Minho Advanced Computing Center

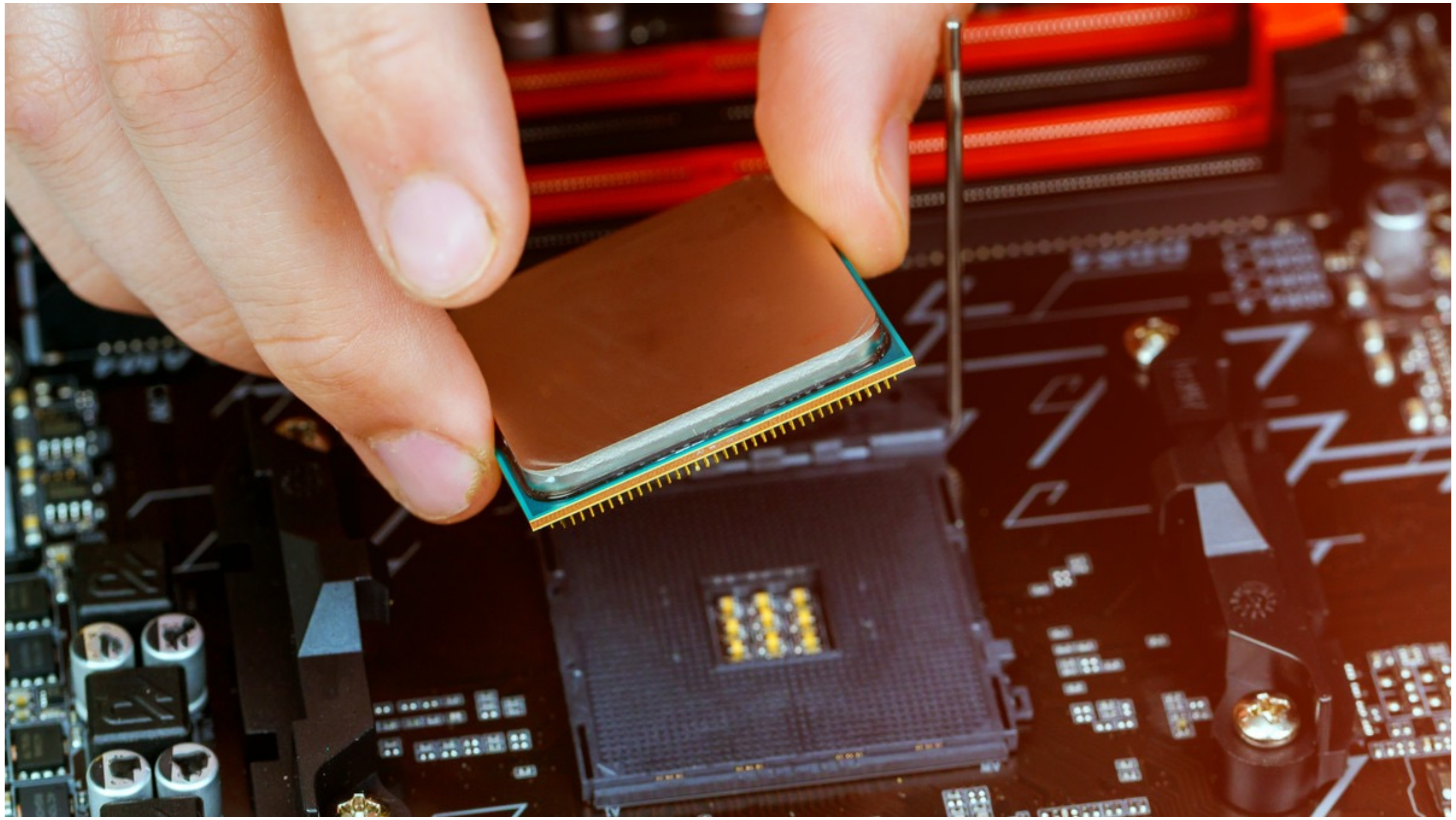
OpenMP

Programming Fundamentals

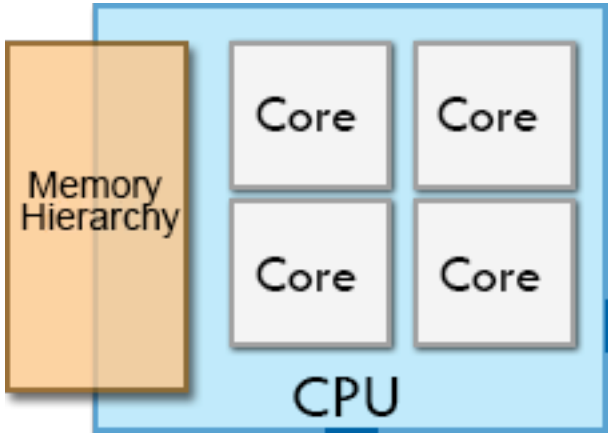
André Pereira
Minho Advanced Computing Center
ampereira@macc.fccn.pt

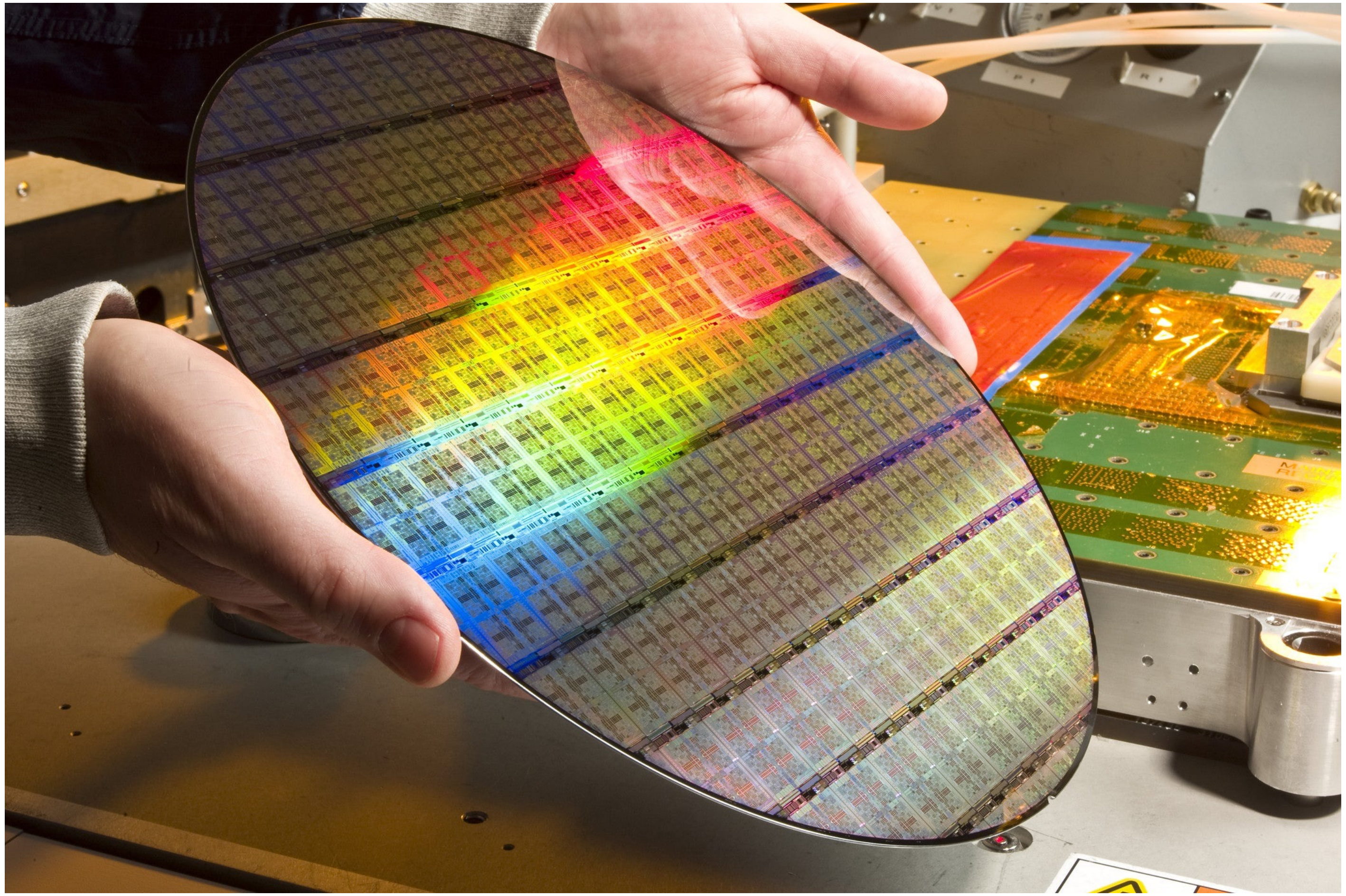
The Basic Architecture of a CPU

The Basic Architecture of a CPU

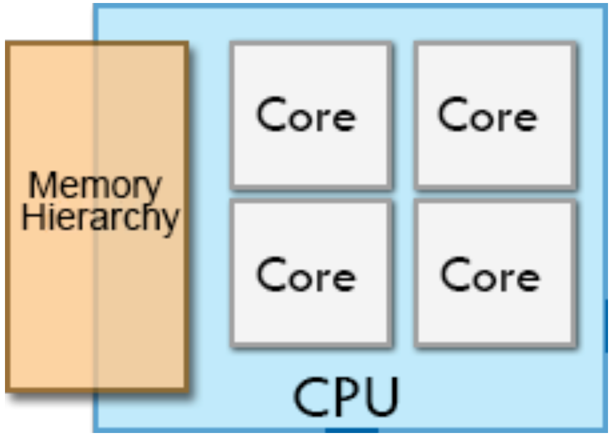


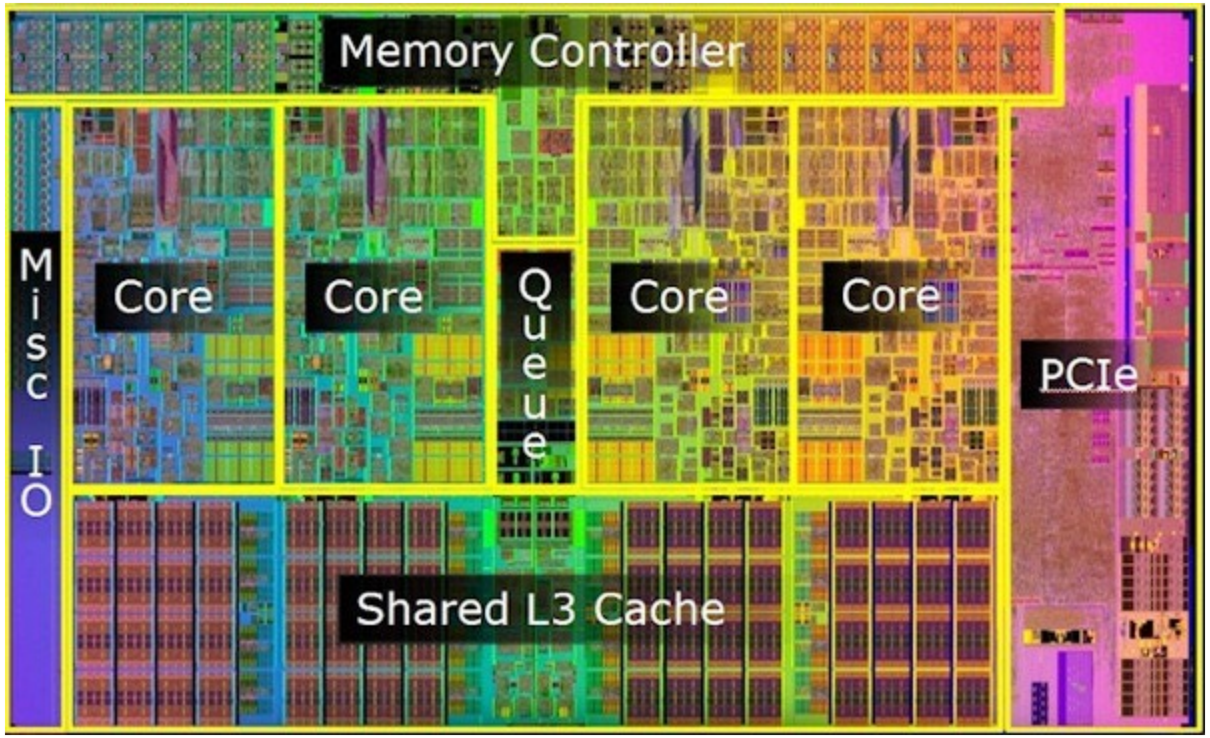
The Basic Architecture of a CPU



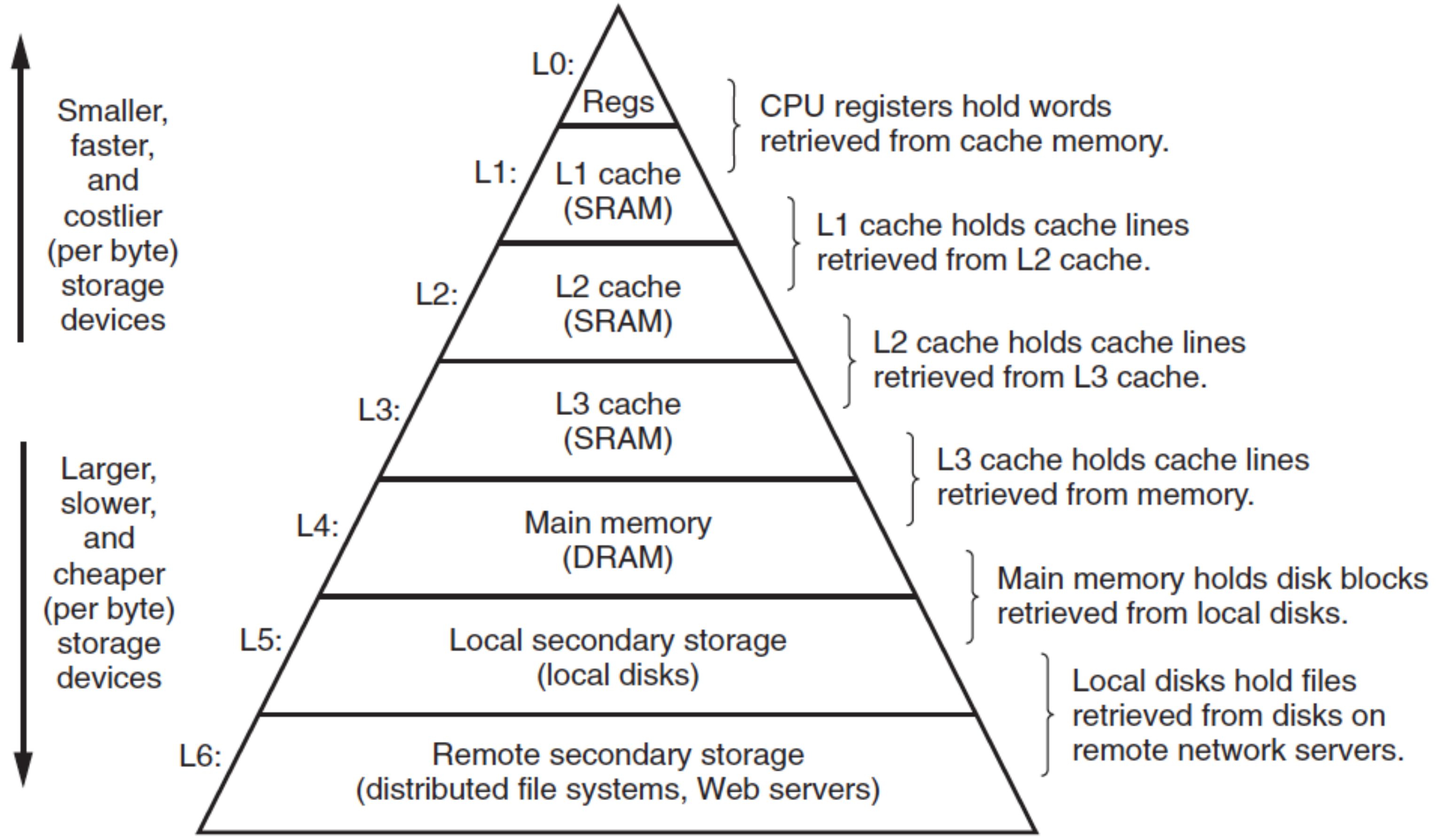


The Basic Architecture of a CPU





The Memory Hierarchy



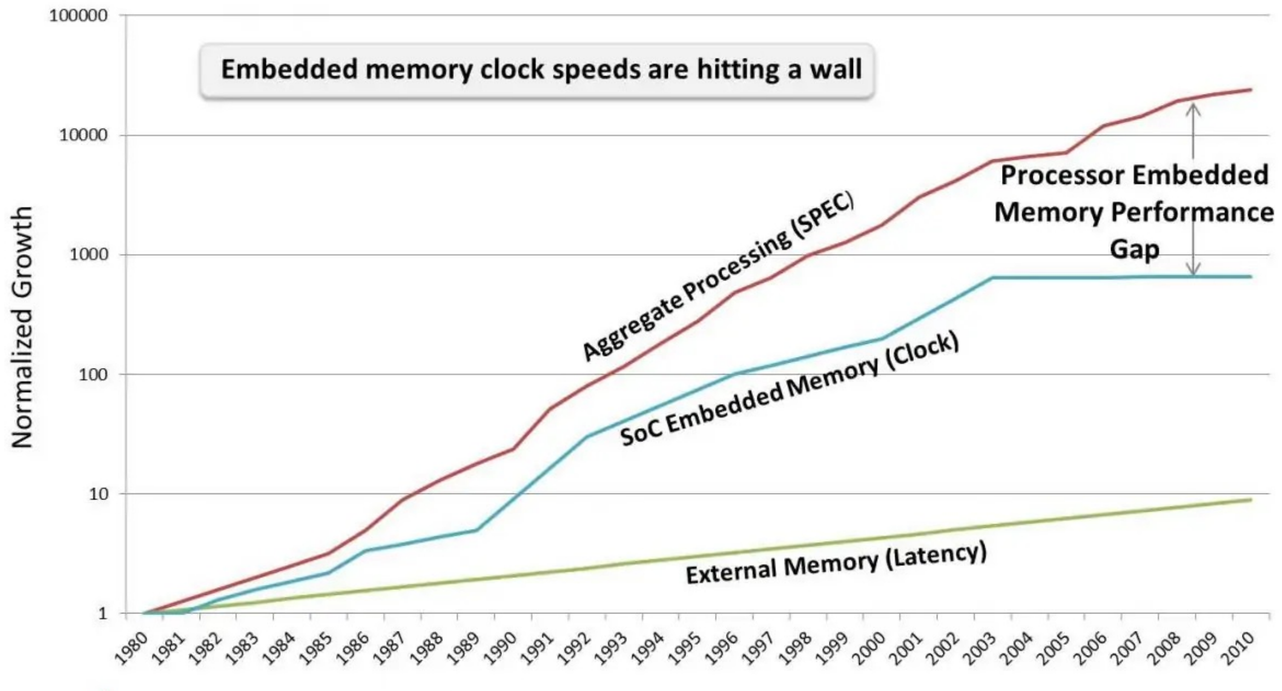
The Memory–CPU Performance Gap

Data locality is crucial

- The closer the data is to the chip the less time is wasted

Key takeaways

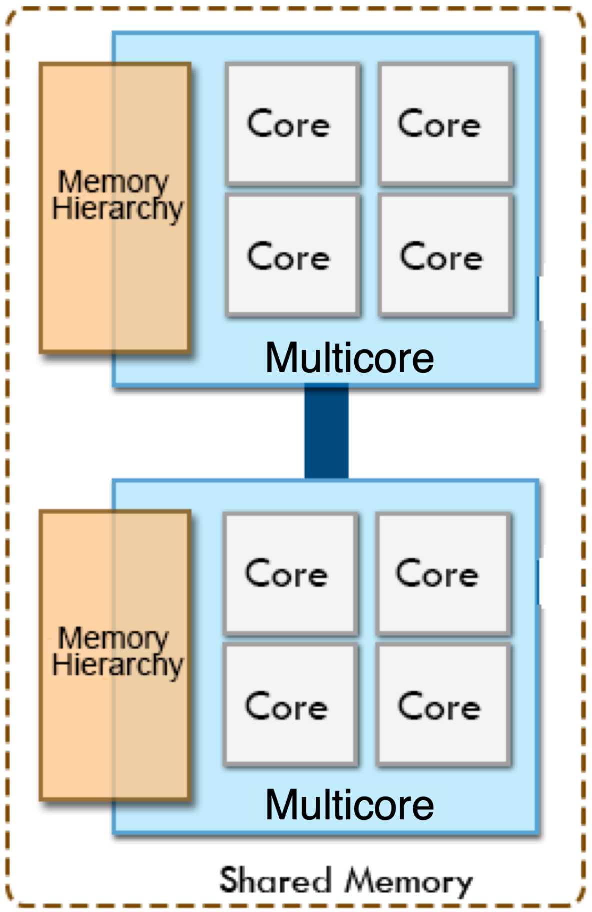
- Contiguous accesses to aligned data minimizes time losses (spatial locality)
  - Ex: traversing an array
- Reuse of data keeps it in the faster memory storage (temporal locality)



*Source: Hennessy and Patterson, 5th Edition

A Shared Memory Server

- One or multiple multicore CPU chips
- Fast CPU interconnection
- Memory address space shared among CPUs
  - Unified address space
  - Memory storage physically separated
  - No explicit access to specific storage
  - Could add performance penalties



A Brief Introduction to C

Why C (or C++)?

- It's a compiled language (faster)
- Closer to the OS level – more control over its behavior
- Performance oriented Python libraries are written in C
- Wider availability of HPC libraries and frameworks

```python
a = [1,2,3]
b = [2,3,4]

map(sum, zip(a,b))
```

or

```python
import numpy
a=numpy.array([0,1,2])
b=numpy.array([3,4,5])
a+b
```

```c
int a[3] = {0, 1, 2};
int b[3] = {3, 4, 5};
int c[3];

for (int i = 0; i < 3; i++)
    c[i] = a[i] + b[i];
```

python

A Brief Introduction to C

Why C (or C++)?

- It's a compiled language (faster)
- Closer to the OS level– more control over its behavior
- Performance oriented Python libraries are written in C
- Wider availability of HPC libraries and frameworks

```python
def vector_add (a, b):
    c = map (sum, zip(a,b))

    return c
```

python

```c
int* vector_add (int a[], int b[], int size) {
    int* c = (int*) malloc (size * sizeof (int));

    for (int i = 0; i < size; i++)
        c[i] = a[i] + b[i];

    return c;
}
```

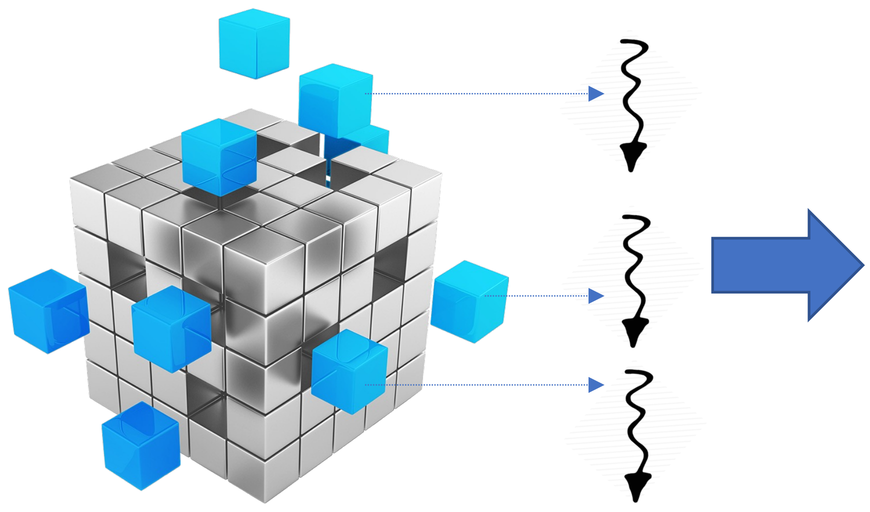
But there are downsides…

- C is more verbose
- Hard typing of variables (is it really a downside?)
- Explicit memory allocation of dynamic data structures
- Fewer QoL improvements as standard
  - C++ helps addressing this issue

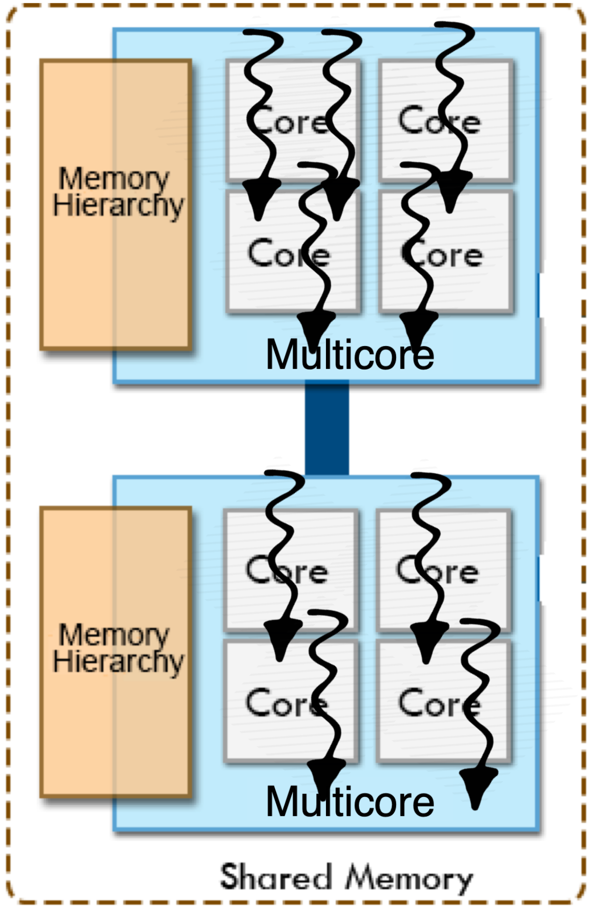
Going Parallel - Threads

Threads
- Entities at the **software** or hardware level
- Execute a section of an application
- Multiple threads can execute concurrently
- Share the same memory address space
  - as opposed to processes

Work sharing
- Divide the workload among threads
  - Each thread processes a subset of the overall workload
- Threads are scheduled to execute in specific CPU cores
  - Mostly handled by the OS



DISTRIBUTE WORK AMONG THREADS



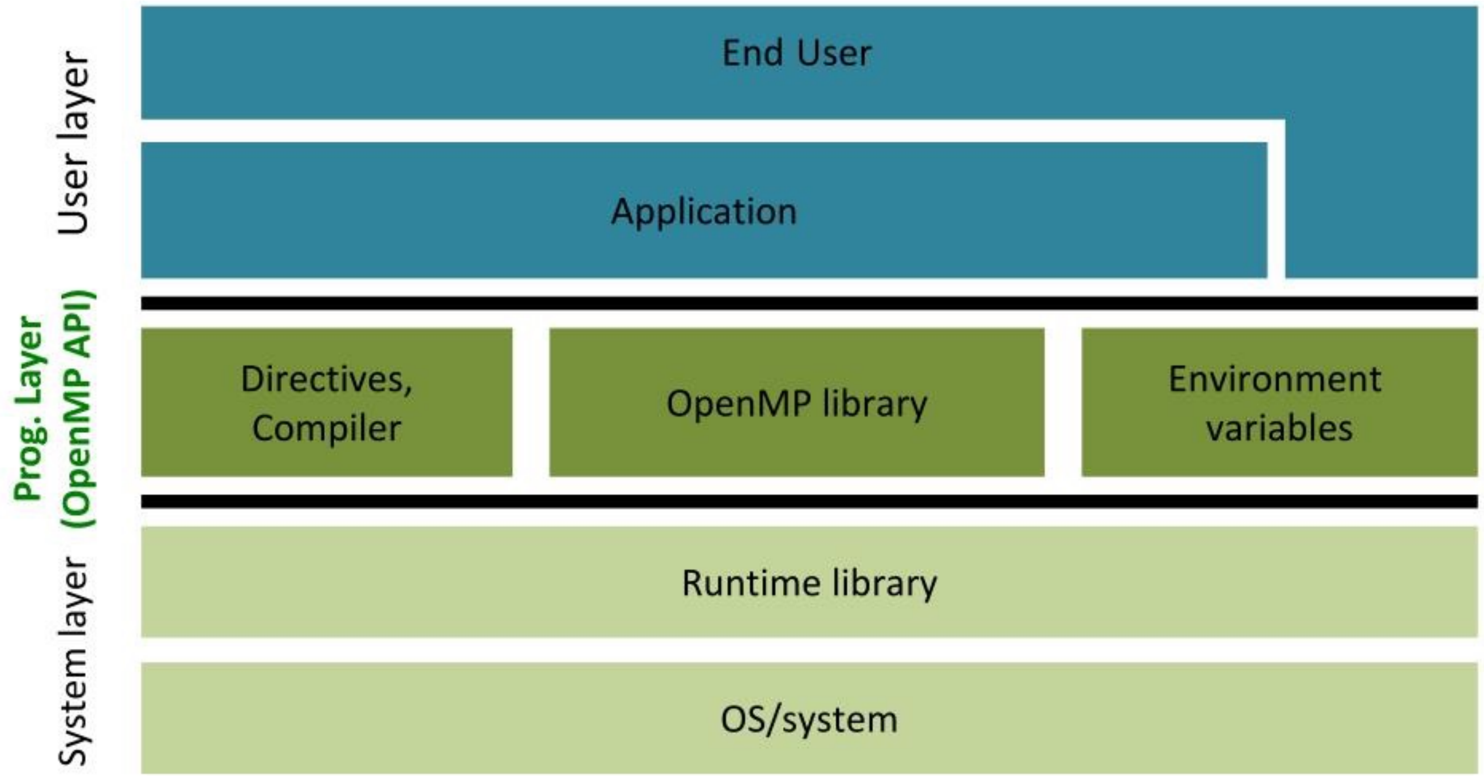
MAP THREADS TO PHYSICAL CORES

OpenMP

Several alternatives for multithread programming

- Posix threads – low level
    - Close to the OS-level
    - Require a lot of micromanagement
    - Limited out-of-the-box functionality
- Frameworks (CILK, Threading Building Blocks, SYCL, …) – high level
    - Feature rich
    - Integrated management and scheduling of complex workloads
    - Application must be designed according to the framework's requirements
- OpenMP – somewhere in the middle
    - Platform independent
    - Often requires minimal modifications to existing sequential code
    - Pragma-based
    - Available for C, C++, and Fortran

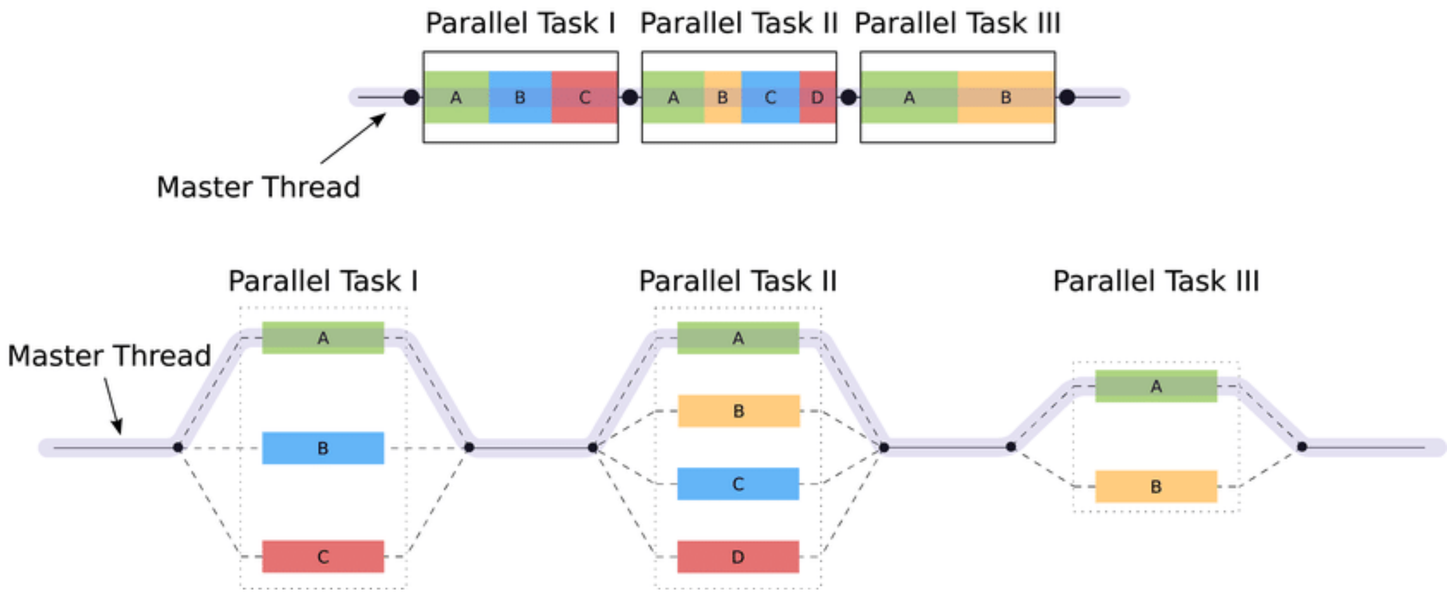
The OpenMP Software Stack



The Fork-Join Model

Interleaving of sequential and parallel sections of the code

- Application begins and ends execution sequentially
- Threads are created and work is distributed at the fork
- Implicit synchronization at the join
- % of code that can be parallelize limits potential improvements
    - See Amdahl's law



OpenMP – Going Parallel

Parallelism with OpenMP is implemented using pragma statements

- Often require minimal modifications to existing sequential code
- Compiler creates parallel machine code based on the pragmas
- Pragmas can be ignored by the compiler to create sequential code
- Pragmas are affected to the section of code next to them

```
#pragma omp parallel
{
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

```
for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
```

```
for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
```

`#pragma omp parallel`

- Creates a parallel section of code
- The code is replicated among the threads created

OpenMP – Going Parallel

OpenMP provides a library of useful functions

- May help control the execution flow of parallel regions
- Helpful to share the workload among threads

```
omp_set_num_threads (4);
#pragma omp parallel
{
    int thread_id = omp_get_thread_num ();
    int n_threads = omp_get_num_threads ();
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

`void omp_set_num_threads (int x)`

- Sets the amount of threads to be created in the next parallel code section

`int omp_get_num_threads (void)`

- Returns the amount of threads of the current parallel code section

`int omp_get_thread_num (void)`

- Returns the identifier of the "current" thread being executed

Hello World - A Practical Example

```c
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {

    printf ("Hello World\n");

    return 0;
}
```

COMPILE

```
❯ make
```

```
[ampereira@c805-001 hello_world]$ make
gcc -c -Wall -Wextra -pedantic -O2 -Wno-unused-parameter  src/hello_world_parallel.c -o
 build/hello_world_parallel.o
gcc -Wall -Wextra -pedantic -O2 -Wno-unused-parameter  -o bin/hello_world build/hello_w
orld_parallel.o
```

Hello World - A Practical Example

```c
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {

    printf ("Hello World\n");

    return 0;
}
```

COMPILE

`❯ make`

Hello World - A Practical Example

```c
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {

    printf ("Hello World\n");

    return 0;
}
```


```
[ampereira@c805-001 hello_world]$ sbatch run.sh
Submitted batch job 65062
[ampereira@c805-001 hello_world]$ cat hello_world.o
Hello World
```

Hello World – Going Parallel

```c
#include <stdlib.h>
#include <stdio.h>

#include <omp.h>

int main (int argc, char* argv[]) {

    omp_set_num_threads (4);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num ();
        int n_threads = omp_get_num_threads ();

        printf ("Hello World from thread %d of %d threads\n",
                thread_id, n_threads);
    }

    return 0;
}
```

COMPILE

```
❯ make
```

```
[ampereira@c805-001 hello_world]$ make
gcc -c -Wall -Wextra -pedantic -O2 -Wno-unused-parameter  src/hello_world_parallel.c -o
 build/hello_world_parallel.o -fopenmp
gcc -Wall -Wextra -pedantic -O2 -Wno-unused-parameter  -o bin/hello_world build/hello_w
orld_parallel.o -fopenmp
```

To use the OpenMP library

- Include the OpenMP header - **`#include <omp.h>`**
- Add the **`-fopenmp`** option to the compiler
  - OpenMP code will be ignored otherwise, and the application will not be parallelized

Hello World – Going Parallel

Minho Advanced Computing Center

```c
#include <stdlib.h>
#include <stdio.h>

#include <omp.h>

int main (int argc, char* argv[]) {

    omp_set_num_threads (4);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num ();
        int n_threads = omp_get_num_threads ();

        printf ("Hello World from thread %d of %d threads\n",
                thread_id, n_threads);
    }

    return 0;
}
```

COMPILE

```
❯ make
```

EXECUTE

```
[ampereira@c805-001 hello_world]$ cat hello_world.o
Hello World from thread 2 of 4 threads
Hello World from thread 0 of 4 threads
Hello World from thread 3 of 4 threads
Hello World from thread 1 of 4 threads
```

OpenMP – Loop Parallelism

Most parallelism potential in scientific and industry code is in loops

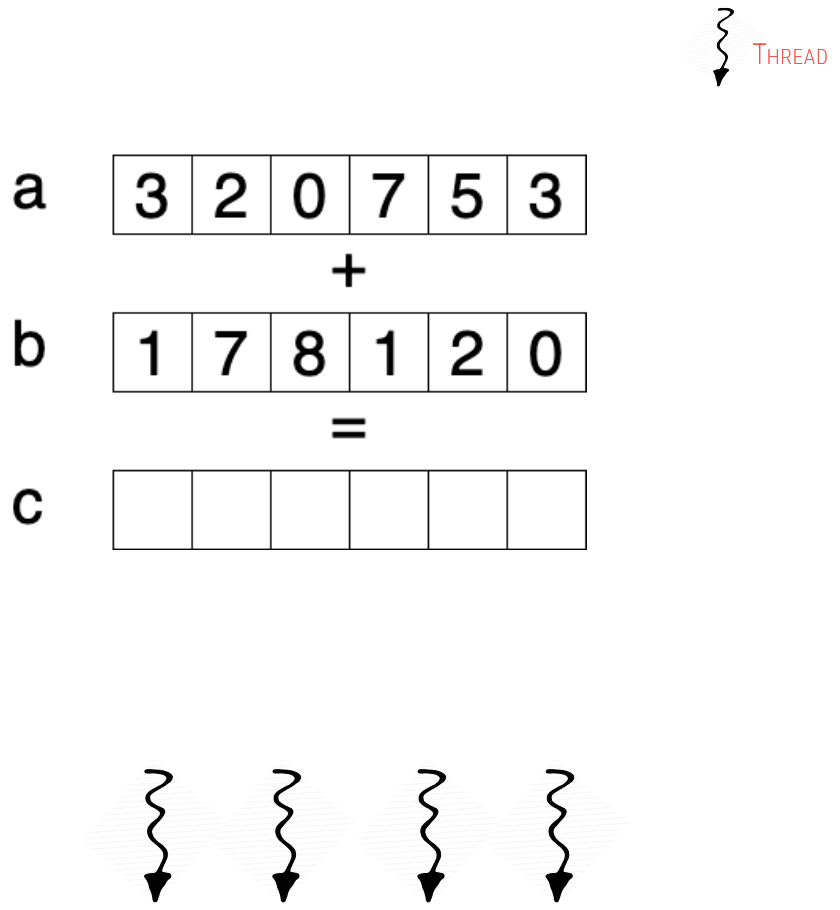
- Iteration through vectors and other list-like structures
- Vector-vector, vector-matrix, and matrix-matrix operations
- Operations on grids and meshes

```
int* vector_add (int a[], int b[], int size) {
    int* c = (int*) malloc (size * sizeof (int));

    for (int i = 0; i < size; i++)
        c[i] = a[i] + b[i];

    return c;
}
```

OpenMP – Loop Parallelism

How is the workload shared among threads?

- **`omp_get_thread_num`** is often useful
- Possible distribution strategies
  - Single element round-robin
  - Chunk division
  - …



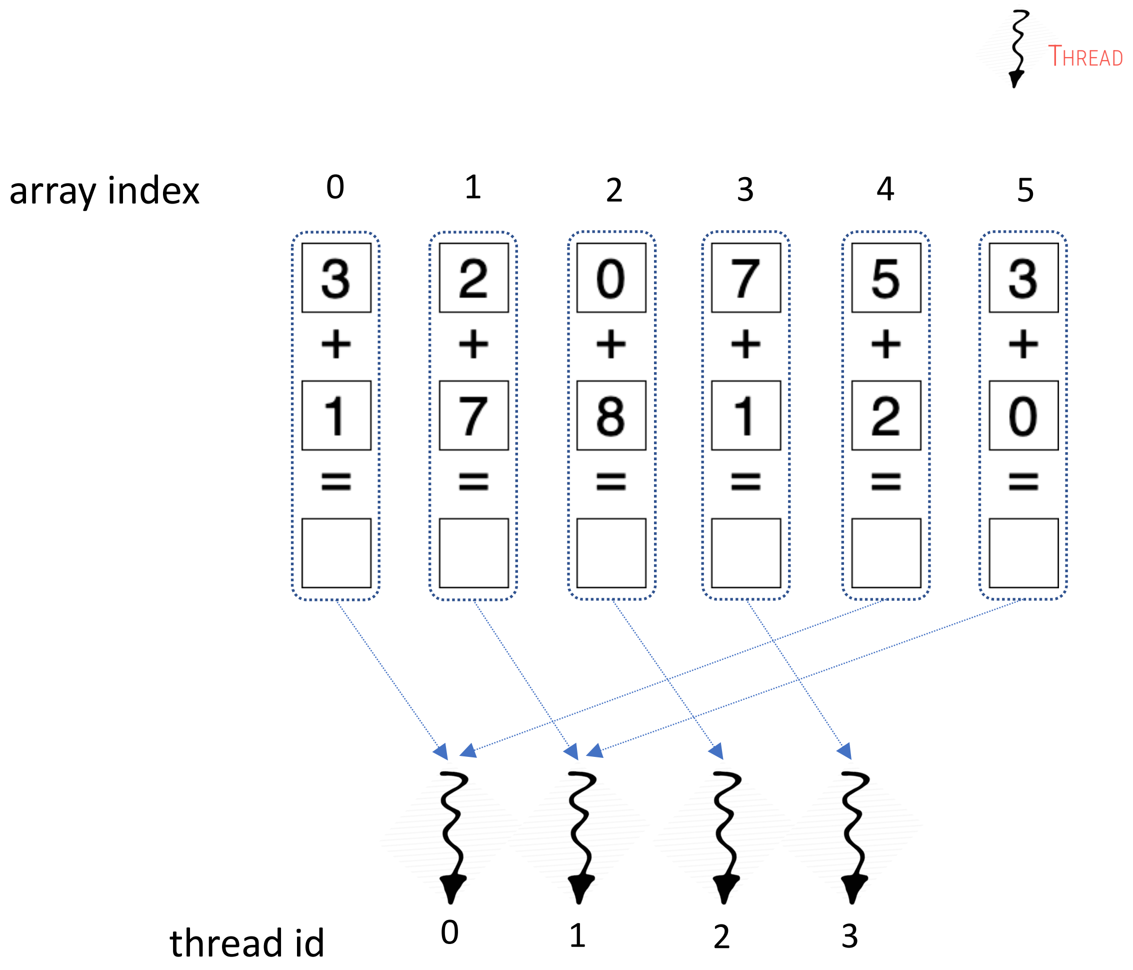
OpenMP – Loop Parallelism

How is the workload shared among threads?

- Omp_get_thread_num is often useful
- Possible distribution strategies

`#pragma omp for` to the rescue

- Automatically distributes the for loop workload among threads
- It's behavior can be tuned by appending
  - **`nowait`**
  - **`schedule(type)`**
  - **`collapse(n)`**
  - …



OpenMP – Loop Parallelism

How is the workload shared among threads?

- Omp_get_thread_num is often useful
- Possible distribution strategies

```
#pragma omp parallel
{
    #pragma omp for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

`#pragma omp for` to the rescue

- Automatically distributes the for loop workload among threads
- It's behavior can be tuned by appending
    - **`nowait`**
    - **`schedule(type)`**
    - **`collapse(n)`**
    - …

OpenMP – Shared and Private Data

Variables declared **outside** of parallel code sections are shared among threads

- Threads can concurrently read or write on the same variable
- These variables can be privatized to each thread through pragmas
    - `private(var_name)`
    - `firstprivate(var_name)`
    - `lastprivate(var_name)`

Variables declared **inside** of parallel code sections are private

ASSUMED PRIVATE

```c
#include <stdlib.h>
#include <stdio.h>

#include <omp.h>

int main (int argc, char* argv[]) {

    omp_set_num_threads (4);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num ();
        int n_threads = omp_get_num_threads ();

        printf ("Hello World from thread %d of %d threads\n",
                thread_id, n_threads);
    }

    return 0;
}
```

OpenMP – Shared and Private Data

Variables declared **outside** of parallel code sections are shared among threads

- Threads can concurrently read or write on the same variable
- These variables can be privatized to each thread through pragmas
  - **`private(var_name)`**
  - **`firstprivate(var_name)`**
  - **`lastprivate(var_name)`**

Variables declared **inside** of parallel code sections are private

SHARED

PRIVATIZED

```c
#include <stdlib.h>
#include <stdio.h>

#include <omp.h>

int main (int argc, char* argv[]) {
    int thread_id, n_threads;
    omp_set_num_threads (4);

    #pragma omp parallel private(thread_id,n_threads)
    {
        thread_id = omp_get_thread_num ();
        n_threads = omp_get_num_threads ();

        printf ("Hello World from thread %d of %d threads\n",
                thread_id, n_threads);
    }

    return 0;
}
```

Minho Advanced Computing Center

Lab Session

Hello World!

```c
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {

    printf ("Hello World\n");

    return 0;
}
```

Copy the exercises to your *scratch*

- `cp -r $SCRATCH/../shared/tr0012022/labs/openmp $SCRATCH`

Get familiar with OpenMP

- Parallelize the *Hello World* example
- Execute the code and verify if the outputs are expected
- Vary the number of threads and see the impact on the outputs

```c
#include <stdlib.h>
#include <stdio.h>

#include <omp.h>

int main (int argc, char* argv[]) {

    omp_set_num_threads (4);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num ();
        int n_threads = omp_get_num_threads ();

        printf ("Hello World from thread %d of %d threads\n",
                thread_id, n_threads);
    }

    return 0;
}
```

Vector Addition

1. A simple parallelization
   - Parallelize the code using **`#pragma omp parallel`**
   - Distribute the iterations among threads according to their id
     - You can use a round-robin distribution
   - Execute and measure the performance of the code

2. Parallel for loop
   - Remove the manual workload distribution
   - Distribute the iterations using a **`#pragma omp for`**
   - Execute and measure the performance of the code

3. Removing implicit synchronizations
   - Append the **`nowait`** directive to **`#pragma omp for`**

Extra: Removing hardcoded number of threads
   - Delete the call to the **`omp_set_num_threads`** function
   - Check the job script to see how the number of threads can be set
   - Execute and measure the performance of the code for 2, 4, and 8 threads. How does the performance vary?

```c
int* vector_add (int a[], int b[], int size) {
    int* c = (int*) malloc (size * sizeof (int));

    for (int i = 0; i < size; i++)
        c[i] = a[i] + b[i];

    return c;
}
```

Thank you for attending!

→

hello@macc.fccn.pt

macc.fccn.pt

@minhoacc