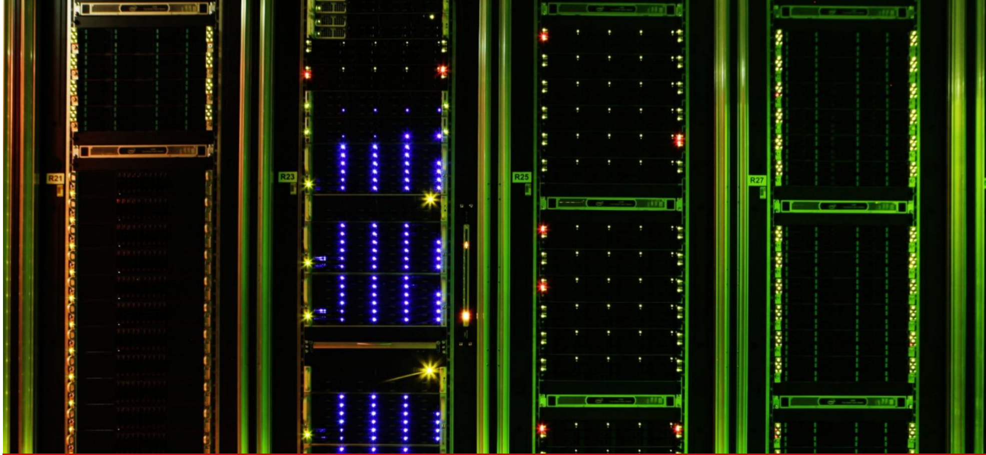


Parallel Programming

Single-core optimization, MPI, OpenMP, and hybrid programming

Nicolas Richart
Emmanuel Lanti

Course based on V. Keller's lecture notes

27th of October - 1st of November 2024

SCITAS

OpenMP

SCITAS

Goal of this section

- Understand the context of shared memory
- Understand more in detail the architecture of a node
- Get familiar with the OpenMP execution and memory model
- Getting some speedup with Task Level Parallelism

EPFL

Releases history, present and future

- October 1997: Fortran version 1.0
- Late 1998: C/C++ version 1.0
- June 2000: Fortran version 2.0
- April 2002: C/C++ version 2.0
- June 2005: Combined C/C++ and Fortran version 2.5
- May 2008: Combined C/C++ and Fortran version 3.0
- **July 2011: Combined C/C++ and Fortran version 3.1**
- July 2013: Combined C/C++ and Fortran version 4.0
- November 2015: Combined C/C++ and Fortran version 4.5
- November 2018: Combined C/C++ and Fortran version 5.0
- November 2020: Combined C/C++ and Fortran version 5.1
- November 2021: Combined C/C++ and Fortran version 5.2

Terminology

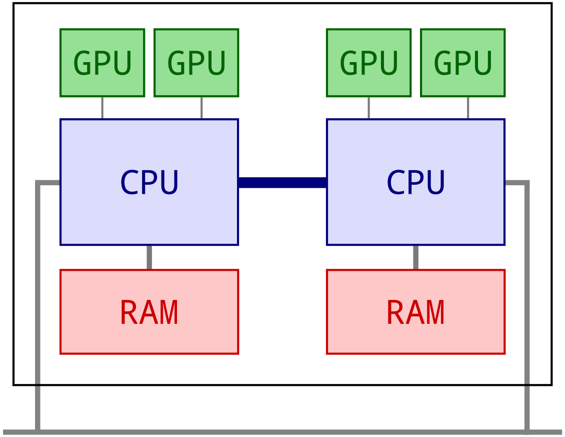
Selected extract of the specification

- Specification:
  - Full specification
  - RefCard
- Terms:

thread	an execution entity with a stack and a static memory (*threadprivate memory*)
OpenMP thread	a *thread* managed by the OpenMP runtime
processor	an hardware unit on which one or more *OpenMP thread* can execute
directive	a base language mechanism to specify OpenMP program behavior
construct	an OpenMP executable directive and the associated statement, loop nest or structured block, if any, not including the code in any called routines. That is, the lexical extent of an executable directive.

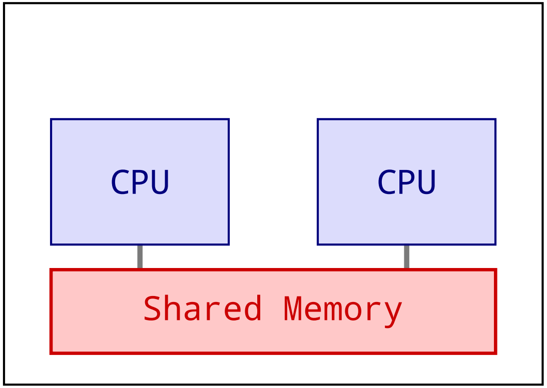
Memory Model

Shared memory



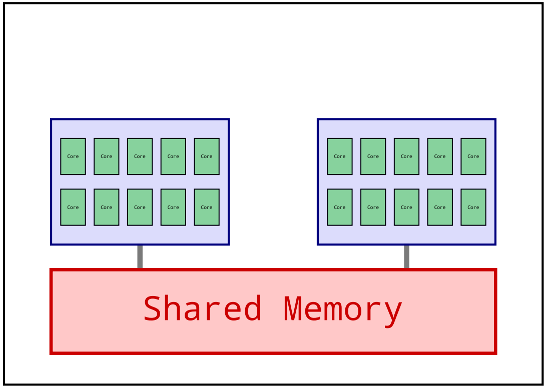
Memory Model

Shared memory



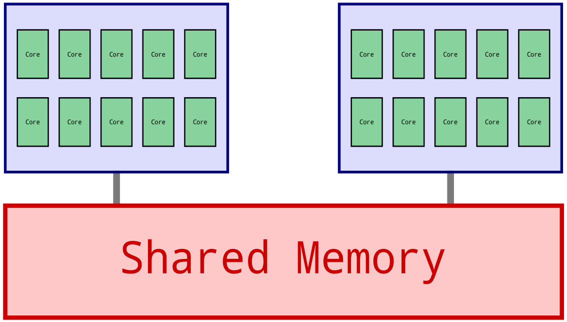
Memory Model

Shared memory



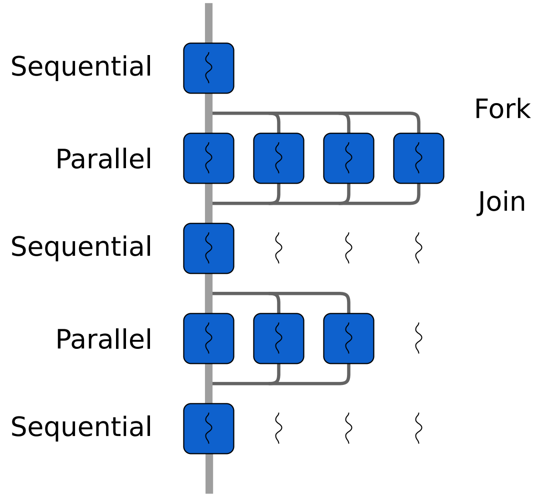
Memory Model

Shared memory



Execution Model

Fork/join



Compiling an OpenMP code

pragmas and compilation

- OpenMP directives are written as pragmas: `#pragma omp`
- Use the conditional compilation flag `#if defined _OPENMP` for the preprocessor

Compiling an OpenMP code

pragmas and compilation

- OpenMP directives are written as pragmas: `#pragma omp`
- Use the conditional compilation flag `#if defined _OPENMP` for the preprocessor

- Compilation using the GNU compiler:

```
$> g++ -fopenmp ex1.c -o ex1
```

- Compilation using the Intel compiler:

```
$> icpc -qopenmp ex1.c -o ex1
```

Hello World in C++

Simple version

openmp/hello.cc

```cpp
#include <iostream>
#include <omp.h>

int main() {

#pragma omp parallel
  {
    auto mysize = omp_get_num_threads();
    auto myrank = omp_get_thread_num();
    std::printf("Hello from thread %i out of %i\n", myrank, mysize);
  }

  return 0;
}
```

Hello World in C++

Simple version

openmp/hello.cc

```cpp
#include <iostream>
#include <omp.h>

int main() {

#pragma omp parallel
  {
    auto mysize = omp_get_num_threads();
    auto myrank = omp_get_thread_num();
    std::printf("Hello from thread %i out of %i\n", myrank, mysize);
  }

  return 0;
}
```

```
$ OMP_NUM_THREADS=4 ./openmp/hello
Hello from thread 2 out of 4
Hello from thread 1 out of 4
Hello from thread 0 out of 4
Hello from thread 3 out of 4
```

Hello World in C++

With condition compilation

openmp/hello_cond.cc

```cpp
int main() {
  int mysize = 1;
  int myrank = 0;

#if defined(_OPENMP)
#pragma omp parallel
  {
    mysize = omp_get_num_threads();
    myrank = omp_get_thread_num();
#endif
    std::printf("Hello from thread %i out of %i\n", myrank, mysize);
#if defined(_OPENMP)
  }
#endif
  return 0;
}
```

Number of concurrent threads

- Default implementation dependent (usually max hardware thread)
- At runtime in the code

```
omp_set_num_threads(nthreads);
```

- With en environment variable

```
$> export OMP_NUM_THREADS=4
```

The `parallel` construct

This is the mother of all constructs in OpenMP. It starts a parallel execution.

Syntax

```
#pragma omp parallel [clause[[,] clause]...]
{
  structured-block
}
```

where *clause* is one of the following:

- **if** or **num_threads** : conditional clause
- **default(private** | **firstprivate** | **shared** | **none)** : default data scoping
- **private(*list*)**, **firstprivate(*list*)**, **shared(*list*)** or **copyin(*list*)** : data scoping
- **reduction(*operator* : *list*)**

Hello π

- In the `pi.cc` add a function call to get the number of threads.
- Compile using the porper options for OpenMP
- Test that it works by varying the number of threads **export OMP_NUM_THREADS**
- To vary the number of threads in a **sbatch** job you can set the number of threads to the number of cpus per task.

```
#!/bin/bash
#SBATCH -c <nthreads>

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
<my_openmp_executable>
```

Worksharing constructs

Work-sharing constructs are possible in three “flavours” :

- **sections** construct
- **single** construct
- **workshare** construct (only in Fortran)

Worksharing constructs

The **sections** construct

Syntax

```
#pragma omp [parallel] sections [clause]
{
  #pragma omp section
  {
    code_block
  }
}
```

where *clause* is one of the following:

- **private(*list*)**, **firstprivate(*list*)**, **lastprivate(*list*)**
- **reduction(*operator* : *list*)**
- Each **section** within a **sections** construct is assigned to one and only one thread

A `sections` construct

Example

openmp/sections.cc

```cpp
#pragma omp parallel sections num_threads(4)
  {
#pragma omp section
    std::printf("Thread %i handling section 1\n", omp_get_thread_num());
#pragma omp section
    std::printf("Thread %i handling section 2\n", omp_get_thread_num());
#pragma omp section
    std::printf("Thread %i handling section 3\n", omp_get_thread_num());
  }
```

A sections construct

Example

openmp/sections.cc

```
#pragma omp parallel sections num_threads(4)
  {
#pragma omp section
    std::printf("Thread %i handling section 1\n", omp_get_thread_num());
#pragma omp section
    std::printf("Thread %i handling section 2\n", omp_get_thread_num());
#pragma omp section
    std::printf("Thread %i handling section 3\n", omp_get_thread_num());
  }
```

```
$ ./openmp/sections
Thread 0 handling section 1
Thread 1 handling section 2
Thread 2 handling section 3
```

Worksharing constructs

The **single** construct

Only one thread (usualy the first entering thread) executes the **single** region.

Syntax

```
#pragma omp single [clause[[,] clause] ...]
{
  structured-block
}
```

where *clause* is one of the following:

- **private(*list*)**, **firstprivate(*list*)**
- **nowait**

The master directive

Deprecated for **masked**

Only the master thread execute the section. It can be used in any OpenMP construct

Syntax

```
#pragma omp master
{
  structured-block
}
```

The Worksharing-loop construct

The **for** construct

Parallelization of the following loop

Syntax

```
#pragma omp for [clause[[,] clause] ... ]
{
  for-loop
}
```

where *clause* is one of the following:

- **schedule(*kind[, chunk_size]*)**
- **collapse(*n*)**
- **ordered**
- **private(*list*)**, **firstprivate(*list*)**, **lastprivate(*list*)**

Example of for construct

openmp/for.cc

```
#pragma omp parallel num_threads(2)
  {
    auto myrank = omp_get_thread_num();
#pragma omp for
    for (int i = 0; i < 6; ++i) {
      std::printf("Thread %i handling i=%i\n", myrank, i);
    }
  }
```

Example of for construct

openmp/for.cc

```cpp
#pragma omp parallel num_threads(2)
  {
    auto myrank = omp_get_thread_num();
#pragma omp for
    for (int i = 0; i < 6; ++i) {
      std::printf("Thread %i handling i=%i\n", myrank, i);
    }
  }
```

```
$ ./openmp/for
Thread 0 handling i=0
Thread 0 handling i=1
Thread 0 handling i=2
Thread 1 handling i=3
Thread 1 handling i=4
Thread 1 handling i=5
```

Compute π in parallel

First try at parallel code

- Add a **parallel for** work sharing construct around the integral computation
- Run the code
- Run the code
- Run the code
- What can you observe on the value of π ?

The Synchronization constructs

The **critical** construct

Restricts execution of the associated structured block to a single thread at a time

Syntax

```
#pragma omp critical [(name) [[,] hint(hint-expression)]]
{
  structured-block
}
```

- **name** optional to identify the construct
- **hint-expression** information on the expected execution
  - `omp_sync_hint_none`
  - `omp_sync_hint_uncontended`
  - `omp_sync_hint_contended`
  - `omp_sync_hint_nonspeculative`
  - `omp_sync_hint_speculative`

Compute π in parallel

Naive reduction

- To solve the data race condition from the previous exercise we can protect the computation of the sum
- Add a **critical** directive to protect the sum
- Run the code
- What can you observe on the execution time while varying the number of threads

The Synchronization constructs

The **barrier** construct

Specifies an explicit barrier.

Syntax

```
#pragma omp barrier
```

The Synchronization constructs

The **atomic** construct

Ensures a specific storage location is accessed atomically.

Syntax

```
#pragma omp atomic [clause[[,] clause] ... ]
statement
```

where *clause* is one of the following:

- *atomic-clauses* **read**, **write**, **update**
- *memory-order-clauses* **seq_cst**, **acq_rel**, **releases**, **acquire**, **relaxed**
- or one of **capture**, **compare**, **hint(hint-expression)**, **fail(seq_cst | acquire | relaxed)**, or **weak**

Data sharing clauses

What are the variables values

- Most common source of errors
- Determine which variables are **private** to a thread, which are **shared** among all the threads
- In case of a **private** variable the variable values can be defined using:
  - **firstprivate** defines the value when entering the region
  - **lastprivate** defines the value when exiting the region (OpenMP 5.1 in C/C++)
- **default(private** | **firstprivate** | **shared** | **none)** can be specified
  default(none) means each variables should appear in a **shared** or **private** list

Data sharing clauses

shared and **private**

These attributes determines the scope (visibility) of a single or list of variables

Syntax

```
shared(list1), private(list2)
```

- The **private** clause: the data is private to each thread and non-initialized. Each thread has its own copy. `#pragma omp parallel private(i)`
- The **shared** clause: the data is shared among all the threads. It is accessible (and non-protected) by all the threads simultaneously. `#pragma omp parallel shared(array)`

Data sharing clauses

firstprivate and lastprivate

These clauses determines the attributes of the variables within a parallel region:

Syntax

```
firstprivate(list1), lastprivate(list2)
```

- The **firstprivate** super-set of **private**, variable is initialized to a copie of variable before the region
- The **lastprivate** super-set of **private** the value of the last thread exiting the region is copied

Data sharing clauses

Example

openmp/private.cc

```
  std::printf("Thread %i sees, a, b, c: %i, %i, %g (before)\n",
              omp_get_thread_num(), a, b, c);

#pragma omp parallel num_threads(3), private(a), firstprivate(b)
  {
    std::printf("Thread %i sees, a, b, c: %i, %i, %g (inside)\n",
                omp_get_thread_num(), a, b, c);
    c = -1e-3;
  }

  std::printf("Thread %i sees, a, b, c: %i, %i, %g (after)\n",
              omp_get_thread_num(), a, b, c);
```

Data sharing clauses

Example

openmp/private.cc

```cpp
  std::printf("Thread %i sees, a, b, c: %i, %i, %g (before)\n",
              omp_get_thread_num(), a, b, c);

#pragma omp parallel num_threads(3), private(a), firstprivate(b)
  {
    std::printf("Thread %i sees, a, b, c: %i, %i, %g (inside)\n",
                omp_get_thread_num(), a, b, c);
    c = -1e-3;
  }

  std::printf("Thread %i sees, a, b, c: %i, %i, %g (after)\n",
              omp_get_thread_num(), a, b, c);
```

```
$ ./openmp/private
Thread 0 sees, a, b, c: 1, 2, 3 (before)
Thread 0 sees, a, b, c: 1839769744, 2, 3 (inside)
Thread 1 sees, a, b, c: 12789424, 2, 3 (inside)
Thread 2 sees, a, b, c: 12801392, 2, -0.001 (inside)
Thread 0 sees, a, b, c: 1, 2, -0.001 (after)
```

Compute π in parallel

Naive reduction improved

- Create a local variable per thread
- Make each thread compute it's own sum
- After the computation of the integral use a **critical** directive to sum the local sum to a **shared** sum

Loop clauses

reduction clause

Syntax

```
reduction(reduction-identifier : list)
```

- *reduction-identifier*: one of the operation +, −, *, &, |, ^, &&, ||
- *list* item on which the reduction applies
- example: `#pragma omp for reduction(+: sum)`

Compute π in parallel

Naive reduction improved

- Use the **reduction** clause
- Compare the timings to the previous versions

Loop clauses

schedule clause

Syntax

```
schedule([modifier [, modifier] : ] kind [, chunk_size])
```

- *kind*
  - **static** iterations divided in chunks sized **chunk_size** assigned to threads in a round-robin fashion
  - **dynamic** iterations divided in chunks sized **chunk_size** assigned to threads when they request them until no chunk remains to be distributed
  - **guided** iterations divided in chunks sized **chunk_size** assigned to threads when they request them. Size of chunks is proportional to the remaining unassigned chunks.
  - **auto** The decisions is delegated to the compiler and/or the runtime system
  - **runtime** The decisions is delegated to the runtime system based on ICV

Loop clauses

collapse clause

Syntax

```
collapse(n)
```

Specifies how many loop are combine into a logical space

Example dgemm

collapse(1)

openmp/dgemm.cc

```
#pragma omp parallel for collapse(1) schedule(static, N / nthreads)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      for (int k = 0; k < N; ++k)
        C[i * N + j] += A[i * N + k] * B[k * N + j];
```

Example dgemm

collapse(1)

openmp/dgemm.cc

```
#pragma omp parallel for collapse(1) schedule(static, N / nthreads)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      for (int k = 0; k < N; ++k)
        C[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
$ OMP_NUM_THREADS=1 ../build/openmp/dgemm
DGEMM with 1 threads, collapse(1): 21.1209 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=2 ../build/openmp/dgemm
DGEMM with 2 threads, collapse(1): 40.2308 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=4 ../build/openmp/dgemm
DGEMM with 4 threads, collapse(1): 72.7659 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=1 ../build/openmp/dgemm
```

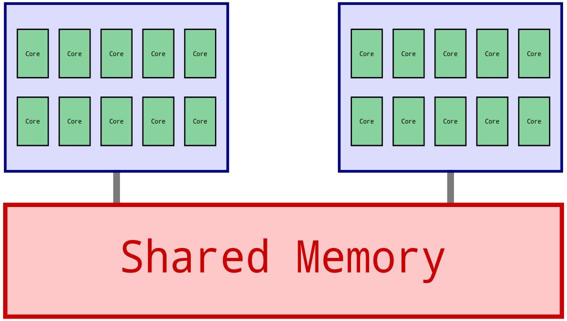
Example dgemm

collapse(2)

```
DGEMM with 1 threads, collapse(2): 20.358 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=2 ../build/openmp/dgemm
DGEMM with 2 threads, collapse(2): 40.0818 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=4 ../build/openmp/dgemm
DGEMM with 4 threads, collapse(2): 72.4462 GFLOP/s (verif 2)
```

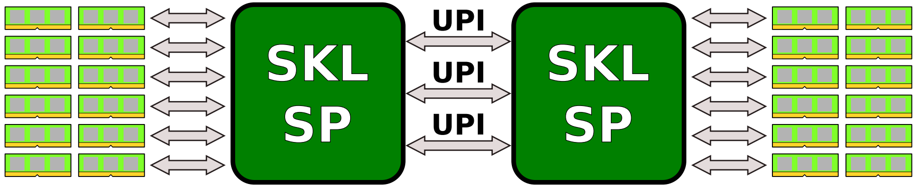
Advanced topics

Idealized model vs reality (NUMA, Sub-NUMA Clusters, Cluster-on-Die)



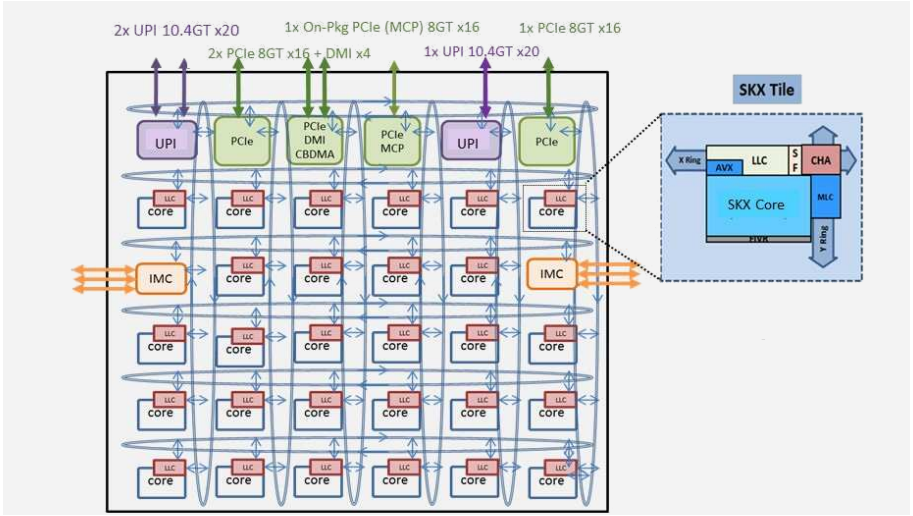
Advanced topics

Idealized model vs reality (NUMA, Sub-NUMA Clusters, Cluster-on-Die)



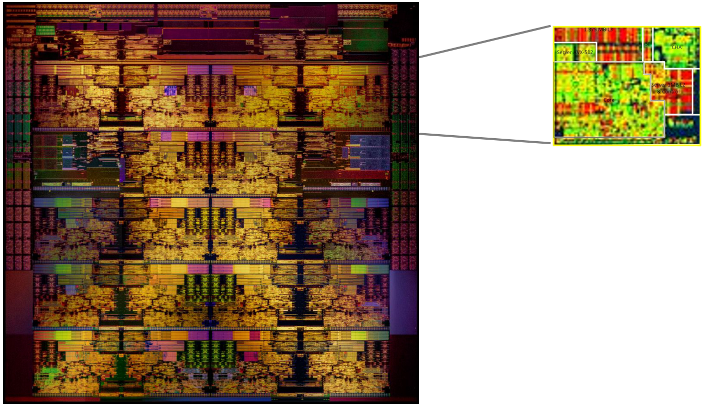
Advanced topics

Idealized model vs reality (NUMA, Sub-NUMA Clusters, Cluster-on-Die)

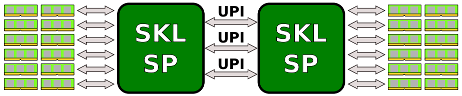


Advanced topics

Idealized model vs reality (NUMA, Sub-NUMA Clusters, Cluster-on-Die)

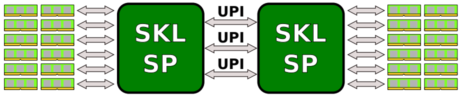


Implications of memory layout



- One thread can only saturate 1 channel
- On memory bound code bandwidth saturate when # of threads $\sim$ # of channels
- If memory allocated on the other processor memory, data go through CPU interconnect (UPI $3 \times 10.4\,\mathrm{GT/s}$)

Implications of memory layout



- One thread can only saturate 1 channel
- On memory bound code bandwidth saturate when # of threads $\sim$ # of channels
- If memory allocated on the other processor memory, data go through CPU interconnect (UPI $3 \times 10.4\,\text{GT/s}$)
- How to mitigate this effects ?
  - Loop schedule
  - Memory first touch
  - Thread placements

Thread Affinity

Thread Affinity Control

- The variable `OMP_PLACES` describes these places in terms of the available hardware.
- The variable `OMP_PROC_BIND` describes how threads are bound to OpenMP places
- The variable `OMP_DISPLAY_AFFINITY` helps to debug the affinity

```
$ OMP_NUM_THREADS=4 OMP_DISPLAY_AFFINITY=true ./openmp/hello
OMP: pid 2115280 tid 2115280 thread 0 bound to OS proc set {0-31}
OMP: pid 2115280 tid 2115285 thread 3 bound to OS proc set {0-31}
OMP: pid 2115280 tid 2115284 thread 2 bound to OS proc set {0-31}
Hello from thread 0 out of 4
Hello from thread 3 out of 4
OMP: pid 2115280 tid 2115283 thread 1 bound to OS proc set {0-31}
Hello from thread 1 out of 4
Hello from thread 2 out of 4
```

Thread Affinity

OMP_PLACES

Possible values for `OMP_PLACES` where each place corresponds to:

threads	a single hardware thread on the device.
cores	a single core (having one or more hardware threads) on the device.
ll_caches	a set of cores that share the last level cache on the device.
numa_domains	a set of cores for which their closest memory on the device is: - the same memory; and - at a similar distance from the cores.
sockets	a single socket (consisting of one or more cores) on the device.

EPFL

Thread Affinity

OMP_PROC_BIND

Possible values for OMP_PROC_BIND:

false	threads not bonded
true	threads are bonded (implementation dependant)
primary	collocate threads with the primary thread
close	place threads close to the master in the places list
spread	spread out threads as much as possible

First touch

- Memory is organized in pages
- When allocating data “nothing” happens
- Pages are allocated on the memory associated to the first thread initializing it

First touch

- Memory is organized in pages
- When allocating data "nothing" happens
- Pages are allocated on the memory associated to the first thread initializing it
- To mitigate the problem, initialize the arrays in same order they are accessed

Data race, false-sharing

- Data race:
  - Data accessed by multiple threads without protection
  - Lead to undetermined results

Data race, false-sharing

- Data race:
  - Data accessed by multiple threads without protection
  - Lead to undetermined results
- False sharing
  - Data smaller than cache-line size
  - Multiple threads accessing data in the same cache line will poison each other caches

Runtime routines

Sub set of the routines in OpenMP

- `omp_get_num_threads()` : number of threads in the current region
- `omp_get_thread_num()` : id of the current thread
- `omp_get_max_threads()` : upper bound to the number of threads that could be used
- `omp_get_wtime()` : wall clock time in seconds
- `omp_get_wtick()` : seconds between successive clock ticks

Parallelize the Poisson code using OpenMP

- Now you can apply what you learn to the `poisson` code.
- Remember that 90% of the time is spend in the dumpers. So make sure you dump only once at the end of the simulation to get a validation image.