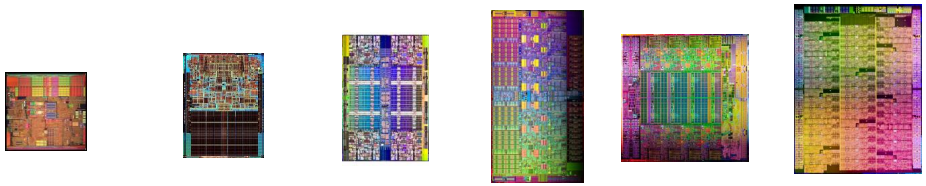
[Potentially] Your first parallel application

- Compute the smallest element in an array as fast as possible

```
small = array[0];
for( i = 0; i < N; i++)
   if( array[i] < small ) )
      small = array[i]
```



Images not intended to reflect actual die sizes

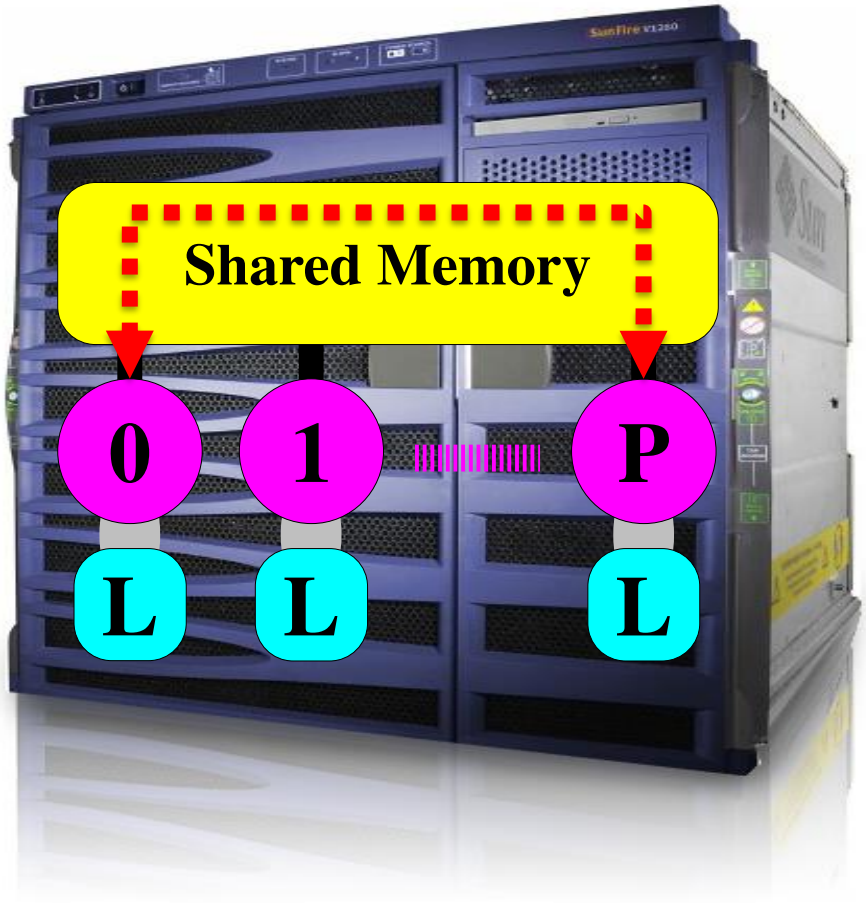
	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v2 series	Intel® Xeon Phi™ Co-processor 7120P
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.7 GHz	1.238 MHz
Core(s)	1	2	4	6	12	61
Thread(s)	2	2	8	12	24	244
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

- Is this the fastest way to compute the min?

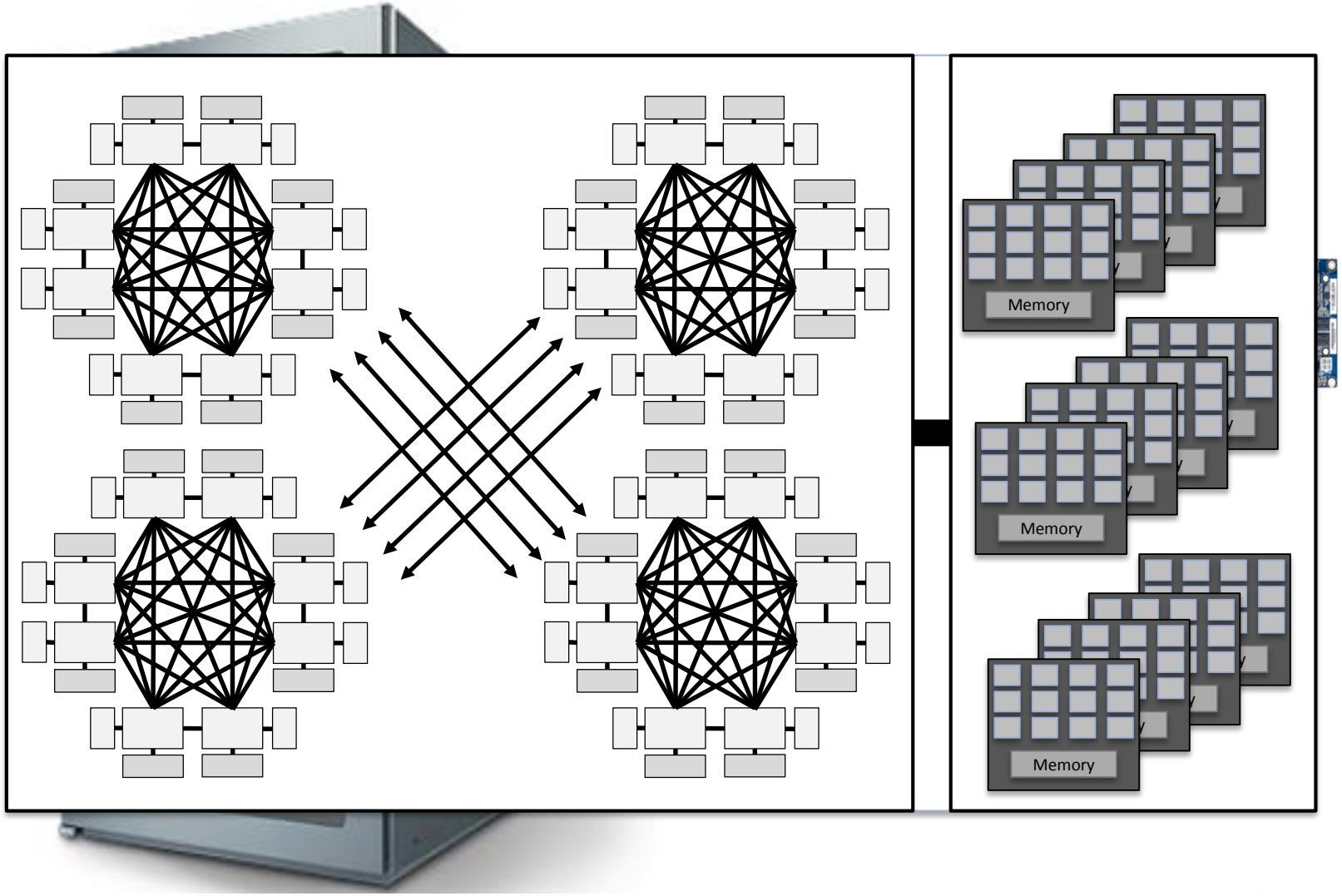
INTRODUCTION TO OPENMP

George Bosilca
bosilca@icl.utk.edu

What is OpenMP: 1997-2013



What is OpenMP: 2013-



http://www.openmp.org

http://www.iwomp.org

http://www.compunity.org

What is OpenMP?

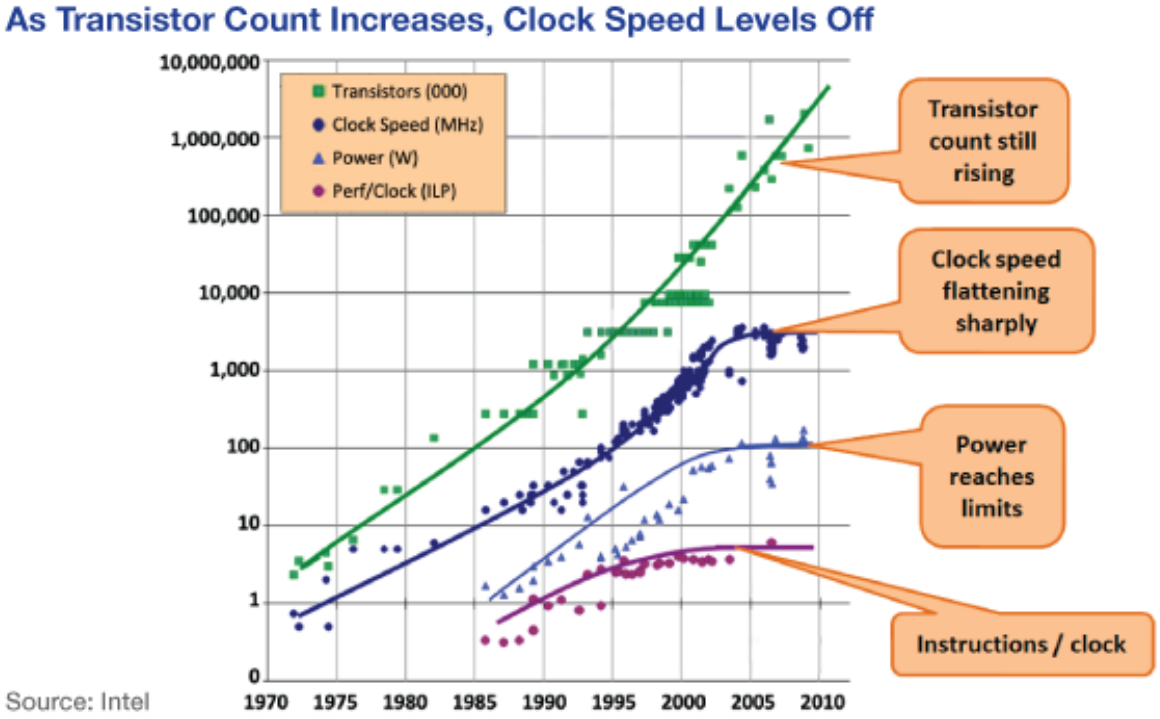
- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C ++, and Fortran
- OpenMP is widely supported by the industry, as well as the academic community
- Consists of Compiler Directives, Runtime routines and Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
- Current version: 4.0 (released July 2013)
  - 4.1 Draft Specs Open for Public Comment

When to consider OpenMP?

- Theoretically: never it should be done automatically by the compiler
- Practically: in all the other cases

When to consider OpenMP?

- Theoretically: never it should be done automatically by the compiler
- Practically: in all the other cases



When to consider OpenMP

- Using an automatically parallelizing compiler
  - The compiler can not find the parallelism
    - The data dependence analysis is not able to determine whether it is safe to parallelize
  - The granularity is not high enough, but you know better
    - Compiler lacks the information to parallelize at a higher level
- Not using an automatically parallelizing compiler
  - No other choice than to parallelize yourself
  - Compilers can still help (e.g. auto-scoping, warnings, etc)

Advantages of OpenMP

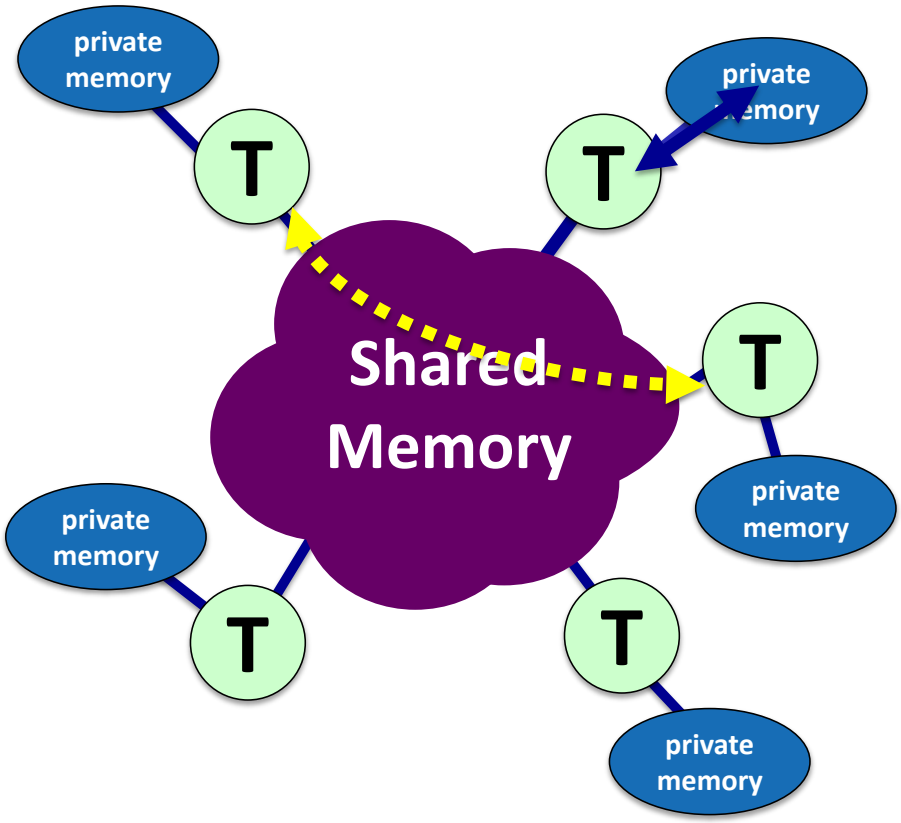
- De-facto and mature standard
- Good performance and scalability
  - If you do it right ....
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
  - But, ...
- Allows the program to be parallelized incrementally
  - But, ...

- OpenMP is ideally suited for multicore architectures
  - Memory and threading model map naturally
  - Lightweight

Components of OpenMP

Directives	Runtime functions	Environment variables
• Worksharing • Tasking • Affinity • Accelerators • Cancellation • Synchronization	• Thread Management • Work Scheduling • Tasking • Affinity • Accelerators • Cancellation • Locking	• Thread Settings • Thread Controls • Work Scheduling • Affinity • Accelerators • Cancellation • Operational

The OpenMP Memory Model



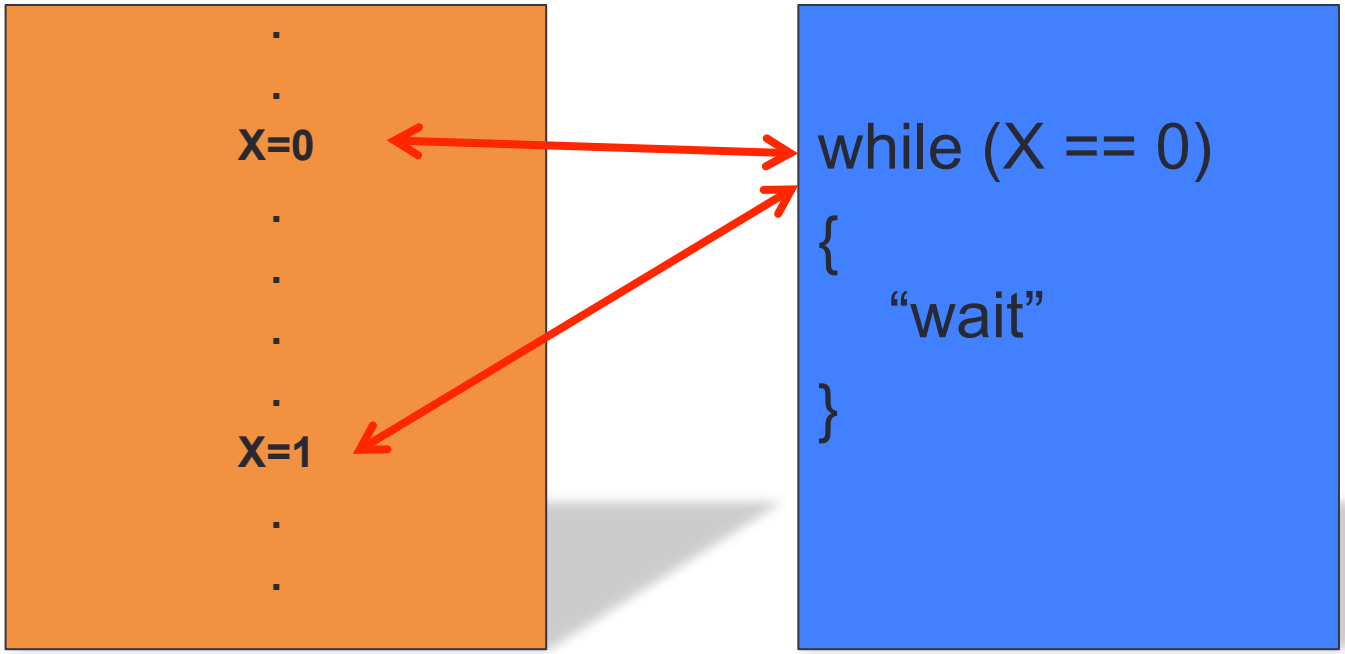
- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application

- Agnostic to the hardware architecture
- Agnostic to memory placement

Keep in mind !

- Need to get this right
  - Part of the learning curve
- Private data is undefined on entry and exit
  - Can use firstprivate and lastprivate to address this
- Each thread has its own temporary view on the data
  - Applicable to shared data only
  - Means different threads may temporarily not see the same value for the same variable ...
- What ?

The Flush Directive



- If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)

The Flush Directive

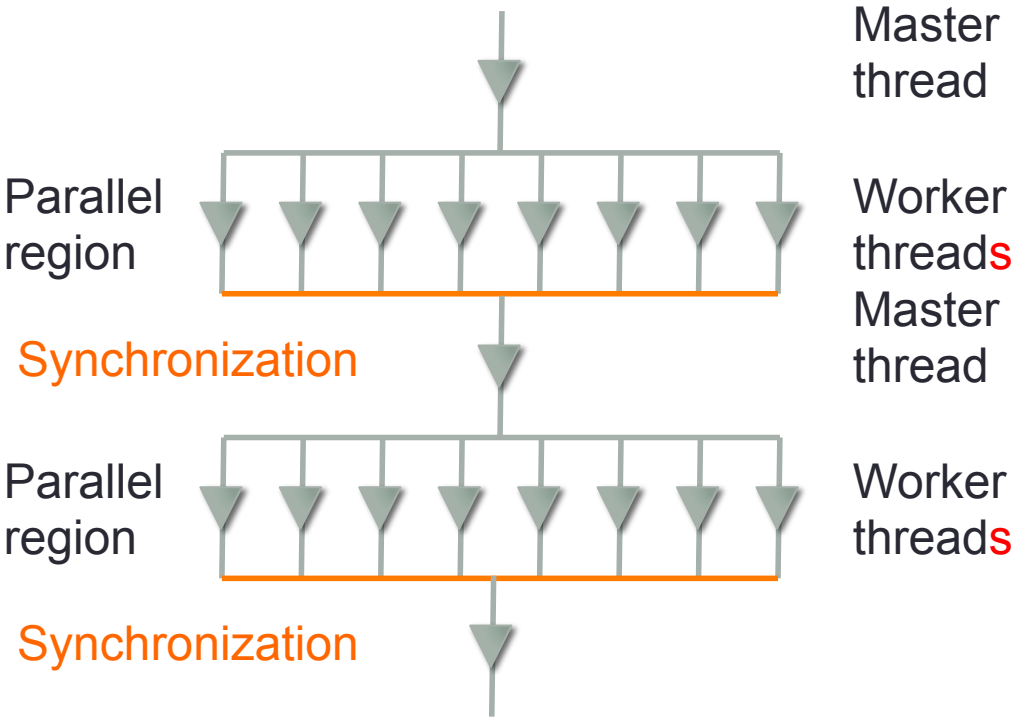
```
void wait_read(int i) {
    #pragma omp flush
    while ( execution_state[i] != READ_FINISHED )
    {
        system("sleep 1");
        #pragma omp flush
    }
} /*-- End of wait_read --*/
```

- Example from “Using OpenMP”
- execution_state[i] is modified outside the scope of the block and potentially on another thread
- Flush ensure that all changes become locally visible

About the Flush

- Implicitly implied on many OpenMP constructs
  - A good thing
  - This is your safety net
- Don’t abuse it
  - For performance reasons a careful usage is more than recommended
- Strong recommendation: never use the flush directive with a list
  - Could give very subtle interactions with compilers
  - Just don’t do it !

OpenMP Execution Model



- Fork and join model
  - A master thread supported by worker threads during #pragma annotated code blocks

The OpenMP Barrier

- Several constructs have an implied barrier
  - This is another safety net (has implied flush by the way)
- In some cases, the implied barrier can be left out through the nowait clause
- This can help fine tuning the application
  - But you'd better know what you're doing
- The explicit barrier comes in quite handy then

```
#pragma omp barrier
```

```
!$omp barrier
```

The Nowait Clause

- To minimize synchronization, some directives support the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct
- In C, it is one of the clauses on the pragma
- In Fortran, it is appended at the closing part of the construct

```
#pragma omp for nowait
{
        .
}
```

```
!$omp do
        .
        .
!$omp end do nowait
```

Defining Parallelism in OpenMP

- A parallel region is a block of code executed by all threads in the team
  - Must be encountered by all threads (BEWARE !)

```
#pragma omp parallel [clause[[,] clause] ...]
{
   “code executed in parallel by each thread”
} // End of parall section (note: implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]
    "this code is executed in parallel"
!$omp end parallel (note: implied barrier)
```

The Worksharing Constructs

```
#pragma omp for
{
        …
}

!$OMP DO
    ….
!$OMP END DO
```

```
#pragma omp sections
{
        …
}

!$OMP SECTIONS
    ….
!$OMP END SECTIONS
```

```
#pragma omp single
{
        …
}

!$OMP SINGLE
    ….
!$OMP END SINGLE
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless the nowait clause has been specified)
- A work-sharing construct does not launch any new threads

The Fortran Workshare Construct

- Fortran have special constructs to work with array sections
- Translate into a specialized Fortran construct

```
!$OMP WORKSHARE
                .
!$OMP END WORKSHARE  [nowait]
```

Example:

```
!$OMP WORKSHARE
   A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

Parallel Sections

```
#pragma omp sections [clauses]
{
   #pragma omp section
   { …. }
   #pragma omp section
   { …. }
   ….
} // (note: implied barrier)
```

```
!$omp sections [clauses]
   !$ omp section
      {….}
   !$ omp section
      {….}
   ….
!$omp end sections [nowait]
```

- Individual section blocks are executed in parallel
  - Independently
  - Workers are equally divided among sections

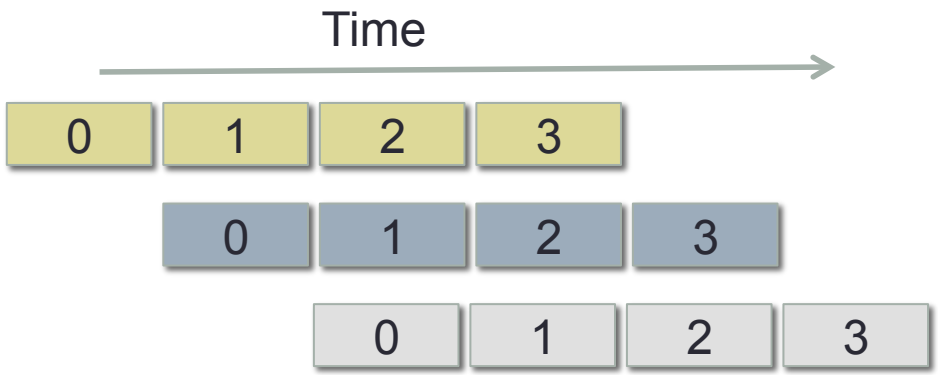
Overlap I/O And Processing

```
#pragma omp parallel sections {
   #pragma omp section {
      for (int i=0; i<N; i++) {
         (void) read_input(i);
         (void) signal_read(i);
      }
   }
   #pragma omp section {
      for (int i=0; i<N; i++) {
         (void) wait_read(i);
         (void) process_data(i);
         (void) signal_processed(i);
      }
   }
   #pragma omp section {
      for (int i=0; i<N; i++) {
         (void) wait_processed(i);
         (void) write_output(i);
      }
   }
} /*-- End of parallel sections --*/
```

Input threads

Processing threads

Output threads



The Single Directive

- Only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
   <code-block>
}
```

- No guarantee on which thread will execute the <code-block>
- By default an implicit barrier between all members of the team is implied
  - Nowait can be used to remove this constraint

```
#pragma omp parallel
 {
   Work1();
   #pragma omp single
   {
     Work2();
   }
   Work3();
 }
```

- If we have T threads:
  - Work1 will be executed T times
  - Work2 will be executed 1 time
  - Work3 will be executed T times

The master Directive

```
#pragma omp master
{ < code-block> }
```

```
!$omp master
    < code-block>
!$omp end master
```

- Similar to the single directive
  - But only the master thread executes the <code-block>
- All others are free to go as needed
- There is implicit barrier on entry or exit

```
#pragma omp parallel
 {
   Work1();
   #pragma omp master
   {
     Work2();
   }
   Work3();
 }
```

- If we have T threads:
  - Work1 will be executed T times
  - Work2 will be executed 1 time by the master thread while the others will continue on Work3
  - Work3 will be executed T times

Additional Directives - Misc

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

Force updates to the variables in the list
Don't use the list

```
#pragma omp ordered
{ < code-block> }
```

```
!$omp ordered
    < code-block>
!$omp end ordered
```

When nothing else works
(aka. debugging purposes)

Additional Directives - Updates

```
#pragma omp critical [(name)]
{ < code-block> }
```

```
!$omp critical [(name)]
    < code-block>
!$omp end critical [(name)]
```

- Generic protection of the <code-block> allowing a single thread at a time in the <code-block>
- The name is used to ensure uniqueness between several critical sections

```
#pragma omp atomic [clause]
```

```
!$omp atomic [clause]
```

- Fine grain protection of a single update
- Uses atomic operations
- Clauses on the atomic: read, write, update, capture, seq_cst

"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." – Leslie Lamport

Loop nesting

- If the parallel directive is in a section where a team already exists, a new team will be created for each thread
  - Each thread will be alone in it's second team
  - Thus, the internal parallel loop is executed sequentially in the context of a single thread.

```
#pragma omp parallel for
 for(int y=0; y<5; ++y)
 {
  #pragma omp parallel for
  for(int x=0; x<6; ++x)
  {
    counter(x,y);
  }
 }
```

The collapse clause

- increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread

```
#pragma omp parallel for collapse(2)
 for(int y=0; y<5; ++y)
 {
  #pragma omp parallel for
  for(int x=0; x<6; ++x)
  {
    counter(x,y);
  }
 }
```

```
#pragma omp parallel for
 for(int x, y, t = 0; t < (5*6); ++t)
 {
   y = t / 6;
   x = t%6;
   counter(x,y);
}
```

Clauses

```
#pragma omp parallel [
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (#)
{
   "code executed in parallel by each thread"
} // End of parall section (note: implied barrier)
```

- private: each thread in the team will have its own copy
- shared: all threads share the same variable (beware read/write conflicts)
- firstprivate: the variable is private but it is initialized from the variable with the same name from the context above
- lastprivate: the last thread will copy its private variable into the global one

Reduction clause

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float a, b;
a= 0.;
b= 1.;
#pragma omp parallel for reduction(+:a) reduction(*:b)
for (i=0; i<n; i++){
 a = a+ a[i];
 b = b* a[i];
}
```

- Each thread has a private a and b, initialized to the operator's identity
- After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction

A word about scheduling

- schedule(static)
  - Each thread receives one set of contiguous iterations
- schedule(static, C)
  - Iterations are divided round-robin fashion in chunks of size C
- schedule(dynamic, C)
  - Iterations handed out in chunks of size C as threads become available
- schedule(guided, C)
  - Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time
- schedule (runtime)
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable

OpenMP environment variables

Environment variables

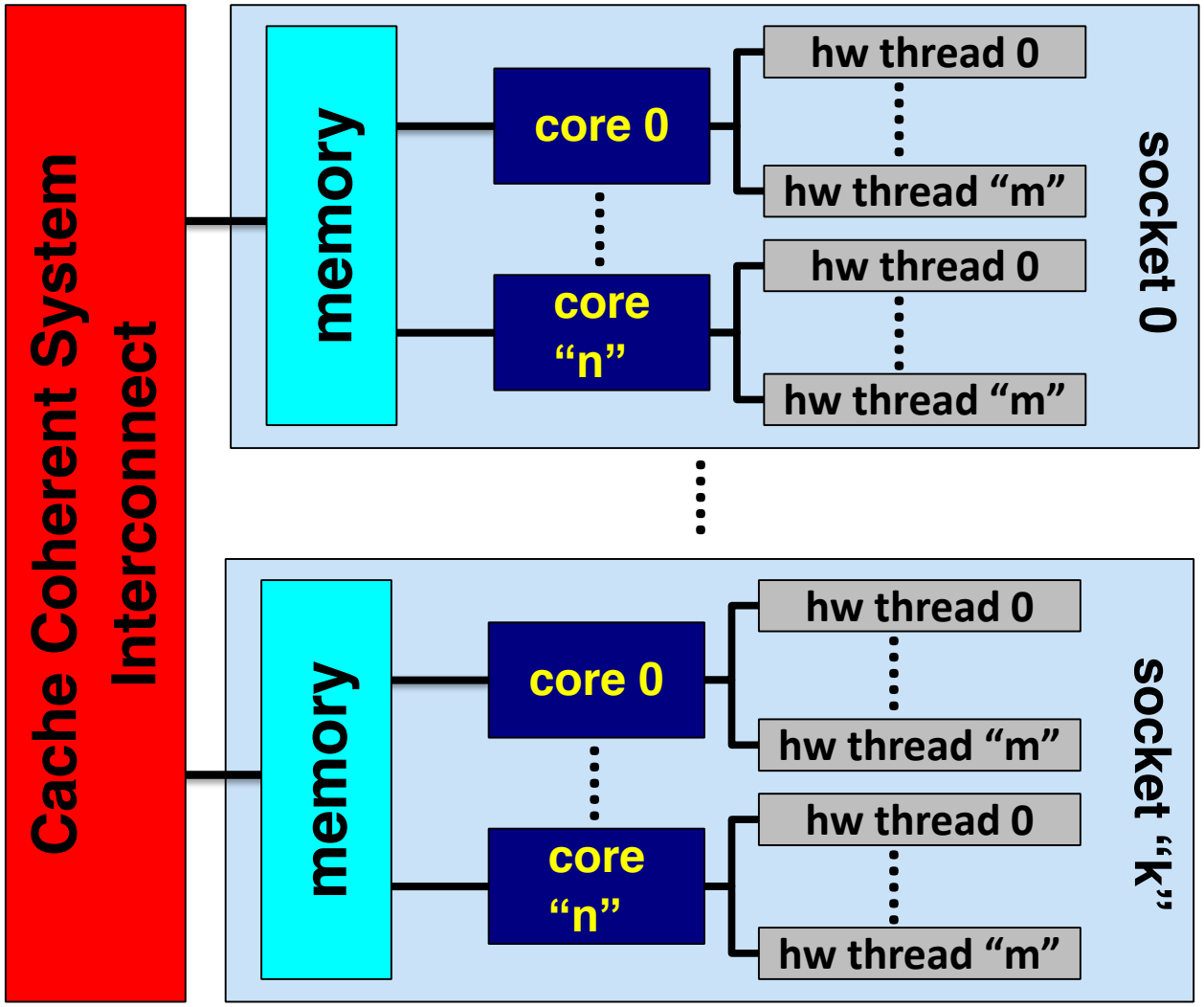
OpenMP Environment Variable	Category
OMP_DISPLAY_ENV	Diagnostics
OMP_NUM_THREADS	Thread Management
OMP_THREAD_LIMIT	Thread Management
OMP_DYNAMIC {true| false}	Thread Management
OMP_NESTED {true| false}	Parallelism
OMP_MAX_ACTIVE_LEVELS	Parallelism
OMP_STACKIZE "size [b|k|m|g]"	Operational

- The names have to be in uppercase; the values are case insensitive
- Be careful when relying on defaults (because they are compiler dependent)

Environment variables

OpenMP Environment Variable	Category
OMP_PLACES	Affinity
OMP_PROC_BIND	Affinity
OMP_DEFAULT_DEVICE	Accelerators
OMP_CANCELLATION	Thread Management
OMP_WAIT_POLICY [active | passive]	Thread Management
OMP_SCHEDULE “schedule, [chunk]”	Work Distribution

Thread affinity



Thread affinity

- Define “hardware thread units”, called places
  - Order within a place is irrelevant
- Define a set of places, called a place list
  - Order of the places in the place list matters!
- Examples:
  - OMP_PLACES = “{0,1,2} , {5,6,7}”
  - OMP_PLACES = “{5,6,7} , {0,1,2}”
  - OMP_PLACES = “{5:2:1} , {0:2:1}”

Thread affinity

- Then define how these units are mapped/bound to the hardware topology
  - This is usually called “binding”
  - Controlled through OMP_PROC_BIND and/or the proc_bind clause
- Note that the low level thread numbers are system dependent
  - Luckily, abstraction is supported though !
- Three abstract names are supported:
  - sockets / cores / threads
- Example:
  - OMP_PLACES=cores
  - Optional: length specifier (e.g. cores(4))

Thread affinity

- Master – Every thread in the team is assigned the same place as the master thread
- Close – Assign threads to places close to the parent thread
  - The threads stay “close”, as defined through the places list
  - Main goal: first use all threads within a core
- Spread – Use a sparse distribution over the places
  - Spread the threads out, as defined by the places list
  - Main goal: use all cores in the system first

- [export|setenv] OMP_PROC_BIND=[master | close | spread]
- proc_bind(master | close | spread)
- This environment variable and the clause define a policy for assigning threads to places