Parallel Programming

ACSE-6: Lecture OpenMP-2

Adriana Paluszny
Royal Society University Research Fellow / Senior Lecturer

Memory (Reminders)

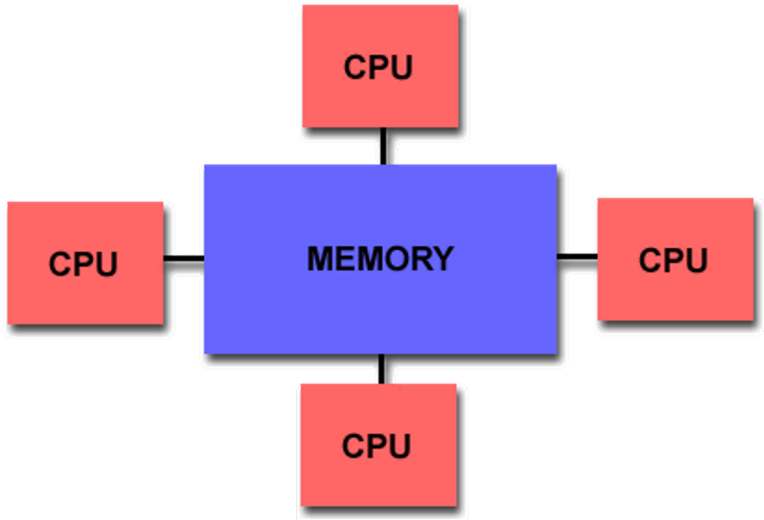
- Memory allocation in terms of stack and heap is not specified in the C++ standard.
- Instead, the standard distinguishes *automatic* and *dynamic* storage duration.
- **Local variables** have automatic storage duration and compilers store them on the **stack**.
- Objects with dynamic memory allocation (created with new) are stored on the free store, conventionally referred to as the heap. In languages that are not garbage-collected, objects on the heap lead to memory leaks if they are not freed.

When threads are created

- When threads are created they are assigned their own **Stack**
- When threads are created they share the **Heap**

- Remember: **Stack** is used for static memory allocation **and** **Heap** for dynamic memory allocation, both stored in the computer's RAM.
- Remember: **Stack** is faster than the **Heap**
- Remember: the sizes of the **Stack** and **Heap** depend on many factors
- **Stack** is accessed through a last-in, first-out (LIFO) memory allocation system. **Heap** Space exists as long as the application runs and is **larger** than **Stack**, which is temporary, but faster.

Shared Memory



- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.

Shared Memory : UMA vs. NUMA

Uniform Memory Access (UMA):

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA -Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

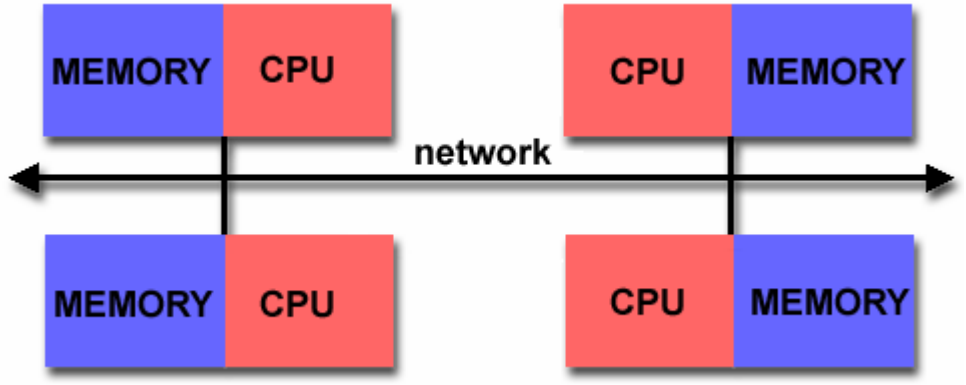
Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Shared Memory: Pro and Con

- Advantages
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

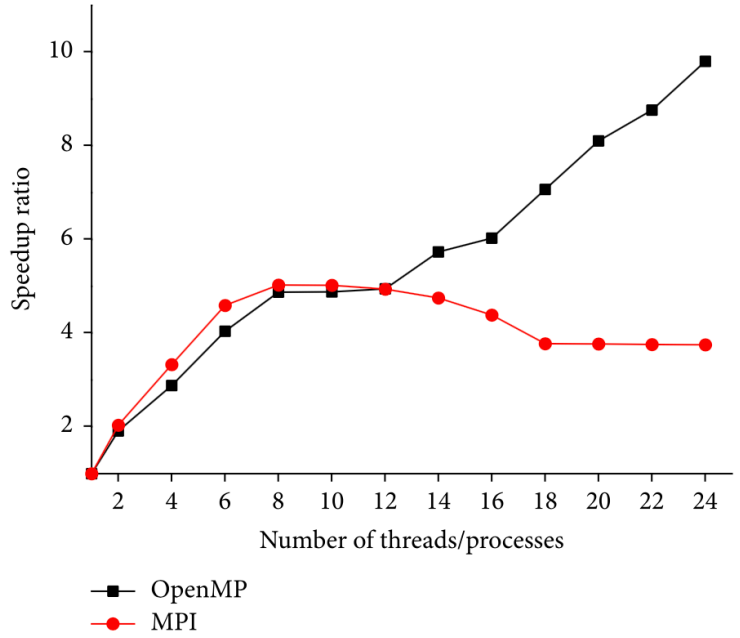


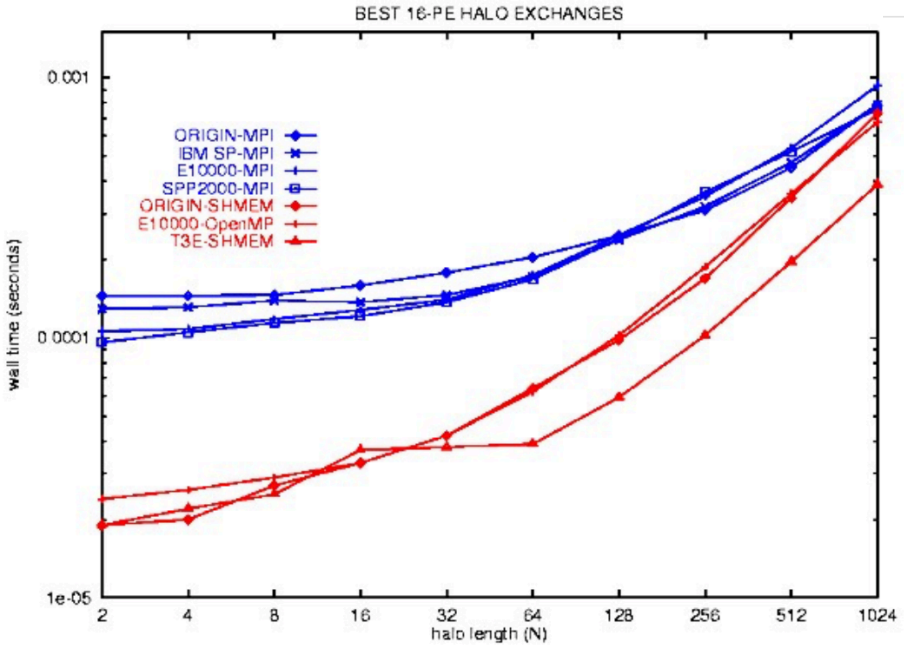
- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.

Distributed Memory: Pro and Con

- Advantages
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

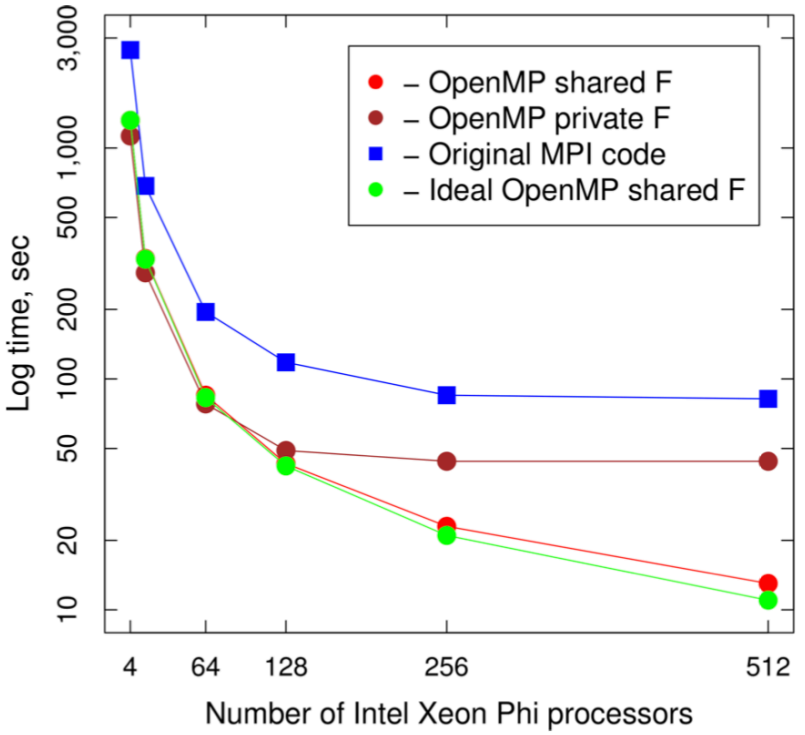
MPI vs OpenMP





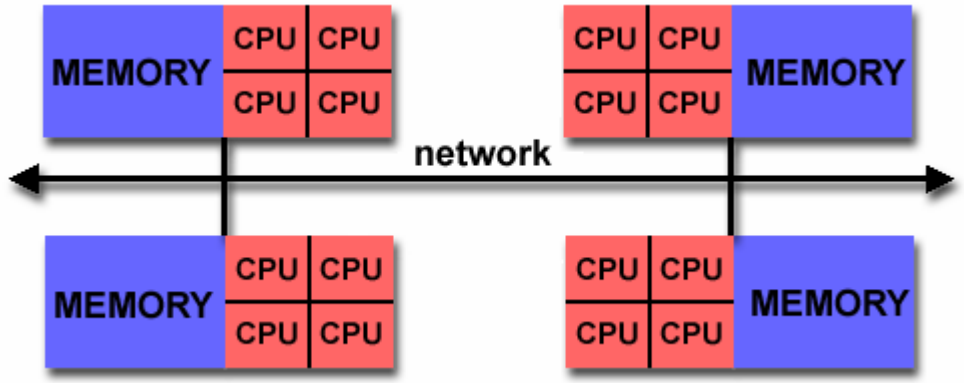
OpenMP is faster, simpler than MPI but it is more difficult/expensive to scale in terms of hardware

MPI/OpenMP parallelization of the Hartree-Fock method



wallcraf@nlr.com

Hybrid Distributed-Shared Memory



- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

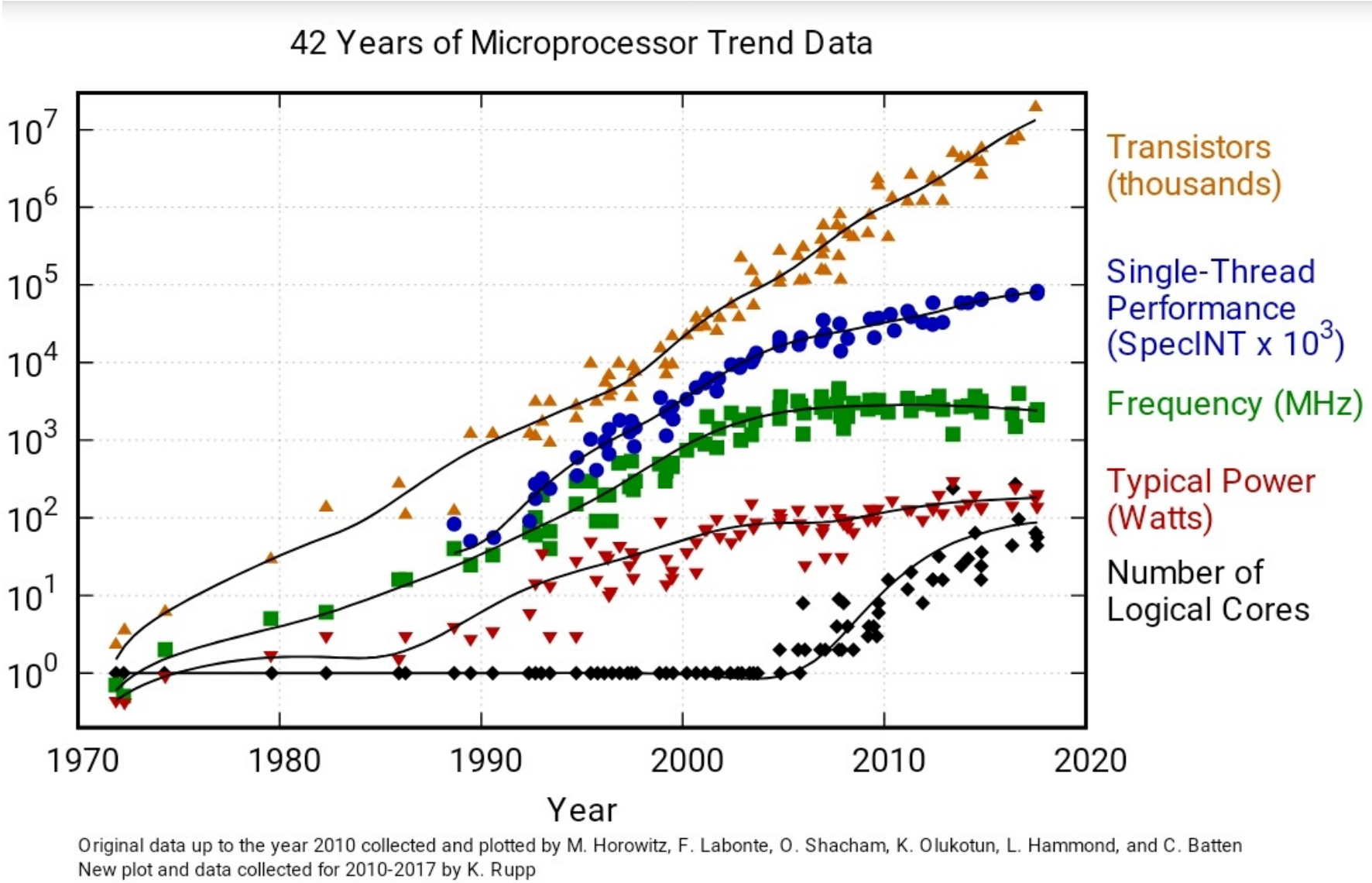
Shared Memory vs. Distributed Memory

- Tools can be developed to make any system appear to look like a different kind of system
  - distributed memory systems can be programmed as if they have shared memory, and vice versa
  - such tools do not produce the most efficient code, but might enable portability
- However, the most natural way to program any machine is to use tools and languages that express the algorithm explicitly for the architecture.

Brief Recap

- Why we need to learn both (MPI+OpenMP)

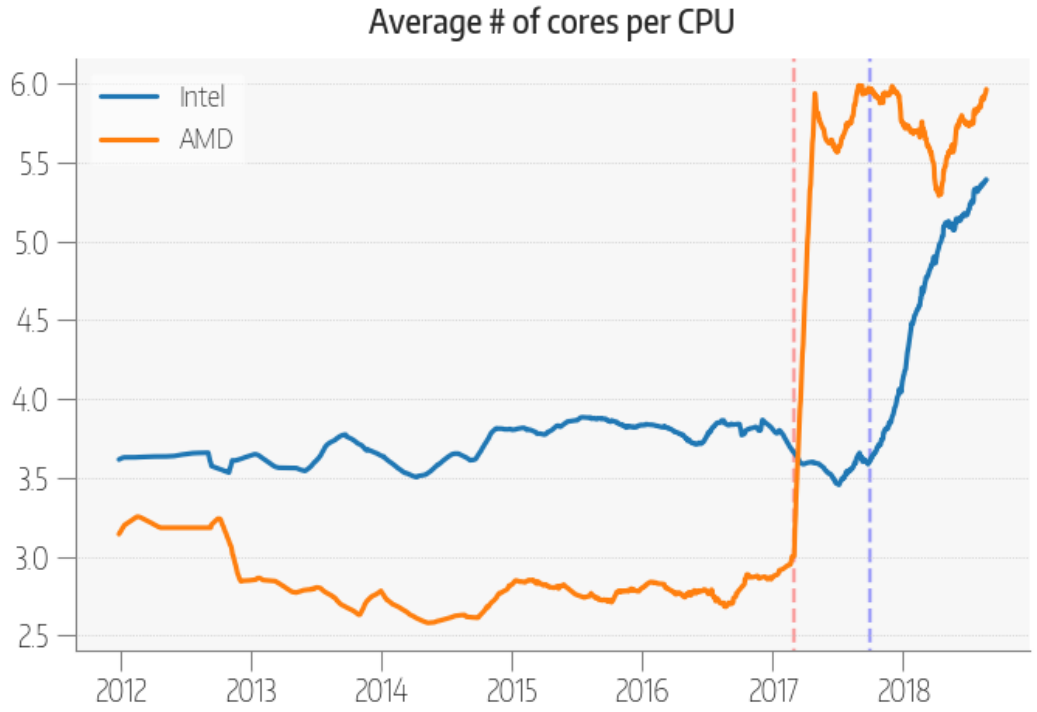
Serial Computing is not moving fast enough

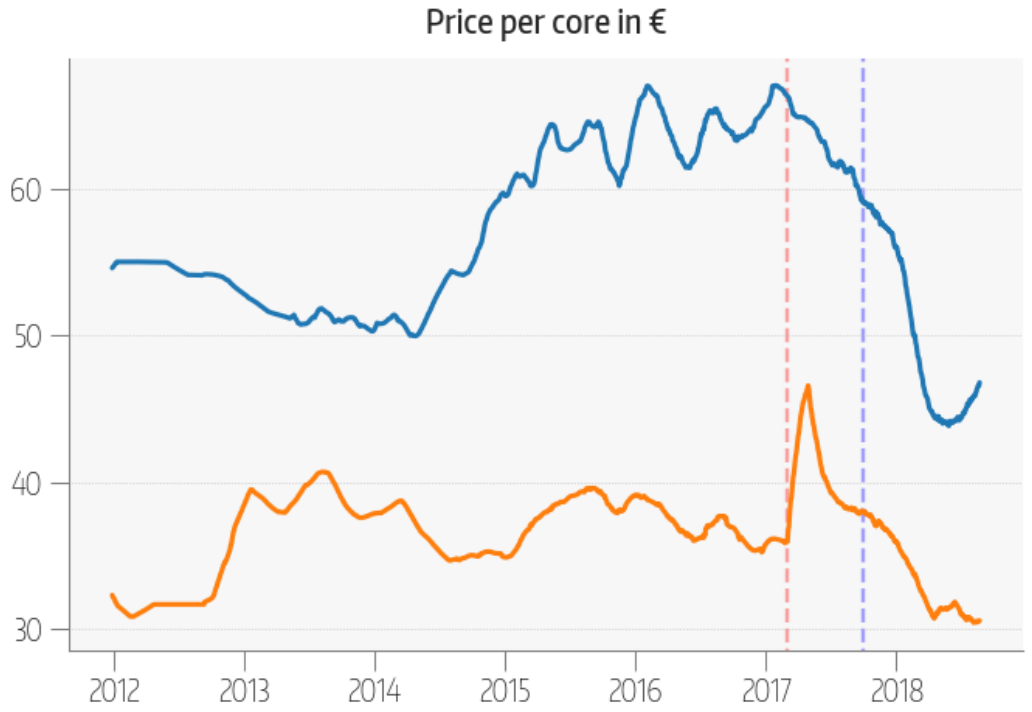


War of the cores

▶ The Core War: AMD versus Intel

After years of standstill, AMD's release of the Zen architecture in March 2017 forced Intel to follow up with Coffee Lake in October 2017, bringing 6 cores to the mainstream. Also, it triggered a price war and forced Intel to reduce prices per core to new lows. For CPUs based on AMD's Bulldozer architecture, `modules` are counted as cores.

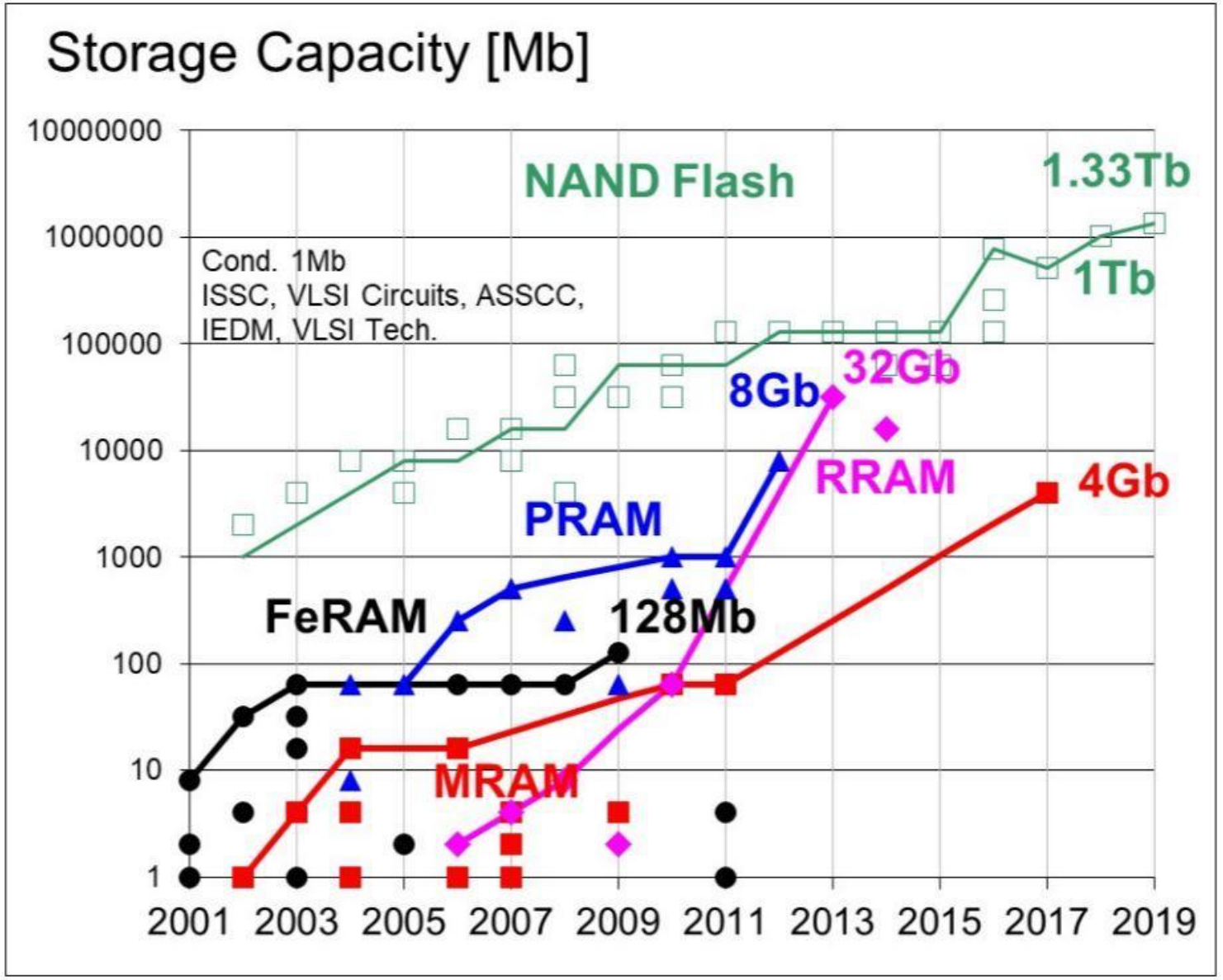




COMPILED BY: REDDIT.COM\USER\INGEBOR @ 08/22/2018

DATA SOURCE: MINDFACTORY.DE @ 1,148,180 TRACKED CPU SALES SINCE 2011

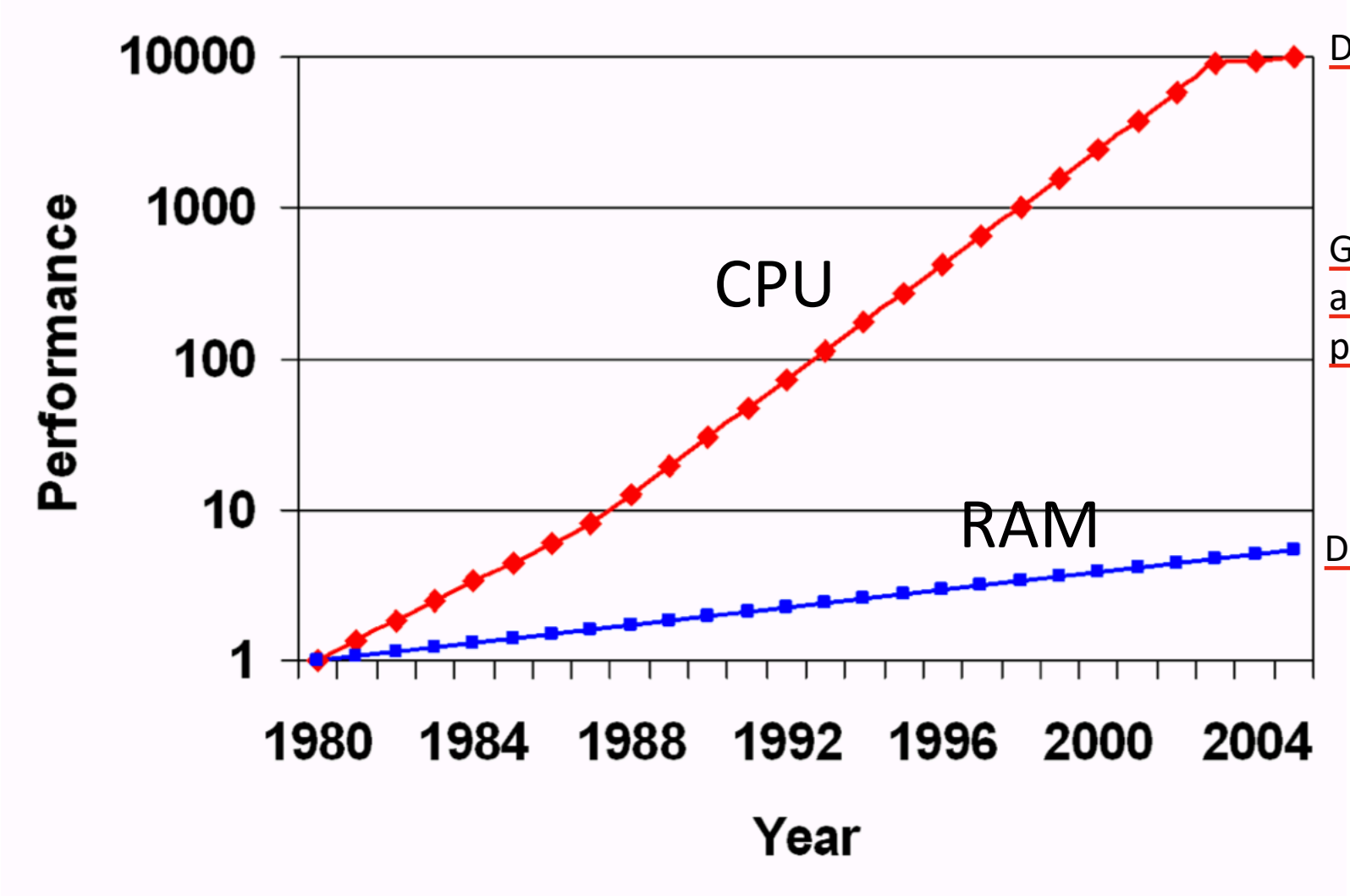
RAM Capacity



Rapid progress in RAM storage capacity – 3D stacked NAND, flash memory outruns competitors.

Source: ISSCC 2019

Processor-Memory performance gap



Double(d?) every 2 years

Gap grows approx. 50% per year

Doubles every 10 years

Programming Parallel Computers

- Programming single-processor systems is (relatively) easy because they have a single thread of execution and a single address space.
- Programming shared memory systems can benefit from the single address space
- Programming distributed memory systems is the most difficult due to multiple address spaces and need to access remote data
- Both shared memory and distributed memory parallel computers can be programmed in a data parallel, SIMD fashion and they also can perform independent operations on different data (MIMD) and implement task parallelism.

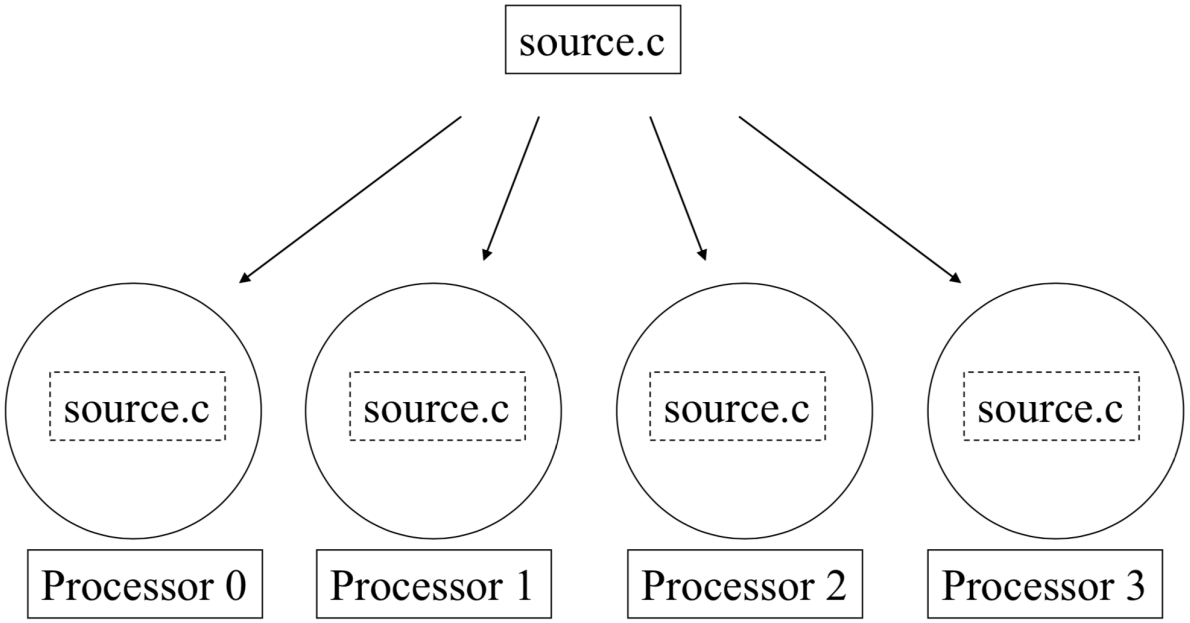
Single Program, Multiple Data (SPMD)

- SPMD: dominant programming model for shared and distributed memory machines.

– One source code is written

– Code can have conditional execution based on which processor is executing the copy

– All copies of code are started simultaneously and communicate and sync with each other periodically



Shared Memory Programming: OpenMP

- Shared memory systems (SMPs and cc-NUMAs) have a single address space:
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, 'SIMD' kinds of codes
  - OpenMP is the new standard for shared memory programming (compiler directives)
  - Vendors offer native compiler directives

OpenMP

- Model for shared memory parallel programming
- Portable across shared-memory architectures
- **Simple**
- Incremental parallelization
  - Parallelize individual computations in a programme while leaving the rest of the programme sequential
- Compiler based
  - Compiler generates thread programmes and synchronisation
- Extension to existing languages (Fortran, C, C++)
  - Mainly using directives (#pragma omp …)
  - Few library routines

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Avoid data races

How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

Most OpenMP programs only use these items

OpenMP pragma, function, or clause	Concepts
`#pragma omp parallel`	Parallel region, teams of threads, structured block, interleaved execution across threads
`int omp_get_thread_num()` `int omp_get_num_threads()`	Create threads with a parallel region and split up the work using the number of threads and thread ID
`double omp_get_wtime()`	Timing blocks of code
`setenv OMP_NUM_THREADS N` `export OMP_NUM_THREADS=N`	Set the default number of threads with an environment variable

`#pragma omp barrier`
`#pragma omp critical`
`#pragma omp atomic`
	Synchronization, critical sections

`#pragma omp for` `#pragma omp parallel for`	Worksharing, parallel loops
`reduction(op:list)`	Reductions of values across a team of threads
`schedule(dynamic [,chunk])` `schedule(static [,chunk])`	Loop schedules
`private(list), shared(list),` `firstprivate(list)`	Data environment
`#pragma omp master` `#pragma omp single`	Worksharing with a single thread
`#pragma omp task` `#pragma omp taskwait`	Tasks including the data environment for tasks.

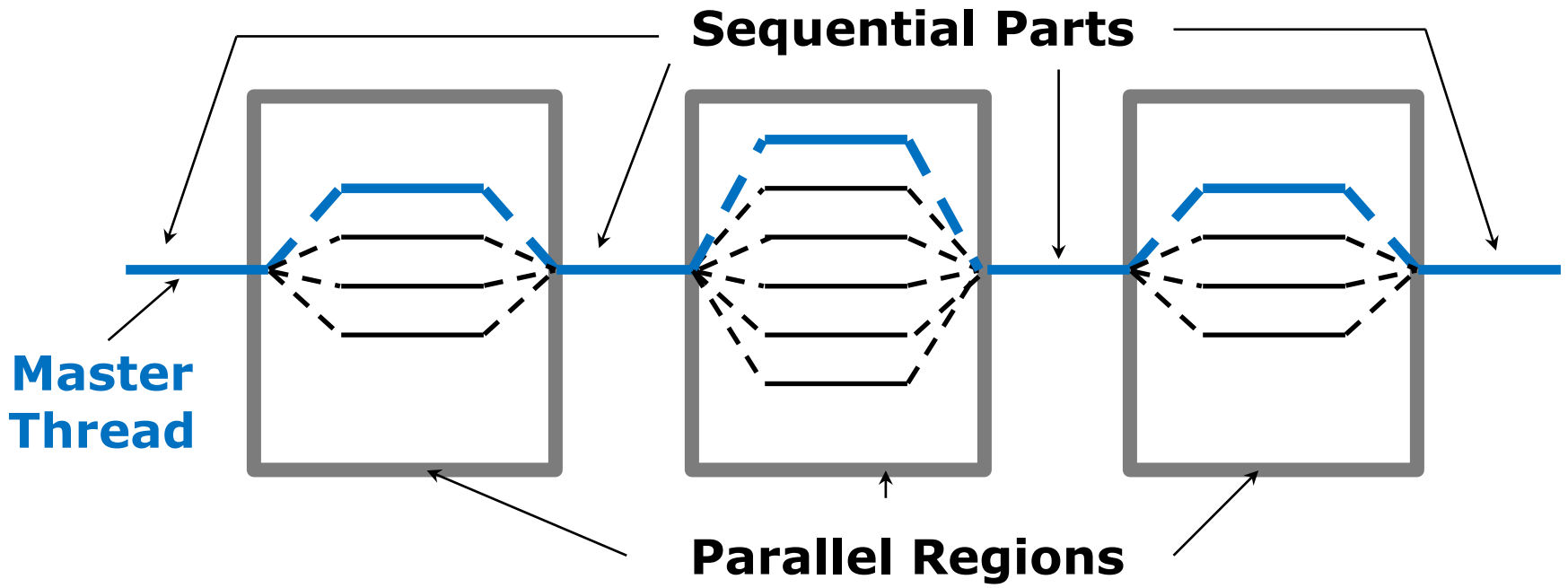
Credits: Tim Mattson

Scope

- In serial programming, the scope of a variable consists of the parts of a program where the variable can be used
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable
- By default:
  - All variables that are visible at the beginning of a parallel block are shared across threads
  - All variables defined inside a parallel block are private to each thread

OpenMP Programming Model:

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



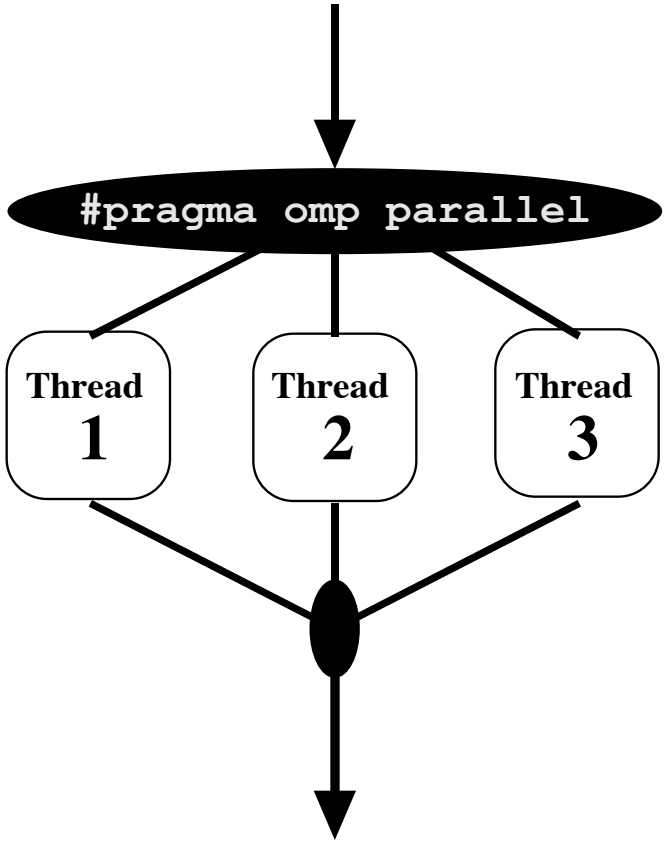
OpenMP Pragma Syntax

- Most constructs in OpenMP* are compiler directives or pragmas.
    - For C and C++, the pragmas take the form:

#pragma omp *construct [clause [clause]…]*

Parallel regions

- Defines parallel region over structured block of code
- Threads are created as **'parallel'** pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise



C/C++ :

```
#pragma omp parallel
{
        commands
}
```

How Many Threads?

- Set environment variable for number of threads

```
set OMP_NUM_THREADS=4
```

- There is no standard default for this variable
  - Many systems:
    - # of threads = # of processors
    - Intel® compilers use this default

For loops

- Splits loop iterations into threads
- Must be in the parallel region
- Must precede the loop

```
#pragma omp parallel
#pragma omp for
      for (i = 0; i < n; i++)
      {
        call_function(i);
      }
```

Types of variables

In a parallel section variables can be private (each thread owns a copy of the variable) or shared among all threads. Shared variables must be used with care because they cause race conditions.

- **`shared`**: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- **`private`**: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.

Variables scope in OpenMP

- **`shared(x)`** – all threads access the same memory location
- **`private(x)`**
  - each thread has its own private copy of x
  - all local instances of x are not initialized
  - local updates to x are lost when exiting the parallel region
  - the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)
- **`firstprivate(x)`**
  - each thread has its own private copy of x
  - all copies of x are initialized with the current value of x
  - local updates to x are lost when exiting the parallel region the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)
- **`default(shared)`** or **`default(none)`**
  - affects all the variables not specified in other clauses
  - default(none)ensures that you must specify the scope of each variable used in the parallel block that the compiler can not figure out by itself (highly recommended!!)

Synchronization

OpenMP lets you specify how to synchronize the threads. Options:

- **`critical`**: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- **`atomic`**: the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
- **`ordered`**: the structured block is executed in the order in which iterations would be executed in a sequential loop
- **`barrier`**: each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- **`nowait`**: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Task

- OpenMP allows the application to create specific tasks

- A **`task`** is composed of:
  - Code that will be executed
  - Data environment (inputs/outputs)
  - A location where the task will be executed (thread)

- 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 conceptual pool associated to the current parallel region

Task

- The task associated with a task construct will be executed only once
- A task is **tied** to the code if the code is executed by the same thread from beginning to end
- A task is **untied the code can be executed by more than one thread.**

Looking for a systematic description of OpenMP?

[see Intel slides]

[see OpenMP 5.0 standard]