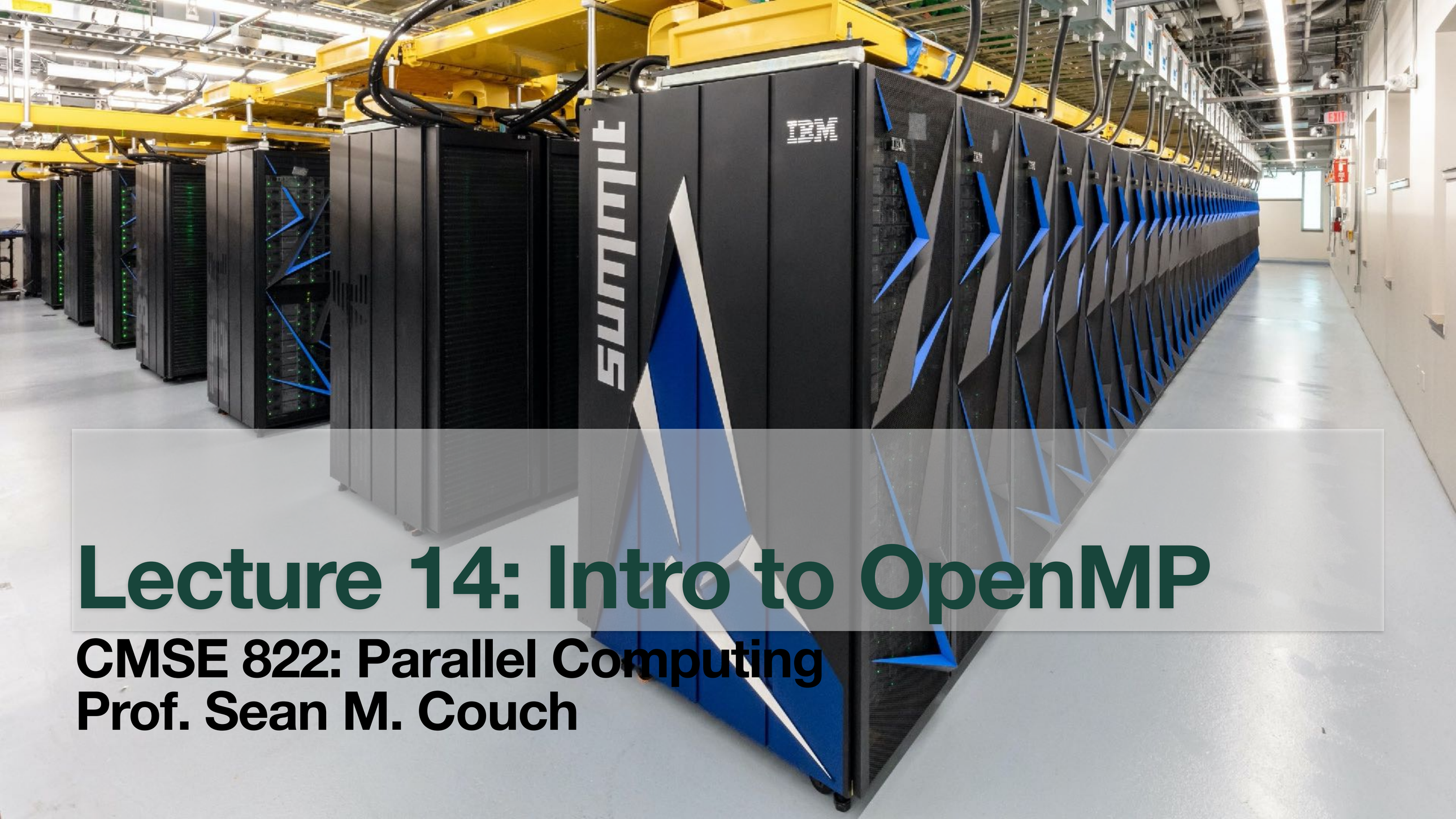
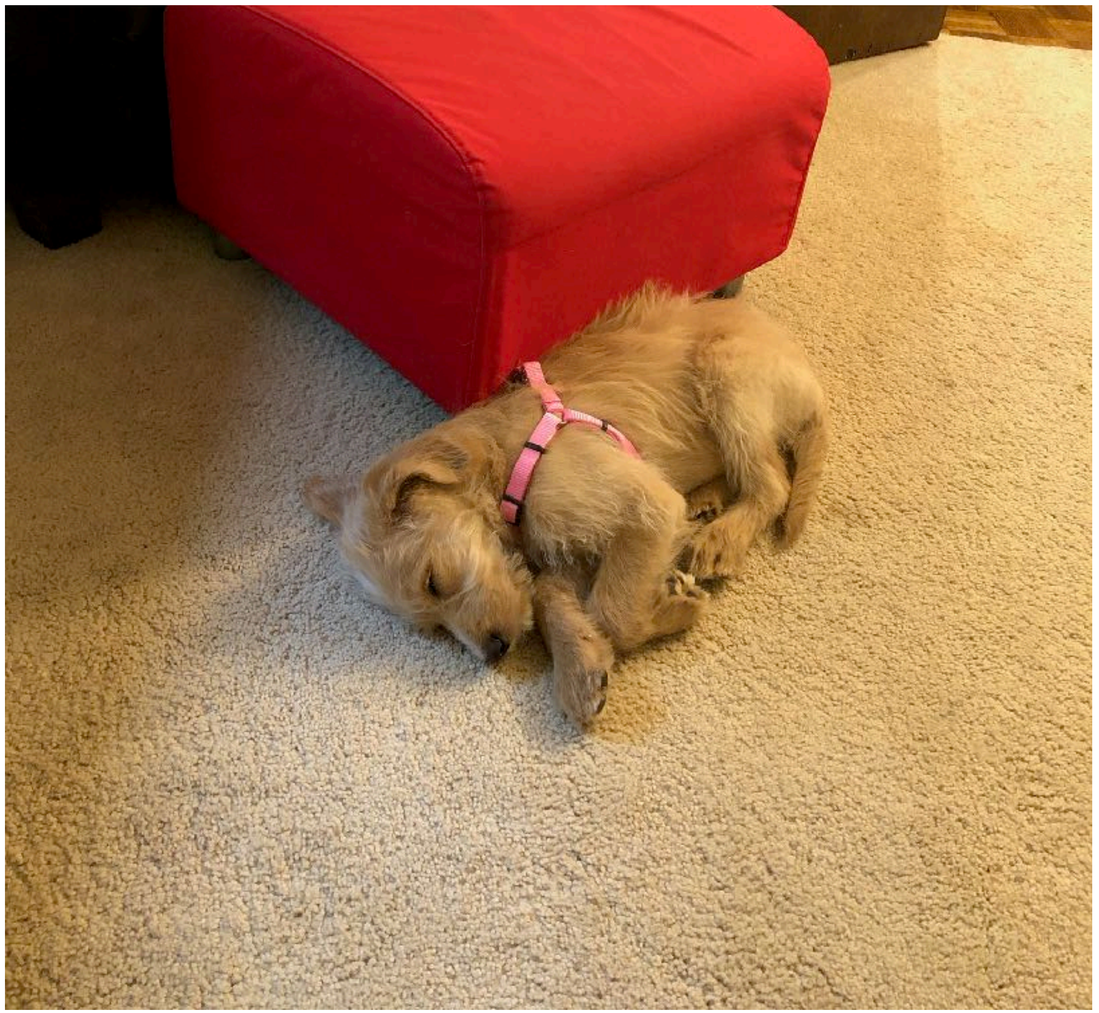


Lecture 14: Intro to OpenMP

CMSE 822: Parallel Computing

Prof. Sean M. Couch

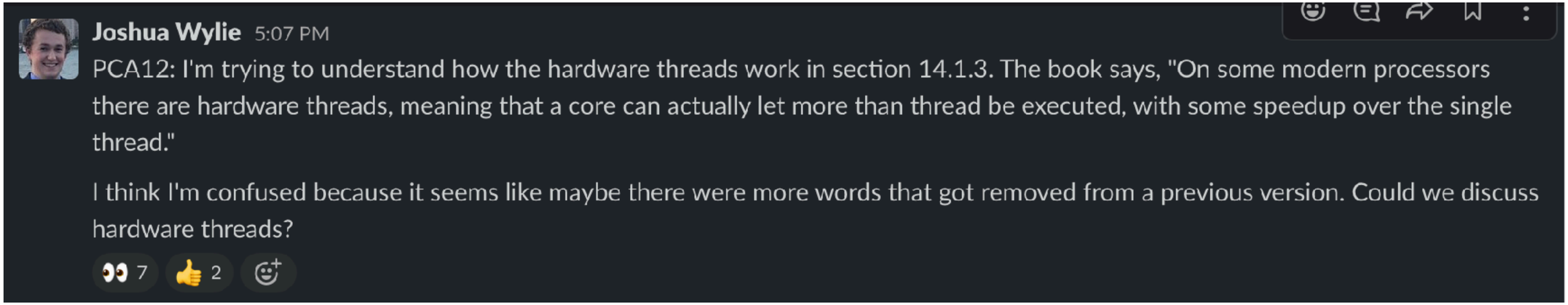
Puppy Time!



© S.M. Couch

PCA Questions

Hardware vs. “Hyper” threads



© S.M. Couch

PCA Questions

Hardware vs. “Hyper” threads

- “hyperthreading” is a special case of hardware threads
- “hardware” thread is a more generic term
- In either case, it means that thread state is duplicated multiple times per hardware core, making some number of “virtual” cores that are mapped to actual physical hardware

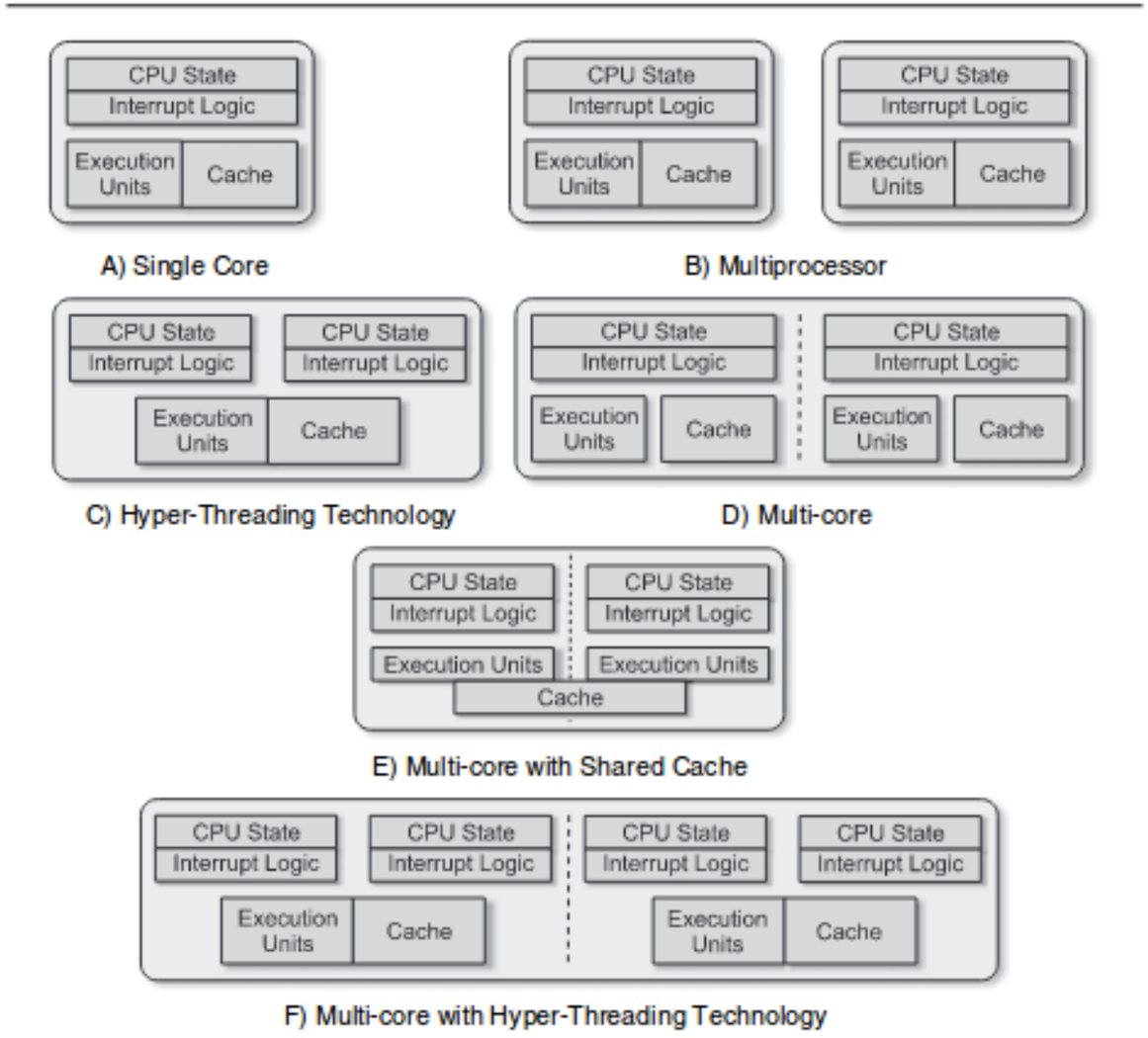


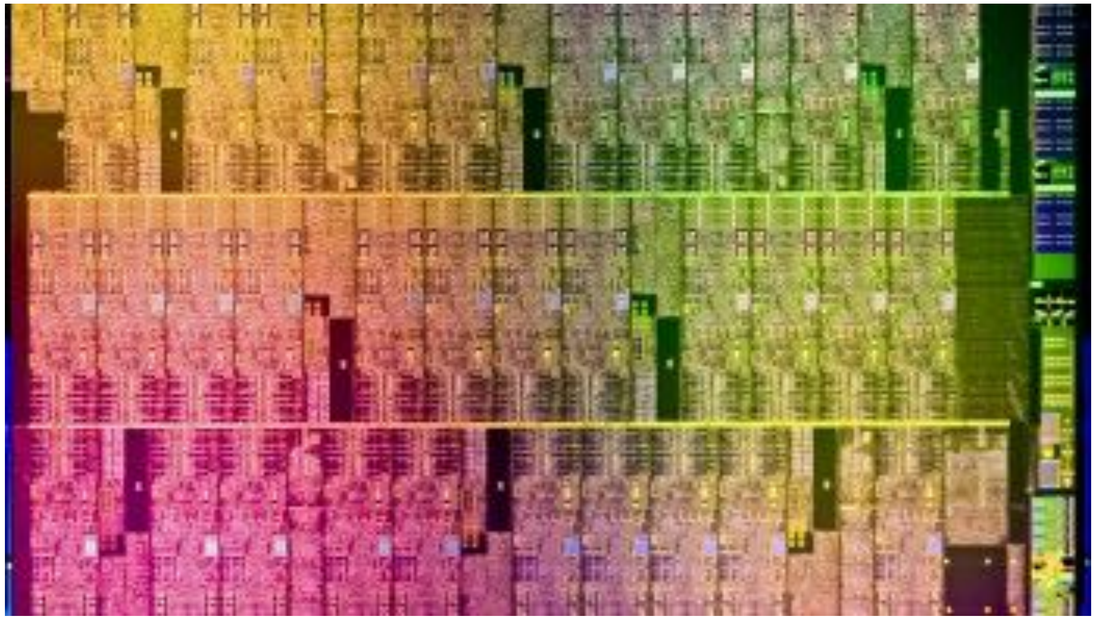
Figure 1.4 Simple Comparison of Single-core, Multi-processor, and Multi-Core Architectures

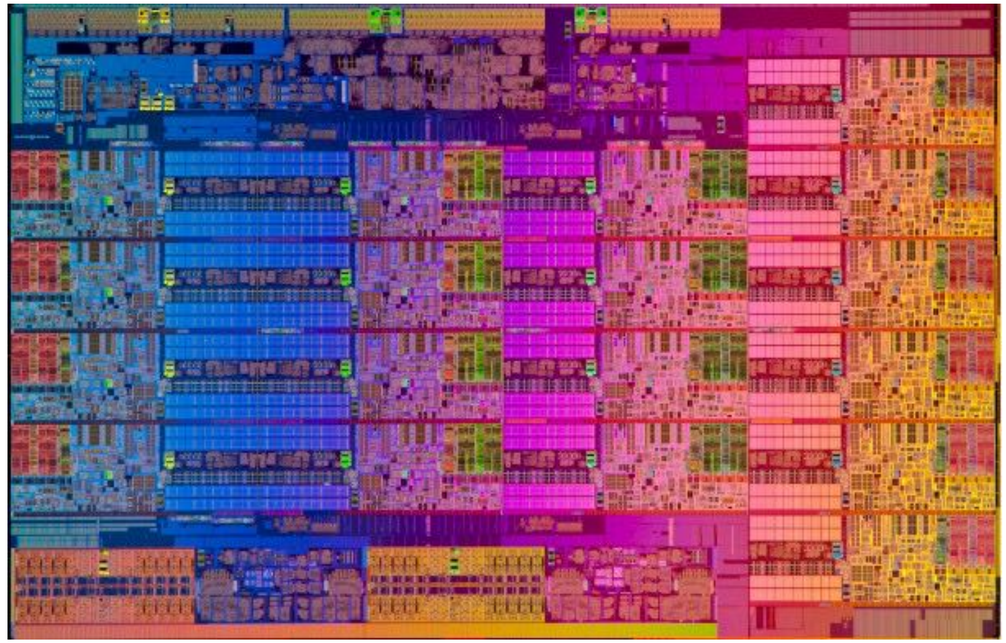
© S.M. Couch

PCA Questions

Hardware vs. “Hyper” threads

Intel Phi (Knights Corner) vs Intel Xeon E5 (Haswell)





~60 cores, “hardware” threading

~16 cores, “hyper” threading

© S.M. Couch

PCA Questions

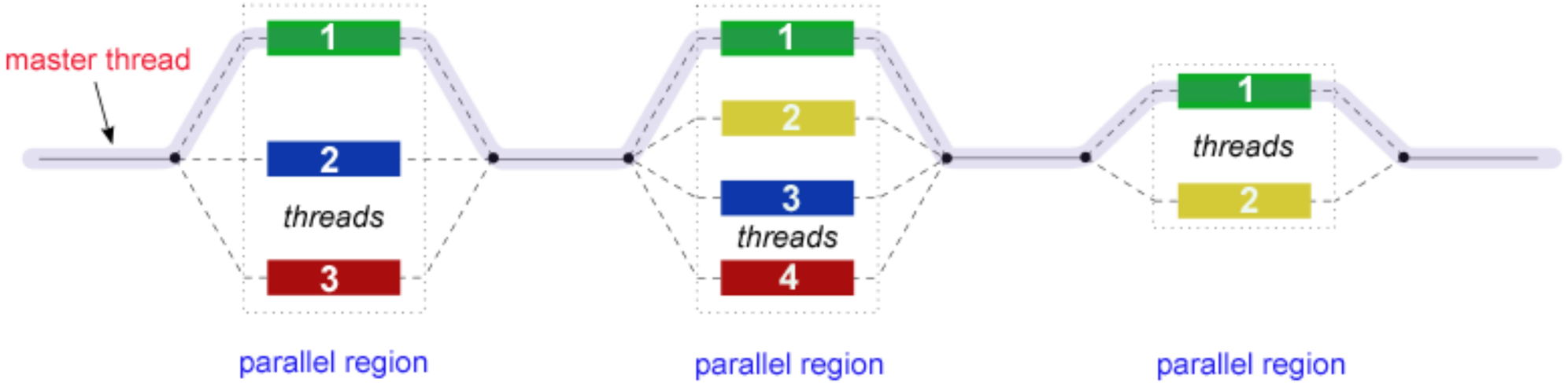
Hardware vs. “Hyper” threads

Intel Phi (Knights Corner)	vs	Intel Xeon E5 (Haswell)
• Multiple FPU pipes		• Multiple FPU pipes
• “in-order” execution		• “out-of-order” execution
• single thread will only use one pipe at a time		• can use multiple pipes with single thread

© S.M. Couch

PCA Questions

Dynamic thread spawning

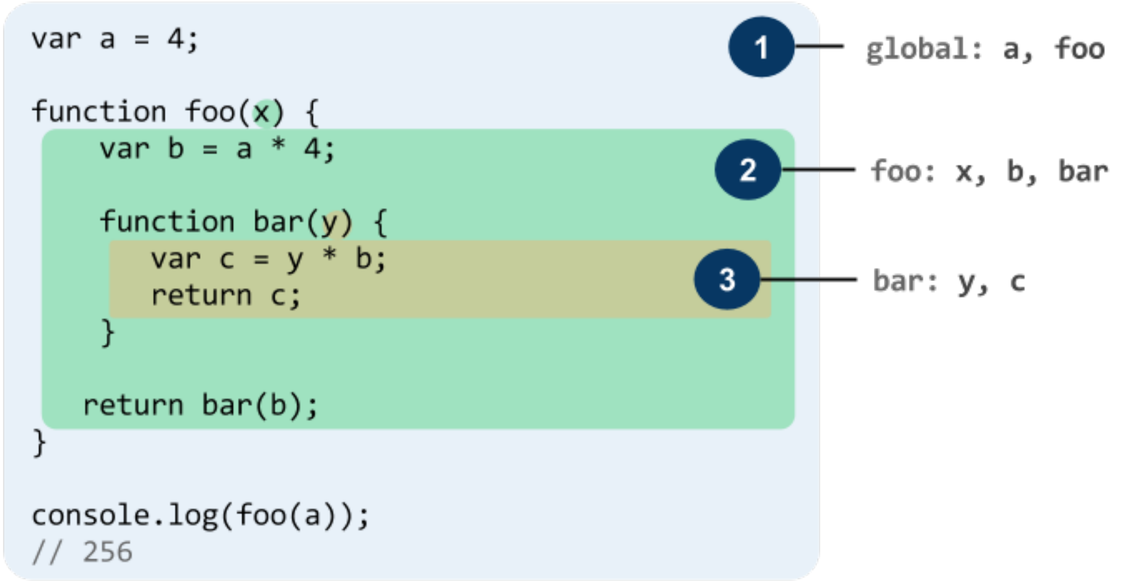


© S.M. Couch

PCA Questions

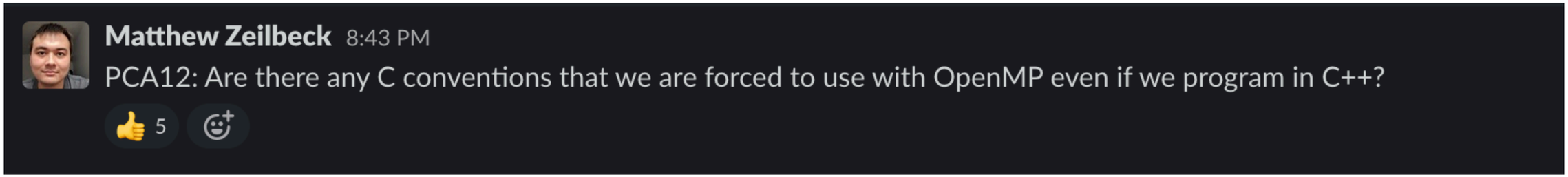
Time slicing and lexical scope

- Time slicing ~ “out-of-order” execution. Multiple threads work are interleaved in time on the core hardware
- Lexical scope: which regions of code can access certain variables



© S.M. Couch

PCA Questions



- No?

© S.M. Couch

OpenMP Tutorial

- See https://computing.llnl.gov/tutorials/openMP/
- openmp.org: standard quick reference sheets!

© S.M. Couch

OpenMP Tutorial

What is OpenMP?

OpenMP Is:

- An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory*** parallelism.

- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

- An abbreviation for: **Open Multi-Processing**

© S.M. Couch

OpenMP Tutorial

- **OpenMP Is Not:**
  - Meant for distributed memory parallel systems (by itself)
  - Necessarily implemented identically by all vendors
  - Guaranteed to make the most efficient use of shared memory
  - Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
  - Designed to handle parallel I/O. The programmer is responsible for synchronizing input and output.

© S.M. Couch

OpenMP Tutorial

- **Goals of OpenMP:**

  - **Standardization:**
    - Provide a standard among a variety of shared memory architectures/platforms
    - Jointly defined and endorsed by a group of major computer hardware and software vendors

  - **Lean and Mean:**
    - Establish a simple and limited set of directives for programming shared memory machines.
    - Significant parallelism can be implemented by using just 3 or 4 directives.
    - This goal is becoming less meaningful with each new release, apparently.

  - **Ease of Use:**
    - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
    - Provide the capability to implement both coarse-grain and fine-grain parallelism

  - **Portability:**
    - The API is specified for C/C++ and Fortran
    - Public forum for API and membership
    - Most major platforms have been implemented including Unix/Linux platforms and Windows

© S.M. Couch

OpenMP Tutorial

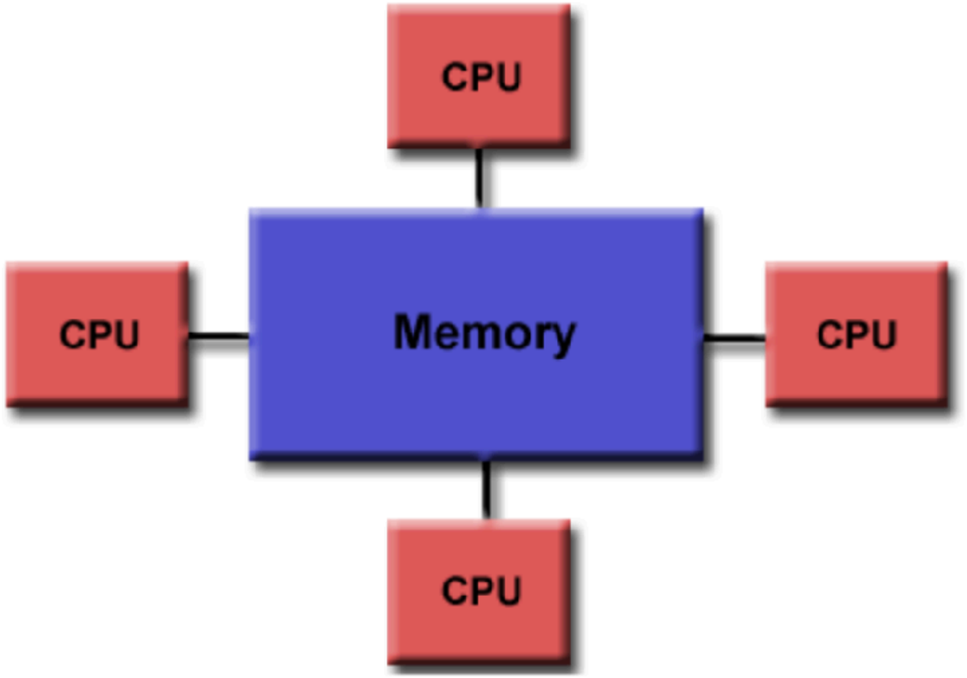
History:

- In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions:
  - The user would augment a serial Fortran program with directives specifying which loops were to be parallelized
  - The compiler would be responsible for automatically parallelizing such loops across the SMP processors
- Implementations were all functionally similar, but were diverging (as usual)
- First attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular.
- However, not long after this, newer shared memory machine architectures started to become prevalent, and interest resumed.
- The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off.
- Led by the OpenMP Architecture Review Board (ARB). Original ARB members and contributors are shown below. *(Disclaimer: all partner names derived from the OpenMP web site)*

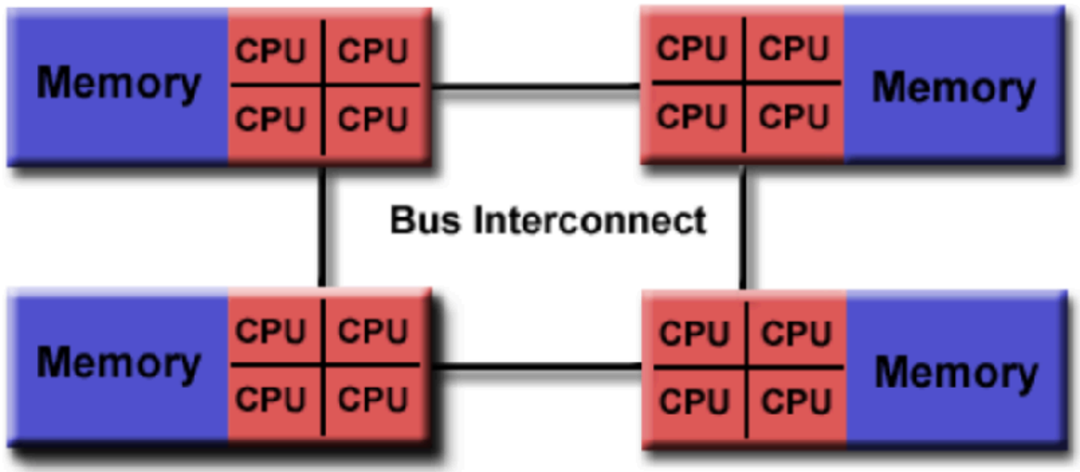
© S.M. Couch

OpenMP Tutorial

- **Shared Memory Model:**
  - OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.



Uniform Memory Access



Non-Uniform Memory Access

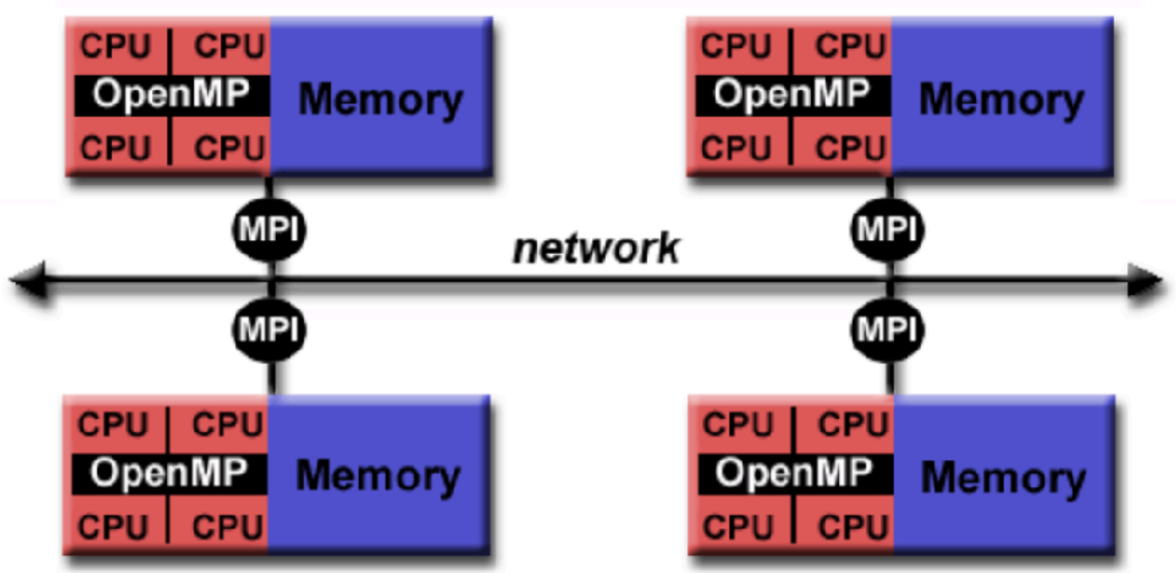
- Because OpenMP is designed for shared memory parallel programming, it largely limited to **single node** parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

© S.M. Couch

OpenMP Tutorial

Motivation for Using OpenMP in HPC:

- By itself, OpenMP parallelism is limited to a single node.
- For High Performance Computing (HPC) applications, OpenMP is combined with MPI for the distributed memory parallelism. This is often referred to as **Hybrid Parallel Programming**.
  - OpenMP is used for computationally intensive work on each node
  - MPI is used to accomplish communications and data sharing between nodes
- This allows parallelism to be implemented to the full scale of a cluster.



Hybrid OpenMP-MPI Parallelism

© S.M. Couch

OpenMP Tutorial

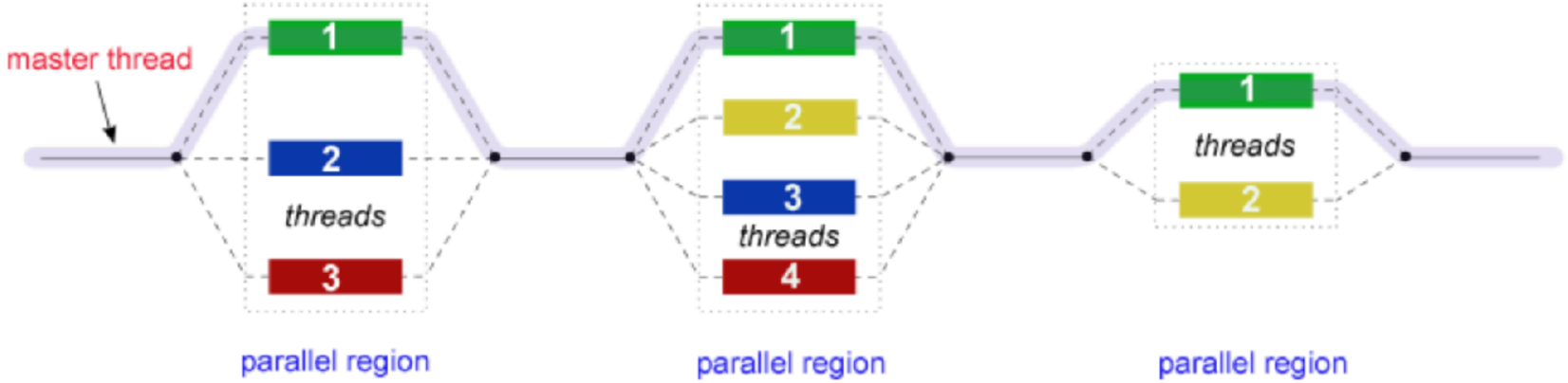
- **Thread Based Parallelism:**
  - OpenMP programs accomplish parallelism exclusively through the use of threads.
  - A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
  - Threads exist within the resources of a single process. Without the process, they cease to exist.
  - Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
- **Explicit Parallelism:**
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives....
  - Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

© S.M. Couch

OpenMP Tutorial

Fork - Join Model:

- OpenMP uses the fork-join model of parallel execution:



- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

© S.M. Couch

OpenMP Tutorial

- **Data Scoping:**
  - Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.
  - All threads in a parallel region can access this shared data simultaneously.
  - OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.
  - This topic is covered in more detail in the Data Scope Attribute Clauses section.
- **Nested Parallelism:**
  - The API provides for the placement of parallel regions inside other parallel regions.
  - Implementations may or may not support this feature.
- **Dynamic Threads:**
  - The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
  - Implementations may or may not support this feature.

© S.M. Couch

OpenMP Tutorial

- **I/O:**
  - OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
  - If every thread conducts I/O to a different file, the issues are not as significant.
  - It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.
- **Memory Model: FLUSH Often?**
  - OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
  - When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.

© S.M. Couch

OpenMP Tutorial

OpenMP API Overview

Three Components:

- The OpenMP 3.1 API is comprised of three distinct components:
  - Compiler Directives (19)
  - Runtime Library Routines (32)
  - Environment Variables (9)

  Later APIs include the same three components, but increase the number of directives, runtime library routines and environment variables.

- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

- Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

© S.M. Couch

OpenMP Tutorial

Compiler Directives:

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- Compiler directives have the following syntax:

```
sentinel      directive-name      [clause, ...]
```

For example:

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

© S.M. Couch

OpenMP Tutorial

Run-time Library Routines:

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines. For example:

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`
C/C++	`#include <omp.h>` `int omp_get_num_threads(void)`

- Note that for C/C++, you usually need to include the **<omp.h>** header file.
- Fortran routines are not case sensitive, but C/C++ routines are.

© S.M. Couch

OpenMP Tutorial

Environment Variables:

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy
- Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

© S.M. Couch

OpenMP Tutorial

C / C++ - General Code Structure

```
1   #include <omp.h>
2
3   main ()  {
4
5   int var1, var2, var3;
6
7   Serial code
8         .
9         .
10        .
11
12  Beginning of parallel region. Fork a team of threads.
13  Specify variable scoping
14
15  #pragma omp parallel private(var1, var2) shared(var3)
16     {
17
18     Parallel region executed by all threads
19                   .
20     Other OpenMP directives
21                   .
22     Run-time Library calls
23                   .
24     All threads join master thread and disband
25
26     }
27
28  Resume serial code
29        .
30        .
31        .
32
33  }
```


© S.M. Couch