Lecture 17 – OpenMP

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Introduction to OpenMP
- Execution model and creating threads
    - Hello World Example
- Data Environment
- Loop Parallelism
- Example: Calculating Pi

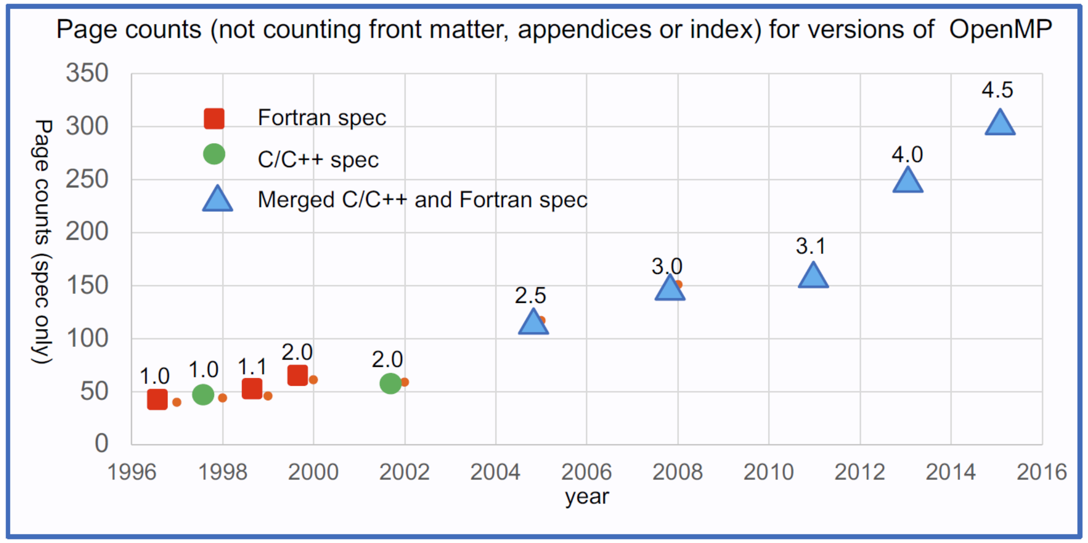
Learning Objectives: By the end of Today's Lecture you should be able to

- (*Knowledge*)
- (*Skill*) use OpenMP compiler directives
- (*Skill*) compile an OpenMP program
- (*Skill*) define a slurm job script for running threaded jobs

What is OpenMP

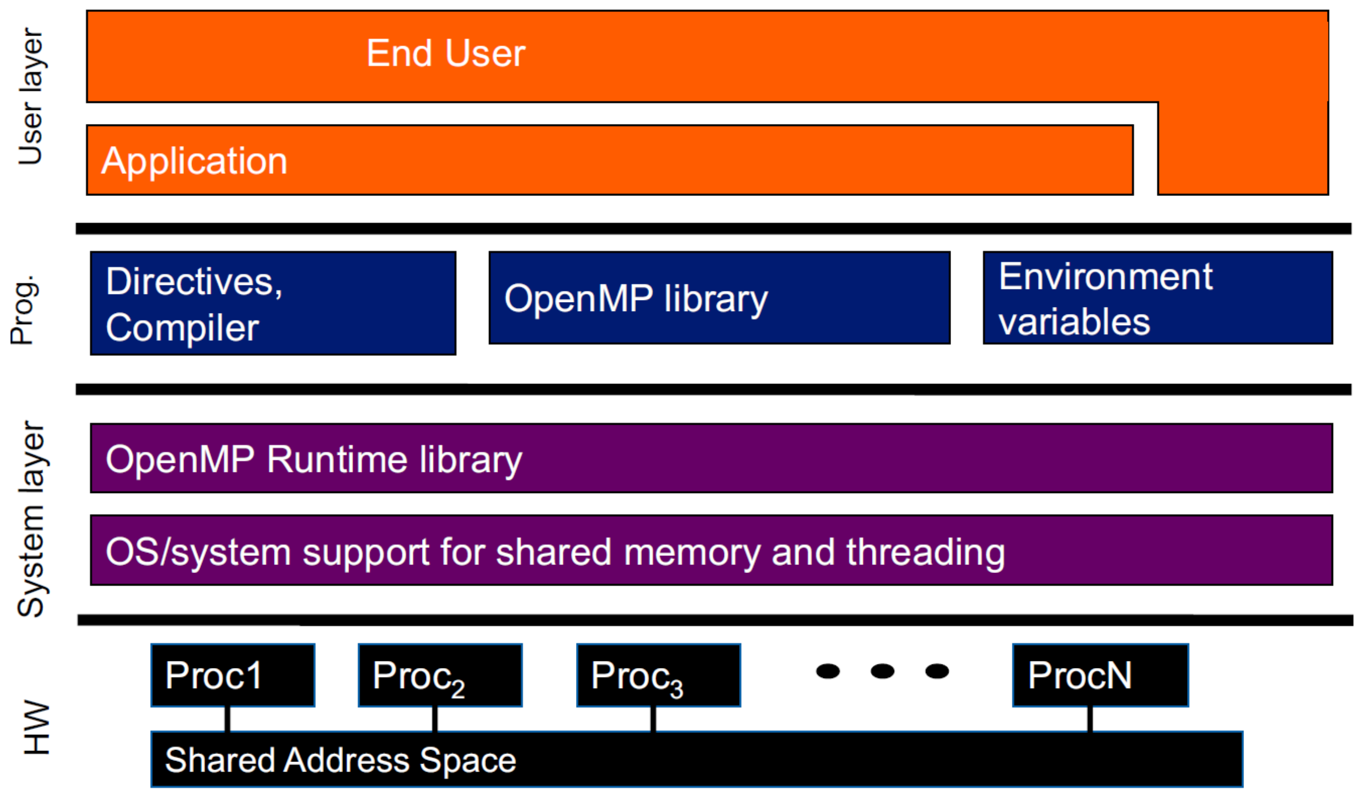
- OpenMP is an Application Programming Interface (API) for writing multithreaded applications
  - A set of compiler directives and library routines
  - Greatly simplifies writing multi-threaded applications in C/C++ and Fortran
  - Standardizes established symmetric multi-processing with vectorization and heterogeneous device programming

OpenMP started in 1997 as a simple interface for scientists. Complexity has grown substantially over the years!



The full spec is overwhelming, so we're going to focus on the essential constructs used by nearly all OpenMP programmers.

OpenMP Software Stack

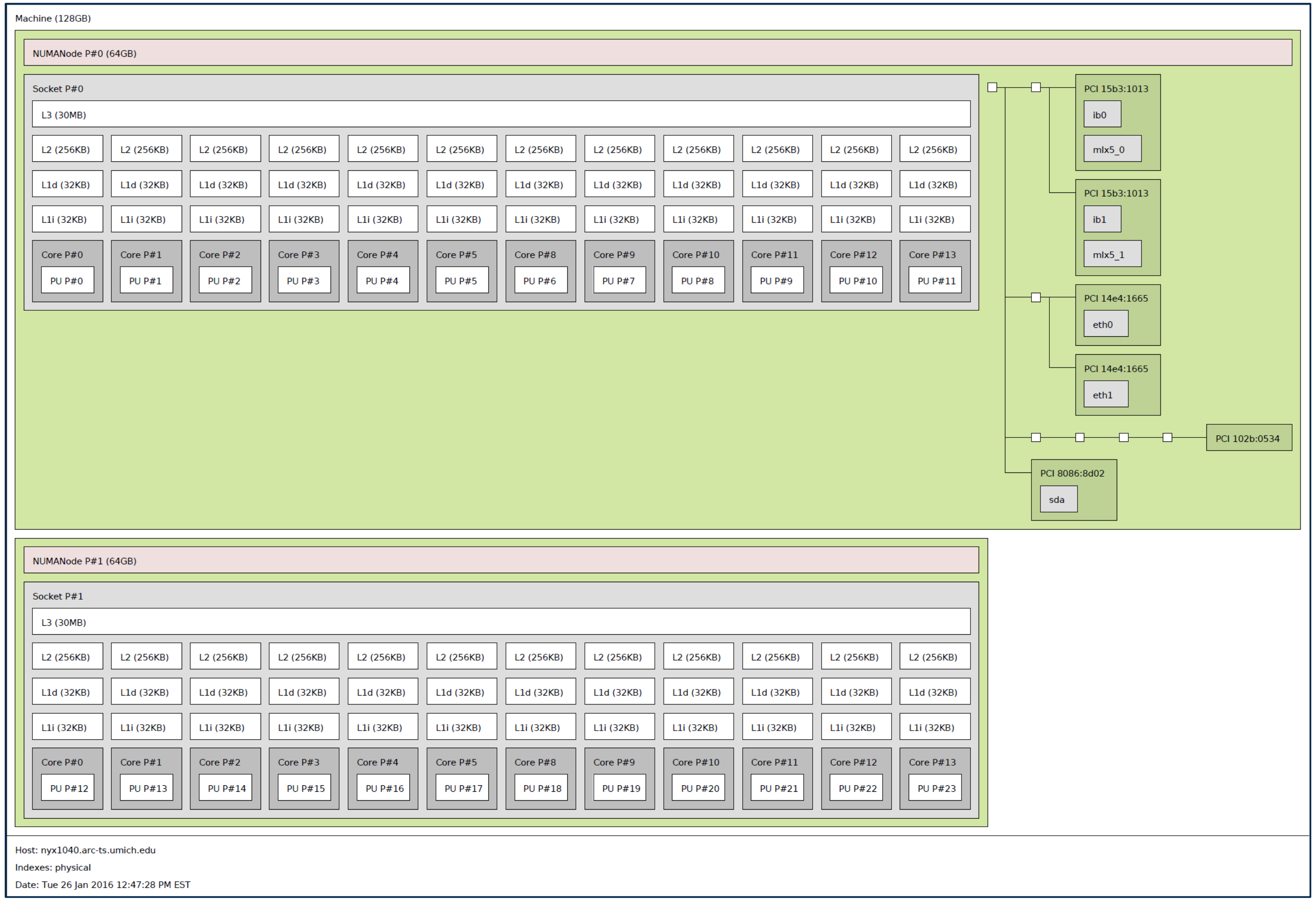


Only showing most common usage.

NUMA and GPU support were added later.

Flux node Architecture

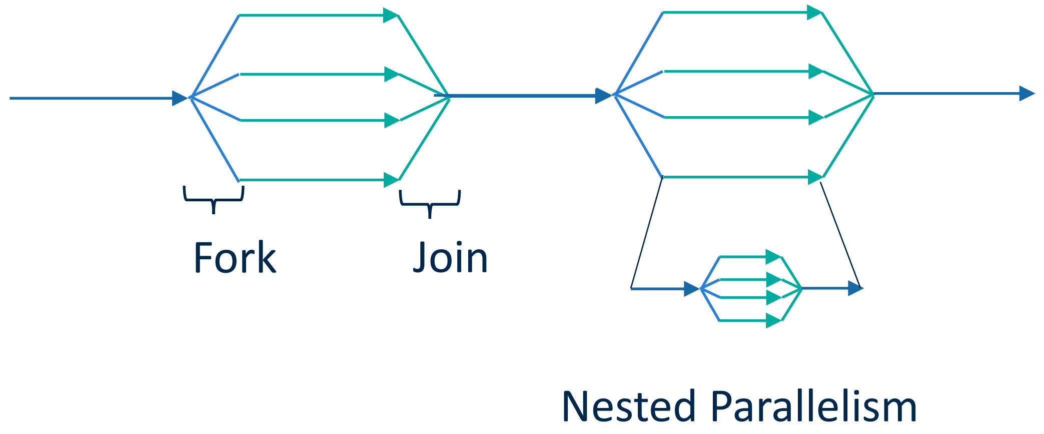
(Extent of hardware to consider with OpenMP)



Basic models of Programming in OpenMP

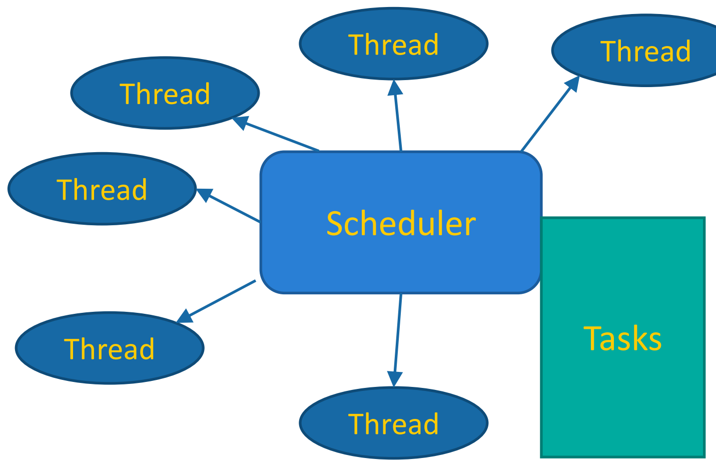
Fork/Join

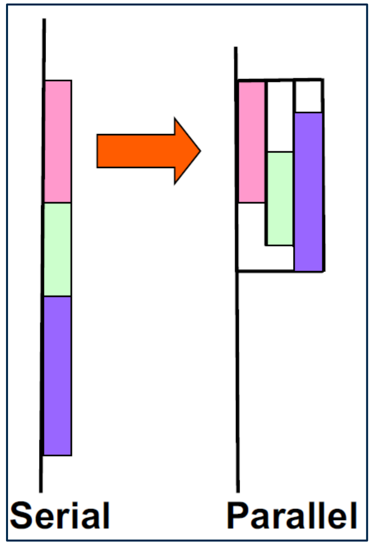
- Simple loop parallelization



Pool of Tasks

- Tasks are independent units of work composed of
  - Code to execute
  - Data to compute with





Basic Syntax

- Most of the constructs in OpenMP are compiler directives.
  - C/C++ `#pragma omp <construct> [<clause> [<clause>] ...]`
  - Fortran `!$OMP <construct> [<clause> [<clause>] ...]`
- Examples
  - `#pragma omp parallel private(x)`
  - `!$OMP parallel private(x)`
- Function interface declarations and compile time constants and types in either:
  - `#include <omp.h>`
  - `USE OMP_LIB`
- Most OpenMP constructions apply to a "structured block".
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - Examples: in C/C++ anything inside "{}"; in Fortran its loops, subroutines, functions, etc.

Enabling OpenMP

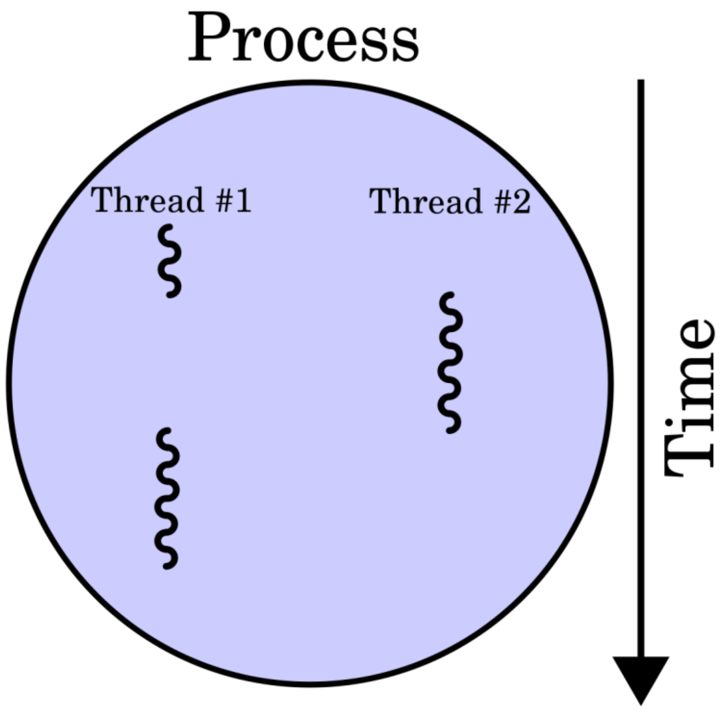
Switches for compiling and linking

Compiler	Flag
GNU gcc/g++/gfortran	-fopenmp
PGI pgcc/pgf90	-mp
Intel (Windows) icl/ifort	/Qopenmp
Intel (Linux/OSX) icc/icpc/ifort	-fopenmp
IBM xlc/xlcxx/xlf77/xlf90/xlf95/xlf2003	-qsmp
NAG nagfor	-openmp
Cray	-h omp

Execution Model

Concept of a Thread

- Ability for the hardware/operating system to execute multiple processes *concurrently*
  - Typically process = thread
  - In multi-threading a process can have multiple threads
  - Usage of “process” and “thread” is confusing
- In Linux the `top` command (short for table of processes) lists all processes
  - These are basically threads
- Bottom line is that *a thread is a software entity*, not a hardware entity



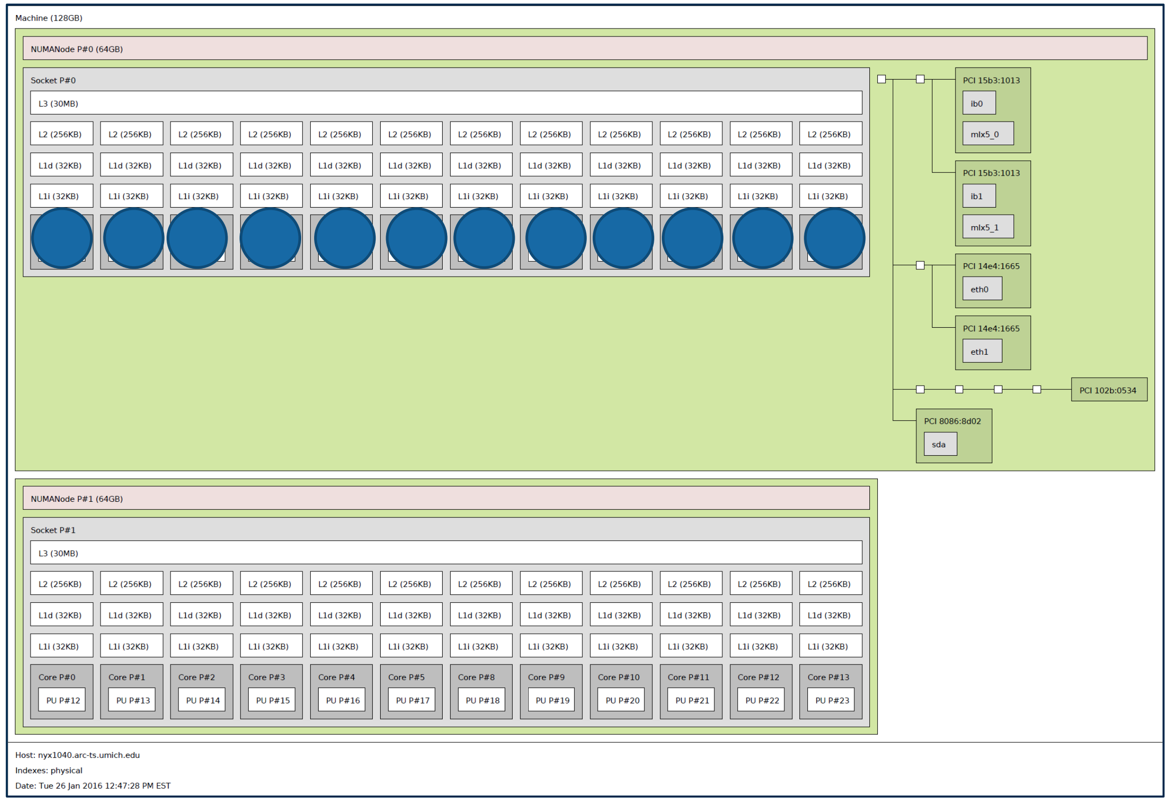
Thread Affinity

- Affinity - association of thread (software) with core (hardware)
  - This is not guaranteed.
  - By default OS and OpenMP runtime library control this.
- Threads can “drift” from core to core during execution
- Fortunately, thread affinity can be controlled
  - `OMP_PROC_BIND` – false|true|master|close|spread
  - `OMP_PLACES` – specify exactly which threads go where e.g. cores, sockets, threads or location list {location:number:stride}[,{location:number:stride}]

Affinity Example (1 socket)

12 threads, one processor

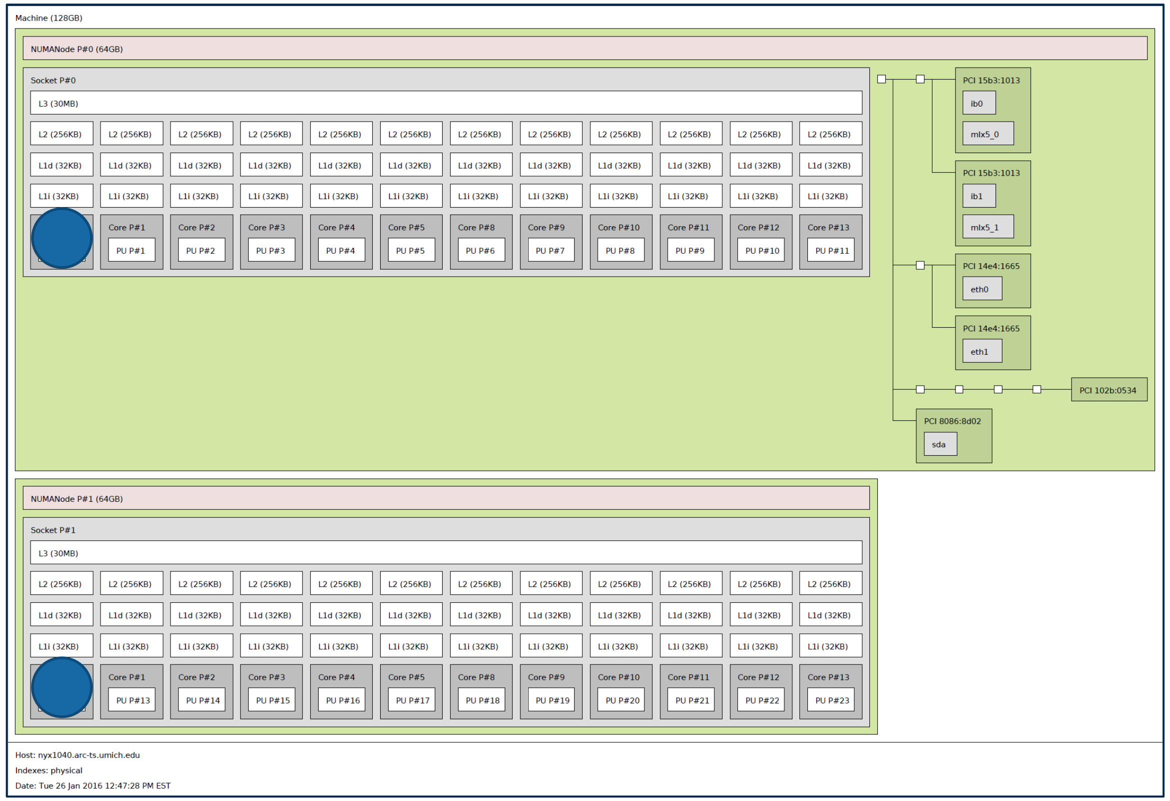
OMP_PROC_BIND=close
OMP_PLACES=cores



Affinity Example (No shared L3)

2 threads, one on each socket

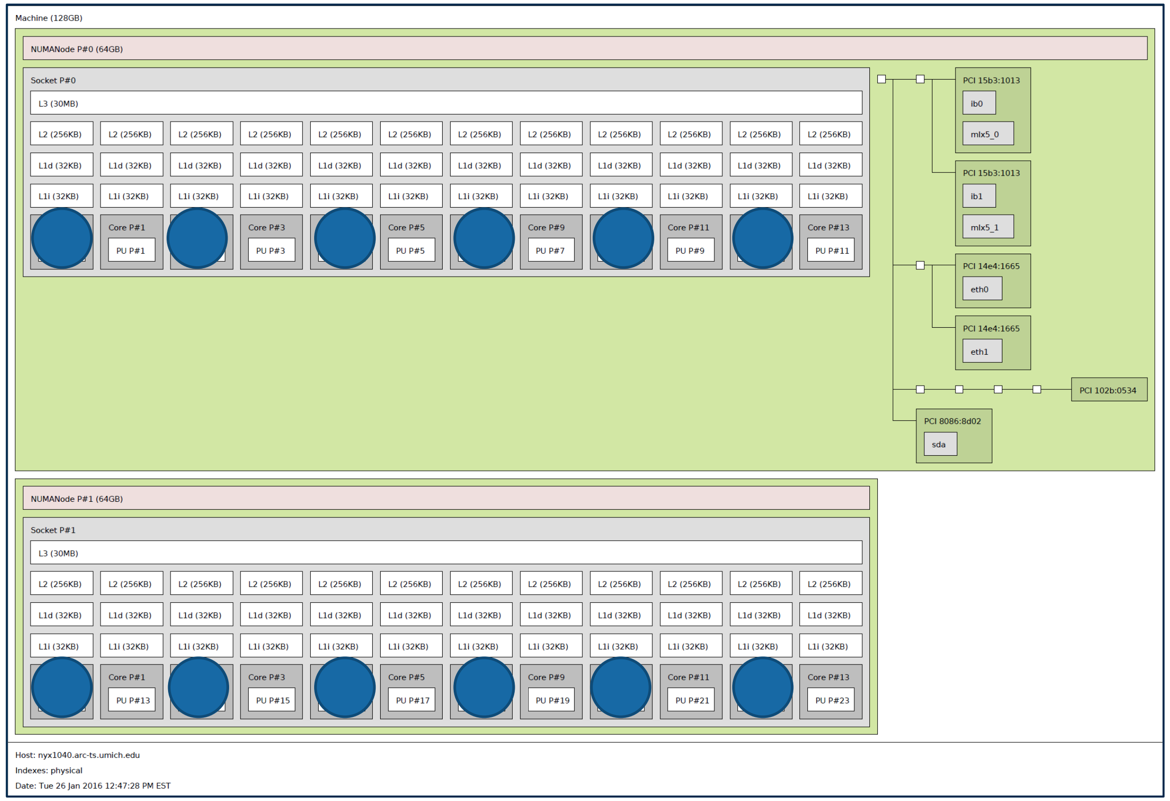
OMP_PROC_BIND=true
OMP_PLACES=sockets



Affinity Example (alternating)

12 threads, every other core

OMP_PROC_BIND=spread
OMP_PLACES=cores



Thread Creation & Destruction

C/C++

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  int nthrds = omp_get_num_threads();
  pooh(ID,A);
}
```

Fortran

```
REAL(8) :: A(1000)
INTEGER :: id,nthrds

omp_set_num_threads(4)
!$OMP PARALLEL
id=omp_get_thread_num();
nthrds=omp_get_num_threads();
CALL pooh(id,A)
!$OMP END PARALLEL
```

Controlling the Number of Threads

- There are a few ways to do this...
- Use the `omp_set_num_threads()`
  - This changes an "internal control variable" the system queries to select the default number of threads in subsequent parallel constructs
- To change without re-compilation one can INSTEAD use environment variables associated with OpenMP
  - When an OpenMP program starts up, it queries an environment variable `OMP_NUM_THREADS` and sets the appropriate internal control variable to the value of `OMP_NUM_THREADS`
  - e.g. `$ export OMP_NUM_THREADS=12`

Hello World Example

C/C++

Serial

```
#include <stdio.h>

int main ()
{





 printf("Hello World \n");


}
```

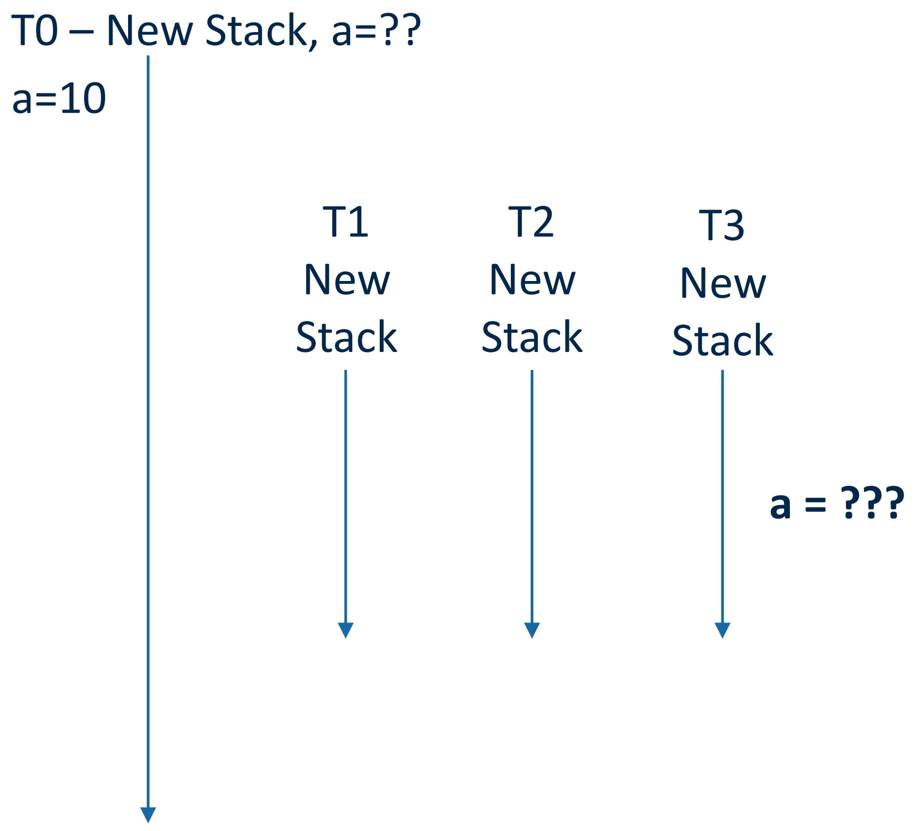
Threaded

```
#include <stdio.h>
#include <omp.h>
int main ()
{
 omp_set_num_threads(4);
  #pragma omp parallel
  {
     int id = omp_get_thread_num();
     printf("Hello World from thread = %d", id);
     printf(" with %d threads\n",omp_get_num_threads());
  }
}
```

Data Environment

Consider the following scenario

```
int a;
a=10
omp_set_thread_num(4);
#pragma omp parallel
{
  int id = omp_get_thread_num();
  printf("On thread = %d, a=%d\n", id, a);
}
```



Data Environment Default Behavior

- Most variables are shared
    - Actual behavior depends on how/where variable is defined
- Global variables default to SHARED
    - In Fortran: COMMON blocks, variables with SAVE attribute, and module variables, dynamically allocated arrays
    - In C/C++: file scope variables, static variables, and dynamically allocated memory
- Default private variables include
    - Stack variables and automatic variables
- Default behavior can be declared explicitly with default clause
    - `default(none|shared|private)`

Controlling Data Environment

- When declaring new parallel sections, OpenMP provides clauses for defining the data environment.
  - `shared` – variable retains one copy in memory, threads do not duplicate anything
  - `private` – specify which variables are private amongst threads
    - Creates local copies of variables. Variables have typical automatic definitions of serial code (e.g. declared but not defined). Note fixed sized arrays are duplicated!
- Special cases
  - `firstprivate` – create local copies and initialize all of them to their state just before the parallel construct. Note this duplicates all arrays!
  - `lastprivate` – variable is set equal to the private version of whichever thread executes the final iteration of for-loop or last section of sections construct.

Parallel Loops

Parallel For - C

```
int main()
{

  ... serial code ...

  #pragma omp parallel for
  for (i=0; i<n; i++)
    a[i] = b[i] + c[i]

  ... more serial code ...

}
```

```
int main()
{

  ... serial code ...

  #pragma omp parallel
  #pragma omp for
  for (i=0; i<n; i++)
    a[i] = b[i] + c[i]

  ... more serial code ...

}
```

Parallel For- Fortran

```
program

... serial code ...

!$omp parallel do
do i = 1,n
  a(i) = b(i) + c(i)
enddo
!$omp end parallel do

... more serial code ...

end program
```

```
program

... serial code ...

!$omp parallel
!$omp do
do i = 1,n
  a(i) = b(i) + c(i)
enddo
!$omp end do
!$omp end parallel

... more serial code ...

end program
```

Loop scheduling

OpenMP lets you control how a threads are assigned iterations of a parallel loop:

- `static` – equal-sized chunks of iterations are assigned to each thread. When a thread finishes, it waits for the others.
- `dynamic` – threads obtain a new chunk when their current chunk is finished.
- `guided` – chunk size starts off large and decreases, for better load balancing.
- `auto` – let the compiler choose.
- `runtime` –the `OMP_SCHEDULE` environment variable determines the scheduling strategy

How to schedule?

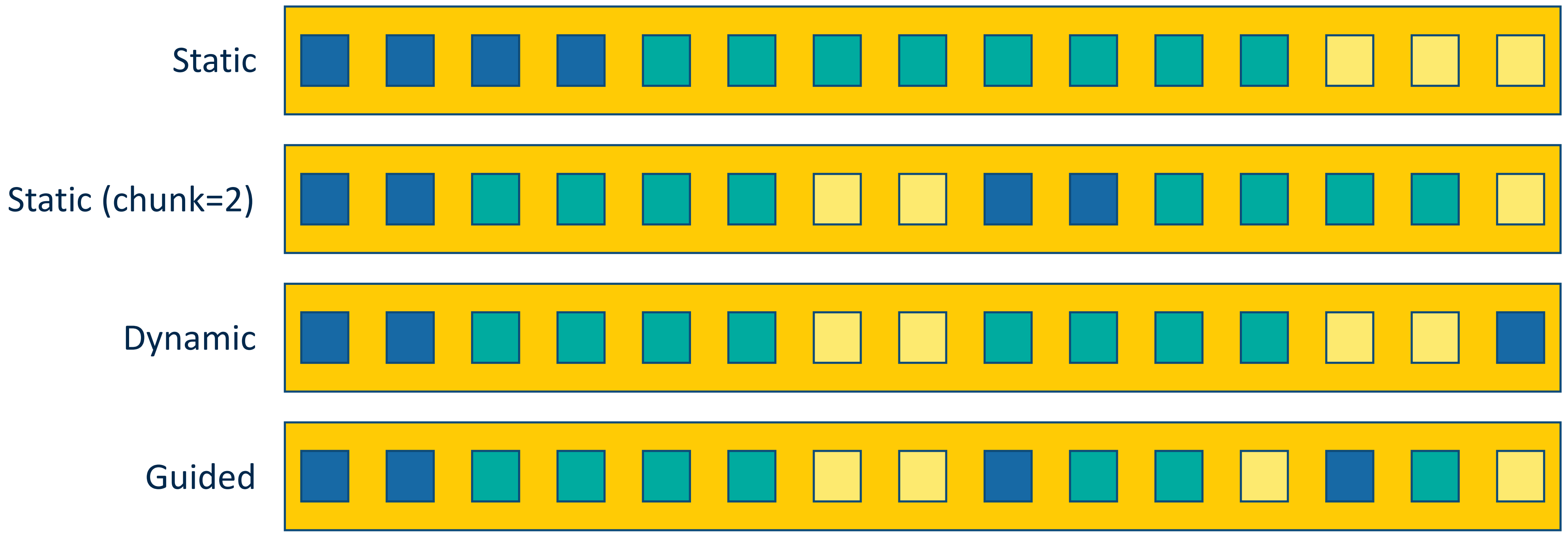
Chunk Size

- A chunk is a block of iterates
  - e.g. do i=1,1000
    can have chunk size of 1 or 10 or 100, etc.
- Chunk size can often be utilized to "tune" certain loops.
- Chunk size can be specified as a variable
  - e.g. chunk=niters/(10*nthreads)
    each thread would receive about 10 chunks

When to use what?

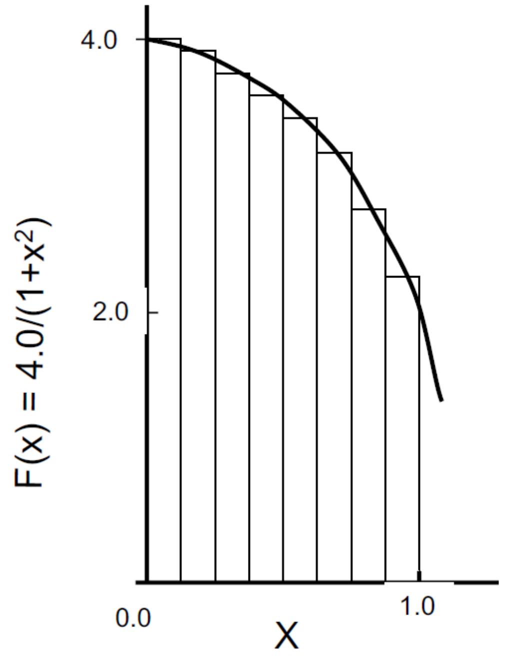
Schedule	When to use
STATIC	Any loop iteration takes about as long as any other loop iteration
DYNAMIC	Large variability in time of each loop iteration
GUIDED	Some variability in time of each loop iteration

Illustration of Different Schedules



Parallel Loop Example

Numerical Integration of pi



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Approximate as a summation of rectangles (midpoint rule)

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Each rectangle has width Δx and height $F(x_i)$ at the middle of the interval i.

Serial Code

```
static long num_steps = 100000000;
double step;
int main()
{
  int i;
  double x, pi, sum = 0.0;
  double start_time, run_time;

  step = 1.0/(double) num_steps;
```

```
  start_time = omp_get_wtime();
  for (i=1;i<= num_steps; i++){
      x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  run_time = omp_get_wtime() - start_time;
  printf("\n pi with %ld steps is %lf in
  %lf seconds\n ",num_steps,pi,run_time);
}
```

Parallel Solution 1

```
#define MAX_THREADS 4
static long num_steps = 100000000;
double step;
int main()
{
  int i;
  double x, pi, fsum = 0.0;
  double sum[MAX_THREADS];
  double start_time, run_time;

  step = 1.0/(double) num_steps;
```

```
  #pragma omp parallel num_threads(MAX_THREADS)
  {
    id = omp_get_thread_num();
    sum[id] = 0.0;
    #pragma omp for
    for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
      sum[id] = sum[id] + 4.0/(1.0+x*x);
    }
  }
  for(i=0; i < MAX_THREADS; i++)
    fsum += sum[i];
  pi = step * fsum;
}
```

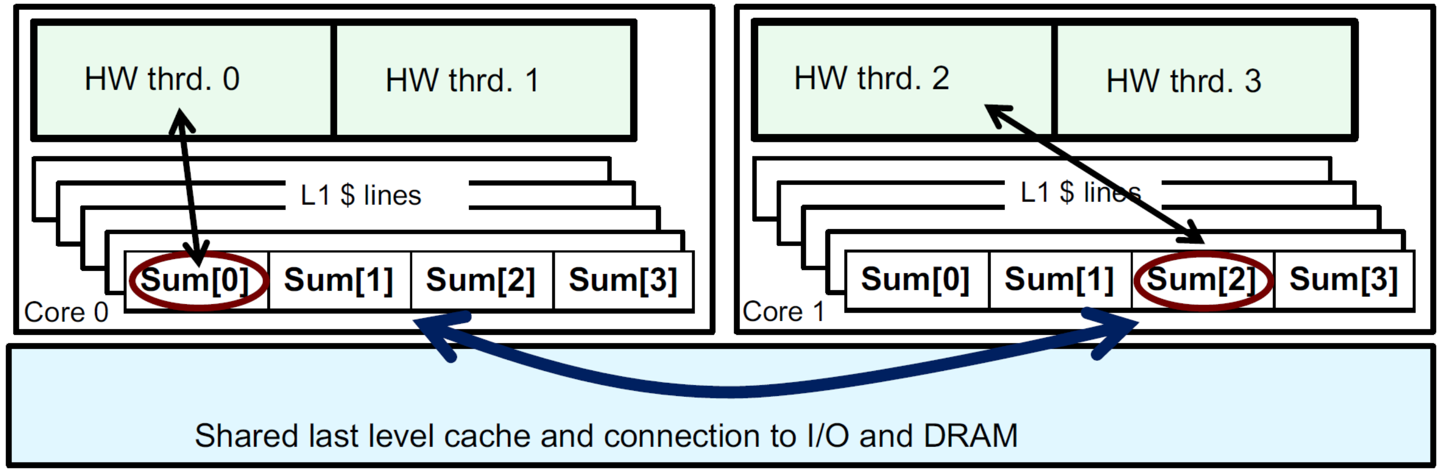
Analysis

- Issue is with computing sum across threads.
  - Threads should compute partial sums and then we sum across threads.
  - Recall from Lecture 12 this is a *reduction operation*
- Alternatively, we can enforce each thread to access `sum` one at a time.
  - This involves *synchronization*

```
for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
  sum = sum + 4.0/(1.0+x*x);
}
```

False Sharing

- If you promote scalars to arrays to avoid race conditions and compute a partial sum, the cache may "slosh" back and forth between threads.
- The reason for this is the array elements are contiguous in memory and hence share a cache lines. This sharing of elements causes cache misses due to conflicts (also sometimes called collisions or interference) which is due to organization.



- The result is the observation of poor scalability.
  - One solution is to pad the array so elements that need to be accessed by each thread appear on different cache lines.

OpenMP clauses for reduction

- `#pragma omp for reduction(op:var[,var2,…])`
- Added for convenience since reductions are very common (e.g. any integral)

```
#pragma omp for reduction(+:sum)
for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
  sum = sum + 4.0/(1.0+x*x);
}
```

Operator	Initial Value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number
And	true
Or	false

And there are many others…