Harold Molina-Bulla DTSC-UC3M

MULTIPROCESSOR / MULTICORE PROGRAMMING

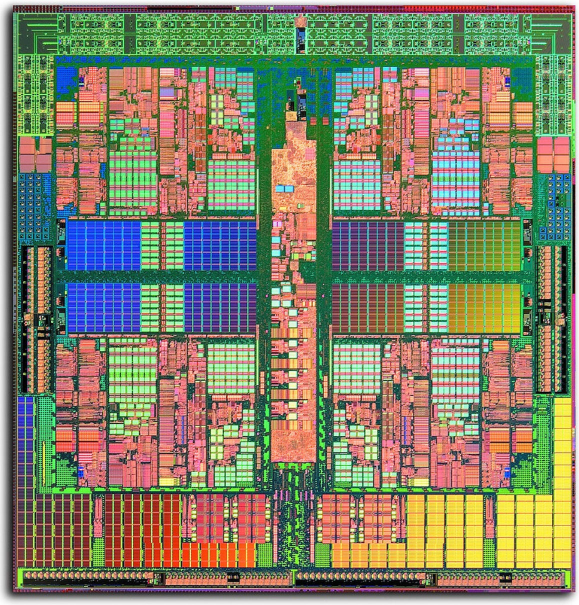
2022

Harold Molina-Bulla DTSC-UC3M

- What is multicore architecture
- What is multiprocessor architecture
- Description of OpenMP
- Programming Multiprocess Programs in Python in a Single Computer

2022

MULTICORE ARCHITECTURE

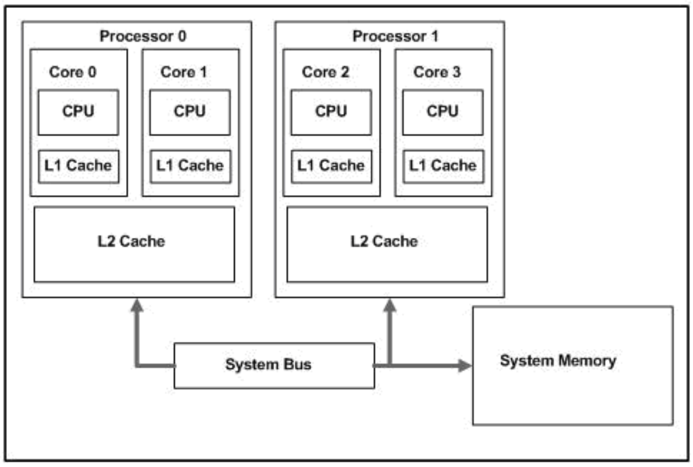


MULTICORE ARCHITECTURE

In a Multicore Processor, the computational cores share the same DIC (Silicon Waffle).

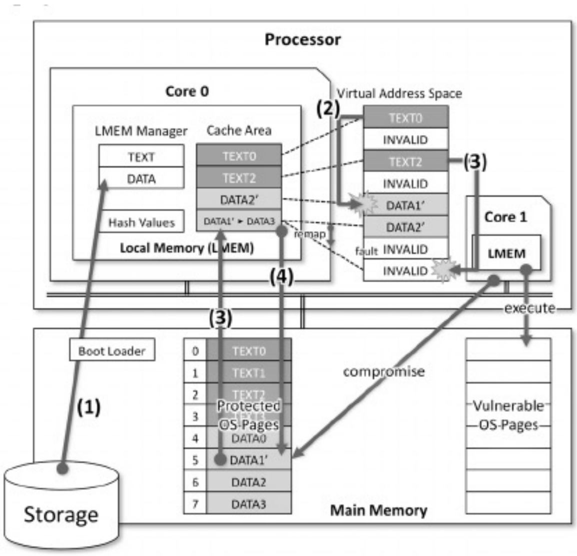
Depending of the Processor Architecture, can share the L2 Memory Cache.

The computational cores uses the same memory and I/O buses



PROCESSOR ARCHITECTURE

- Each CPU-Core has his own Local Memory L1.
- If the data is not stored in L1 memory, it looks in the L2 Cache Memory, otherwise, looks in the Main Memory (L3) or Hard Disk (L4).
- Possible Issues: several cores trying to access to the same memory position.
- Memory Access Violation. Program failure



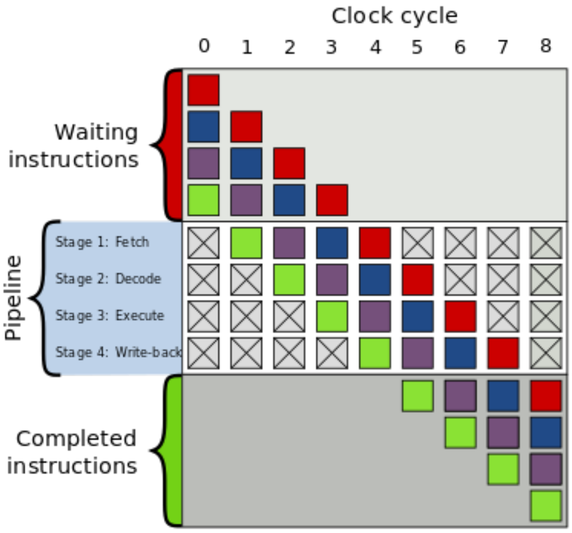
SUPERSCALAR PROCESSOR CORES

Superscalar processors executes one instruction per clock cycle (in average).

But to execute one instruction, it takes much more than 1 clock cycle.

How it solves?

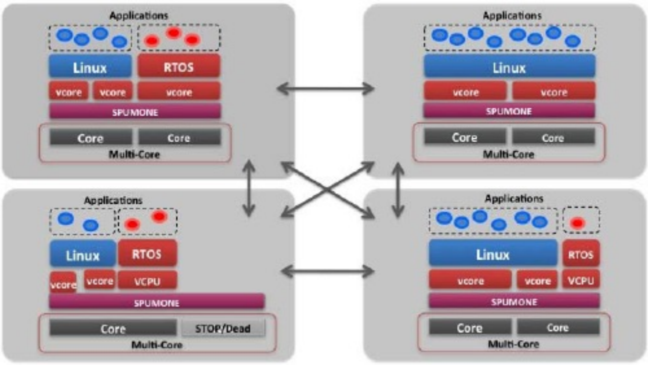
- Solution provided by Henry Ford at the beginning of 20th century:
  - Execution pipes
- Introduces security issues (after 20 years of using it).



Harold Molina-Bulla DTSC-UC3M

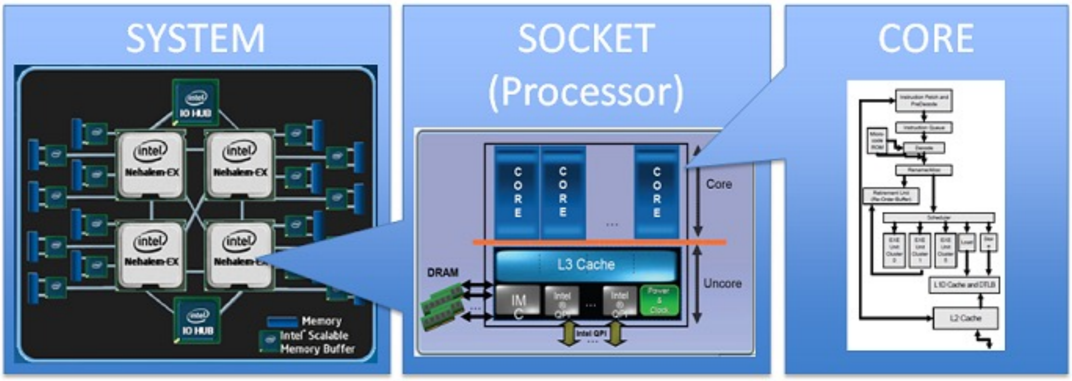
MULTICORE – MULTI - PROCESSOR ARCHITECTURE

- In a multi-processor architecture, there are more than one processor connected in the motherboard.
- Is there a necessary an extra hardware, which synchronizes memory access, I/O access.
- There are an external Memory Management Unit who controls the access to external shared memory.
- There are specialized memory hardware: multichannel memory

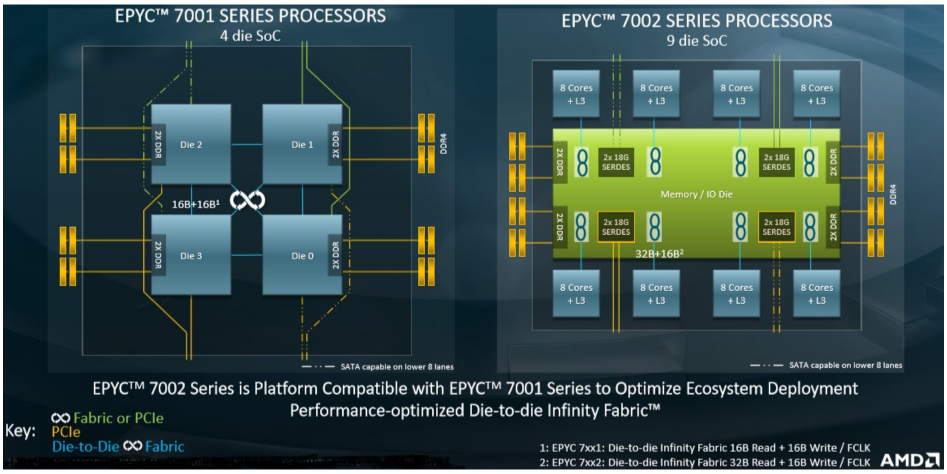


Harold Molina-Bulla DTSC-UC3M

INTEL XEON MULTICORE/MULTIPROCESSOR



Harold Molina-Bulla DTSC-UC3M



EPYC™ 7002 Series is Platform Compatible with EPYC™ 7001 Series to Optimize Ecosystem Deployment

Performance-optimized Die-to-die Infinity Fabric™

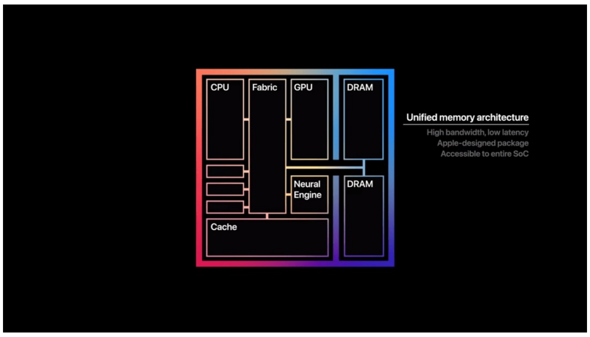
Key: ∞ Fabric or PCIe
PCIe
Die-to-Die ∞ Fabric

1: EPYC 7xx1: Die-to-die Infinity Fabric 16B Read + 16B Write / FCLK

2: EPYC 7xx2: Die-to-die Infinity Fabric 32B Read + 16B Write / FCLK

Harold Molina-Bulla DTSC-UC3M

APPLE SILICON ARCHITECTURE



Unified memory architecture

High bandwidth, low latency
Apple-designed package
Accessible to entire SoC

Harold Molina-Bulla DTSC-UC3M

HOW TO PROGRAMMING IN MULTICORE/MULTIPROCESSOR

- Common programs are single thread programs…that means:
  - It uses one execution program control
  - Each instruction is executed just after the previous one, even it does not need the results of them.
  - Do not uses all the resources of the computer.
  - It is the safetest programming method.
    - It does not have problems of IO conflicts, memory access, memory hazzards, etc
  - It is Operating System Dependent: Not all O.S. are multitask/multicore compliant.

Harold Molina-Bulla DTSC-UC3M

TYPES OF PARALLEL PROGRAMMING BY DATA HANDLE

- Depending on the relation between data and instructions, we can classify our parallel routines in:

	SINGLE INSTRUCION	MULTIPLE INSTRUCTION
SINGLE DATA	X	MISD
MULTIPLE DATA	SIMD	MIMD

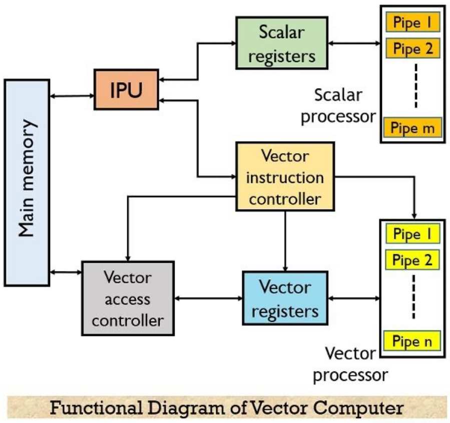
Harold Molina-Bulla DTSC-UC3M

HIGHLY OPTIMIZED SUPERSCALAR INSTRUCTIONS

- Only will be used in SIMD operations
  - Vector operations.
  - High optimized CPU instruction, only available in middle-low level languages:
    - C/C++.
    - Assembler.
  - Needs:
    - Base memory address source data (up to 2 source memory addresses).
    - Base target memory address.
    - Data type and length (or a specific data type: float/double data type).
  - With just one instruction, execute it over all data defined.

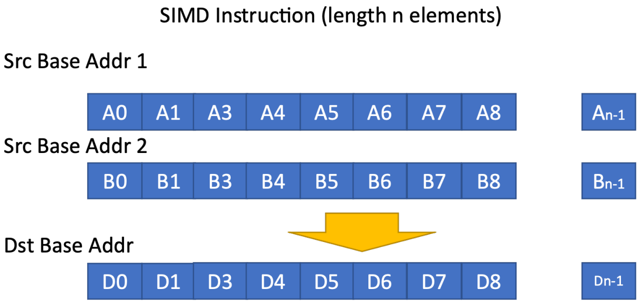
Harold Molina-Bulla DTSC-UC3M

HIGHLY OPTIMIZED SUPERSCALAR INSTRUCTIONS



Functional Diagram of Vector Computer

Electronics Desk



Harold Molina-Bulla DTSC-UC3M

FINE GRANULARITY BY CPU INSTRUCTION

- CPU Instructions:
    - SSE*X* Instructions family
    - AVX-*X*
    - Can check CPU extension in Linux using the command:
        - cat /proc/cpuinfo
- Fastest than user programmed loop.
- Highly tied to CPU version.

Harold Molina-Bulla DTSC-UC3M

TYPES OF PARALLEL PROGRAMMING BY GRANULARITY

- The granularity means the size of the code to be executed by each one of the parallel threads.
    - Fine granularity: We paralelize small operations: for example each one of the terms multiplications in a matrix multiplication.
    - Middle granularity: We paralelize group of operations: for example block of cells in a matrix multiplications.
    - Coarse granularity: We paralelize big group of instructions, even full processes.

Harold Molina-Bulla DTSC-UC3M

TYPES OF PARALLEL PROGRAMMING BY GRANULARITY

- Balance between the Granularity and the communications/control overhead
  - Nothing is free of cost in the world.
  - In a fine granularity level we will need to invoke the control routines to syncronize our process
    - More communications/control overhead
  - In a coarse granularity we reduce the control routines executions
    - May not exploid the parallelization advantages
    - Data exchange and communications can be expensive (due data size exchange)

Harold Molina-Bulla DTSC-UC3M

REQUISITES FOR MULTICORE/MULTIPROCESS PROGRAMMING

- We need Multitask Operating System
- We need a multicore computer (could be multicore in a single processor chip or multiprocessor computer).
- Specific libraries for parallel programming/execution.
  - First solutions: multithread libraries
    - Allows to define multiple threads of execution. The user should define an syncronize the execution.
    - Each execution thread is independent, using a shared memory
    - Sinchronization is hard to define. Uses semaphores and locks
    - Solution implemented when programming in Oracle JAVA

Harold Molina-Bulla DTSC-UC3M

REQUISITES FOR MULTICORE/MULTIPROCESS PROGRAMMING

- Most advanced solutions:
  - OpenMP library
    - Based on compilers for Supercomputers
    - Introduces new structures and metacommands to programming in middle level, common purpouse programming languages, like C/C++ or FORTRAN
    - Using metalanguaje instructions, the programmer defines where the program must be paralellized by the compiler, and introduce extra code to distribute the execution among the execution cores.

Harold Molina-Bulla DTSC-UC3M

OPENMP

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:

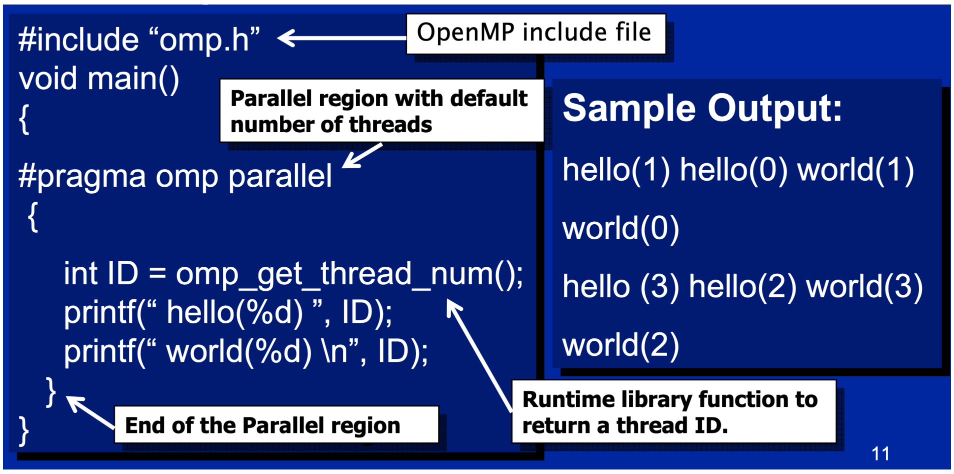
  race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:

  Use synchronization to protect data conflicts.
- Synchronization is expensive so:

  Change how data is accessed to minimize the need for synchronization.

Harold Molina-Bulla DTSC-UC3M

OPENMP



```
#include "omp.h"
void main()
{
#pragma omp parallel
 {
     int ID = omp_get_thread_num();
     printf(" hello(%d) ", ID);
     printf(" world(%d) \n", ID);
   }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

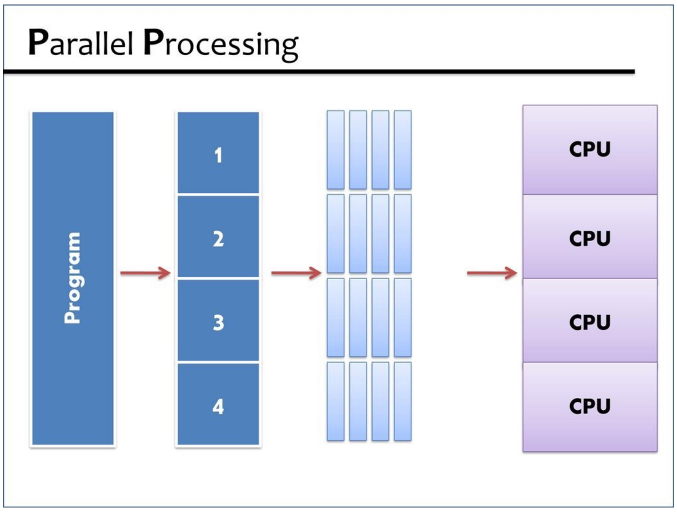
Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

11

Harold Molina-Bulla DTSC-UC3M

PARALLEL PROGRAMING IN PYTHON



Harold Molina-Bulla DTSC-UC3M

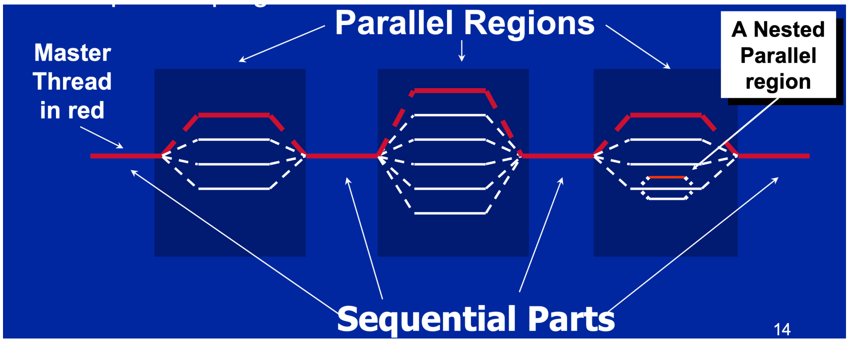
STEPS TO PARALLEL PROGRAMING

- Create threads
- Synchronize parallel tasks
- Parallel loops
- Synchronize single masters
- Memory models

Harold Molina-Bulla DTSC-UC3M

PARALLEL PROGRAMED PROGRAM

- Our programs will have sequential parts and parallel regions, even nested parallel regions.



Harold Molina-Bulla DTSC-UC3M

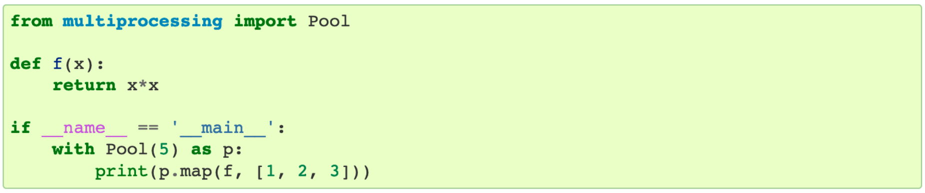
FIRST STEP: CREATE EXECUTION THREADS

- To implement parallel programs in Python we will use multiprocessing module
  - Threads
  - Pool of threads
  - Synchronize threads
  - Sharing memory
  - Queues of execution
- Very powerfull tool, if it is well programmed

Harold Molina-Bulla DTSC-UC3M

CREATE PARALLEL THREADS

- We need create a pool of threads, to execute our code in parallel
- With multiprocess module, we will use the class "Pool"

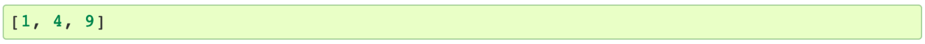


```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output



```
[1, 4, 9]
```

CREATE PARALLEL THREADS

- Parameters of Pool class constructor:
  - Num of paralellel processes.
    - BIG QUESTION: How many processors we will use? How many processors we have?
    - Recommendation: use the same number of physical cores in your computer.
    - What happens if we ask for more processors than we have?
  - Second question: How will we design our code in order to improve or, at least, do not degradate the execution?