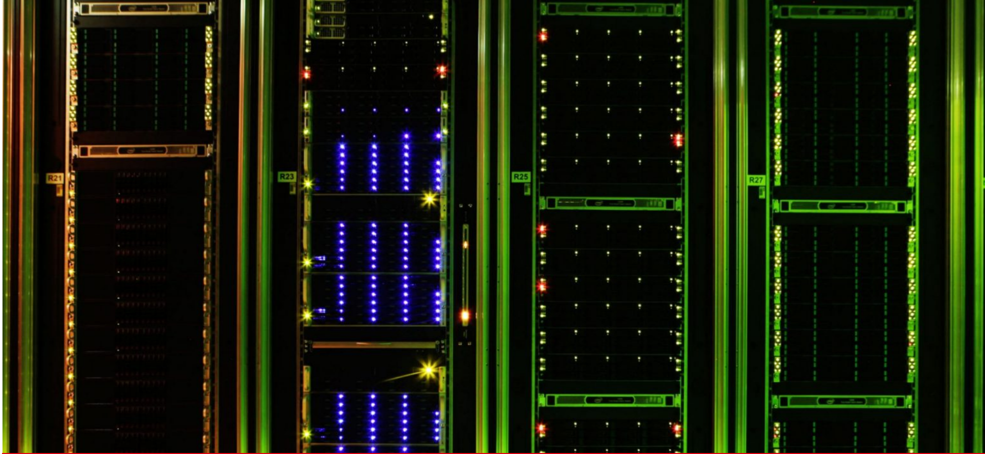


MATH-454 Parallel and High Performance Computing

Lecture 3: Thread Level Parallelism with OpenMP

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

March 13 2025

SCITAS

EPFL

OpenMP

SCITAS

Goals 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.

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
- November 2024: Combined C/C++ and Fortran version 6.0

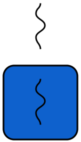
Terminology

Selected extract of the specification

- Specification:
  - Full specification
  - RefCard
- Terms:

thread an execution entity with a stack and a static memory (*threadprivate memory*)

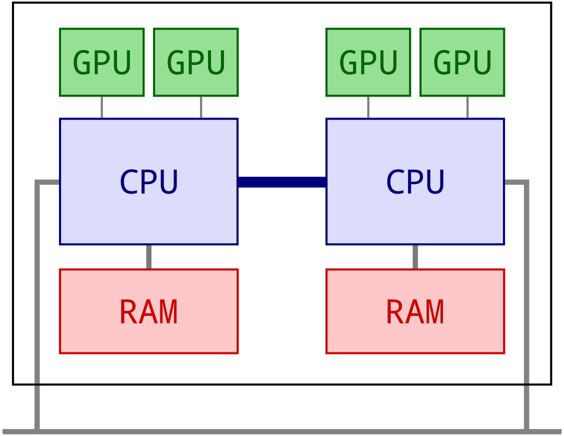
OpenMP thread a *thread* managed by the OpenMP runtime

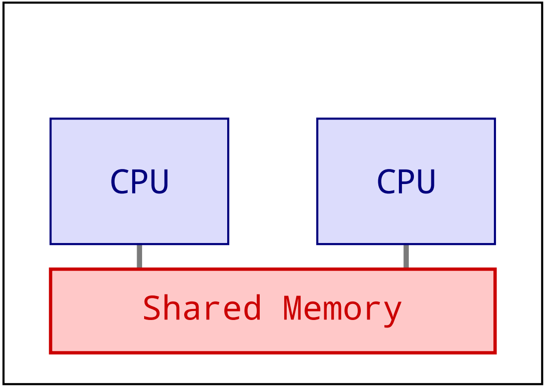


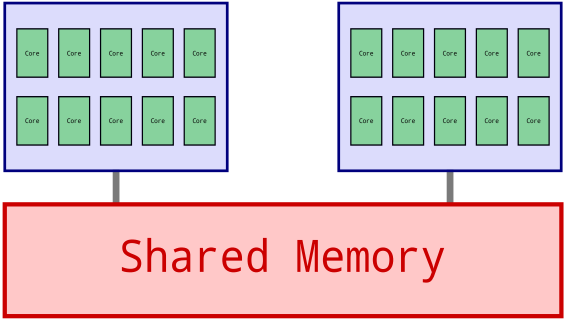
processor an hardware unit on which one or more *OpenMP threads* can execute

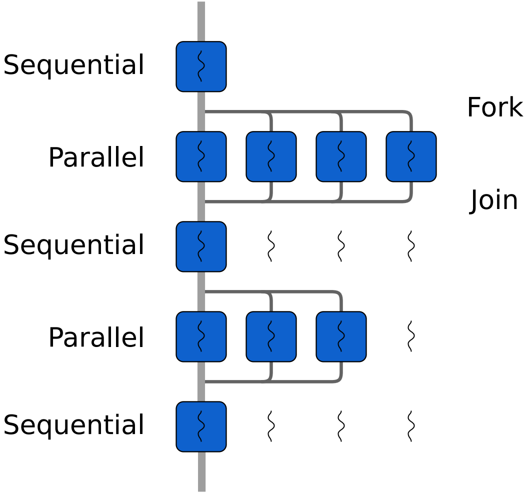
directive a base language mechanism to specify OpenMP program behavior

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.


GPU
GPU
GPU
GPU
CPU
CPU
RAM
RAM


CPU
CPU
Shared Memory


Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Shared Memory


Sequential
Parallel
Sequential
Parallel
Sequential
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

- 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 -fopenmp 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

```
$ 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

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

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

Number of concurrent threads

- Default value is implementation dependent (usually max hardware threads)
- At runtime in the code

```
omp_set_num_threads(nthreads);
```

- With an environment variable

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

Runtime routines

Subset 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

For calling these functions you need to `#include <omp.h>` !!

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***)**

Worksharing constructs

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

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

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

```
#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.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
```

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

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

Syntax

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

The master directive

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

Syntax

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

`master` directive was deprecated in OpenMP version 5. For modern compilers, use:

Syntax

```
#pragma omp masked filter(0)
  {
     structured-block
  }
```

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
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
```

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

Specifies an explicit barrier.

Syntax

```
#pragma omp barrier
```

I.e., all threads wait there until the last one reaches that directive.

openmp/barrier.c

```c
#pragma omp parallel
  {
    const int myrank = omp_get_thread_num();

    printf("[Thread %d] I print my first message.\n", myrank);

    #pragma omp barrier

    #pragma omp single
    {
      printf("The barrier is complete.\n");
    }

    printf("[Thread %d] I print my second message.\n", myrank);
  }
```

```
$ OMP_NUM_THREADS=3 ./openmp/barrier
[Thread 0] I print my first message.
[Thread 2] I print my first message.
[Thread 1] I print my first message.
The barrier is complete.
[Thread 0] I print my second message.
[Thread 1] I print my second message.
[Thread 2] I print my second message.
```

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**, **capture**
- *memory-order-clauses* **seq_cst**, **acq_rel**, **releases**, **acquire**, **relaxed**

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
- If not defined, the variables will be **shared**
- 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
- **default(private | firstprivate | shared | none)** can be specified
  default(none) means each variable should appear in a **shared** or **private** list

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)
```

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

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
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)
```

Syntax

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

- *reduction-identifier*: one of the operation +, −, *, &, |, ^, &&, ||
- *list* item on which the reduction applies

Loop clauses

reduction clause

Syntax

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

- *reduction-identifier*: one of the operation $+$, $-$, $*$, &, |, ^, &&, ||
- *list* item on which the reduction applies

openmp/reduction.c

```c
  global_sum = 0;

#pragma omp parallel for reduction(+:global_sum)
  for (i = 0; i < size_vec; i++) {
    global_sum += vec[i];
  }

  printf("sum = %i\n", global_sum);
```

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

Syntax

```
collapse(n)
```

Specifies how many loops are combined 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)

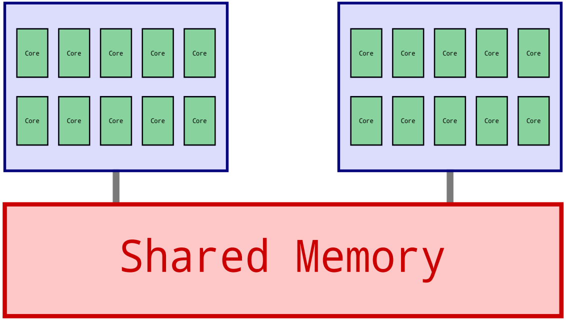
openmp/dgemm.cc

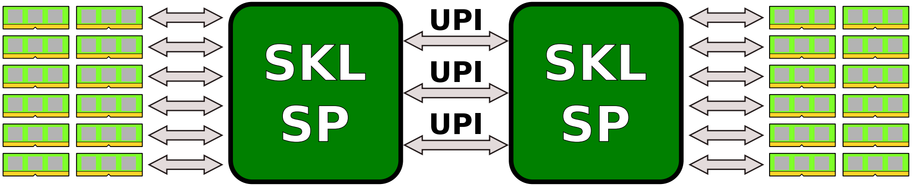
```
#pragma omp parallel for collapse(1) schedule(static, N / nthreads)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      for (int k = 0; k < N; ++k)
        C[i * N + j] += A[i * N + k] * B[k * N + j];
```

```
$ 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)
```

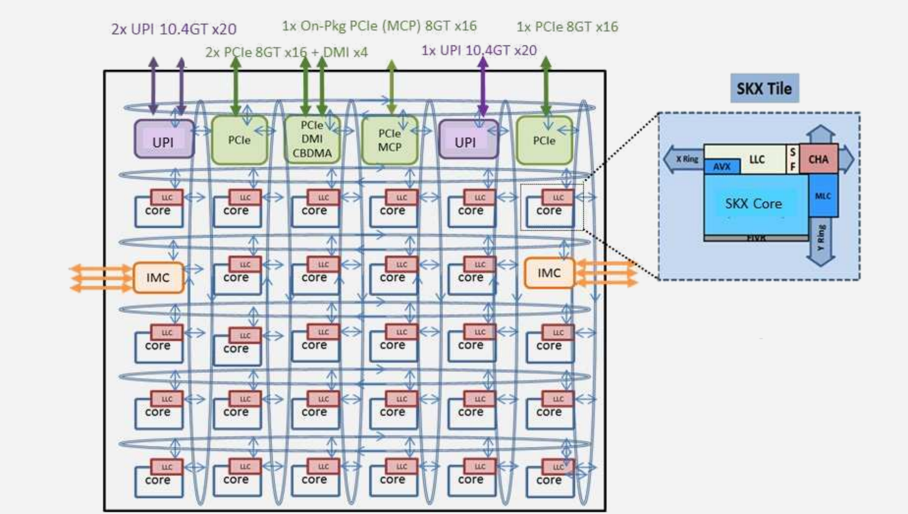
Example dgemm
collapse(2)

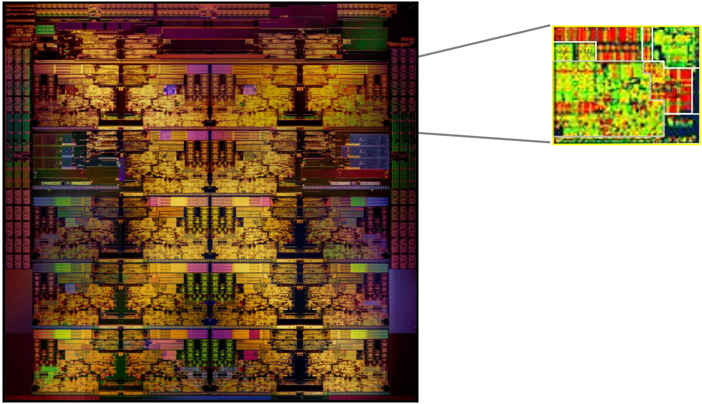
```
$ OMP_NUM_THREADS=1 ../build/openmp/dgemm
  DGEMM with 1 threads, collapse(2): 20.358 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=2 ../build/openmp/dgemm
  DGEMM with 2 threads, collapse(2): 40.0818 GFLOP/s (verif 2)
$ OMP_NUM_THREADS=4 ../build/openmp/dgemm
DGEMM with 4 threads, collapse(2): 72.4462 GFLOP/s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


Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Shared Memory


SKL
SP
UPI
UPI
UPI
SKL
SP

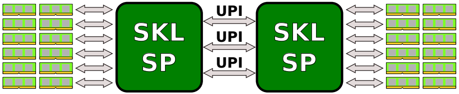
Idealized model vs reality (NUMA, Sub-NUMA Clusters, Cluster-on-Die)





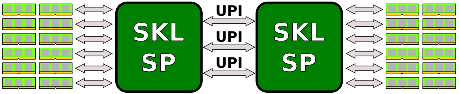
1.375 MB L3
Server AVX-512
SF
CHA
Server 1MB L2
& Control
Core

Implications of memory layout



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

- 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 {0-35}
OMP: pid 2115280 tid 2115285 thread 3 bound to OS proc set {0-35}
OMP: pid 2115280 tid 2115284 thread 2 bound to OS proc set {0-35}
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 {0-35}
Hello from thread 1 out of 4
Hello from thread 2 out of 4
```

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.

Thread Affinity

OMP_PROC_BIND

Possible values for `OMP_PROC_BIND`:

- **false** threads not bonded
- **true** threads are bonded (implementation dependant)
- **master** collocate threads with the master 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

openmp/firsttouch.cc

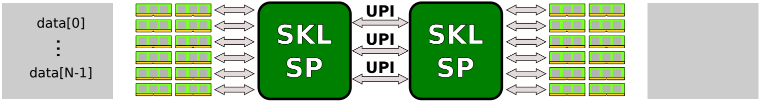
```
  std::vector<int> data(N);

  // Serial initialisation (page allocation by master thread)
  for (std::size_t i = 0; i < N; ++i)
  {
    data[i] = i % 10;
  }

#pragma omp parallel num_threads(2) for reduction(+ : sum)
  for (std::size_t i = 0; i < data.size(); ++i)
  {
    if (data[i] < 4)
      sum += data[i];
  }
```

First touch

Example

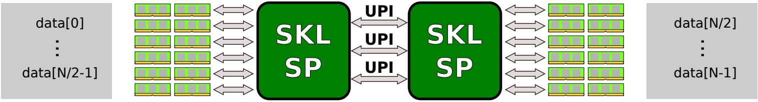


openmp/firsttouch.cc

```cpp
  std::vector<int> data(N);

  // Parallel initialisation (pages allocation)
#pragma omp parallel for num_threads(2)
  for (std::size_t i = 0; i < N; ++i)
  {
    data[i] = i % 10;
  }

#pragma omp parallel for num_threads(2) reduction(+ : sum)
  for (std::size_t i = 0; i < data.size(); ++i)
  {
    if (data[i] < 4)
      sum += data[i];
  }
```



data[0]
:
data[N/2-1]
SKL SP
UPI
UPI
UPI
SKL SP
data[N/2]
:
data[N-1]

First touch
Example

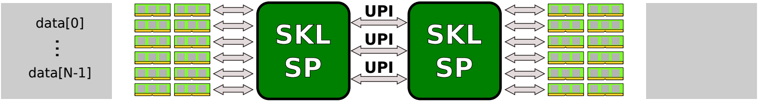
openmp/firsttouch.cc

```
  // Serial initialisation (by master thread) occurs here.
  std::vector<int> data(N, 0);

  // Pages are already allocated (too late)
#pragma omp parallel for num_threads(2)
  for (std::size_t i = 0; i < N; ++i)
  {
    data[i] = i % 10;
  }

#pragma omp parallel for num_threads(2) reduction(+ : sum)
  for (std::size_t i = 0; i < data.size(); ++i)
  {
    if (data[i] < 4)
      sum += data[i];
  }
```

First touch
Example



EPFL

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

Data race, false-sharing