University of Leeds

Lecture 2: Introduction to shared memory pa

Previous lectures

In the last introductory lecture we saw:

- s.
-
- Some general concepts:
  - **Concurrency** (more general than pa
  - **Shared** *versus* **distributed mem**
  - Potential performance issues related t
  - **Flynn's taxonomy**.

This lecture

This l ,
relev

- Processes *versus* threads and the threa
- Languages and frameworks suitable for t
- How to set up and run OpenMP.

Multi-core CPUs

A **core** is a single **processing unit** that executes instructions.

- Has components that *fetch*, *decode etc.* instructions.
-
- https://eduassistpro.github.i

As its name suggests, **multi-core** proc
one such unit.

- **MIMD** in Flynn's taxonomy (*sing*
- Most common now are **dual core**, **quad core** and **octa core**.
- High-performance chips can have many more, *e.g.* SW26010 (used in China's Sunway TaihuLight supercomputer) has 260.

Simultaneous multithreading

Some chips employ **simultaneous multithreading**[1]:

- Two (or more) threads run on the same core.
- If one thread stops execution (*e.g.* to wait for memory

Appe
requi

- Performance improvements only 15%-

When interrogating a framework for the max available threads, you may get **more** than the number of cores.

[1] Known as **hyperthreading** on Intel chips.
[2] Rauber and Rünger, *Parallel Programming* 2nd ed. (Springer, 2013).

The processor-memory gap

Memory access rates are increasing far slower than processor performance (taking in to account number of cores):

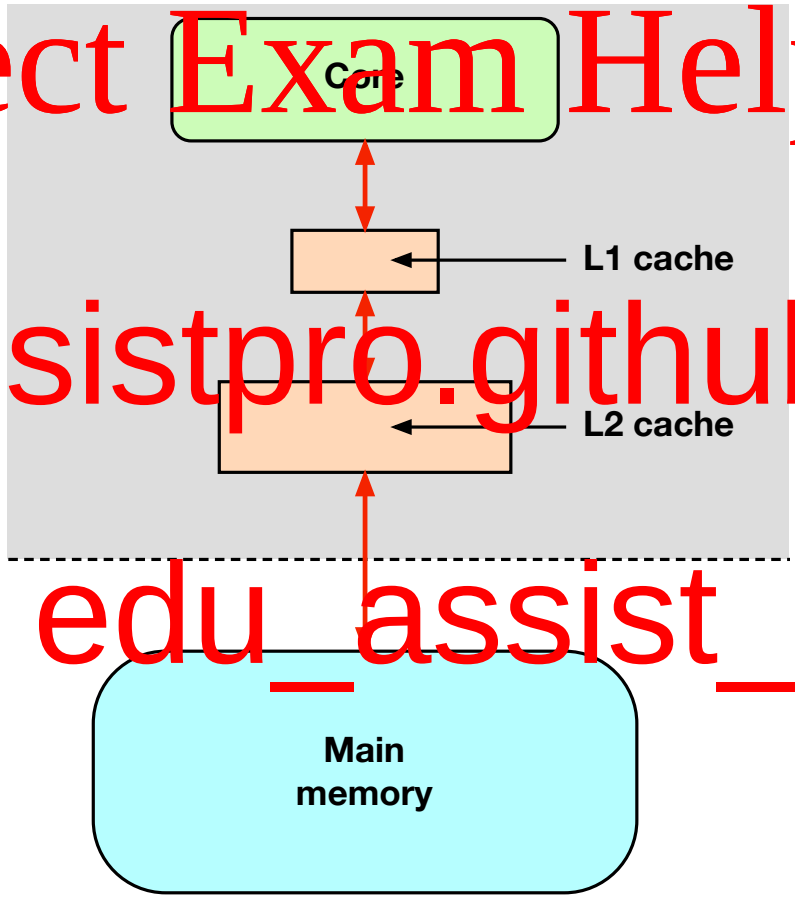
- This is the **processor-memory gap**.

Year

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kauffman, 2006).

Single-core memory caches: A reminder

- **Small, fast on-chip** memory.
- A
  r
  (
- Subsequent accesses return the cache data (a **cache hit** - *fast*), or from main memory (a **cache miss** - *slow*).
- Multiple caches **levels** (*e.g.* L1, L2, L3) arranged **hierarchically**.



Schematic two-level cache

Multi-core memory caches

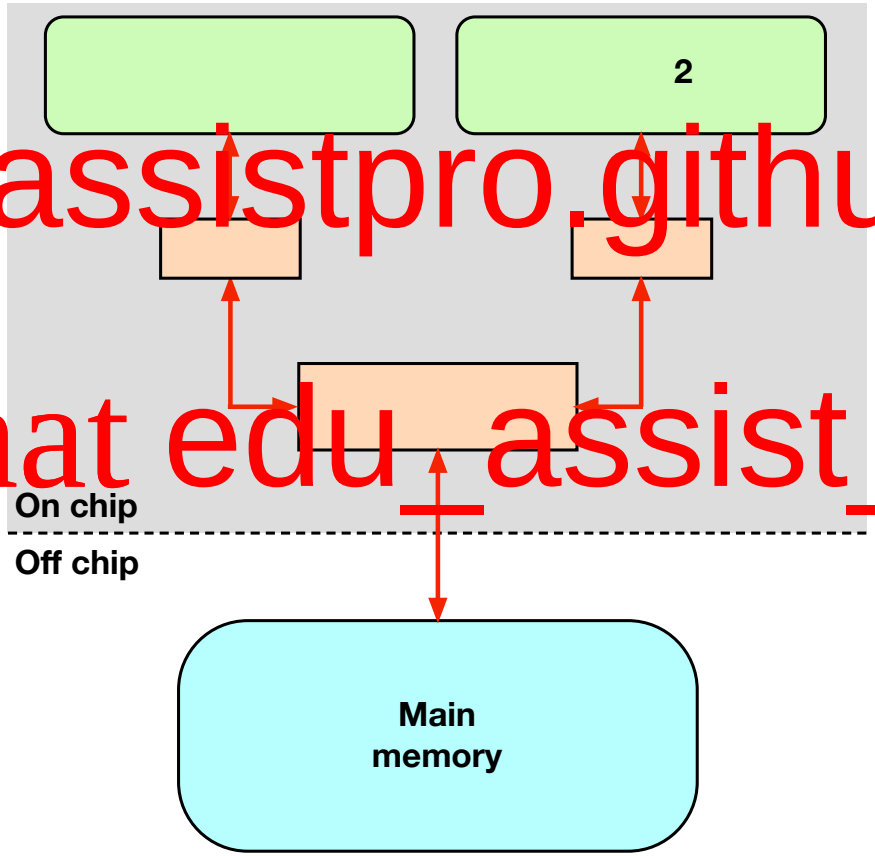
Different manufacturers choose different ways to incorporate caches into multi-core designs.

Ofte **hierarchy**

- https://eduassistpro.github.i
- Share higher level caches.

For *e.g.* quad cores:

- L1 for each core.
- L2 for pairs.
- L3 for all cores.



Cache coherency

1. Core 1 reads an address x, resulting in a line in its L1.
2. Core 2 does the same, resulting in a line in its L1.
3.
4. https://eduassistpro.github.i

Maintaining consistent memory views for all caches is called **cache coherence**.

A common way to maintain **cache coherency** is **snooping**:

- The *cache controller* **detects writes** to caches, and updates higher-level caches.

False sharing

- Maintaining cache coherency incurs a performance loss
  - If two cores repeatedly write to the same memory location,

How y
locations **on the same cache line**, upd

- *i.e.* hardware **performance loss**

This unnecessary cache coherency is known as **false sharing**

Potential benefit of cache sharing

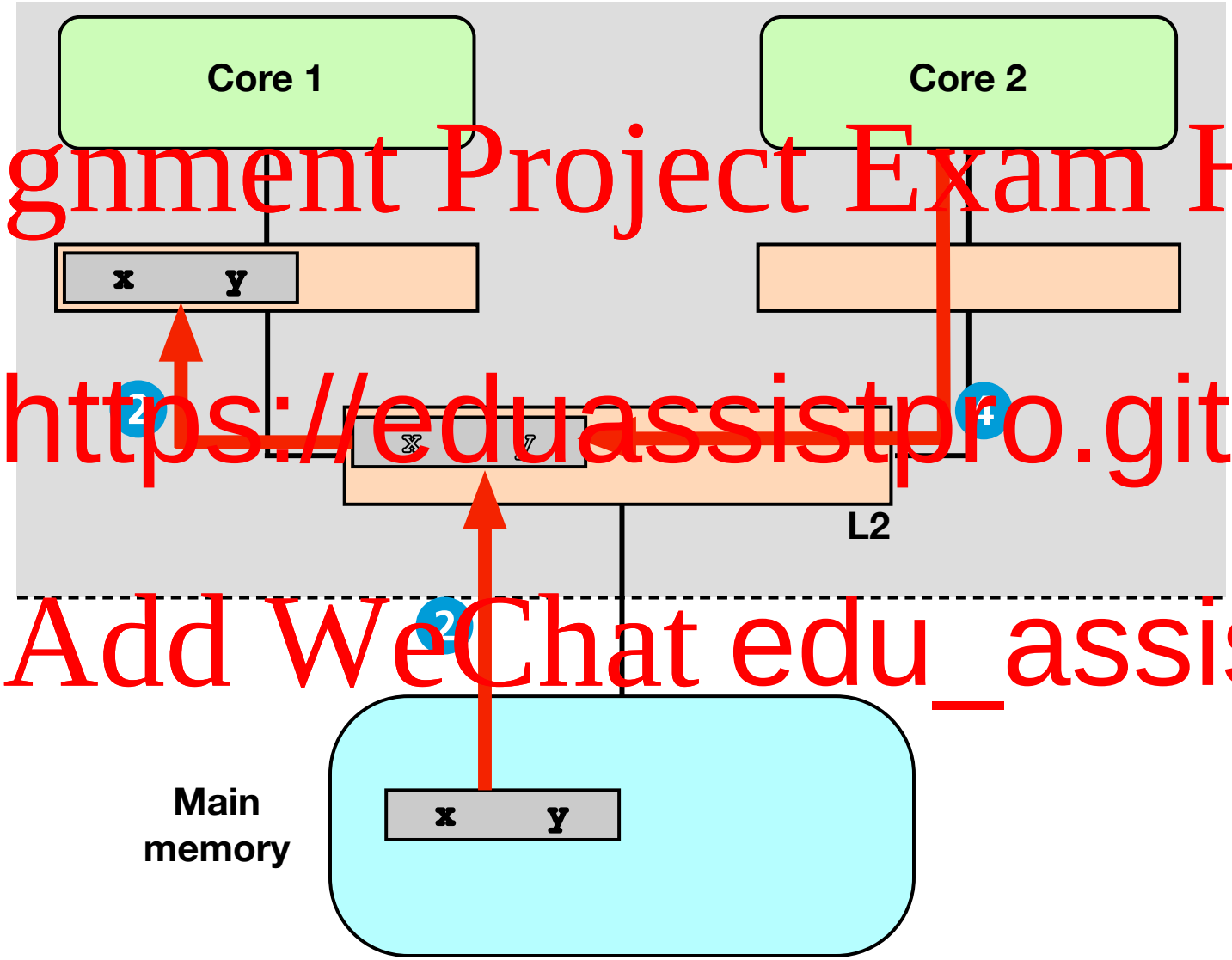
It is also possible for multiple cores to benefit from shared caches:

1. Core 1 reads an address `x` from main memory.
2.
3.
4. `x`.

eed to access main memory.

It is therefore possible for fewer accesses to main me compared to the equivalent serial code.

This can result in parallel speed up **more than the number of cores** (known as superlinear speedup; cf. Lecture 4).



Processes *versus* threads

Control flows can be either **processes** or **threads**:

Processes:

-
- s).
- **Expensive** to generate (large heap me

Threads:

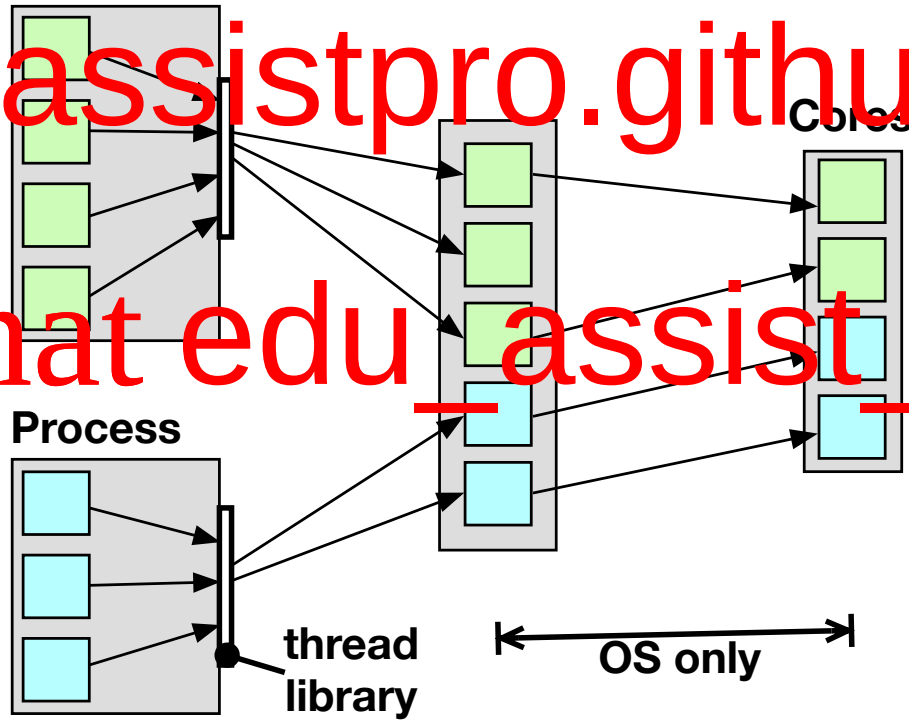
- Threads of one process **share** its address space.
- Implicit communication *via* this **shared memory**.
- **Cheap** to generate (no heap memory).

Kernel *versus* user-level threads

- The threads that execute on the core(s) are **kernel threads**.
  - Only the OS has direct control over kernel threads.

Progr
genera
threads.

- Managed by a **thread library**.
- Mapped to kernel threads by the OS **scheduler**.



Thread programming

The choice of programming framework/API/library/etc. **must** be suitable for the architecture on which it will run.

- _ __ _ lism.

It is pos

- Java supported threads early on, through t and `Runnable` interface.
- The C library `pthread` implement

- C++11 has language-level concurrency support.
- Python has a threading library, although need to work around its **global interpreter lock** to exploit multi-cores.

Higher-level threading support

Higher-level options that do not require explicit thread control also exist to reduce development times.

For C /. in *Structured Parallel Programming* (Mo

- **Cilk Plus**
- **TBB** (Threading Building Blocks
- **ArBB** (Array Building Blocks).
- **OpenCL**, although primarily used for GPUs.

The first three are not (yet?) widely implemented in compilers.

OpenMP

For the SMP component of this module we will use **OpenMP**.

-
- Currently up to v5.2, although compilers may only support earlier versions

More information available from `http://www.openmp.org`

Compiling C with OpenMP

For this module we will use gcc (GNU Compiler Collection)[1].

- Compile with `-fopenmp`
- Must include `omp.h`

All pa

- https://eduassistpro.github.i
- You will each get your own INDIVIDUAL ac
- Full instructions for logging into your acco
- A Linux OS is provided along with the al libraries this module:
  - You can run jobs interactively on 2 cores while debugging (can still have more threads!)
  - You can run batch jobs on up to 16 cores via `Slurm`

[1] Easy to install on Macs with `homebrew`.

helloWorld.c

Code on Minerva: helloWorld.c


```
1  #include <stdio.h>
2  #include <omp.h>  // Run-time OpenMP library routines.
3
4  int main
5  {
6    // Th  s
7    #pr
8    {
9      // Get this thread number, and the maximum.
10     int threadNum  = omp_get_thread_num ();
11     int maxThreads = omp_get_max_threads();
12
13     // Simple message to stdout.
14     printf( "Hello from thread %i of %i!\n", threadNum,
       maxThreads );
15   }
16   return 0;
17 }
```

Compiling C: Reminder

If the source code is called `helloWorld.c`:

```
gcc -f
```


Opti

- `-fopenmp` tells compiler to expect Open
- `-Wall` turns on all warnings; recommer
- `-o helloWorld` is the executable nam .
- `helloWorld.c` is the source code.
- Sometimes need *e.g.* `-lm` for the maths library.

#pragma omp parallel

#pragma directives provide information beyond the language

- All OpenMP pragmas start: `#pragma omp ...`

Here the ne
from { to }) **in parallel**.

- The code inside this scope is run by
- Outside of this scope there is only

This is why the `printf` statement is repeated multiple times, **even though it only appears once in code**.

We will look at this in more detail next time.

#include <omp.h>

Include omp.h to use OpenMP runtime library routines:

int om _ _ _

- res.
- May exceed apparent core number with **multithreading** (see earlier).

`int omp_get_thread_num()`:

- Returns the thread number **within the current scope**.
- `0 <= omp_get_thread_num() < omp_max_thread_num()`

Setting the number of threads

If you don't want to use the default number of threads:

```
void o  _   _   _
```


Alternatively, use shell **environment va**

- For bash: `export OMP_NUM_THRE`
- Avoids the need to recompile.
- List all environment variables using `env`.
- To see all OMP variables: `env | grep OMP`

Summary and next lecture

Today we have started looking at **shared memory parallelism** (SMP):

- Various languages, frameworks *etc.* support SMP.
- OpenMP is commonly supported by C/C

Next time we will look in more detail at what is actually going on at the thread level, for a more interesting example.