasks are created
- Shared variables
  - M[], X, n
- Private variables
  - i,j
- Scheduling options
  - static, dynamic, etc
  - chunk size
- Implicit barrier where tasks join

OpenMP compilation, execution

- **Compilation**

  $ gcc -fOpenMP my_OpenMP.c -o my_OpenMP

  NOTE: the compiler ignores the OpenMP directives if –fOpenMP is omitted

- **Execution**

  $ export OMP_NUM_THREADS=3

  $ ./my_OpenMP

Cilk

- **Developed by Leiserson at MIT, starting in 1994**
- **An extension to C**
  - **for constructing multithreaded parallel programs**
  - **Uses a set of keywords: *cilk, spawn, sync,* etc.**
- **Uses a source-to-source compiler and a standard C compiler**
- **Intel TBB is based on CILK**

Cilk

- **Software developer exposes parallelism**
  - **identifies functions free of side effects, which can be treated as independent tasks and executed in parallel.**
    - » **annotate such functions with the *cilk* keyword**
    - » **invoke them with the *spawn* keyword**
  - **Places the *sync* keyword which indicates that execution of a function/main cannot proceed until all previously spawned functions have completed and returned their results to the parent**

Cilk

- **Runtime takes care of task distribution/scheduling**
  - **Tasks are stored in dequeues, one per processor**
  - **Employs *work stealing***
    - » **each processor fetches tasks from the top of its own queue**
    - » **when a processor queue is empty it picks another processor at random and "steals" a task from the bottom of its dequeue,**
      - **LIFO order**

- **While OMP parallel for does what is called *work sharing***

- **Work stealing is better for dynamic workloads**
  - **DAGs**
  - **Widely varying execution times per task**

cilk example: Fibonacci series

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

```
cilk Int fib (int n) {
  int x, y;
  if (n<2) return n;
  x = spawn fib (n-1);
  y = spawn fib (n-2);
  sync;
  return x+y;
  }
```

- Sample output

```
fib(1)=1
fib(2)=1
fib(3)=2
fib(4)=3
fib(5)=5
fib(6)=8
fib(7)=13
fib(8)=21
fib(9)=34
fib(10)=55
```

- **Compile**

  $ cilkc -O3 fib.cilk -o fib

- **Execution**

  $ ./fib –nproc 4 30

cilk example: Fibonacci series

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

```
cilk Int fib (int n) {
   int x, y;
   if (n<2) return n;
   x = spawn fib (n-1);
   y = spawn fib (n-2);
   sync;
   return x+y;
   }
```

- Sample output

  fib(1)=1
  fib(2)=1
  fib(3)=2
  fib(4)=3
  fib(5)=5
  fib(6)=8
  fib(7)=13
  fib(8)=21
  fib(9)=34
  fib(10)=55

  ⋮

- **Compile**

  $cilkc -O3 fib.cilk -o fib

- **Execution**

  $./fib –nproc 4 30

In general the software developer must restructure/express the code in a "*divide & conquer*" fashion.
The topology of the dag is implicitly conveyed by the placement of the *spawn* keywords.