Parallel Computing for Science & Engineering (PCSE 374C/394C)

OpenMP

Instructors:
Victor Eijkhout, Cyrus Proctor

OpenMP-- Overview

- Standard is ~15 years old.
- The "language" is easily comprehended.
  You can start simple and expand.
- Light Weight from System Perspective
- Very portable –GNU and vendor compilers.
- Spend time finding parallelism can be the most difficult part.
  The parallelism may be hidden.
- Writing Parallel OpenMP code examples is relatively easy.
- Developing parallel algorithms and/or parallelizing serial code is much harder.
- Expert level requires awareness of scoping and synchronization.

OpenMP executable runs on an SMP*

- Shared Memory systems:
  - One Operating System
  - Instantiation of ONE process
  - Threads are forked (created) from within your program.
  - Multiple threads on multiple cores

*SMP = Symmetric Multi-Processor:
The execution of the operating system
has equal access to any of the processors"

What is OpenMP (Open Multi-Processing)

- **De facto standard for Scientific Parallel Programming on Symmetric Multi-Processor (SMP) Systems.**

- **It is an API (Application Program Interface) for designing and executing parallel Fortran, C and C++ programs**
  - Based on threads, but
  - Higher-level than POSIX threads (Pthreads) (http://www.llnl.gov/computing/tutorials/pthreads/#Abstract)
- **Implemented by:**
  - **Compiler Directives**
  - **Runtime Library (interface to OS and Program Environment)**
  - **Environment Variables**
- **Compiler option required to interpret/activate directives.**

- **http://www.openmp.org/ has tutorials and description.**
- **Directed by OpenMP ARB (Architecture Review Board)**

OpenMP History

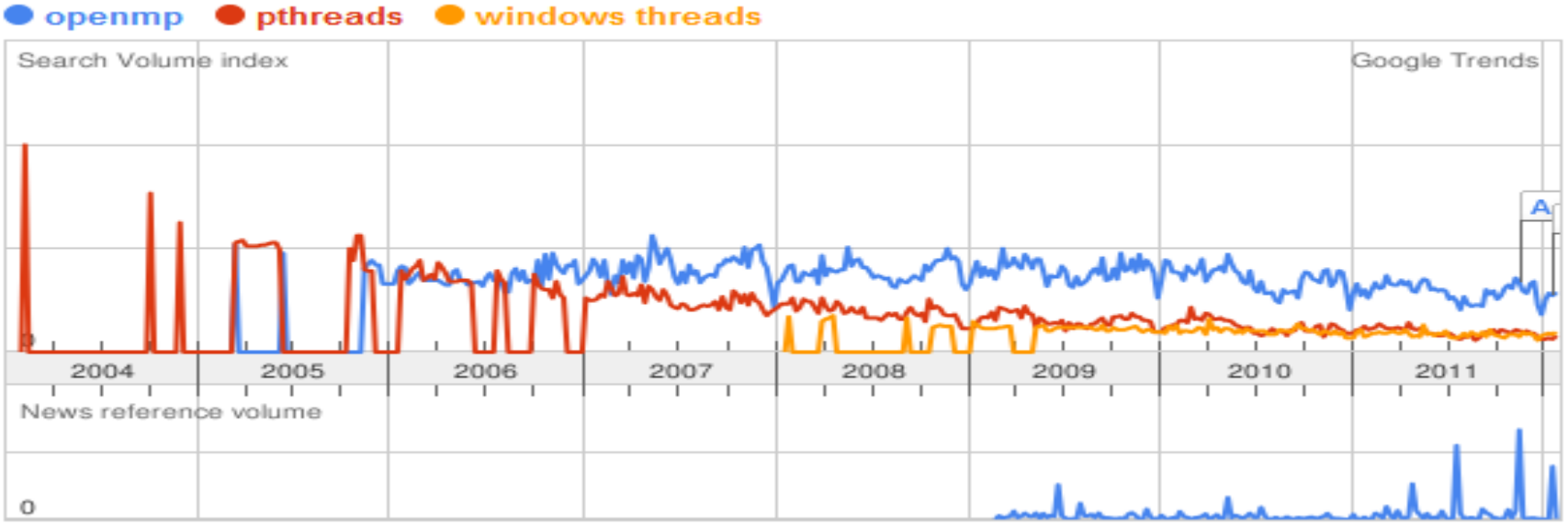
- Primary OpenMP participants

  AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, MS, TI, CAPS, NVIDIA

  ANL, LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH, TACC

- OpenMP Fortran API, Version 1.0, published Oct. 1997
- OpenMP C API, Version 1.0, published Oct. 1998
- OpenMP 2.0 API for Fortran, published in 2000
- OpenMP 2.0 API for C/C++, published in 2002
- OpenMP 2.5 API for C/C++ & F90 published in 2005
- OpenMP 3.0 Tasks published May 2008
- OpenMP 3.1 published July 2011
- OpenMP 4.0 Affinity, Accelerator Support July 2013

OpenMP History



OpenMP 3.0: The World is still flat, no support for NUMA (yet)!
OpenMP is hardware agnostic, it has no notion of data locality.
The Affinity problem: How to maintain or improve the nearness of threads and their most frequently used data.
Or:
Where to run threads?
Where to place data?

http://terboven.wordpress.com/

Thread binding is in OpenMP 4.0
(other techniques are already available)

Advantages/Disadvantages of OpenMP

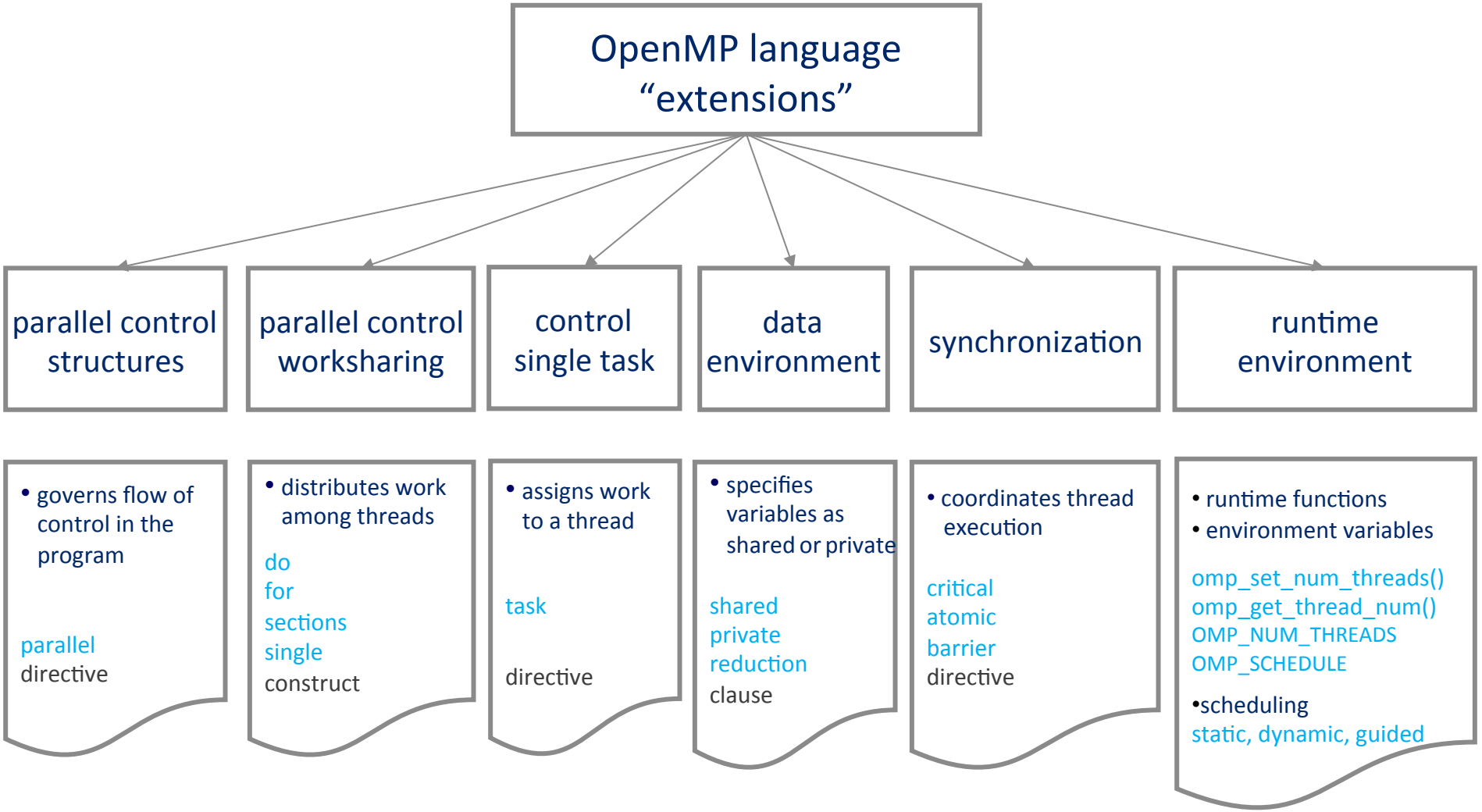
- Pros
  - Shared Memory Parallelism is easier to learn.
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
  - Converting serial code to OpenMP parallel can be easier than converting to MPI parallel.
  - SMP hardware is prevalent now.
    - Supercomputers **and** your desktop/laptop
    - GPUs (Graphics Cards), MICs (Many-cores CPUs)
- Cons
  - Scalability limited by memory architecture.
  - Available on SMP systems "only".
  - Beware: "Upgrading" large serial code may be hard.

Processes on an SMP System

- The OS starts a process
  - One instance of your computer program, the “a.out”
- Many processes may be executed on a single core through “time sharing” (time slicing).
  - The OS allows each process to run for awhile.
- The OS may run multiple processes concurrently on different cores.
- Security considerations
  - Independent processes have no direct communication (exchange of data) and are not able to read another process’s memory.
- Speed considerations
  - Time sharing among processes has a large overhead.

OpenMP Threads

- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process.
- Each thread is given some private memory only seen by the thread.
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Don't do this. (But TS with user threads is less expensive than TS with processes).
- Implementation of threads differs from one OS to another.



Directive Sentinels: "!$omp" and "#pragma omp" not shown

OpenMP Syntax

- OpenMP Directives: **Sentinel**, **construct** and **clauses**

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

  !$omp ***construct*** **[*clause* [[,]*clause*]…]** **F90**

- Example

  #pragma omp parallel num_threads(4) **C**

  !$omp parallel num_threads(4) **F90**

- Function prototypes and types are in the file:

  #include <omp.h> **C**

  use omp_lib **F90**

- Most OpenMP constructs apply to a "structured block", that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Directives

- OpenMP directives begin with special comments/pragmas that open-aware compilers interpret. Directive sentinels are:

F90	**!$OMP**
C/C++	**# pragma omp**

Syntax: *sentinel* parallel *clauses* *uses defaults w.o. clauses*

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel
```

C/C++

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

Fortran Parallel regions are enclosed by enclosing directives.

C/C++ Parallel regions are enclosed by curly brackets.

Parallel Region

```
    ...
1   !$omp parallel
2         code statements
3         call work(…)
4   !$omp end parallel
```

```
...
#pragma omp parallel
{  code statements
        work(…)
}
```

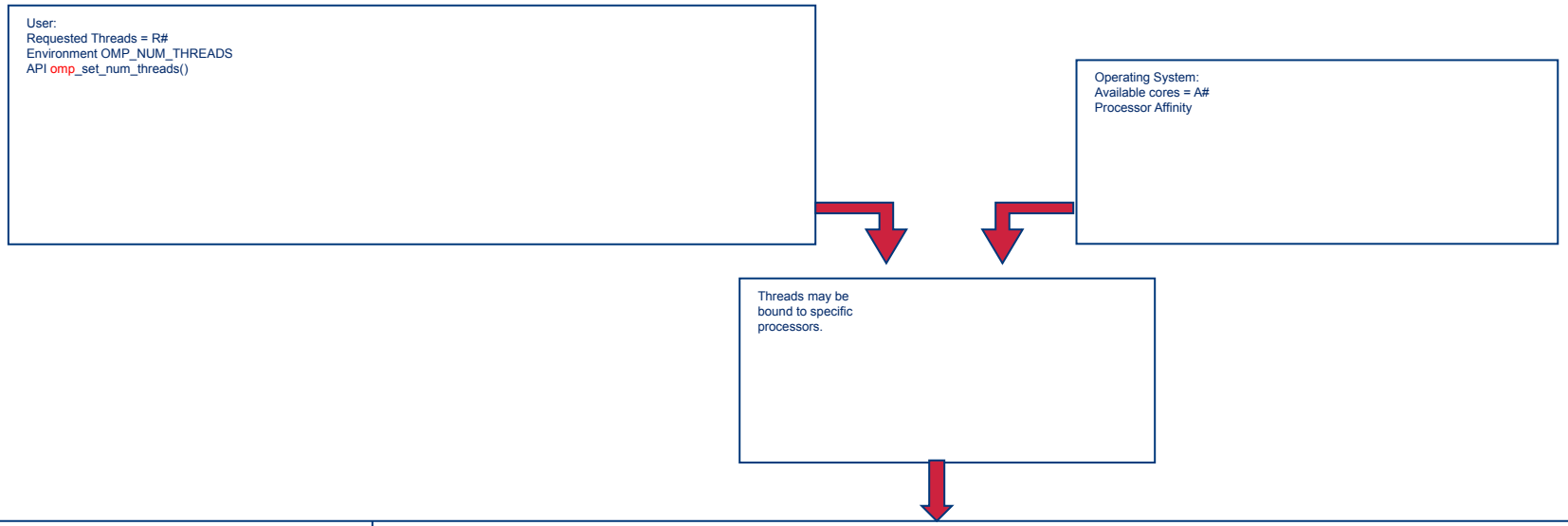
Line 1 Team of threads formed.

Lines 2-3 This is the parallel region
Each thread executes code block and subroutine call or function.
No branching (in or out) in a parallel region.

Line 4 All threads synchronize at end of parallel region (implied barrier).

In example above, user must explicitly create independent work (tasks) in the code block and routine (using thread id and total thread count).

OpenMP Setting Number of Threads



Cores Dedicated (batch system)	R# = A#	1-to-1 map between threads and cores
	R# > A#	Many-to-1 map between threads and cores Execution may be unbalanced
Cores Shared	R# <= A#	Time slicing with others-- longer exec. times Unbalanced Execution
	R# > A#	

OpenMP Thread and Memory Location

Where do threads/processes and memory allocations go?

Default: Decided by policy when process exec'd or thread forked, and on write to memory . Processes and threads can be rescheduled to different sockets and cores.

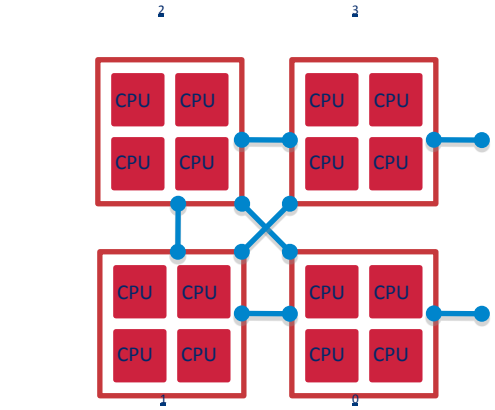
Ways Process Affinity and Memory Policy can be changed:

1.) Dynamically on a running process (knowing process id)
2.) At process execution (with wrapper command)
3.) Within program through F90/C API

NUMA Operations

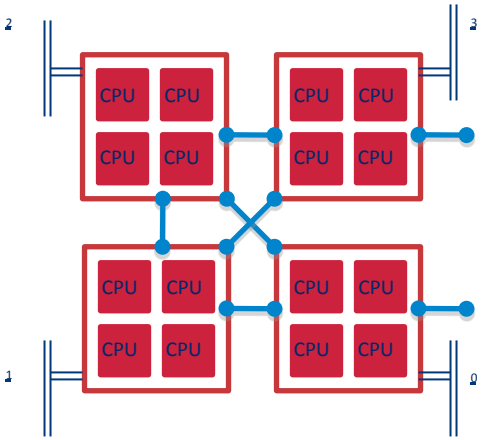
- Process Affinity and Memory Policy can be controlled at socket and core level with numactl.

Command: numactl <options socket/core> ./a.out



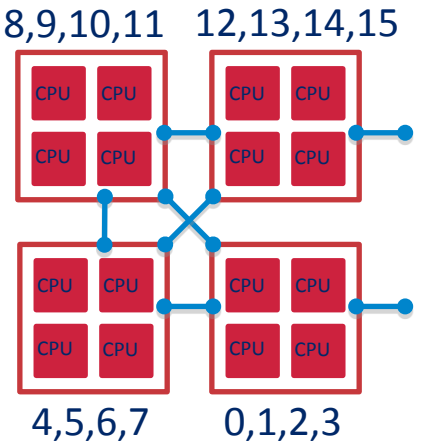
Process: Socket References

process assignment
-N



Memory: Socket References

memory allocation
–l –i --peferred –m
(local, interleaved, pref., mandatory)



Process: Core References

core assignment
–C