



An introduction to OpenMP

Ben Cumming

CSCS Summer School 2014

Before We Start

- The OpenMP web site is a good source of information:

openmp.org

- tutorials and examples from beginner to advanced
- the standard (which is easy to understand for a standard)
- quick reference guides



Multicore

The Free Lunch

- For a long time high-performance computing had a "free-lunch"
 - The density of transistors in chips increased, decreasing the size of integrated circuits
 - same number of transistors with less power
 - more transistors to add functionality
 - The clock speeds steadily rose, increasing the number of operations per second (from MHz to GHz)
- But the free lunch has been over for a few years now
 - We are reaching the limitations of transistor density
 - Increasing clock frequency requires too much power

we used to focus on floating point operations **per second**
now we also think about floating point operations **per Watt**

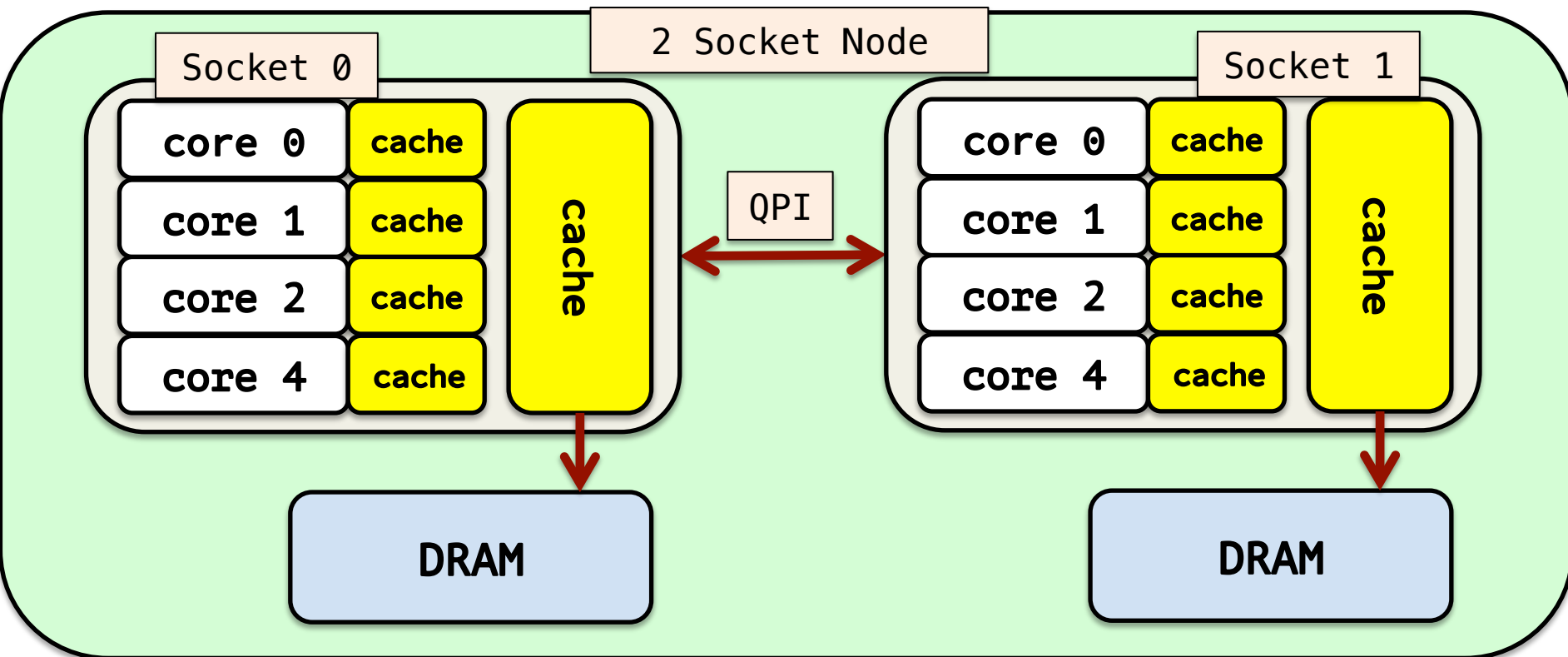
The Solution: Multicore

- The number of transistors is still increasing
 - Sandy Bridge 28 nm, Haswell 22 nm, Broadwell 14 nm
 - Projected hard limit of 5-7 nm
- This has lead to three trends
 - more cores
 - the Sandy Bridge processor on Daint has 8 cores
 - Xeon phi KNL will have 72 cores
 - reducing clock speed
 - simplify/specialize cores
 - An extreme example of this is GPUs, which have in the order of 100/1000s of cores specialized for tasks common in graphics



Multicore HPC Nodes

- A CPU-based node can have multiple sockets, each with multiple cores.
 - cores on a socket share cache and uniform DRAM access

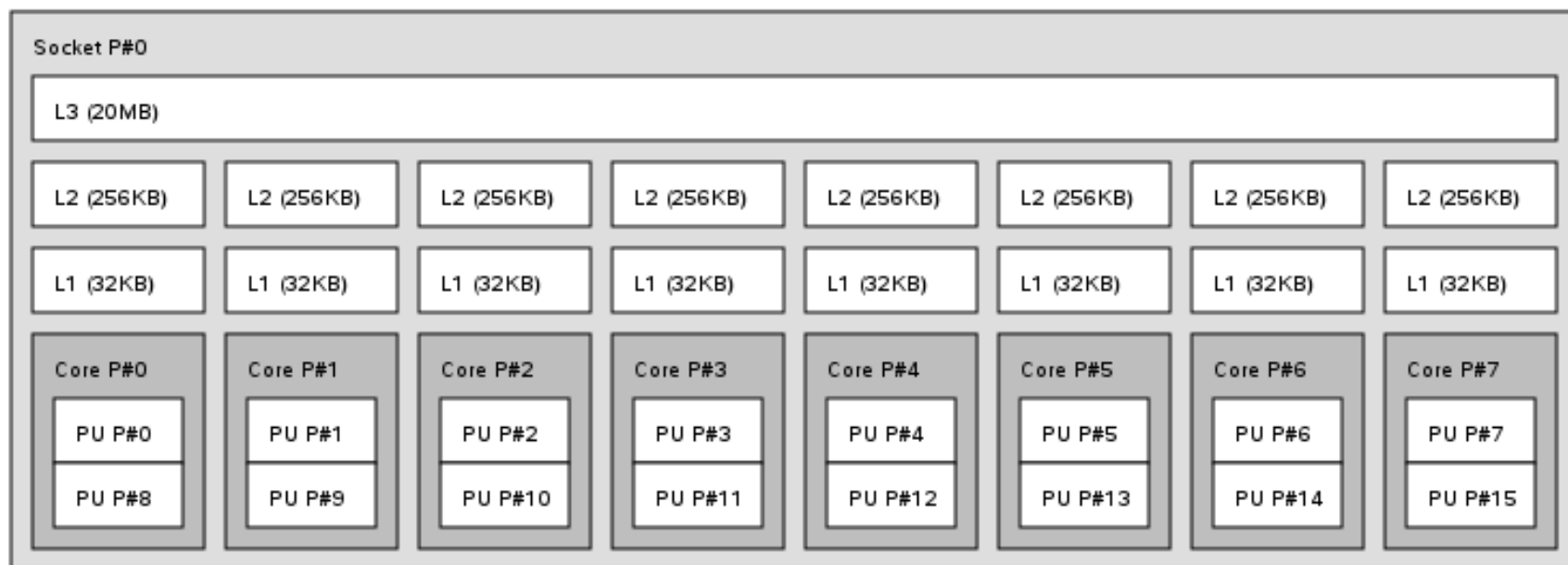




Piz Daint : using lstopo

- the `lstopo` command can be used to visualize node topology

Machine (31 GB)





Piz Dora: using Istopo

- Piz Dora has two Broadwell sockets with 18 cores each
- 3 years between Daint and Dora



The Flat MPI Model

- MPI is the dominant parallelization model in HPC
 - The problem being solved is broken into chunks, one chunk per MPI process
 - Processes communicate via message passing
- The flat MPI model was well-suited to few cores per socket/node
 - Each MPI process sees one core with its cache and memory.
 - It is a logical abstraction for the underlying hardware
- You will learn about MPI later in this course

The Hybrid MPI-OpenMP Model

- The flat MPI model assigns one process per core
 - for 8/16/32/64 ranks per multi-core node
 - can have scaling problems with many nodes/MPI ranks: the amount of data passed around in messages increases as number of ranks increases
 - to take advantage of shared cache and DRAM on a socket, why not use threads on the socket/node, and pass messages between sockets/nodes?
- The hybrid MPI-OpenMP model has light-weight threads that **share on-node memory**.

This course is about OpenMP, however you could use other threading technologies like **pthread**s, **Intel Threading Building Blocks**, **HPX** or **C++11 threads**.

NUMA

- On a multi-socket node each socket has its own DRAM memory
- However the operating system presents all memory as one unified memory space
 - threads can read and write to memory that is local to other sockets.
 - non-local accesses (to local memory on another socket) are slower than local accesses
- This memory architecture is called **Non-Uniform Memory Access (NUMA)**
 - “**NUMA effects**” arise when threads frequently access memory on a different NUMA domain.

Avoiding NUMA Effects

- Avoid them: hybrid MPI-OpenMP with **at least one** rank per NUMA node
 - threads in an MPI rank all run on the same NUMA node
 - explicit message passing via MPI between NUMA nodes
- Use first-touch memory allocation
 - by default the Linux kernel assigns memory pages to the NUMA node that first "touches" the memory
 - it is not the allocation of memory, but the first write to the memory that is important
 - the thread that will be accessing a part of memory should initialize it
 - memory allocation methods that initialize memory values are to be avoided, e.g. `calloc` and `std::allocator<>`

NUMA First Touch Example

```
const int N = 1000000000;  
double* x = (double*)malloc(N*sizeof(double));  
double* x = (double*)malloc(N*sizeof(double));  
double* y = (double*)malloc(N*sizeof(double));  
  
// initialize memory as it will be used  
#pragma omp parallel for  
for(i=0; i<n; ++i) {  
    x[i] = ...;  
    y[i] = ...;  
    z[i] = ...;  
}  
  
#pragma omp parallel for  
for(i=0; i<n; ++i)  
    z[i] = x[i] + y[i];
```

Affinity

- **CPU affinity** is the pinning of a process or thread to a core
 - If the operating system interrupts the thread, it doesn't migrate it to another core
 - For most HPC scenarios where only one application is running on a node, these interruptions are short
 - Some multi-core architectures (e.g. IBM BG/Q) require multiple threads/core to achieve peak performance
 - On x86 the best results are obtained with 1 or 2 threads per core
- **Memory affinity** is allocation of memory as closely as possible to the cores that will be accessing it
 - achieved by first touch or MPI between NUMA nodes



Controlling Thread Affinity on Daint

- On Cray systems the `srun` command used to launch jobs provides flags for setting thread affinity
 - see the man page for all options `> man srun`
- The best configuration can be system and application dependent
- letting the runtime schedule threads is useful when using libraries that create additional threads

```
> srun -n1 -c8 -hint=nomultithread ./a.out  
> srun -n2 -c4 -hint=nomultithread ./a.out
```

8 cores

2 MPI ranks with 4 cores each



OpenMP

What is OpenMP?

Open Multi
Processing

- The OpenMP standard is an API with **compiler directives**, a **run time library**, and **environment variables** for writing **parallel shared memory** applications in C, C++ and Fortran.
 - supported by compiler vendors in HPC, including GNU, Intel, Cray and PGI compilers. Also Clang and MSVC.
- OpenMP compilers allow programmers to tell the compiler where and how to parallelize key loops and tasks with directives
 - The user does not have the complication of explicitly managing threads, as would be the case with a low level threading library like pthreads.

Goals of OpenMP

- Standardization
- Lean
 - concise and simple set of directives
 - possible to get good speedup with a handful of directives
 - but each new release gets more complicated (now at 4.5)
- Ease of use
 - you can incrementally add it to code without major changes
 - in practice refactoring is required to get performance
- Portability
 - Supported by range of compilers and platforms

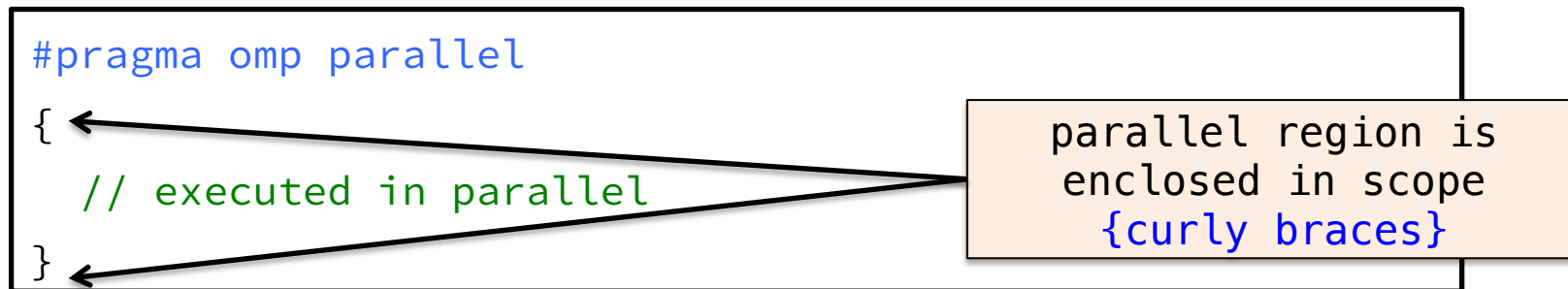
Should I use OpenMP?

- OpenMP + MPI often doesn't perform as well as flat MPI on multicore
 - at least not without significant developer effort
- But for many-core architectures like Xeon Phi flat MPI won't scale as well
 - lightweight threading is required
- OpenMP is the defacto standard for Fortran, and path of least resistance existing codes
- Other options exist for C and C++
 - look into them if developing new code
 - e.g. Intel TBB, Stella HPX, Cilk++

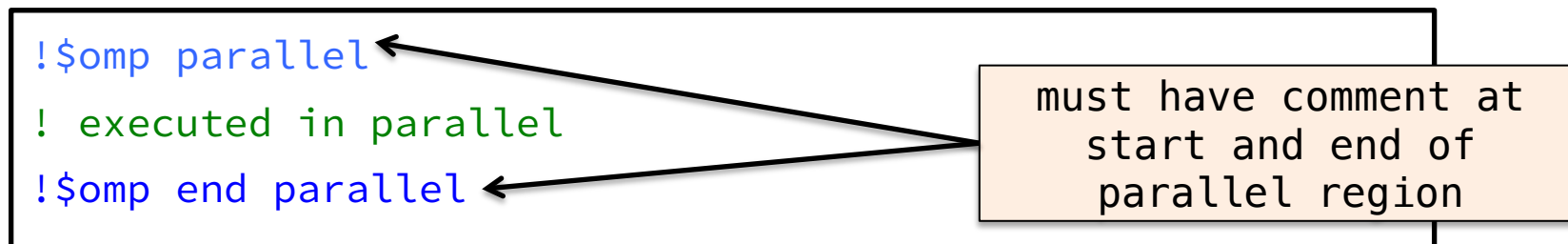


OpenMP Compiler Directives

- In C and C++ parallel regions are scopes that are marked with `#pragma omp parallel`



- In Fortran parallel regions are indicated with specially-formatted comments

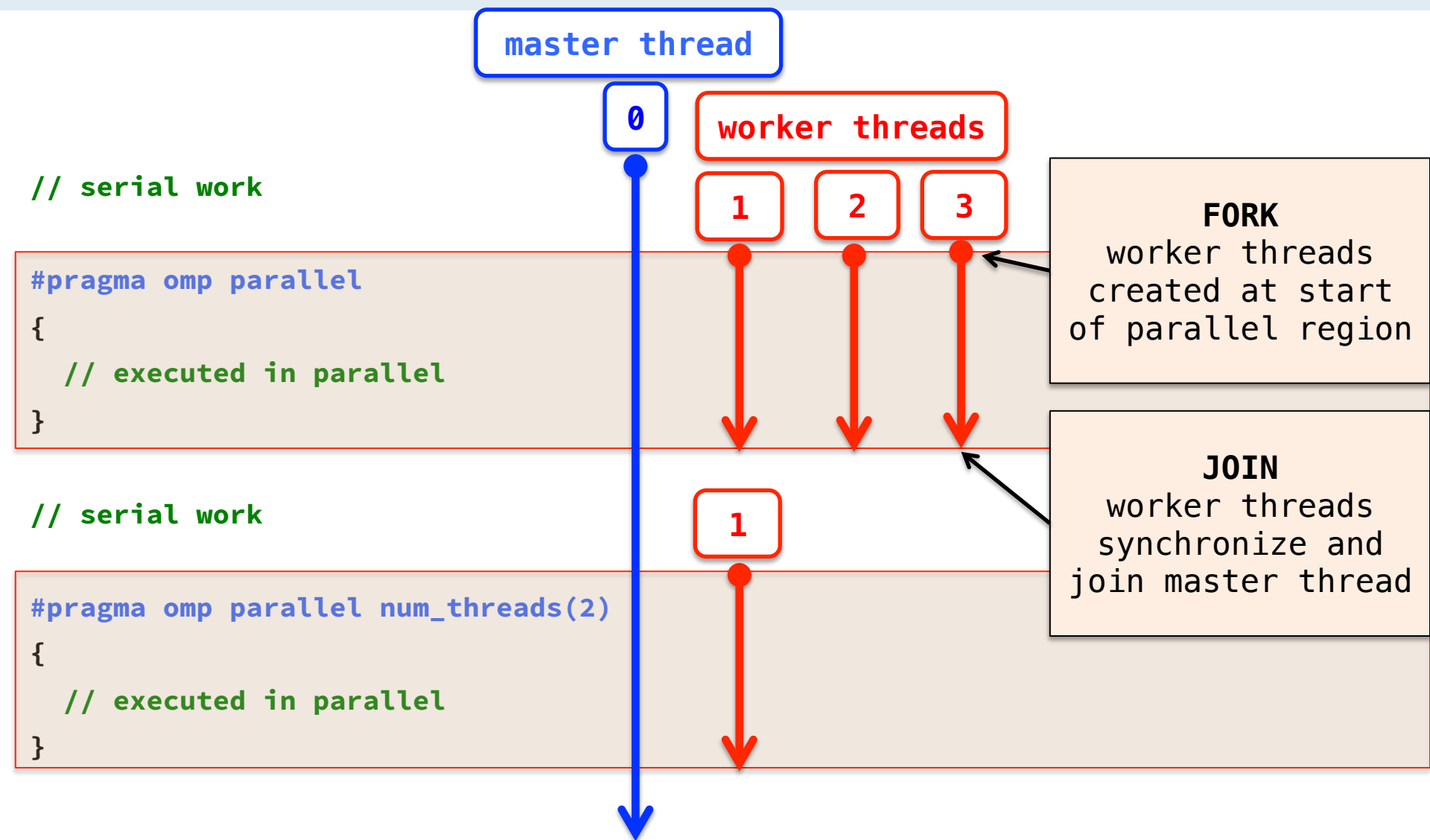


Fork and Join Model

- OpenMP uses a **fork and join** model for threading
- The application starts with a master thread
 - **FORK**: a team of parallel worker threads is started at the beginning of each parallel block
 - the block is executed in parallel by each thread
 - **JOIN**: the worker threads are synchronized at the end of the parallel block and join with the master thread.
- Threads are numbered 0:N-1, where N is the total number of threads
- The master thread is always numbered 0.



Fork and Join Illustrated



Compiling OpenMP

- Most compilers require a flag to enable OpenMP compilation
 - without a flag the `#pragma` or `!$omp` directives are ignored by the compiler and a serial application is created
- compilers that don't understand OpenMP will simply ignore the directives (no portability problems).

Cray	: on by default for <code>-O1</code> and greater, disable with <code>-h noomp</code>
Intel	: off by default, enable with <code>-openmp</code>
GNU	: off by default, enable with <code>-fopenmp</code>
PGI	: off by default, enable with <code>-mp</code>



Running OpenMP applications

- The default number of threads is set with an environment variable `OMP_NUM_THREADS`
- There has to be at least one core per thread
 - multiple threads on a single core have to share resources on the core
 - Typically we assign as many cores as there are threads, and let the runtime schedule them

```
> ftn -fopenmp app.ftn -o app.exe
```

compile openmp code

```
> export OMP_NUM_THREADS=8
```

set 8 threads

```
> srun -n1 -c8 --hint=nomultithread ./app.exe
```

run on 8 cores (on Cray)

Exercises: before starting

- The source code for exercises and slides are in the repository
 - to get a copy in your local path

```
> ssh -X username@ela.cscs.ch  
> ssh -X daint  
> cd $SCRATCH  
> git clone https://github.com/eth-cscs/SummerSchool2016
```

let us know if you have a problem
getting the code

- for these examples we will use the gnu compiler, for which
a script to set up the environment is provided

```
> source setup.sh  
> CC hello_world.cpp -fopenmp  
> ftn hello_world.f90 -fopenmp
```

we use the GNU compiler because the Cray compiler aggressively replaces simple loops with BLAS calls in the absence of OpenMP directives, which make it difficult to compare OpenMP versions of a code.

Exercise 1: Compiling and running

- Open the test code [hello_world.cpp/f90](#)
 - what do you expect the output to be?

- Compile:

choose C++ or Fortran version

```
> module swap PrgEnv-cray PrgEnv-gnu  
> CC hello_world.cpp -fopenmp  
> ftn hello_world.f90 -fopenmp
```

- Then run


```
> OMP_NUM_THREADS=1 srun -c1 -n1 -hint=nomultithread ./hello_world.exe
```

```
...
```

```
...
```

```
> OMP_NUM_THREADS=8 srun -c8 -n1 -hint=nomultithread ./hello_world.exe
```

shorthand for setting number of threads



- Is the output what you expected? Why?

Exercise 1: Affinity

- go to the [openmp/exercises/affinity](#) path
- Compile:

```
> cd SummerSchool2016/openmp/exercises/affinity  
> make
```

- Experiment with affinity
 - try different numbers of threads and srun flags

```
> srun ./test.mpi  
> OMP_NUM_THREADS=1 srun -hint=nomultithread ./test.mpi  
> OMP_NUM_THREADS=1 srun ./test.mpi  
> OMP_NUM_THREADS=8 srun -c8 ./test.mpi  
> OMP_NUM_THREADS=8 srun -c8 -n1 ./test.mpi  
> OMP_NUM_THREADS=8 srun -c8 -n1 -hint=nomultithread ./test.mpi
```

Runtime Library

- OpenMP has runtime library routines for controlling your application, including
 - `int omp_get_thread_num()`
 - get id of current thread
 - `int omp_get_num_threads()`
 - number of threads in **current** parallel region
 - `int omp_get_max_threads()`
 - default number of threads in parallel regions: corresponds to OMP_NUM_THREADS environment variable
 - `double omp_get_wtime()`
 - accurate timing function: returns double
- There are many others, however these are the most commonly used

Runtime Library

- The runtime library requires that the omp header/module is included

```
#include <omp.h>
...
int threads = omp_get_max_threads();
int outside = omp_get_num_threads();
int inside;
#pragma omp parallel
{
    inside = omp_get_num_threads();
}
printf("%d in, %d out, %d max \n",
       inside, outside, threads);
```

```
use omp_lib
...
integer :: threads, inside, outside
threads = omp_get_max_threads()
outside = omp_get_num_threads()
!$omp parallel
inside = omp_get_num_threads()
!$omp end parallel
print *, inside, ' in ', outside, ' out ',
        threads, ' max'
```

```
> OMP_NUM_THREADS=8 ./a.out
8 in, 1 out, 8 max
```

Synchronization

- Sometimes you need to synchronize threads inside a parallel region, for example
 - tasks that have to be done by only one thread
 - when multiple threads have to update memory, and we need to ensure that the value in memory is consistent
 - threads have to wait for others to finish before continuing
- OpenMP provides directives that can be used to indicate such regions
- **WARNING:** synchronization and serial code regions can quickly limit the potential speed up from parallelism.

Synchronization Example

- The intention of the code below is to print a hello world message, similarly to the hello world example earlier:

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    printf("hello world from thread %d", tid);
}
```

```
> OMP_NUM_THREADS=8 ./a.out
???
```

- each thread gets a private copy of `tid`, but the output could get messed up because all threads write to `stdout` at the same time.

master Directive

- The master directive indicates sections that are to be executed only by the master thread
 - the master thread is always thread 0

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp master
    printf("hello world from thread %d", tid);
}
```

```
> OMP_NUM_THREADS=2 ./a.out
hello world from thread 0
```

- only the master thread prints its `tid` value.

single Directive

- Block will only be executed by the **first thread** to arrive at the block
 - varies from one run to the next

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp single
    printf("hello world from thread %d", tid);
}
```

```
> OMP_NUM_THREADS=8 ./a.out
hello world from thread 3
> OMP_NUM_THREADS=8 ./a.out
hello world from thread 6
```

- only one thread will print message.

critical Directive

- All threads will execute block, **one at a time**
 - in order that the threads arrive at the block: varies each time application is run

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp critical
    printf("hello world from thread %d", tid);
}
```

```
> OMP_NUM_THREADS=3 ./a.out
hello world from thread 1
hello world from thread 0
hello world from thread 2
```

- each thread prints one message, **in order of thread arrival at the critical block**

barrier Directive

- All threads wait at barrier until all of the threads are at barrier, before continuing

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp barrier
    printf("hello world from thread %d", tid);
}
```

```
> OMP_NUM_THREADS=3 ./a.out
hello world from thread 0 hello world from thread 2
hello world from thread 1
```

- in this case, this doesn't solve anything: all threads try to write to stdout at the same time when they simultaneously leave the barrier

Exercise 2

- Go back to the [hello_world.cpp/f90](#) example, and add appropriate synchronization directives
 - do you see expected behavior now?
- Look at [sum_threads.cpp/f90](#)
 - what was the intended output?
 - run the example and compare the actual output.
 - can you add a synchronization directive to get the expected result?

```
> source setup.sh
> CC sum_threads.cpp -fopenmp
... or
> ftn sum_threads.f90 -fopenmp
> OMP_NUM_THREADS=8 srun -c8 -n1 -hint=nomultithread ./a.out
```

Shared Memory Model

- OpenMP uses a shared memory model
- All threads can read and write to the same memory locations simultaneously
- By default variables are shared, so one copy is used by all threads
- The result of computations where multiple threads attempt to read/write to a variable are **undefined**
 - see the **sum_threads** example in the previous exercises for a simple example
 - this is a very common parallel programming bug called a **race condition**

Variable Scoping

- OpenMP provides clauses that describe how variables should be shared between threads
 - **shared**: all variables access the same copy of a variable.
 - this is the default behavior
 - **WARNING**: take care when writing to shared variables
 - **private**: each thread gets its own copy of the variable
 - private copy is uninitialized
 - use **firstprivate** to initialize variable with value from master

Variable Scoping

```
int tid;  
const int num_threads = omp_get_num_threads();  
#pragma omp parallel shared(num_threads) private(tid)  
{  
    tid = omp_get_thread_num();  
    #pragma omp critical  
    printf("hello world from thread %d of %d", tid, num_threads);  
}
```

Private Variables in C99/C++

- Variables that are declared inside a parallel region are private by default
 - this isn't possible in Fortran because variables must be declared at the top of each subroutine/program/function

```
int tid;  
#pragma omp parallel private(tid)  
{  
    tid = omp_get_thread_num();  
    // ... use tid for local computation  
}
```

using private clause

```
#pragma omp parallel  
{  
    int tid = omp_get_thread_num();  
    // ... use tid for local computation  
}
```

best practice: declare variable in scope



Exercises 3

- Now we finally have all the tools needed to fix [hello_world.cpp/f90](#)

Work Sharing: for/do loops

- A common target for parallelization is loops without loop carried dependencies
 - for example, adding two vectors:

```
double *x, *y, *z;  
int n;  
for(int i=0; i<n; ++i) {  
    z[i] = x[i] + y[i];  
}
```

C++

```
real(kind=8) :: x(:), y(:), z(:)  
integer      :: i, n  
do i=1,n  
    z(i) = x(i) + y(i)  
end do
```

Fortran

- we could attempt to parallelize this using the techniques that we have learnt so far...



for/do loops the hard way

```
double *x, *y, *z;
int n;
...
for(int i=0; i<n; ++i) {
    z[i] = x[i] + y[i];
}
```

serial

What a mess!
And error-prone
too: does this
approach still work
if $n < \text{num_threads}$?

```
double *x, *y, *z;
int n;
...
#pragma omp parallel
{
```

calculate loop
bounds for this
thread's chunk

```
int tid = omp_get_thread_num();
int num_threads = omp_get_num_threads();
int work = n/num_threads;
int s = tid*work;
int e = (tid==num_threads-1) ? n : s+work;
for(int i=s; i<e; ++i)
    z[i] = x[i] + y[i];
}
```

parallel



parallel for

- OpenMP provides a directive for `for/do` loops

```
double *x, *y, *z;
int n, i;
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<n; ++i) {
        z[i] = x[i] + y[i];
    }
}
```

C++

loop index
variable i is
private by
default

```
real(kind=8) :: x(:), y(:), z(:)
integer      :: i, n
!$omp parallel
!$omp do
do i=1,n
    z(i) = x(i) + y(i)
end do
!$omp end do
!$omp end parallel
```

Fortran

- Compiler handles loop bounds for you
- there is a compact single-line directive:

```
#pragma omp parallel for
for(i=0; i<n; ++i) {
    z[i] = x[i] + y[i];
}
```

C++

```
!$omp parallel do
do i=1,n
    z(i) = x(i) + y(i)
end do
!$omp end parallel do
```

Fortran

Example: Vector Normalize

1. Open `dot.cpp/f90`
 - this code finds the dot product of two vectors
 - add OpenMP directives to parallelize the code
2. Open `normalize.cpp/f90`
 - write a parallel version of the function/subroutine `normalize_vector()`
 - find the norm of the vector v
 - scale the vector by the norm



Reductions

- Reduction operations reduce a set of values to a single value according to an operation **op**

```
a = initial value  
for i = 1,n  
    a = a op expr
```

e.g. C++ sum reduction

```
double sum = 0.0, v[n];  
for(int i=0; i<n; ++i)  
    sum = sum + v[i];
```

- OpenMP provides a clause of the form **reduction(op:list)** for performing reductions

- a is a scalar variable in list
- expr is a scalar expression that does not reference a
- only certain expressions allowed
e.g. (+,-,*,/,binary ops)

```
double sum = 0.0, v[n];  
#pragma omp parallel for  
#pragma omp reduction(+:sum)  
for(int i=0; i<n; ++i)  
    sum = sum + v[i];
```

Exercise

- Revisit the [dot.cpp/f90](#) exercise, and rewrite it to use a reduction.
 - use [test.sh](#) to see how it scales from 1 to 8 threads
 - try arrays of length 10'000, 100'000 and 1'000'000
 - how does this affect scaling?
- We can apply everything that we have learnt to the example [pi.cpp/f90](#), which computes pi using the trapezoidal rule
 - use [test.sh](#) to test its scaling from 1 to 8 threads
 - is its scaling better or worse? why?



Nested Loops

- If you apply directives nested loops, the outer loop will be parallelized.
- the collapse directive will merge the loops and parallelize both

```
double sum = 0.0, v[n];  
#pragma omp parallel for  
for(int j=0; j<n; ++j) {  
    for(int i=0; i<m; ++i)  
        A[j][i] = f(B[j][i]);  
}
```

n parallel work items

```
double sum = 0.0, v[n];  
#pragma omp parallel for collapse(2)  
for(int j=0; j<n; ++j) {  
    for(int i=0; i<m; ++i)  
        A[j][i] = f(B[j][i]);  
}
```

n*m parallel work items

- use if n is small relative to OMP_NUM_THREADS and/or the amount of work in f is significant



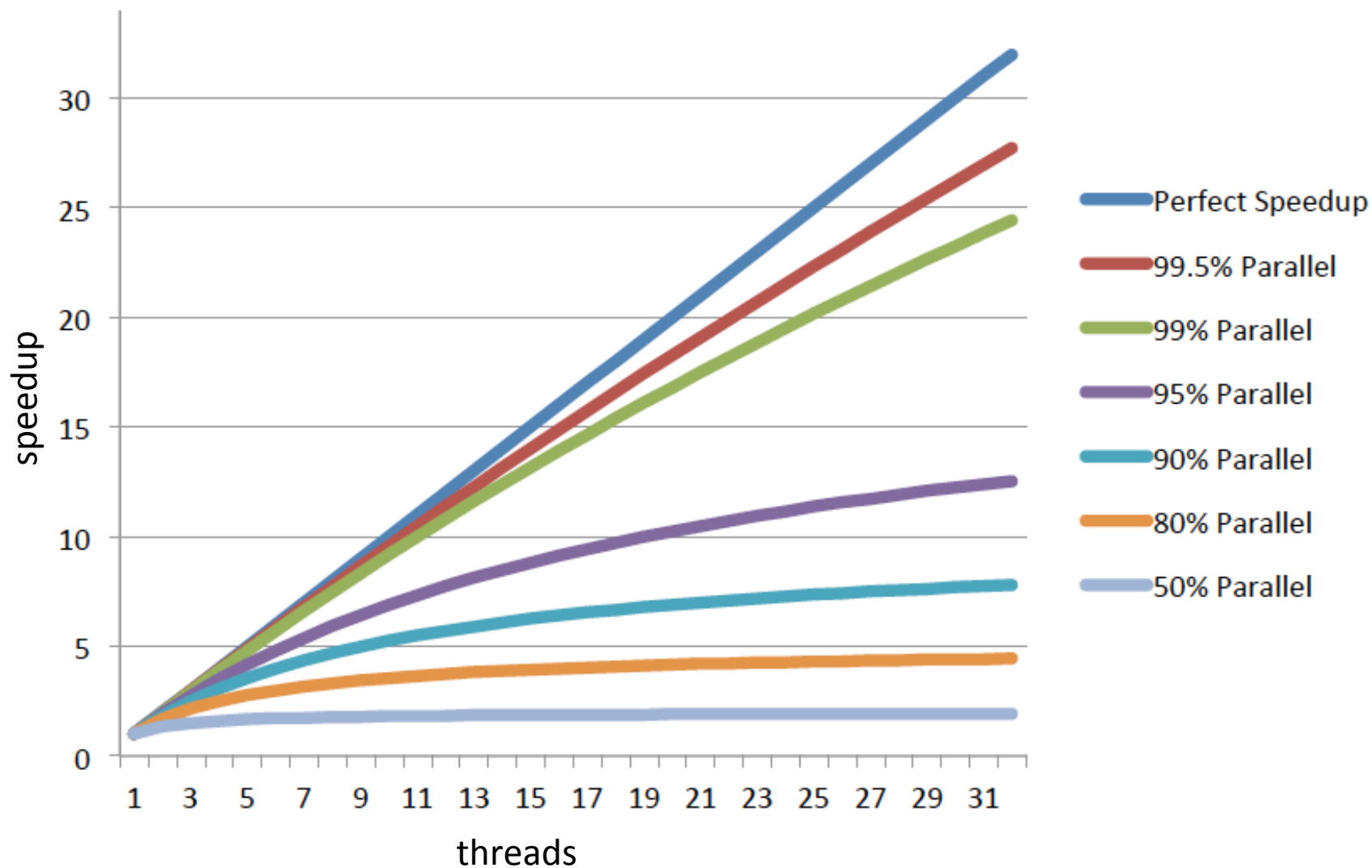
Limitations to Parallel Speedup

Limitations To Parallel Speedup

- The amount of speedup that you will get from OpenMP is dependent on many factors including
 - having enough work to keep all threads busy
 - the ratio of sequential to parallel work
- Amdahl's Law is used to define the maximum possible speedup when only parts of a code can be parallelized
 - $T(n) = T(1) * (B + (1-B)/n)$
 - i.e. as $n \rightarrow \infty$, the best speedup possible is $B * T(1)$
 - where $T(n)$ is time to solution for n threads, and B is the fraction of the algorithm that is strictly serial



Amdahl Illustrated



Amdahl's Law

- Our application would ideally scale to many threads
- But just 1% serial code in an algorithm means that it won't scale more than 100 times: no matter how many threads/nodes we use.
- While adding OpenMP directives to an existing sequential code is easy, the modifications required to get really good scaling might require significant restructuring
 - there is no free lunch!

OpenMP 4.0 and Accelerators

- OpenMP 4.0 adds directives for using accelerators
 - Including GPUs and Intel Xeon Phi (MIC)
 - GCC 5.0 and the Intel compilers support of Xeon Phi offload
 - Currently there is no OpenMP 4.0 GPU offload
- Similar concept to the OpenACC directives that you will see for GPUs next week

Should You use OpenMP?

- **Disclaimer:** this might be controversial
- you should consider OpenMP if
 - you are using Fortran
 - you have to port an application that relies on OpenMP libraries
- However, if using C++ there are alternatives
 - namely Intel TBB & Stellar HPX
 - these are vanilla C++: not language extensions
 - Can be used to express more different types of parallelism more easily
 - e.g. task based parallelism