

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

## Computação Paralela (2022/2023)

Escolaridade:

1h TP / semana      2h P / semana

Docentes (aulas TP e P):

Rui Costa

americo.costa@ua.pt

Física (13.3.32.2)

Nuno Lau

nunolau@ua.pt

IEETA (IRIS Lab / 2.07)

Página web em [elearning.ua.pt](http://elearning.ua.pt)

login: utilizador universal

Slides adaptados da bibliografia

- Identificar e classificar os diferentes modelos e os diferentes níveis da programação paralela
- Reconhecer programas suscetíveis de ser paralelizados e escolher a melhor abordagem
- Desenvolver, depurar e otimizar um programa paralelo
- Utilizar eficazmente clusters computacionais

# Programa (Aulas TeoricoPráticas)

---

1. Computação paralela de alto desempenho
2. Modelos de programação paralela
3. OpenMP
4. CUDA
5. MPI
6. Otimização

As aulas práticas seguem uma filosofia do *saber fazer* e visam a realização de pequenos trabalhos sobre os diferentes assuntos.

- Paralelização com threads
- Extensões multimédia de processadores
- OpenMP
- CUDA
- MPI

- **Componente TeoricoPrática**
  - Teste final (durante época de exames)
  - Nota mínima: 8
- **Componente Prática**
  - 2 trabalhos
  - 50%.TP1 + 50%.TP2
  - Nota mínima: 8
- **Nota final**
  - $NF = 40\%.CTP + 60\%.CP$

- Efetuar uma implementação precisa dos problemas propostos
- Os trabalhos são desenvolvidos também fora das aulas práticas
- **O plágio será fortemente penalizado**

- Rauber and G. Rünger, Parallel Programming: for Multicore and Cluster Systems, 2nd edition, Springer, 2013.
- Hager and G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2010.
- J. Quinn, Parallel Programming: in C with MPI and OpenMP, McGraw-Hill, 2003.
- S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1997.
- Cheng, M. Grossman, and T. McKercher, Professional CUDA C Programming, Wrox, 2014.

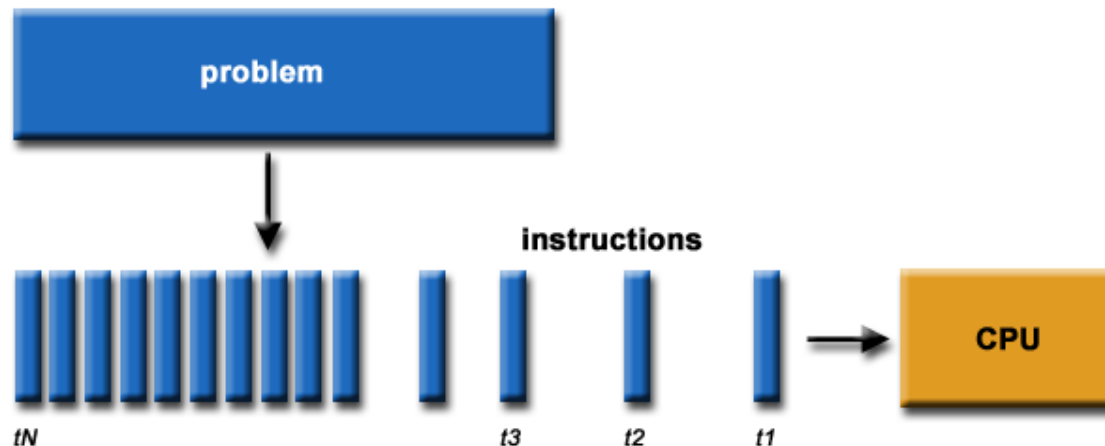


# Questões?

---

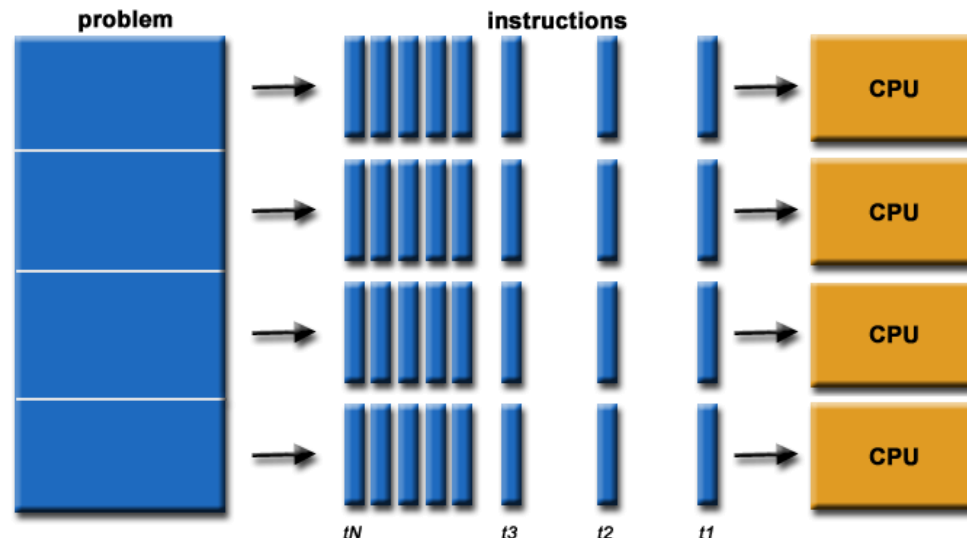
# What is Parallel Computing? (1)

- Traditionally, software has been written for **serial** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



# What is Parallel Computing? (2)

- In the simplest sense, **parallel computing** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs/Cores



- The compute resources can include:
  - A single computer with multiple processors/cores
  - A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA ...)
  - An arbitrary number of computers connected by a network
  - A combination of the above

- The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously
  - Execute multiple program instructions at any moment in time
  - Solved in less time with multiple compute resources than with a single compute resource

# Parallel Computing: what for? (1)

- Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world:
  - many complex, interrelated events happening at the same time, yet within a sequence.
- Some examples:
  - Planetary and galactic orbits
  - Weather and ocean patterns
  - Tectonic plate drift
  - Rush hour traffic in Paris
  - Automobile assembly line
  - Daily operations within a business
  - Building a shopping mall
  - Ordering a hamburger at the drive through.

# Parallel Computing: what for? (2)

- Traditionally, parallel computing has been considered to be "the high end of computing" and has been motivated by numerical simulations of complex systems and "Grand Challenge Problems" such as:
  - Weather and climate
  - Chemical and nuclear reactions
  - Biological, human genome
  - Geological, seismic activity
  - Mechanical devices - from prosthetics to spacecraft
  - Electronic circuits
  - Manufacturing processes

# Parallel Computing: what for? (3)

- Commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Example applications include:
  - Parallel databases
  - Data mining
  - Machine learning
  - Oil exploration
  - Web search engines, web based business services
  - Computer-aided diagnosis in medicine
  - Management of national and multi-national corporations
  - Advanced graphics and virtual reality, particularly in the entertainment industry
  - Networked video and multi-media technologies
  - Collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time



# Why Parallel Computing? (1)

---

- This is a legitime question!
- Parallel computing is complex on any aspect!
- The **primary reasons** for using parallel computing:
  - **Save time and/or money**
    - wall clock time
    - using multiple "cheap" computing resources instead of paying for time on a supercomputer
  - **Solve larger problems**
  - **Provide concurrency** (do multiple things at the same time)

# Computação Paralela

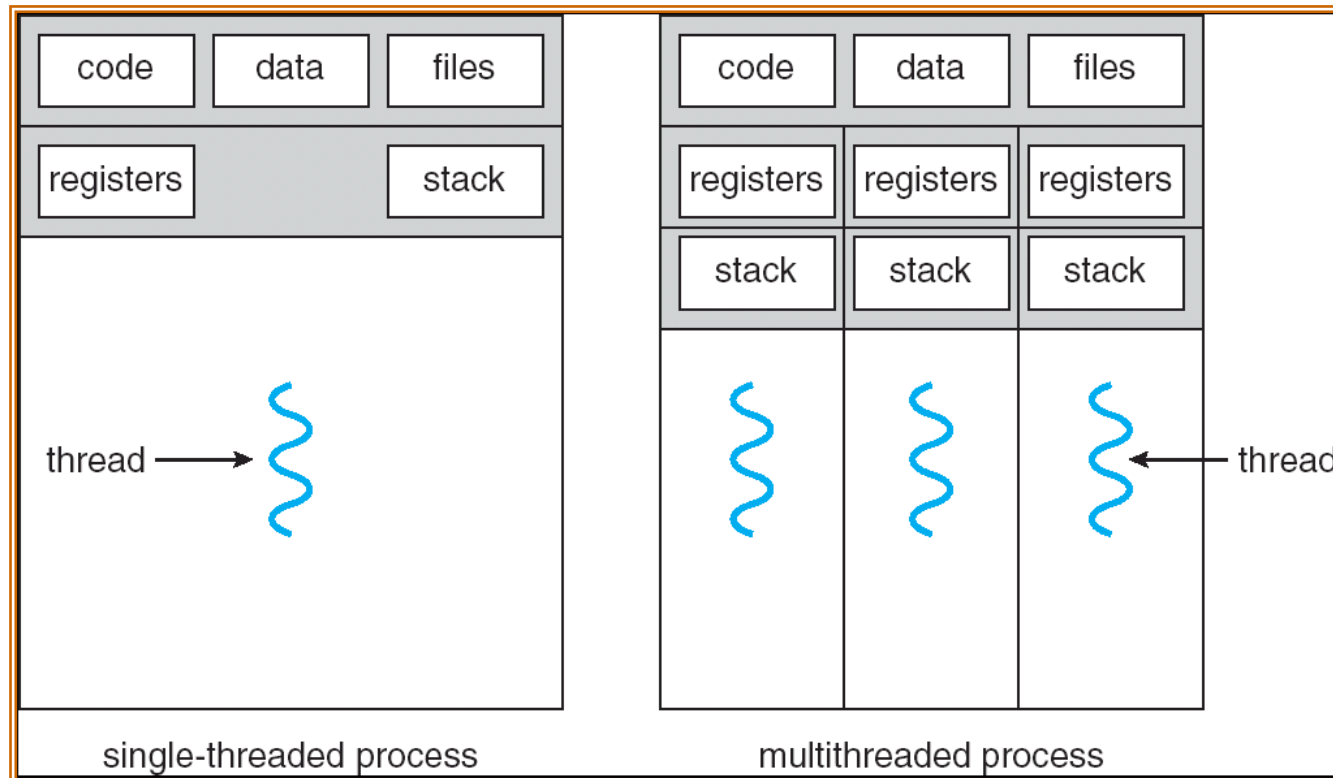
Mest. Int. Engenharia Computacional  
Mest. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

- Programas têm geralmente de executar diversas atividades distintas
- Usando *threads*, o programador pode desenvolver o programa como um conjunto de fluxos de execução sequenciais, um para cada atividade
- Cada *thread* comporta-se como tendo o seu processador próprio.
- Todas as *threads* do mesmo processo partilham espaço de endereçamento (memória)

# Processos *Single* e *Multi threaded*



# Processos e *Threads*

<b>Per-process items</b>	<b>Per-thread items</b>
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

- Num servidor web, cada pedido de página pode ser processado numa *thread* separada
- Há uma (*dispatcher*) *thread* que recebe todos os pedidos e os distribui pelas (*worker*) *threads*

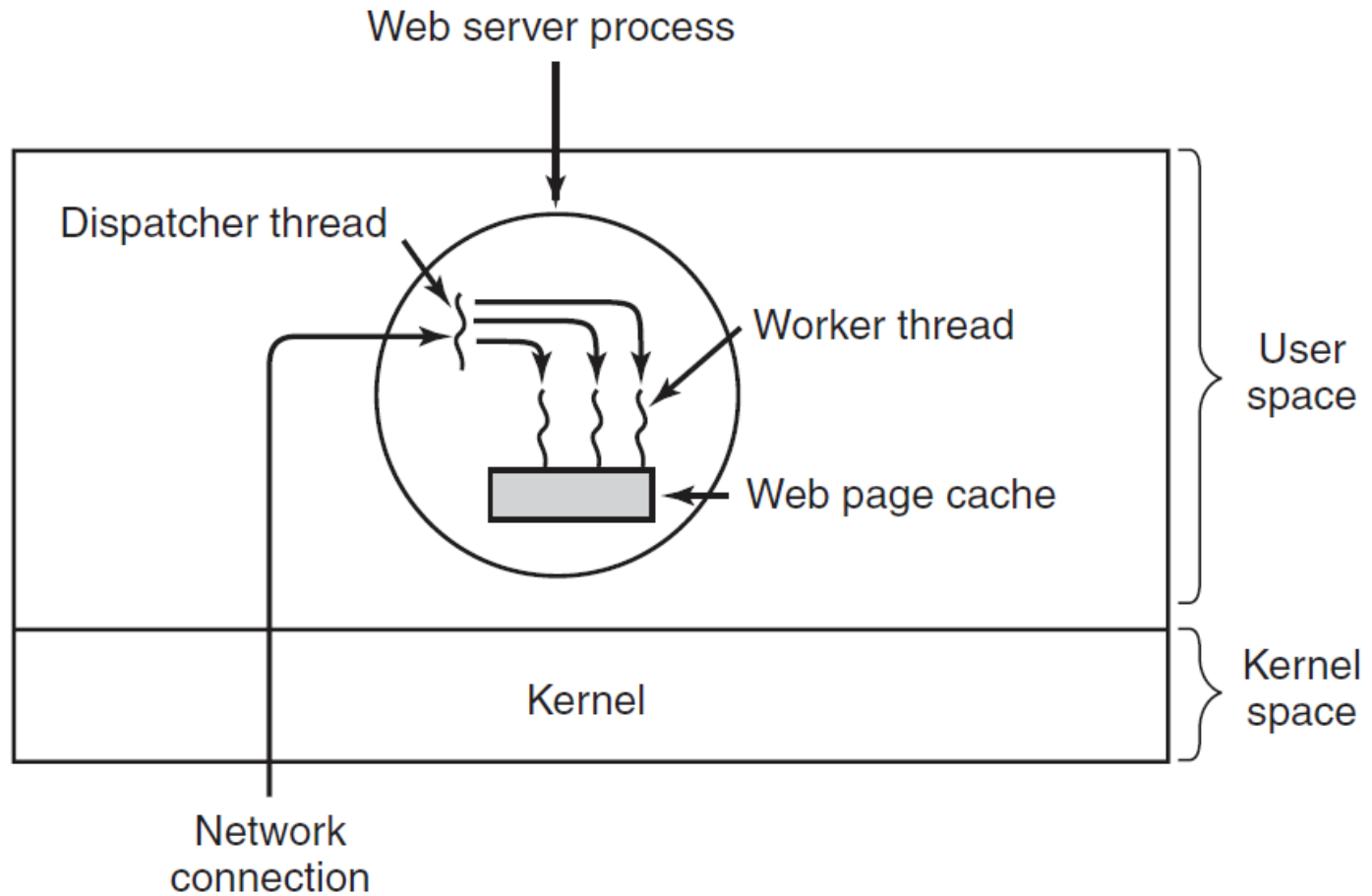
```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

Dispatcher thread

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

Worker threads

# Servidor Web Multithreaded



- POSIX standard para a criação e sincronização de *threads*
- API define comportamento, mas não implementação
- Comum em sistemas UNIX (Linux, Mac OS X)



Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

# Criar POSIX *Threads*

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine) (void *), void *arg);`

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 5

void *PrintMsg(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! Thread ID, %d\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;

    for( i = 0; i < NUM_THREADS; i++ ) {
        printf( "main() : creating thread, %d\n", i);
        rc = pthread_create(&threads[i], NULL, PrintMsg, (void *)i);

        if (rc) {
            printf("Error: unable to create thread, %d\n", rc);
            exit(1);
        }
    }
    pthread_exit(NULL);
}
```

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

# Why Parallel Computing? (1)

---

- This is a legitime question!
- Parallel computing is complex on any aspect!
- The **primary reasons** for using parallel computing:
  - **Save time and/or money**
    - wall clock time
    - using multiple "cheap" computing resources instead of paying for time on a supercomputer
  - **Solve larger problems**
  - **Provide concurrency** (do multiple things at the same time)

# Why Parallel Computing? (2)

---

- Other reasons might include:
  - **Taking advantage of non-local resources**
    - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
    - Example: SETI@Home
  - **Overcoming memory constraints**
    - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

# Limitations of Serial Computing

- Both physical and practical reasons pose significant constraints to simply building ever faster serial computers:
- Transmission speeds
  - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/ns) and the transmission limit of copper wire (9 cm/ns). Increasing speeds necessitate increasing proximity of processing elements
- Limits to miniaturization
  - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be
- Economic limitations
  - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive

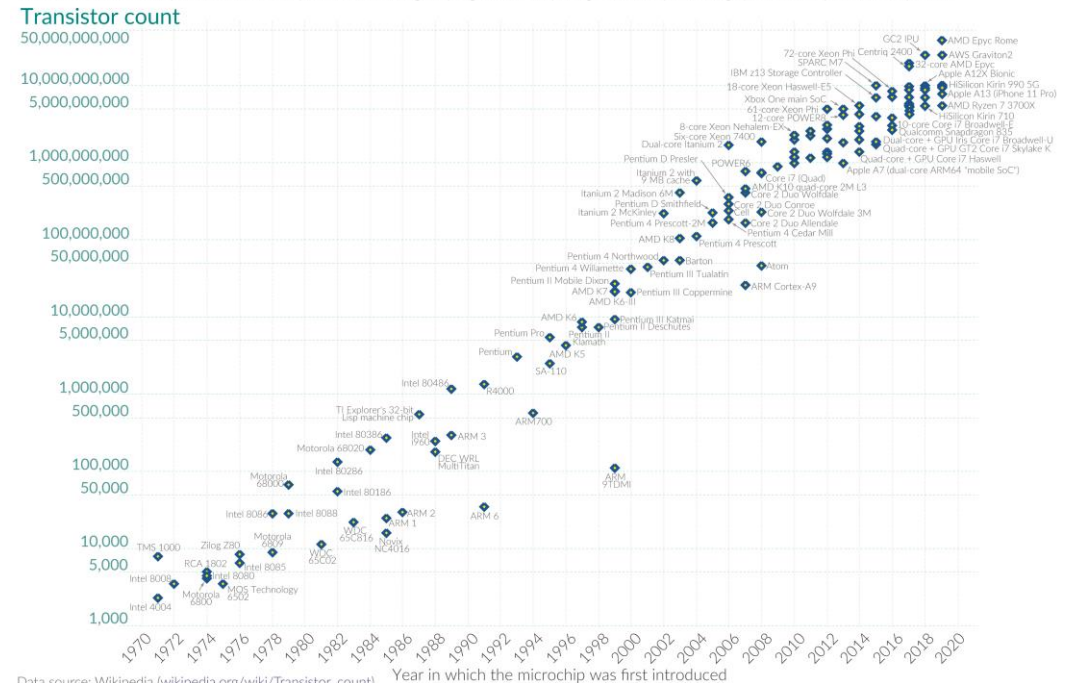
# Moore's Law

- The number of transistors on integrated circuits doubles approx every 2 years
- Chip performance double every 18-24 months
- Power consumption is proportional to frequency
- Clock speed saturates at 3 to 4 GHz.
  - End of free lunch
  - Future is parallel!

**Moore's Law: The number of transistors on microchips doubles every two years**

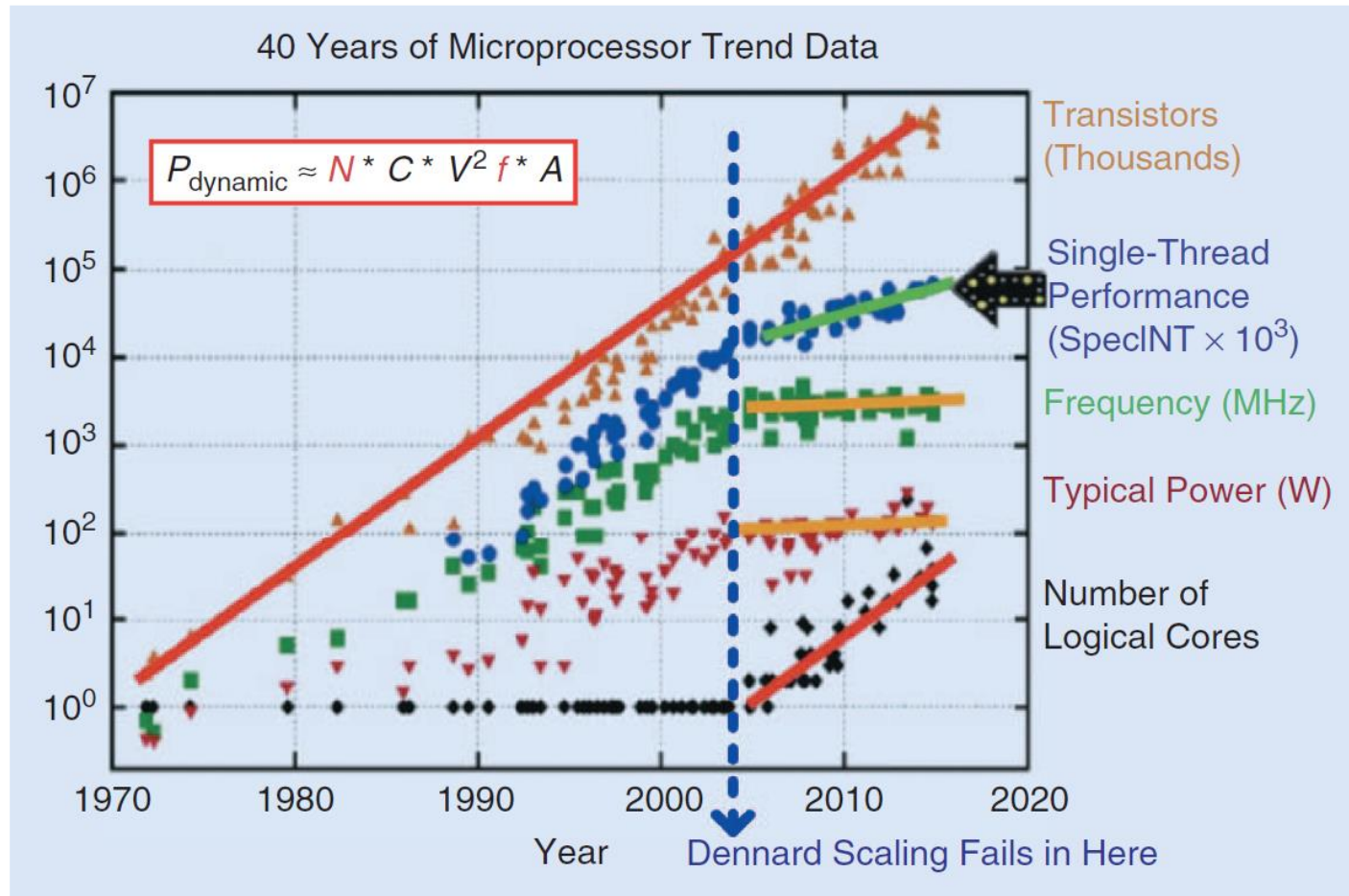
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data



Data source: Wikipedia ([wikipedia.org/wiki/Transistor\\_count](https://wikipedia.org/wiki/Transistor_count))  
OurWorldinData.org – Research and data to make progress against the world's largest problems. Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

# Moore's Law





- during the past 20 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.
- Mixing general purpose solutions (your PC...) and very specialized solutions as GPGPU from Nvidia, Tensor Processing Units from Google, Vision Processing Unit ...

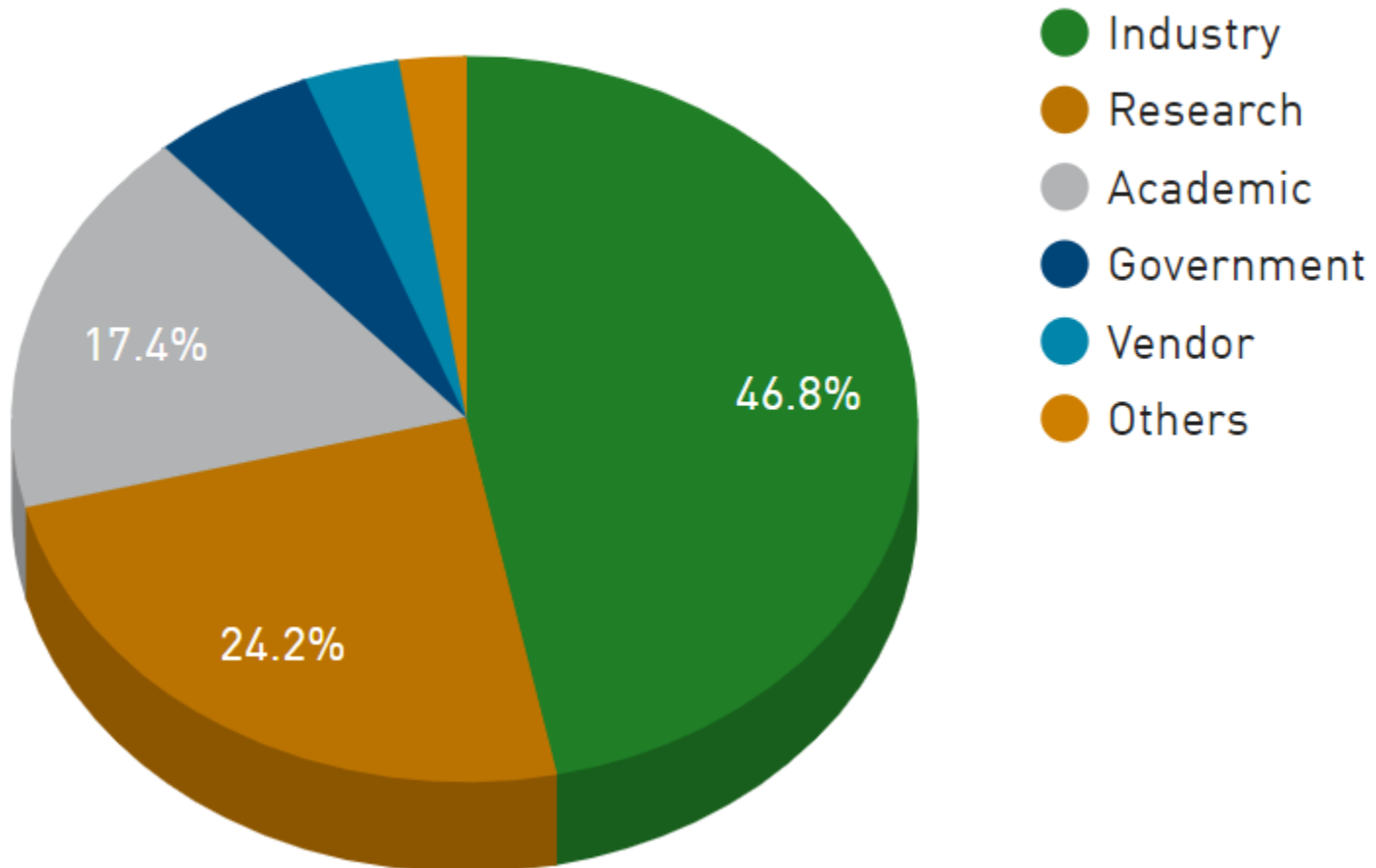
# Who and What? (1)

---

- [Top500.org](http://Top500.org) provides statistics on parallel computing users
- Charts that follow are just a small sample
- Some things to note:
  - Sectors may overlap
    - For example, Research may be classified Academic. Respondents have to choose between the two.

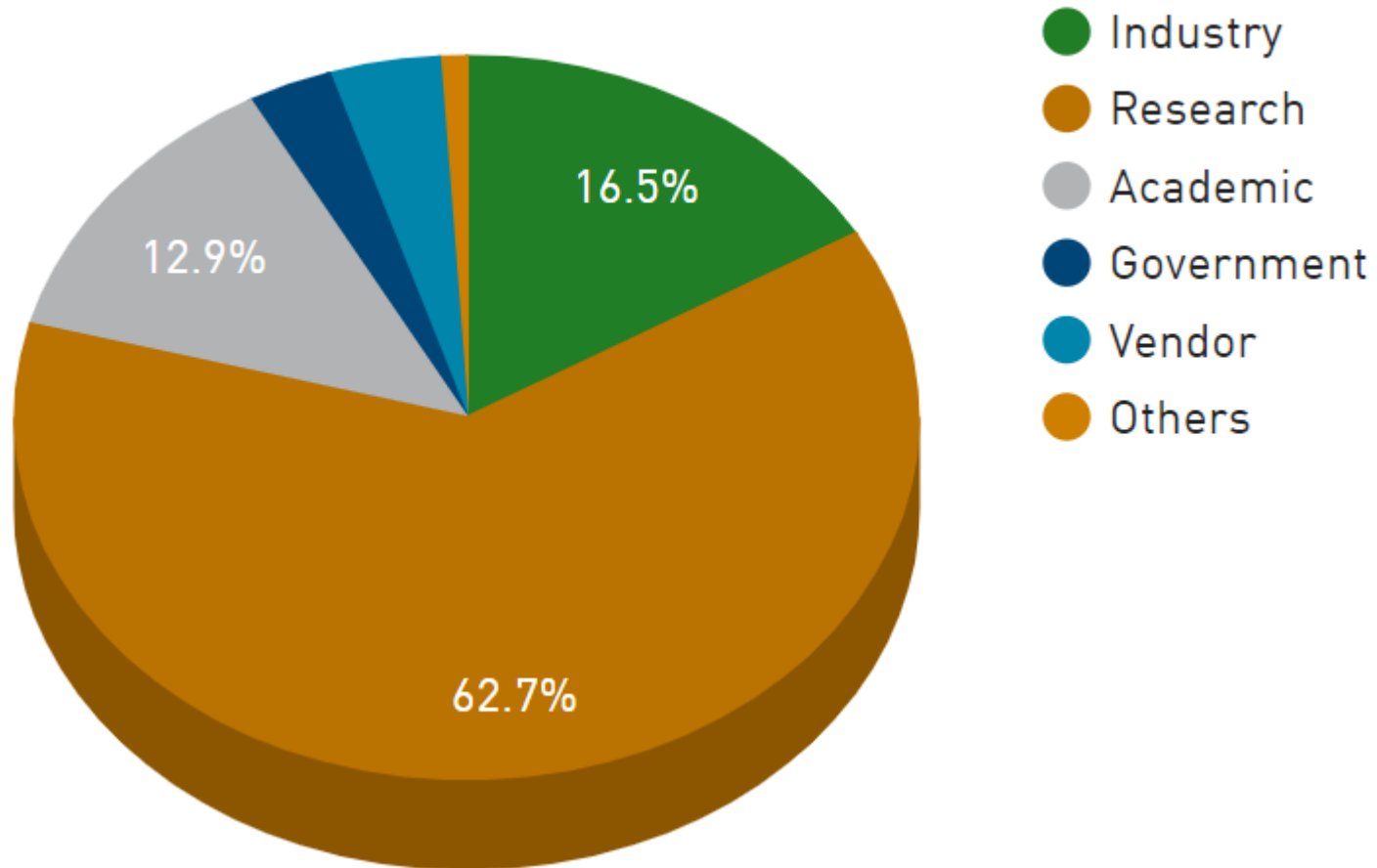
# Who and What? (2)

**Segments System Share**



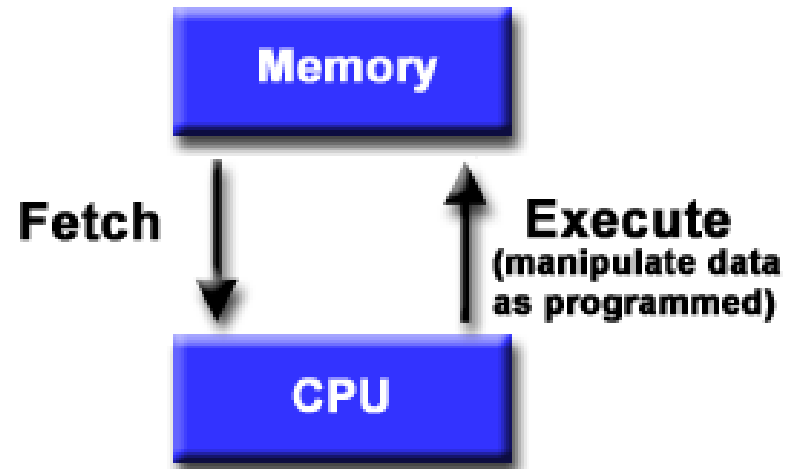
# Who and What? (3)

Segments Performance Share



- For over 60 years, virtually all computers have followed a common machine model known as the **von Neumann computer**. Named after the Hungarian mathematician John von Neumann.
- A von Neumann computer uses the **stored-program concept**. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

- Basic design
  - Memory is used to store both program and data instructions
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.



- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called **Flynn's Taxonomy**.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

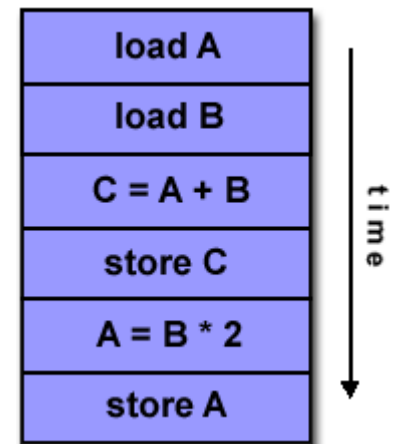
- The matrix below defines the 4 possible classifications according to Flynn

<b>S I S D</b> Single Instruction, Single Data	<b>S I M D</b> Single Instruction, Multiple Data
<b>M I S D</b> Multiple Instruction, Single Data	<b>M I M D</b> Multiple Instruction, Multiple Data



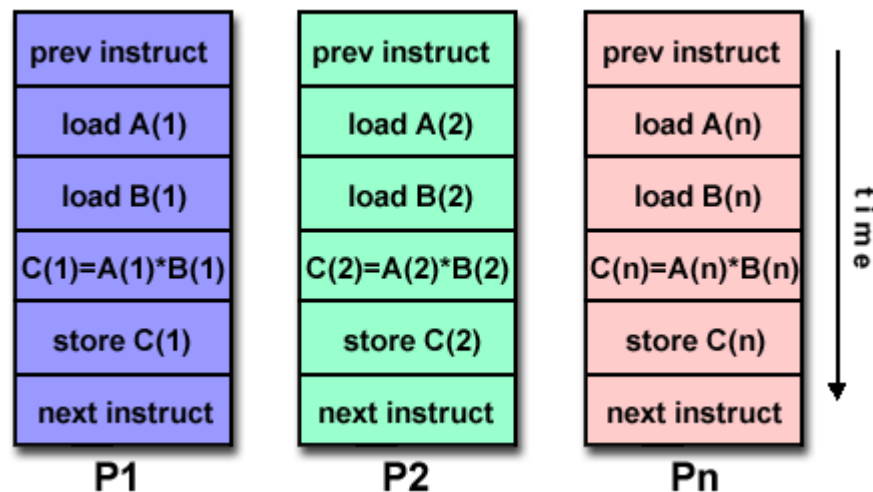
# Single Instruction, Single Data (SISD)

- **A serial (non-parallel) computer**
- **Single instruction:** only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single data:** only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples:
  - PCs with one core, single CPU workstations and mainframes



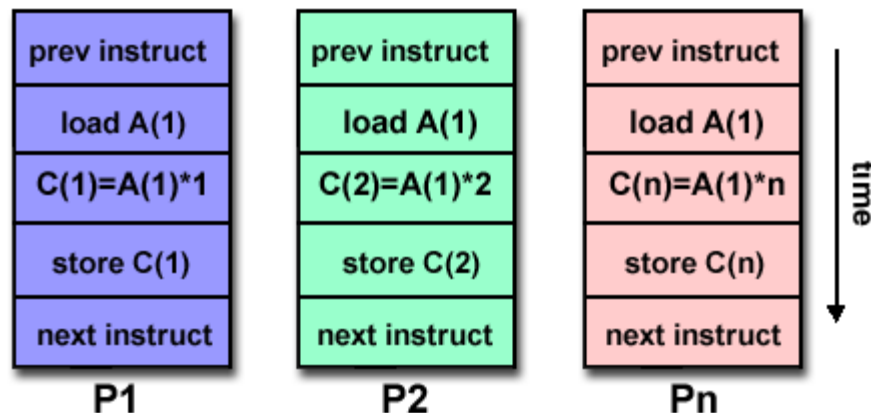
# Single Instruction, Multiple Data (SIMD)

- **A type of parallel computer**
- **Single instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple data:** Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820, MMX, SSE



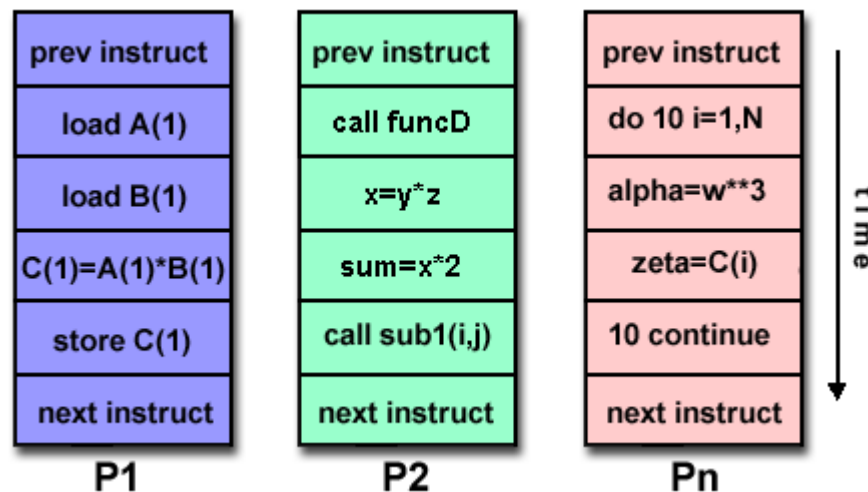
# Multiple Instruction, Single Data (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- **Few actual examples** of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971) that was capable of this type of processing.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - Multiple cryptography algorithms attempting to crack a single coded message.



# Multiple Instruction, Multiple Data (MIMD)

- Currently, **the most common type of parallel computer**. Most modern computers fall into this category.
- **Multiple Instruction**: every processor may be executing a different instruction stream
- **Multiple Data**: every processor may be working with a different data stream
- Execution can be **synchronous** or **asynchronous**, **deterministic** or **non-deterministic**
- Examples:
  - most current PCs and supercomputers, networked parallel computer "grids" and multi-processor SMP computers.



Like everything else, parallel computing has its own "jargon". Some of the more commonly used terms associated with parallel computing are listed below. Most of these will be discussed in more detail later.

- **Task**
  - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.
- **Parallel Task**
  - A task that can be executed by multiple processors safely (yields correct results)
- **Serial Execution**
  - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

- **Parallel Execution**

- Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

- **Shared Memory**

- From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

- **Distributed Memory**

- In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

- **Communications**

- Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

- **Synchronization**

- The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
- Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

- **Granularity**

- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- **Coarse:** relatively large amounts of computational work are done between communication events
- **Fine:** relatively small amounts of computational work are done between communication events

- **Observed Speedup**

- Observed speedup of a code which has been parallelized, defined as:
$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$
- One of the simplest and most widely used indicators for a parallel program's performance.



- **Parallel Overhead**

- The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
  - Task start-up time
  - Synchronizations
  - Data communications
  - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
  - Task termination time

- **Massively Parallel**

- Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but nowadays it can reach millions of processors

- **Scalability**

- Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
  - Hardware - particularly memory-cpu bandwidths and network communications
  - Application algorithm
  - Parallel overhead related
  - Characteristics of your specific application and coding

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

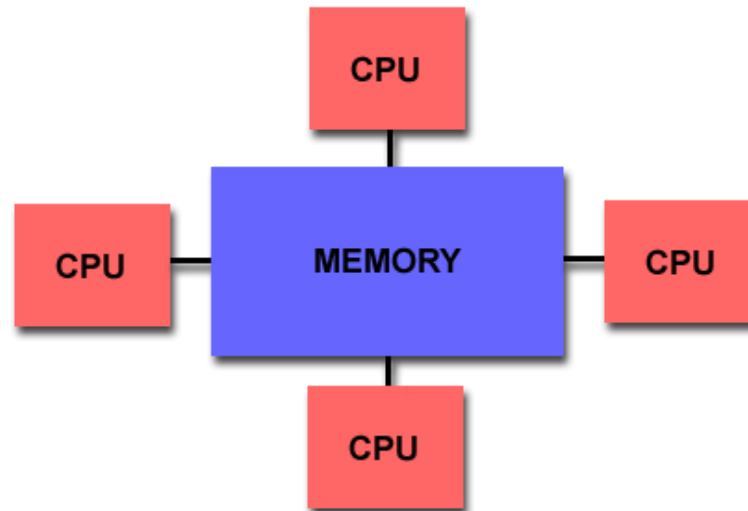
Rui Costa, Nuno Lau

# Parallel Computer Memory Architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

# Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.



- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.

- **Uniform Memory Access (UMA):**
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - **Equal access and access times to memory**
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- **Non-Uniform Memory Access (NUMA):**
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - **Not all processors have equal access time to all memories**
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

- **Advantages**

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

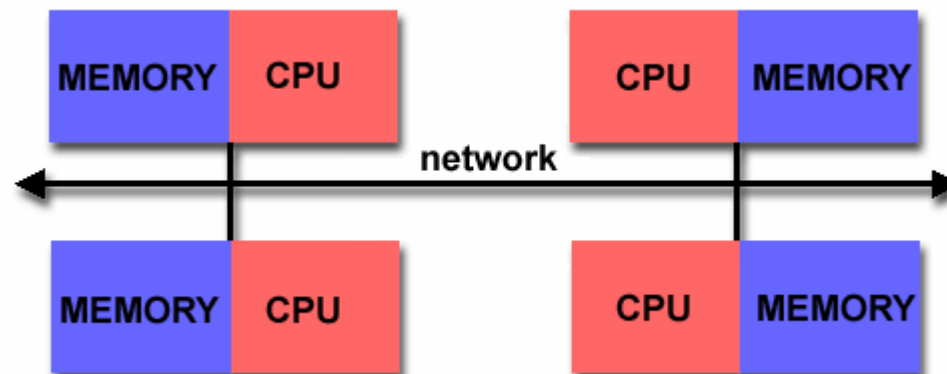
- **Disadvantages:**

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.



# Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a **communication network** to connect inter-processor memory.
- **Processors have their own local memory.** Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of **cache coherency does not apply**.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.



- **Advantages**

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

- **Disadvantages**

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

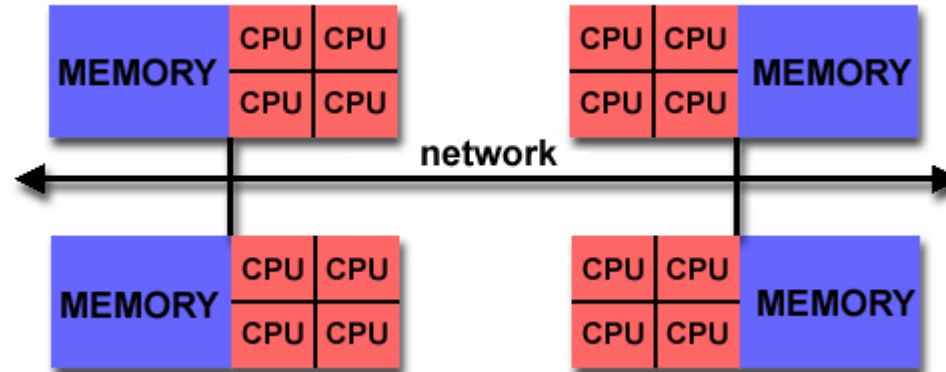
# Hybrid Distributed-Shared Memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

# Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

# Superscalar Compiler and Processor

Sequential  
source code

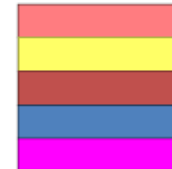
```
a = foo(b);  
for (i=0, i<
```

Superscalar compiler

Find independent  
operations

Schedule  
operations

Sequential  
machine code



Superscalar processor

Check instruction  
dependencies

Schedule  
execution

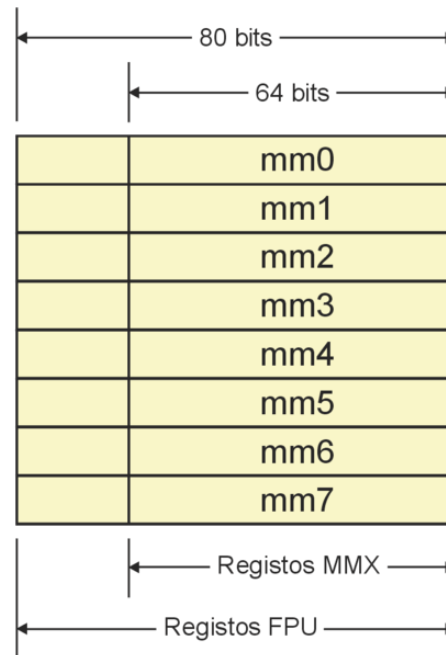
- Extensions to the base ISA that allow a form of vector computation
- “Vectors” are implemented in dedicated registers
  - MMX: 64 bits
  - SSE: 128 bits
  - AVX: 256 bits
  - AVX-512: 512 bits
- Registers of  $N$  bits may be used as vectors of  $2 \times (N/2)$  elements,  $4 \times (N/4)$  elements, etc.
- One multimedia instruction applies simultaneously to all elements of a register

- MMX
  - 57 instructions added to Pentium
- Extended MMX
- SSE
  - Pentium III
  - 71 instructions (52 FP SIMD, 19 MMX)
- SSE2
  - 144 new instructions (Pentium 4)
  - 128 bit registers
  - Cache-control
- SSE3
  - 13 new instructions (Pentium 4 - 2004)
  - Horizontal processing of values in a register
- SSSE3
  - 32 new instructions (Core)
- SSE4
  - 54 instructions (Penryn - 2008)

- 3D Now!
  - AMD
  - 45 instructions (21 FP SIMD, 19 MMX, 5 DSP)
- AltiVec
  - Motorola
  - 162 instructions
- AVX
  - Sandy Bridge – 2011
  - 256 bit registers
  - Instructions with 3 operands
- AVX2
  - Haswell New Instructions, 2013
  - Broadcast/permute operations on data elements
  - Vector shift instructions with variable-shift count per data element
  - Instructions to fetch non-contiguous data elements from memory
- AVX-512
  - Proposed in 2013, First Processor 2016
  - 512 bit registers
  - Several extensions: Foundation, Prefetch, Vector Neural Network, etc.

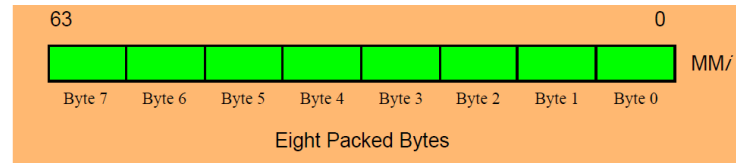


- 8 registers of 64 bits
  - MM0, MM1, ..., MM7
  - Are implemented on the same hardware as the FP registers: ST0, ..., ST7
    - This allowed to maintain compatibility with existing operating systems

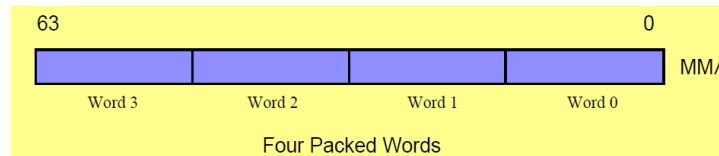


# MMX data types

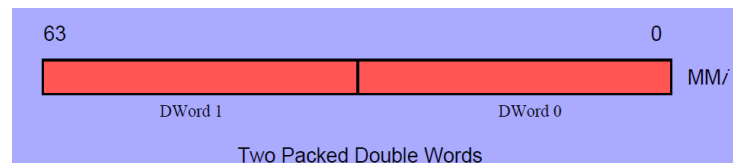
- 8 bytes array



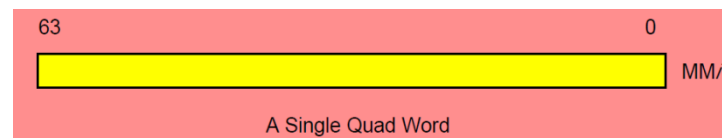
- 4 words array (16 bit/word)



- 2 double words array (32 bits)

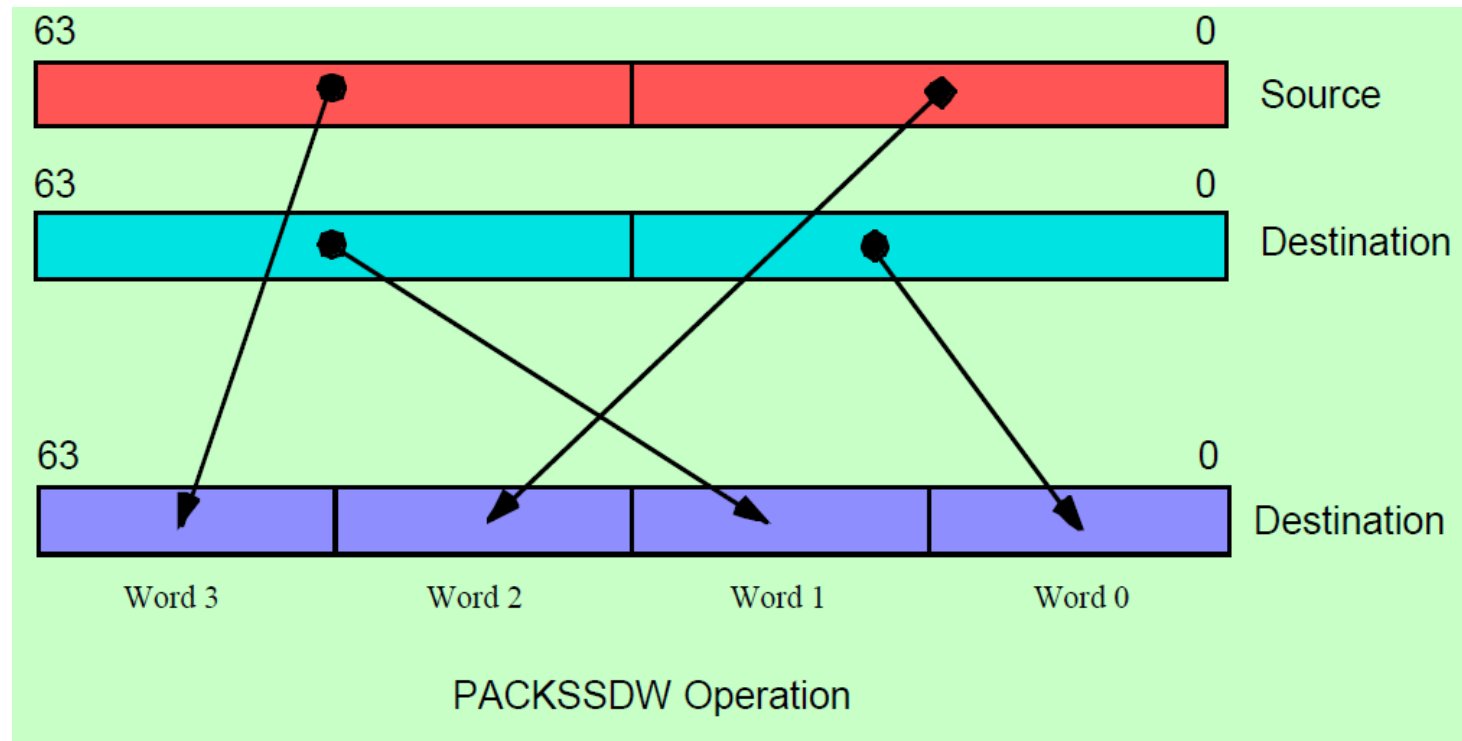


- Quadword (64 bits)

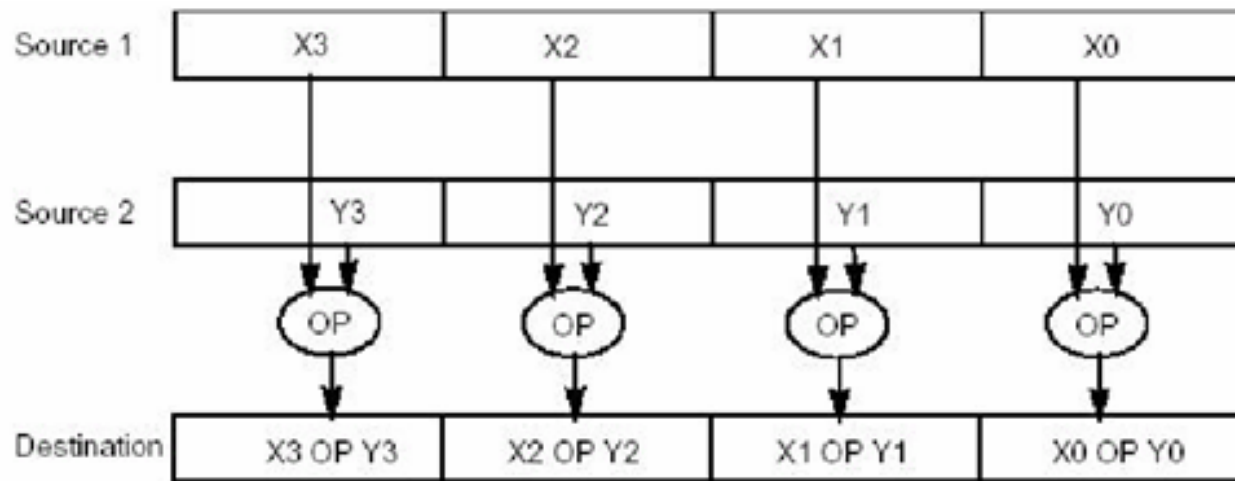


- Data transfer
  - Move data between registers, memory and MMX registers
  - `movd` (32 bits) and `movq` (64 bits)
- Conversion
  - Converts larger data types to smaller data types and vice-versa
- Packed arithmetic
- Comparisons
- Logic operations
- Shift and Rotate
- EMMS
  - Prepares the processor to execute FP code again

- Conversion



- Packed arithmetic
  - SIMD: *Single Instruction Multiple Data*
  - paddb (8bit), paddw (16bit), paddd (32bit)
  - paddb, paddsb (*signed saturation*), paddusb (*unsigned saturation*)

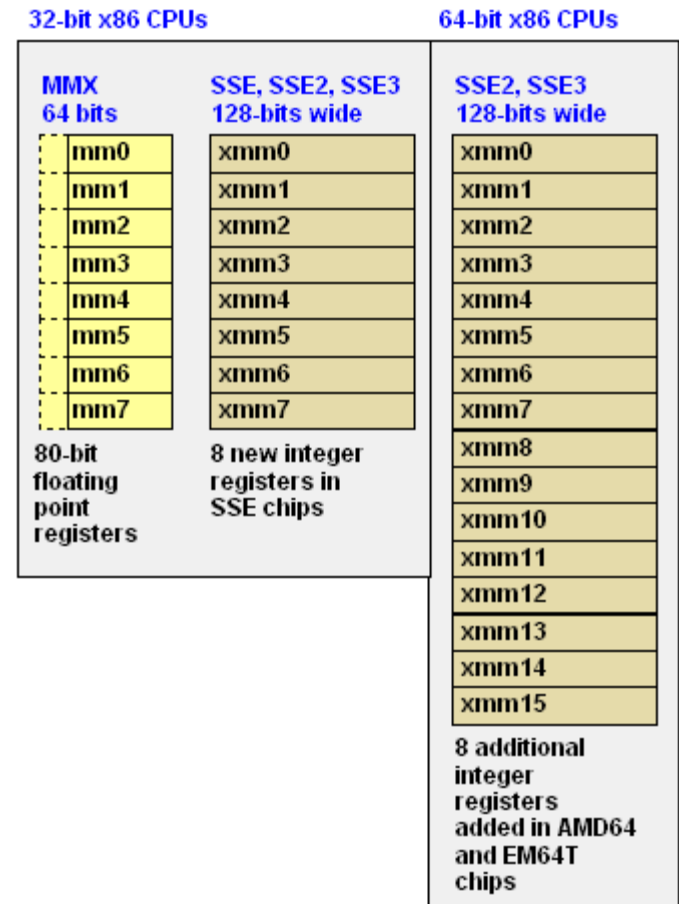


# MMX instructions

Packed Arithmetic	Wrap Around	Signed Sat	Unsigned Sat
Addition	PADD	PADDs	PADDUS
Subtraction	PSUB	PSUBs	PSUBUS
Multiplication	PMULL/H		
Multiply & add	PMADD		
Shift right Arithmetic	PSRA		
Compare	PCMPcc		
Conversions	Regular	Signed Sat	Unsigned Sat
Pack		PACKSS	PACKUS
Unpack	PUNPCKL/H		
Logical Operations	Packed	Full 64-bit	
And		PAND	
And not		PANDN	
Or		POR	
Exclusive or		PXOR	
Shift left	PSLL	PSLL	
Shift right	PSRL	PSRL	
Transfers and Memory Operations	32-bit	64-bit	
Register-register move	MOVD	MOVQ	
Load from memory	MOVD	MOVQ	
Store to memory	MOVD	MOVQ	
Miscellaneous			
Empty multimedia state	EMMS		

# SSE/SSE2 data types

- 8 registers of 128 bits
  - XMM0, ..., XMM7
- 4 FPs simple precision (SSE)
- 2 FPs double precision (SSE2)
- 16 bytes (SSE2)
- 8 words (SSE2)
- 4 double words (SSE2)
- 1 inteiro 128bit (SSE2)



## Code:

```
int A[size], B[size], C[size];
```

```
...
```

```
for (i = 0 ; i < size ; i++)
```

```
    C[i] = A[i] + B[i];
```

## Parallelism?



# Sum arrays

```
movdqa (%eax,%edx,4), %xmm0    # load A[i] to A[i+3]
movdqa (%ebx,%edx,4), %xmm1    # load B[i] to B[i+3]
padd    %xmm0, %xmm1           # CCCC = AAAA + BBBB
movdqa %xmm1, (%ecx,%edx,4)     # store C[i] to C[i+3]
addl    $4, %edx               # i += 4
```

- `movdqa (%eax,%edx,4), %xmm0`
  - **mov**: transferência
  - **dq**: double quad
  - **a**: align
  - **(%eax,%edx,4)**:  $\text{eax} + 4 * \text{edx}$

# Sum elements of an array

---

## Code:

```
int A[size], total=0;  
...  
for (i = 0 ; i < size ; i++)  
    total += A[i];
```

## Parallelism?

# Sum elements of an array

## Restructured code:

```
int A[size], temp[4], total;
temp[0]=temp[1]=temp[2]=temp[3]=0;
for (i = 0 ; i < size ; i+=4) {
    temp[0] += A[i];    temp[1] += A[i+1];
    temp[2] += A[i+2]; temp[3] += A[i+3];
}
total = temp[0]+temp[1]+temp[2]+temp[3];
```

## Parallelism?

# Internal product of 2 vectors

## Code:

```
int A[size], B[size], iprod;  
...  
for (i = 0 ; i < size ; i++)  
    iprod += A[i]*b[i];
```

## Parallelism?

## Loop unrolled code:

```
int A[size], B[size], iprod=0;
for (i = 0 ; i < size ; i+=3) {
    iprod += A[i]*B[i];
    iprod += A[i+1]*B[i+1];
    iprod += A[i+2]*B[i+2];
}
```

## Parallelism?

## Loop unrolled code:

```
int A[size], B[size], iprod=0;
for (i = 0 ; i < size ; i+=3) {
    temp0 = A[i]*B[i];
    temp1 = A[i+1]*B[i+1];
    temp2 = A[i+2]*B[i+2];
    iprod += temp0 + temp1 + temp2
}
```

## Parallelism?

- Internal product of 2 vectors (**a** and **b**)

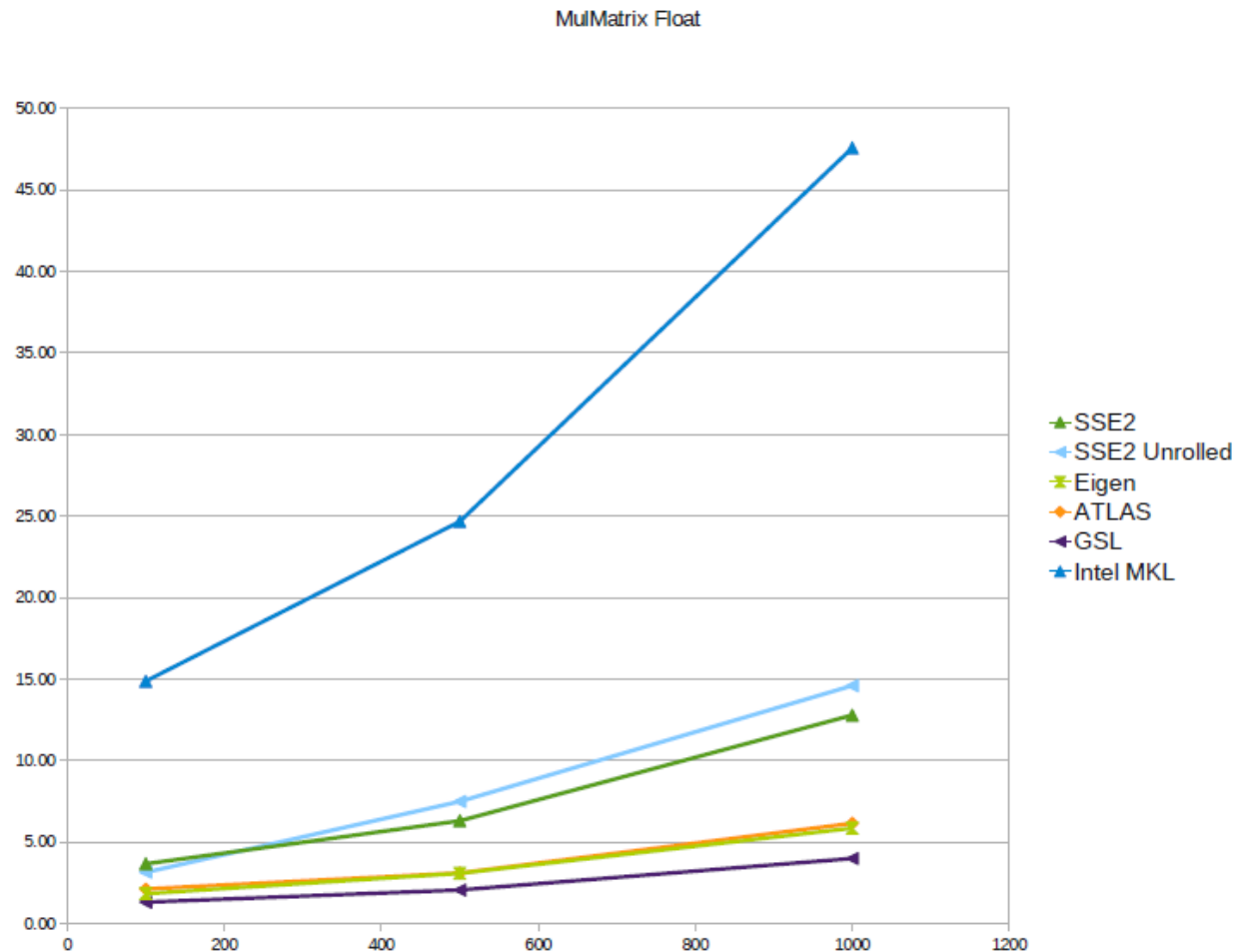
```
length6 = (size/24)*24
for(; i < length6; i += 24){
    __asm__ volatile
    (// instruction comment
      "\n\t movdqa 0x00(%0),%%xmm2 \t#" "\n\t movdqa 0x10(%0),%%xmm3 \t#"
      "\n\t movdqa 0x20(%0),%%xmm4 \t#" "\n\t movdqa 0x30(%0),%%xmm5 \t#"
      "\n\t movdqa 0x40(%0),%%xmm6 \t#" "\n\t movdqa 0x50(%0),%%xmm7 \t#"

      "\n\t mulps 0x00(%1),%%xmm2 \t#" "\n\t mulps 0x10(%1),%%xmm3 \t#"
      "\n\t mulps 0x20(%1),%%xmm4 \t#" "\n\t mulps 0x30(%1),%%xmm5 \t#"
      "\n\t mulps 0x40(%1),%%xmm6 \t#" "\n\t mulps 0x50(%1),%%xmm7 \t#"

      "\n\t addps %%xmm2,%%xmm0 \t#" "\n\t addps %%xmm3,%%xmm0 \t#"
      "\n\t addps %%xmm4,%%xmm0 \t#" "\n\t addps %%xmm5,%%xmm0 \t#"
      "\n\t addps %%xmm6,%%xmm0 \t#" "\n\t addps %%xmm7,%%xmm0 \t#"

      :
      : "r" (a+i), // %0
      : "r" (b+i) // %1
    );
}
```

- Comparison with other libraries





# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

- **Open specifications for Multi Processing** *via collaborative work between interested parties from the hardware and software industry, government and academia*

**Acknowledgement:** This lecture is based on “A ‘Hands-on’ Introduction to OpenMP”, Tim Mattson, Intel Corp., [timothy.g.mattson@intel.com](mailto:timothy.g.mattson@intel.com)

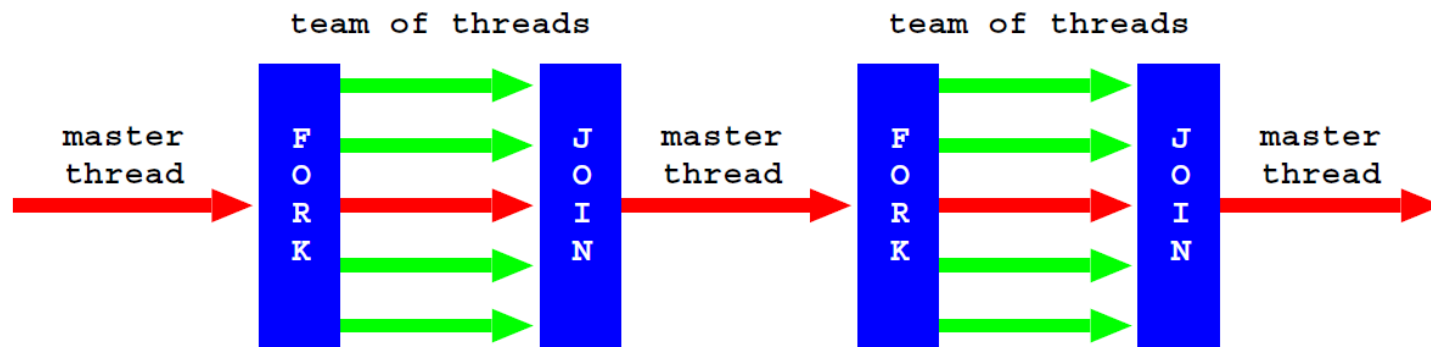
- **Shared Memory Programming Model**
- Cooperation of several hardware and software companies (AMD, Intel, arm, Fujitsu, IBM, HP, NASA, NEC, NVIDIA, Siemens, SUSE, ...)
- Parallel Programming API for multiprocessor / multicore architectures
- Languages: C/C++ or Fortran
- OS: Unix/Linux or Windows
- **OpenMP is a specification not an implementation!**

- Portable programming model for shared memory architectures
- Simple set of programming directives
- Enable incremental parallelism for sequential programs
- Efficient implementations for different problems

- Explicit parallelism
  - Programmer is responsible for annotating execution tasks and synchronization points
  - Embedded compiler directives
- Implicit multithreading
  - Process is a set of threads that communicate through shared variables
  - Creation and termination of threads is performed implicitly by the execution environment
    - Global address space is shared by all threads
    - Variables may be shared or private for each thread
    - Control, management and synchronization of variables involved in parallel tasks is transparent to the programmer

# OpenMP fork-join execution

- Program initiates with a single (master) thread
- Executes sequentially until parallel region is defined by OpenMP constructor, and then:
  - Master thread forks “team of threads”
  - Parallel region code is executed concurrently by all threads (including master thread)
  - There is an implicit barrier at the end of parallel region
  - “team of threads” terminates and master thread continues sequentially



From CP@FCUP

- API includes
  - Compiler directives: `#pragma omp <directive>`
  - Library of functions declared at `omp.h`
  - Environment variables (`OMP_NUM_THREADS; ...`)
- OpenMP code
  - Must include `omp.h` header file  
`#include <omp.h>`
  - To compile with `gcc` use `-fopenmp` option  
`gcc -fopenmp myprog.c ...`

# omp parallel directive

```
#pragma omp parallel [clause, ...]
```

- Defines parallel region
- Number of threads determined by:
  - Only master if `if(expr)` clause is used and `expr` is false
  - Number defined by `num_threads(expr)` clause
  - Number defined by last call to `omp_set_num_threads(expr)`
  - Number defined by environment variable `OMP_NUM_THREADS`
  - Implementation dependent



# Hello.c

```
#include <stdio.h>
#include <omp.h>

int main ()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();

        printf("Hello (%d) ", ID);
        printf("World (%d) \n", ID);
    }
}
```

- **`int omp_get_thread_num(void) ;`**
  - Returns thread **id** from **0** to **n-1**, where **n** is the number of threads
  - Master thread has id **0**
- **`int omp_get_num_threads(void) ;`**
  - Returns the number of active threads
- **`void omp_set_num_threads(int num_threads) ;`**
  - Specifies the maximum number of threads to be used in the next parallel regions
  - Can only be called from sequential code

# Shared and private variables

```
#pragma omp parallel shared(list1) private(list2)
```

- Shared variables
  - Shared by all threads
  - Beware of race conditions!
- Private variables
  - Duplicated in each thread
  - Initial value is undefined
  - Value after parallel region is undefined
- Variables are shared by default
  - Can be changed by using **default(mod)** clause
    - mod: **private**, **shared**, **none**
- Variables declared inside parallel region are private

# Shared and private variables

```
int n = ...;
int *v = (int *) malloc(sizeof(int) * n);
int i, j;
#pragma omp parallel num_threads(n) default(none) \
                    shared(v,n) private(i,j)
{
    int tid = omp_get_thread_num();
    j = 0;
    for (i = 0; i < n * 100000; i++) j += tid;
    v[tid] = j;
}
```

- private variables: `i`, `j` and `tid`
- shared variables: `v` and `n`
- Which is the final value of `v[tid]`?

```
#pragma omp parallel firstprivate(list)
```

- **firstprivate** variables
  - Duplicated in each thread
  - Initial value is copied from value of variable with the same name
  - Can be used to improve efficiency
    - Latency of reading private variable may be lower than latency of reading shared variable by all threads

# Example: Dot product

```
int dot_product(int* u, int* v, int n) {  
    int r = 0;  
    for (int i = 0; i < n; i++) {  
        r += u[i] * v[i];  
    }  
    return r;  
}
```

- Partition iterations / vector among threads
- Reduction of partial results

# Example: Dot product

First version (incorrect!):

```
int dot_product(int* u, int* v, int n) {
    int r = 0;
    #pragma omp parallel
    {
        int rank = omp_get_thread_num();
        int my_n = n / omp_get_num_threads();
        for (int i = rank * my_n; i < (rank+1)*my_n; i++)
            r += u[i] * v[i];
    }
    return r;
}
```

- Manual partition
- Error prone

# Example: Dot product

Better version:

```
int dot_product(int* u, int* v, int n) {
    int r = 0;
    #pragma omp parallel
    {
        #pragma omp for reduction(+:r)
        for (int i = 0; i < n; i++)
            r += u[i] * v[i];
    }
    return r;
}
```

- **#pragma omp for**
  - distributes iterations among threads
- Reduction is automatically performed by using **reduction** clause
- Scheduling may be controlled through **schedule** clause



# Example: Dot product

Even better version:

```
int dot_product(int* u, int* v, int n) {  
    int r = 0;  
    #pragma omp parallel for reduction(+:r)  
    for (int i = 0; i < n; i++) {  
        r += u[i] * v[i];  
    }  
    return r;  
}
```

- **#pragma omp parallel for**
  - creates parallel region and distributes iterations among threads

- OpenMP **reduction** clause:
  - `reduction (op : list)`
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double ave=0.0, A[MAX]; int i;  
#pragma omp parallel for reduction (+:ave)  
for (i=0;i< MAX; i++) {  
    ave + = A[i];  
}  
ave = ave/MAX;
```

# OpenMP Reduction

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only

Operator	Initial value
&	1
	0
^	0
&&	1
	0

# OpenMP Reduction

```
int a = 0, b = 5, c = -2, d = 3;
#pragma omp parallel num_threads(3) \
    reduction(+:a,b) reduction(min:c,d)
{
    a = b = c = d = omp_get_thread_num();
}
```

- Final values a=3, b=8, c=-2, d=0

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

# Parallel Programming Models

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other Models

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.



- Although it might not seem apparent, **these models are NOT specific to a particular type of machine or memory architecture**. In fact, any of these models can (theoretically) be implemented on any underlying hardware.
- **Shared memory model** on a distributed memory machine: **Kendall Square Research (KSR) ALLCACHE** approach.
  - Machine memory was physically distributed, but appeared to the user as a single shared memory (global address space). Generically, this approach is referred to as "virtual shared memory".
  - Note: although KSR is no longer in business, there is no reason to suggest that a similar implementation will not be made available by another vendor in the future.
  - Message passing model on a shared memory machine: MPI on SGI Origin.
- The **SGI Origin** employed the **CC-NUMA** type of shared memory architecture, where every task has direct access to global memory. However, the ability to send and receive messages with MPI, as is commonly done over a network of distributed memory machines, is not only implemented but is very commonly used.

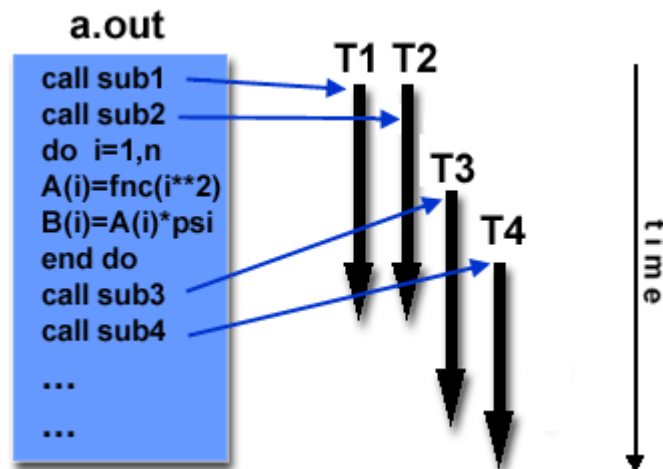
- Which model to use is often a combination of what is available and personal choice.
- **There is no "best" model**, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

- In the **shared-memory programming model**, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No common distributed memory platform implementations currently exist. However, as mentioned previously in the Overview section, the KSR ALLCACHE approach provided a shared memory view of data even though the physical memory of the machine was distributed.

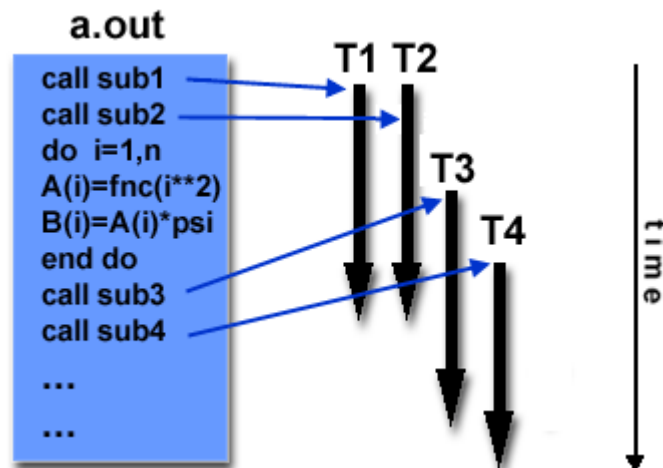
# Threads Model

- In the **threads model** of parallel programming, **a single process can have multiple, concurrent execution paths.**
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of parallel subroutines:
  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run.
  - **a.out** performs some serial work, and then **creates a number of tasks (threads)** that can be scheduled and run by the operating system concurrently.
  - **Each thread has local data**, but also, **shares the entire resources of a.out**. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of **a.out**.

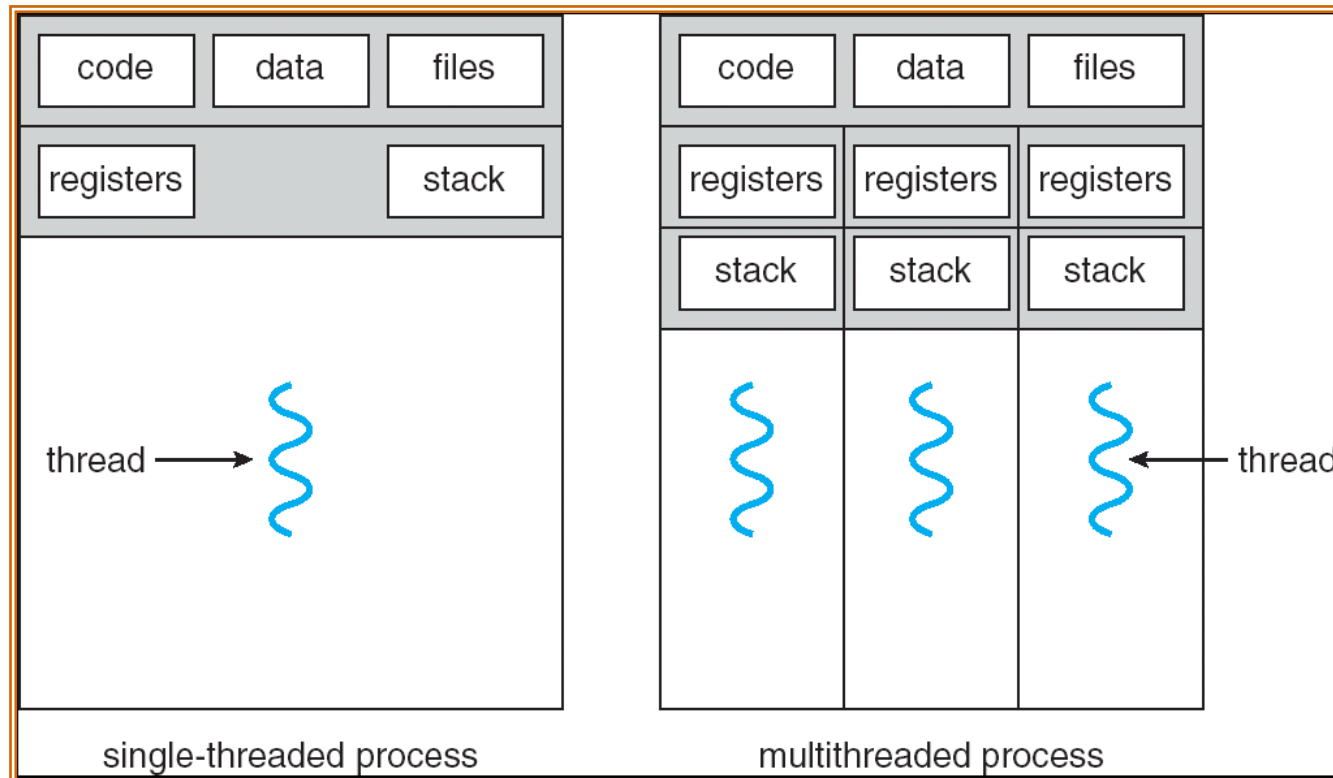


# Threads Model

- A thread's work may best be described as a parallel subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
- **Threads communicate** with each other **through global memory** (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
- Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.
- Threads are commonly associated with shared memory architectures and operating systems.



# Single and Multi threaded Processes

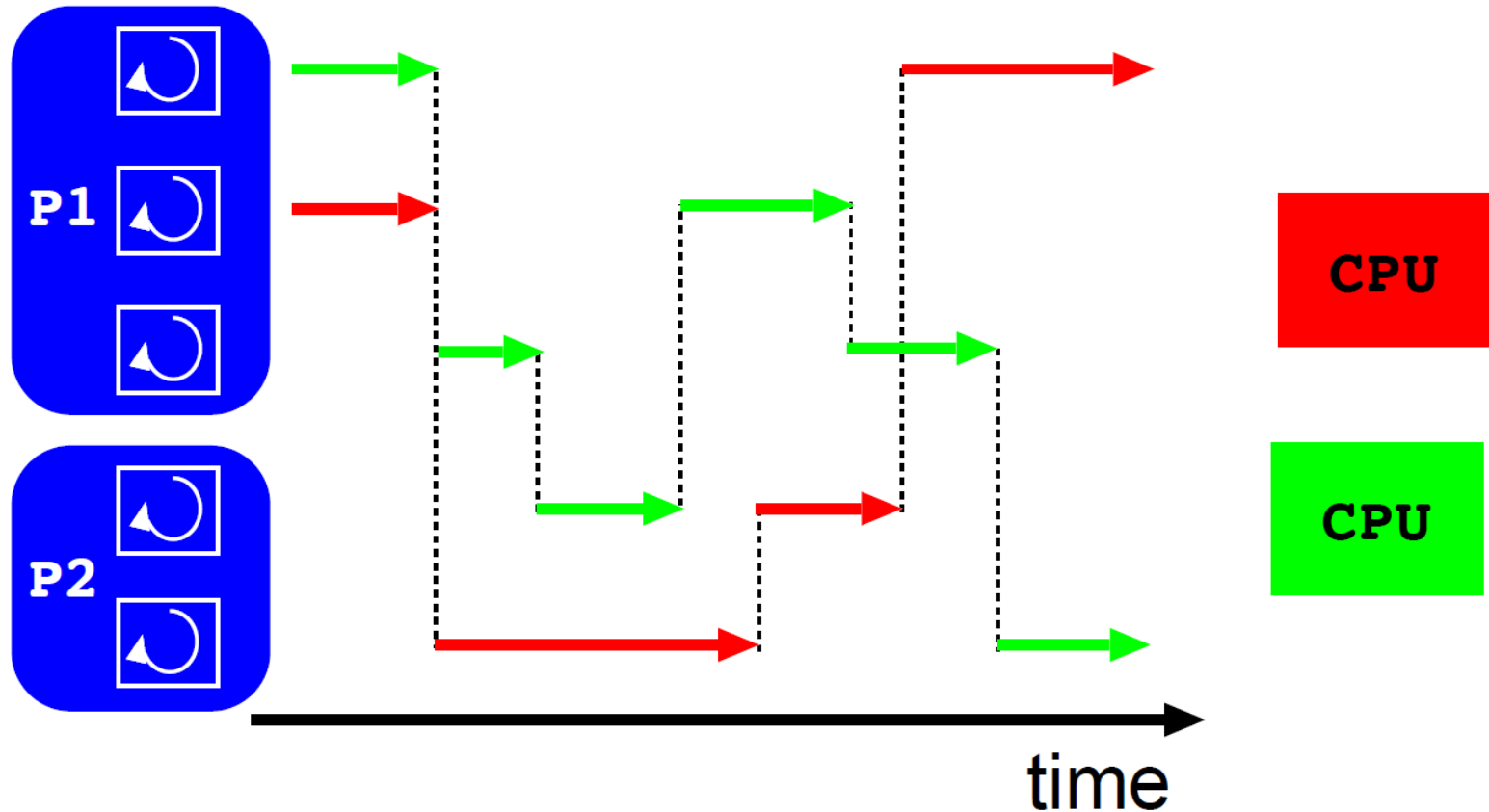


# Processes and Threads

<b>Per-process items</b>	<b>Per-thread items</b>
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



# Processes, Threads and CPUs



from CP@FCUP

# Threads Model Implementations

- From a programming perspective, threads implementations commonly comprise:
  - A **library of subroutines** that are called from within parallel source code
  - A **set of compiler directives** embedded in either serial or parallel source code
- In both cases, the programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

- **POSIX Threads**

- Library based; require parallel coding
- Specified by the IEEE POSIX 1003.1c standard (1995).
- C Language only
- Commonly referred to as Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
- Very explicit parallelism; requires significant programmer attention to detail.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

# Create POSIX Threads

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 5

void *PrintMsg(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! Thread ID, %d\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;

    for( i = 0; i < NUM_THREADS; i++ ) {
        printf( "main() : creating thread, %d\n", i);
        rc = pthread_create(&threads[i], NULL, PrintMsg, (void *)i);

        if (rc) {
            printf("Error: unable to create thread, %d\n", rc);
            exit(1);
        }
    }
    pthread_exit(NULL);
}
```

- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998. Version 2.0 in 2002; version 3.0 in 2008, version 4.0 in 2013; version 5.1 in 2020
  - Portable / multi-platform, including Unix and Windows platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP (see Parallel Programming in .NET).

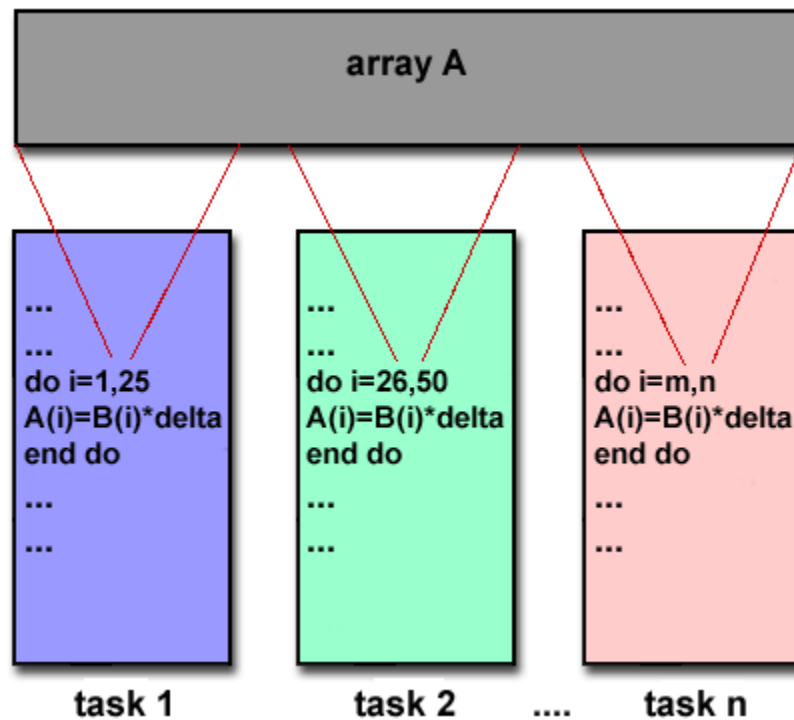
- The **message passing model** demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. MPI-3 was approved in 2012, MPI-4 was approved in June 2021. MPI specifications are available on the web at <https://www.mpi-forum.org/docs/>.



# Message Passing Model Implementations: MPI

- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.



- The **data parallel model** demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing. Data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

- **Single Program Multiple Data (SPMD)**
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data

- **Multiple Program Multiple Data (MPMD)**
- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

- **Open specifications for Multi Processing** *via collaborative work between interested parties from the hardware and software industry, government and academia*

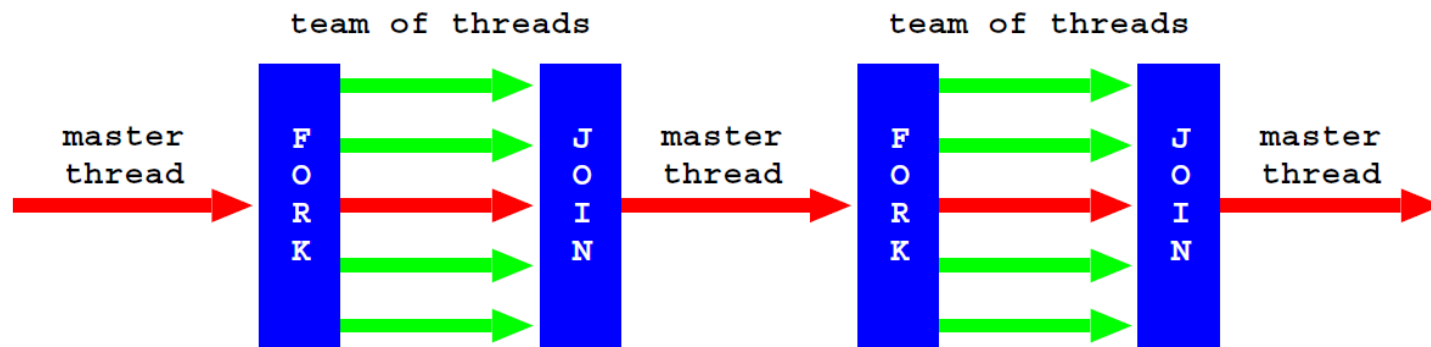
**Acknowledgement:** This lecture is based on “A ‘Hands-on’ Introduction to OpenMP”, Tim Mattson, Intel Corp., [timothy.g.mattson@intel.com](mailto:timothy.g.mattson@intel.com)



- **Shared Memory Programming Model**
- Cooperation of several hardware and software companies (AMD, Intel, arm, Fujitsu, IBM, HP, NASA, NEC, NVIDIA, Siemens, SUSE, ...)
- Parallel Programming API for multiprocessor / multicore architectures
- Languages: C/C++ or Fortran
- OS: Unix/Linux or Windows
- **OpenMP is a specification not an implementation!**

# OpenMP fork-join execution

- Program initiates with a single (master) thread
- Executes sequentially until parallel region defined by OpenMP constructor, and then:
  - Master thread forks “team of threads”
  - Parallel region code is executed concurrently by all threads (including master thread)
  - There is an implicit barrier at the end of parallel region
  - “team of threads” terminates and master thread continues sequentially



From CP@FCUP

- OpenMP has constructs to enable mutual exclusion among threads
- 3 different ways:
  - **omp critical**
    - Defines critical region
    - Can be used for any code
  - **omp atomic**
    - Mutual exclusion for atomic variable updates
    - Only for simple memory updates
      - $x \text{ binop} = \text{expr}$ ,  $x++$ ;  $++x$ ;  $x--$ ;  $--x$
  - Explicit use of locks

- **omp critical**

```
int dot_product(int* u, int* v, int n) {  
    int r = 0;  
    #pragma omp parallel  
    {  
        int private_r = 0;  
        #pragma omp for  
        for (int i = 0; i < n; i++)  
            private_r += u[i] * v[i];  
        #pragma omp critical  
        r += private_r;  
    }  
    return r;  
}
```

- **omp atomic**

```
int dot_product(int* u, int* v, int n) {  
    int r = 0;  
    #pragma omp parallel  
    {  
        int private_r = 0;  
        #pragma omp for  
        for (int i = 0; i < n; i++)  
            private_r += u[i] * v[i];  
        #pragma omp atomic  
        r += private_r;  
    }  
    return r;  
}
```

- Locks

```
int dot_product(int* u, int* v, int n) {  
    int r = 0;  
    omp_lock_t lock;  
    omp_init_lock(&lock); // initialize lock  
    #pragma omp parallel  
    {  
        int private_r = 0;  
        #pragma omp for  
        for (int i = 0; i < n; i++)  
            private_r += u[i] * v[i];  
        omp_set_lock(&lock); // acquire lock  
        r += private_r;  
        omp_unset_lock(&lock); // release lock  
    }  
    omp_destroy_lock(&lock); // destroy lock  
    return r;  
}
```

- **omp master**
  - Executed only by master thread
  - Ignored by others; no synchronization
- **omp single**
  - Executed by a single thread
  - Implicit barrier for all threads
    - Except `nowait` clause is used
- **omp barrier**
  - Explicit barrier

- **omp master**
  - Executed only by master thread
  - Ignored by others; no synchronization
- **omp single**
  - Executed by a single thread
  - Implicit barrier for all threads
    - Except `nowait` clause is used
- **omp barrier**
  - Explicit barrier



# OpenMP `schedule` clause

- **`schedule (static [ , chunk ] )`**
  - Each thread has fixed number of iterations
- **`schedule (dynamic [ , chunk ] )`**
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled
- **`schedule (guided [ , chunk ] )`**
  - Threads dynamically grab blocks of iterations
  - The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
- **`schedule (runtime)`**
  - Schedule and chunk size taken from the `OMP_SCHEDULE` environment variable (or the runtime library).
- **`schedule (auto)`**
  - Schedule is left up to the runtime to choose (does not have to be any of the above).

# nowait ordered lastprivate clauses

- **nowait**
  - threads do not need to synchronize at the end of cycle
- **ordered**
  - Allows #pragma ordered blocks that must execute in the cycle iteration order
- **lastprivate(list)**
  - Variables in list are private
  - Their last value is transported to after the cycle

# nowait ordered lastprivate clauses

```
#pragma omp for nowait
for (int i = 0; i < n; i++) a[i] = x*i;
#pragma omp for
for (int i = 0; i < n; i++) b[i] = i*i;
#pragma omp for
for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
#pragma omp for ordered lastprivate(x)
for (int i = 1; i < n; i++)
#pragma omp ordered
{
    c[i] += c[i-1]; x = c[i];
    printf("c[%d] = %d\n", c[i]);
}
```

- threads may go to 2<sup>nd</sup> cycle even before all threads terminated 1<sup>st</sup> cycle (**nowait**)
- In the last cycle, each thread only executes after previous iterations are concluded (**ordered**)
- **x** gets the value of c[n-1] (**lastprivate**)

# OpenMP `omp sections` directive

```
#pragma omp sections [clause, ...]
{
    #pragma omp section
    { ... }
    ...
    #pragma omp section
    { ... }
}
```

- The `omp sections` directive defines a set of code sections that can perform concurrently, allowing functional parallelism between sections.
- Each section is identified by an `omp section` directive and is performed by just one thread.
- All threads, including those that are not involved in any section, synchronize at the end of the `omp sections` region, unless the `nowait` clause is specified.

# OpenMP `omp sections` directive

```
x = f1 ();  
a = f2 (x) ;  
b = f3 (x, 1) ;  
c = f4 (x, a) ;  
d = f5 (a, b) ;  
e = f6 (d) ;
```

- Assuming that **f1** to **f6** can execute concurrently correct, how to parallelize the code?
- Dependencies allow for the following processing order:
  - x** = **f1** () for just one thread.
  - a** = **f2** (**x**) and **b** = **f3** (**x**, 1) for 2 threads in parallel.
  - c** = **f4** (**x**, **a**) and **d** = **f5** (**a**, **b**) for 2 threads in parallel.
  - e** = **f6** (**d**) for only one thread.

# OpenMP omp sections directive

```
#pragma omp parallel
{
    #pragma omp single
    x = f1();
    #pragma omp sections
    {
        #pragma omp section
        a = f2(x);
        #pragma omp section
        b = f3(x,1);
    }
    #pragma omp sections
    {
        #pragma omp section
        c = f4(x, a);
        #pragma omp section
        d = f5(a,b);
    }
    #pragma omp single
    e = f6(d);
}
```

- A task has
  - Code to execute
  - Data environment (it owns its data)
  - An assigned thread that executes the code and used the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering parallel construct packages up a set of implicit tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and tied to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!



# OpenMP task example

- **omp task** used to process elements of a linked list:

```
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task
            process (p)
            p = next(p) ;
        }
    }
}
```

# Computação Paralela

Mest. Engenharia Computacional  
Mest. Int. Engenharia Computacional

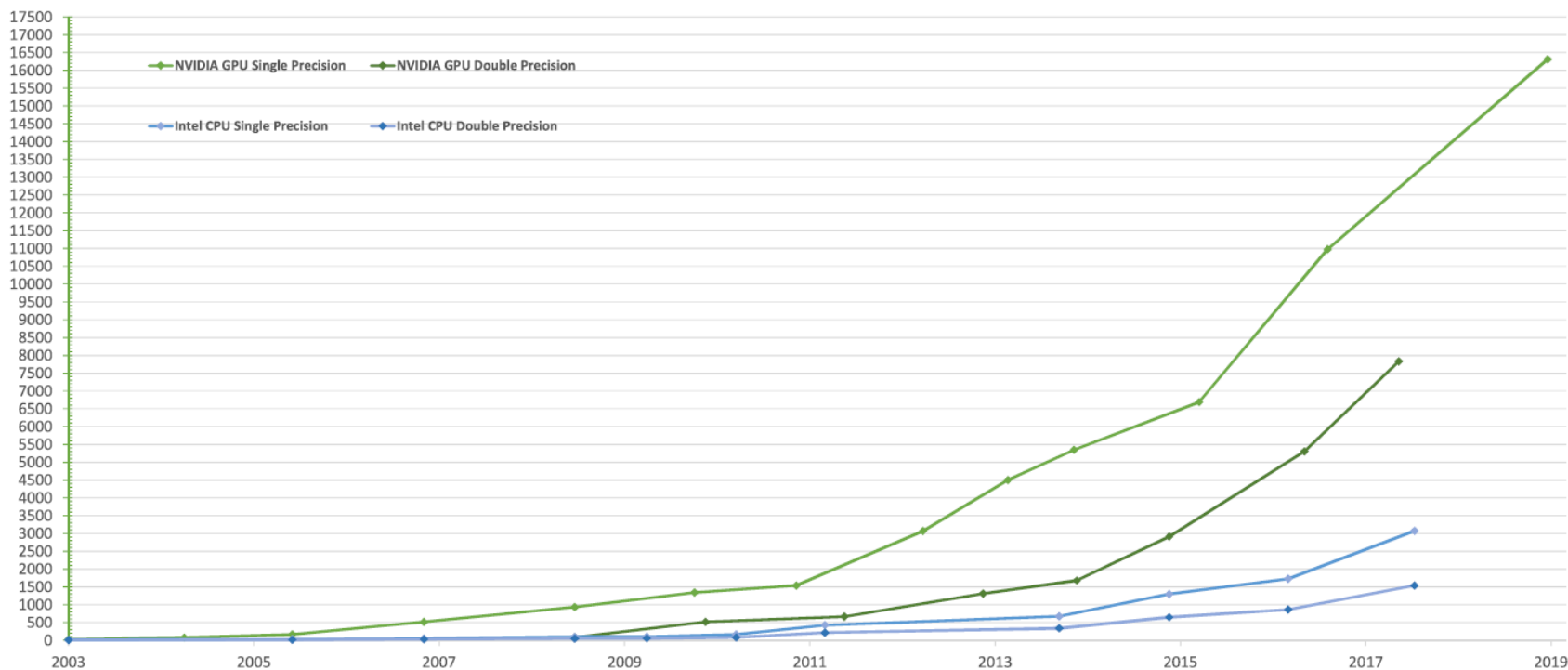
Ano letivo 2022/2023

Rui Costa, Nuno Lau

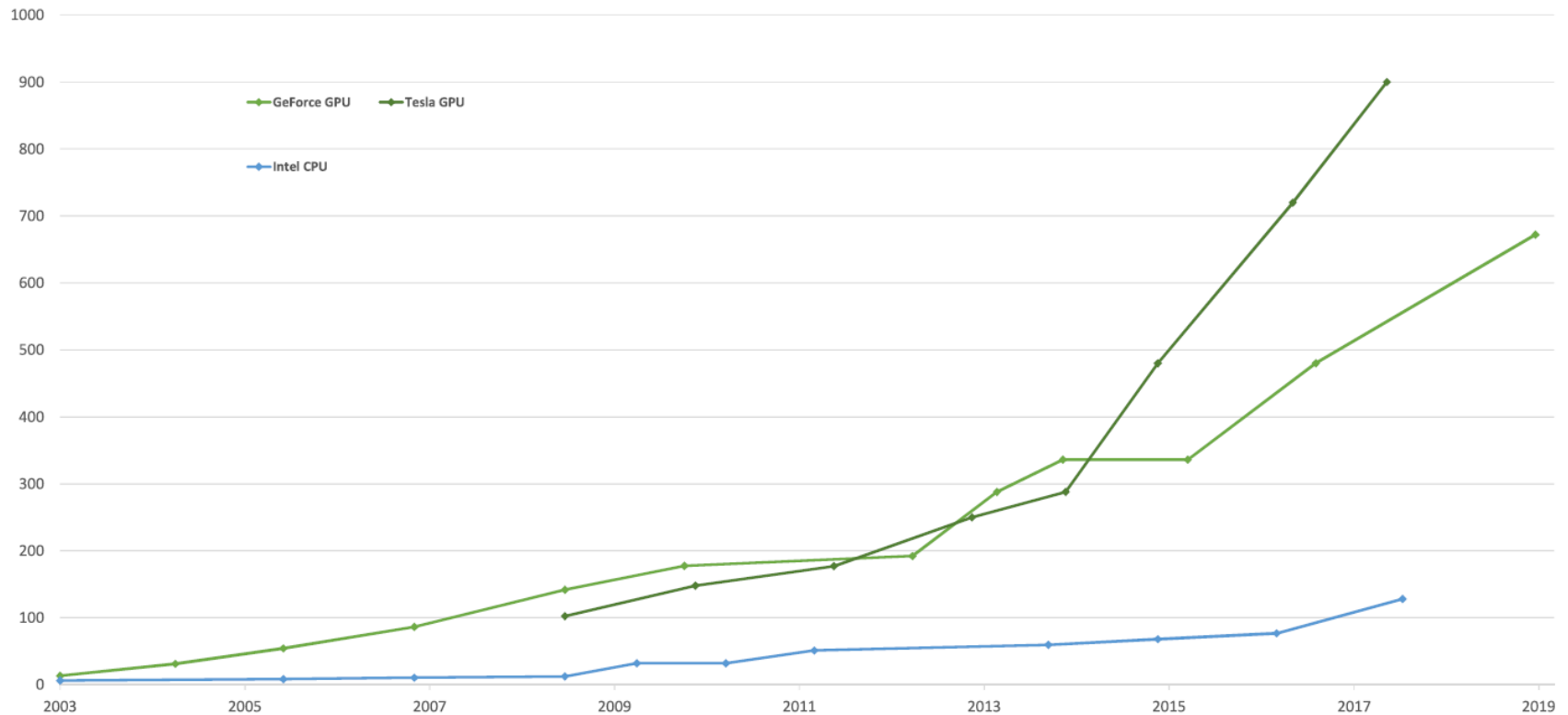
- Parallel general purpose computation model
- Introduced by NVIDIA in 2006
- Allows using the GPU for the execution of general purpose applications

**Acknowledgement:** This lecture is based on “CUDA C++ Programming Guide”, v11.6, NVIDIA and older versions

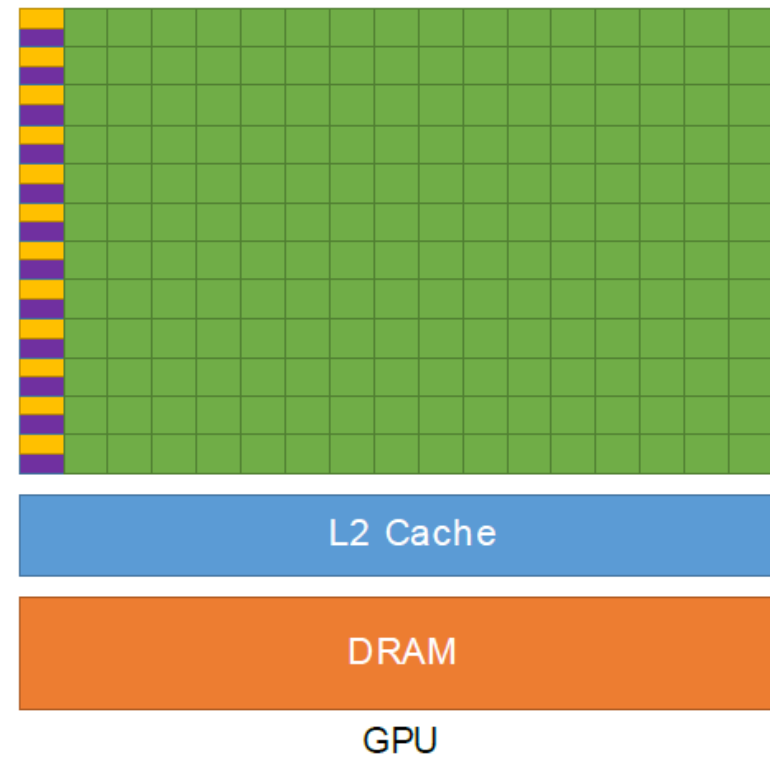
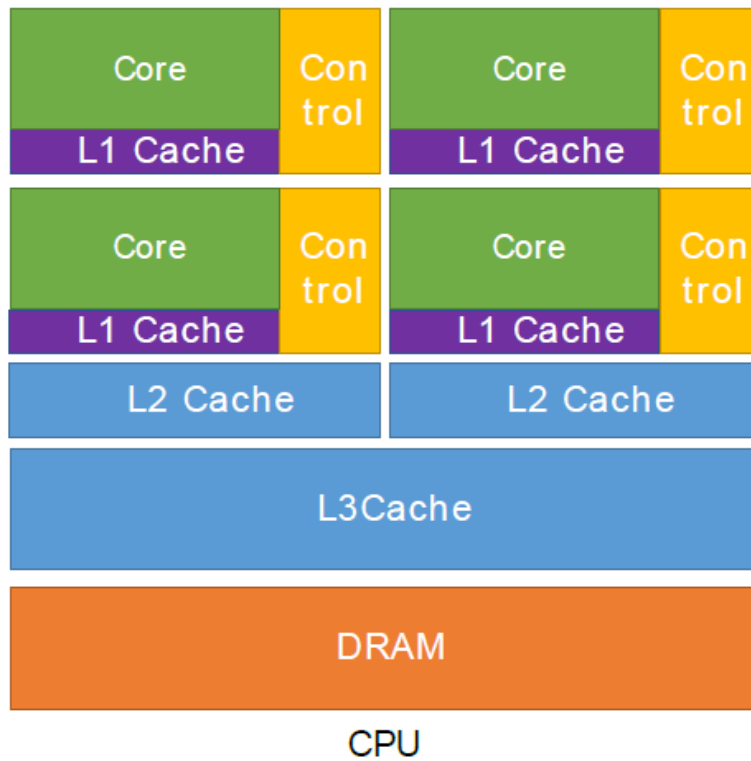
## Theoretical GFLOP/s



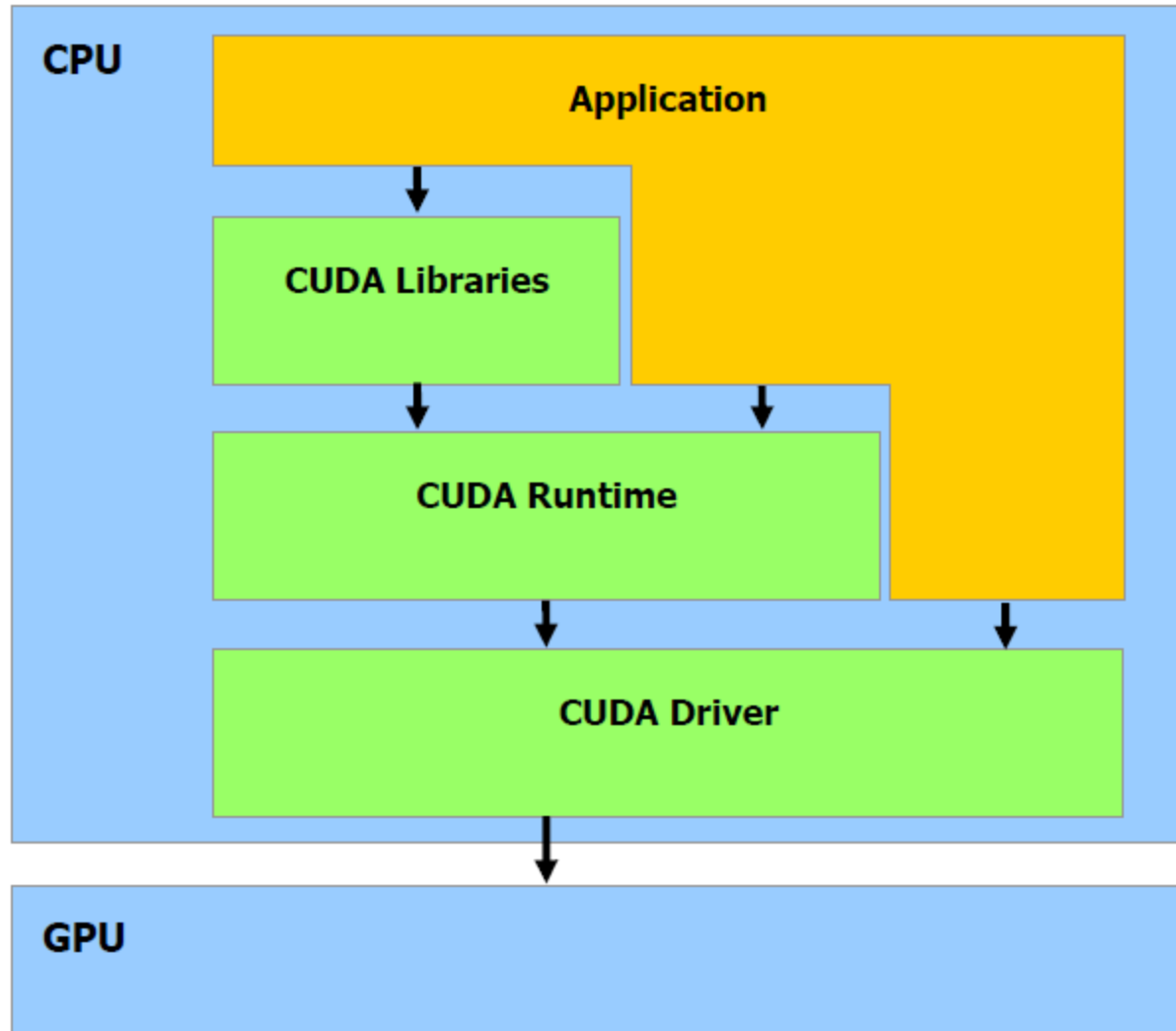
## Theoretical GB/s








# CUDA – CPU and GPU Architecture



# CUDA Architecture

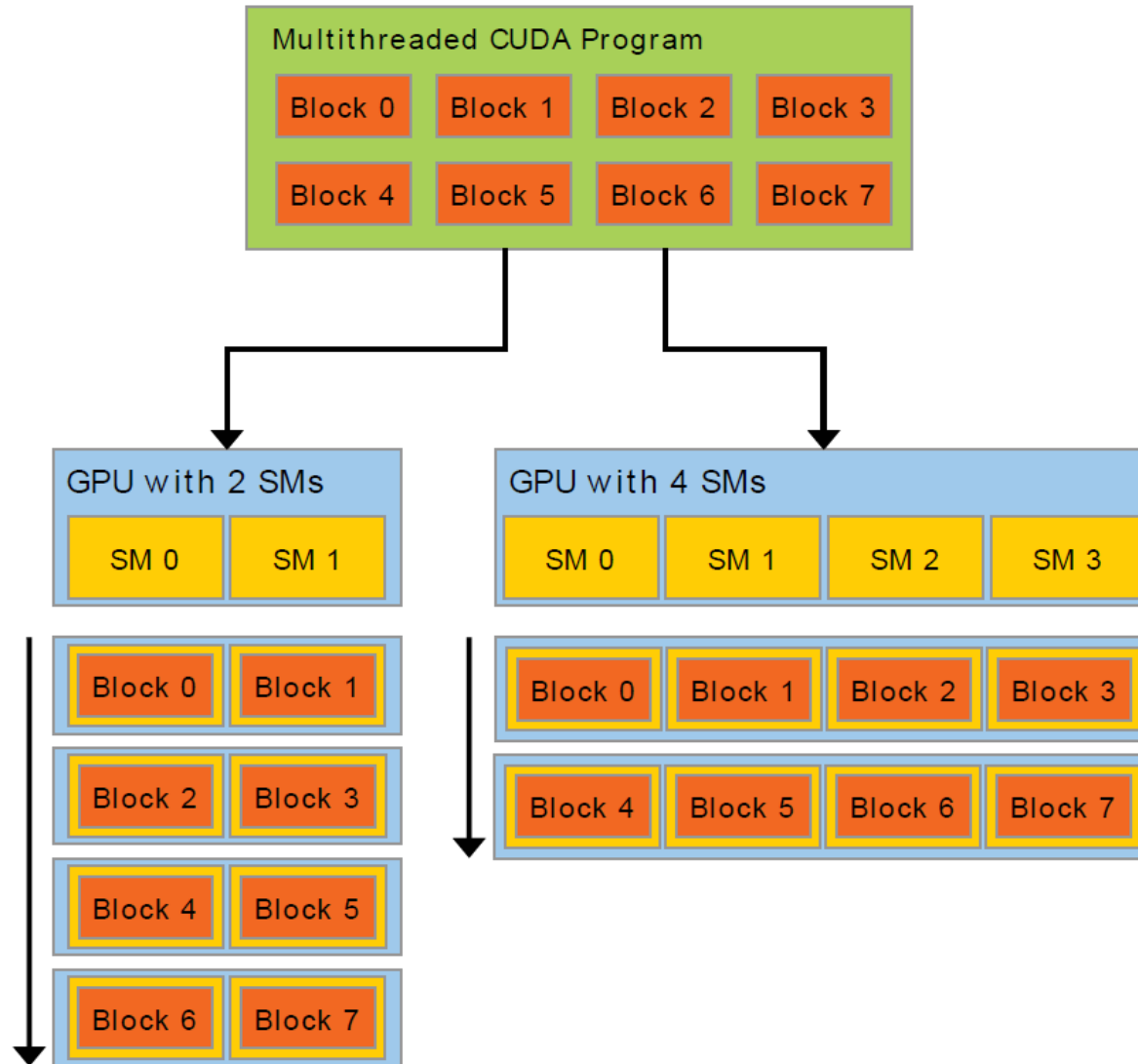


# CUDA Computing Applications

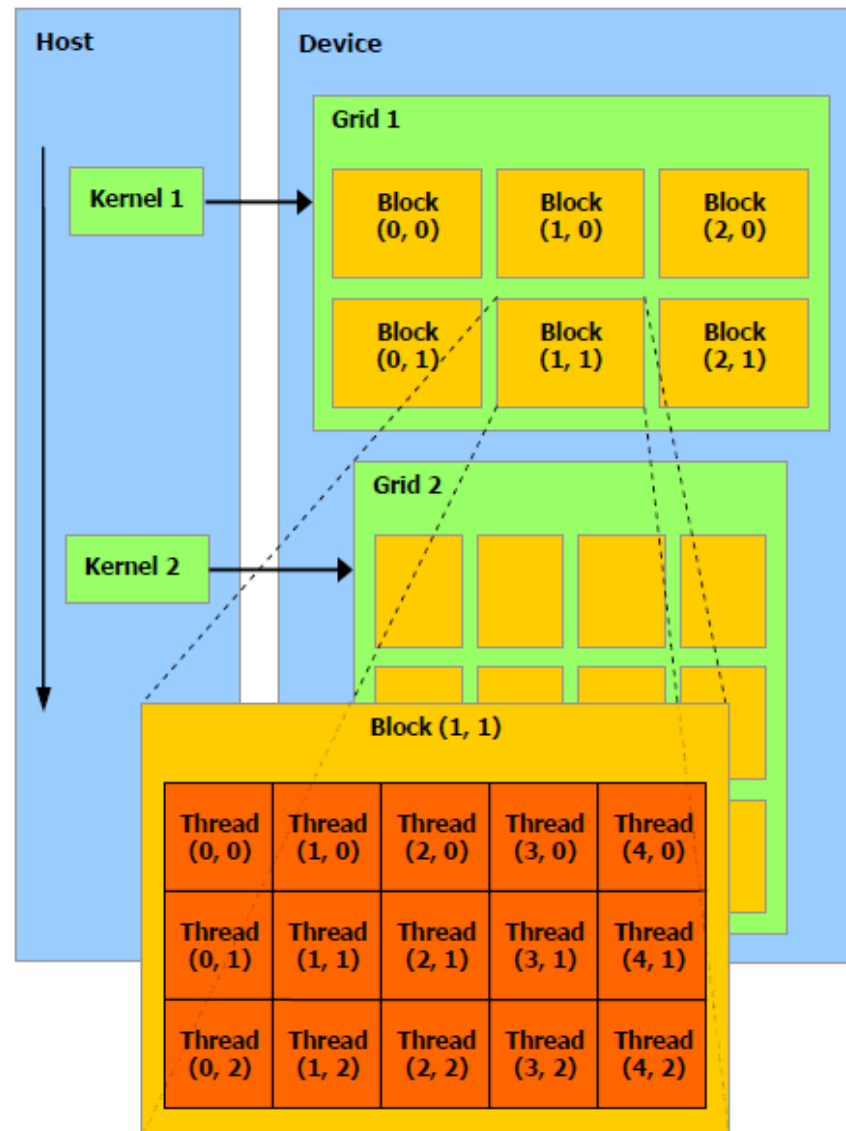
GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
			CUDA-Enabled NVIDIA GPUs			
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series		
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series		
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series		
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		



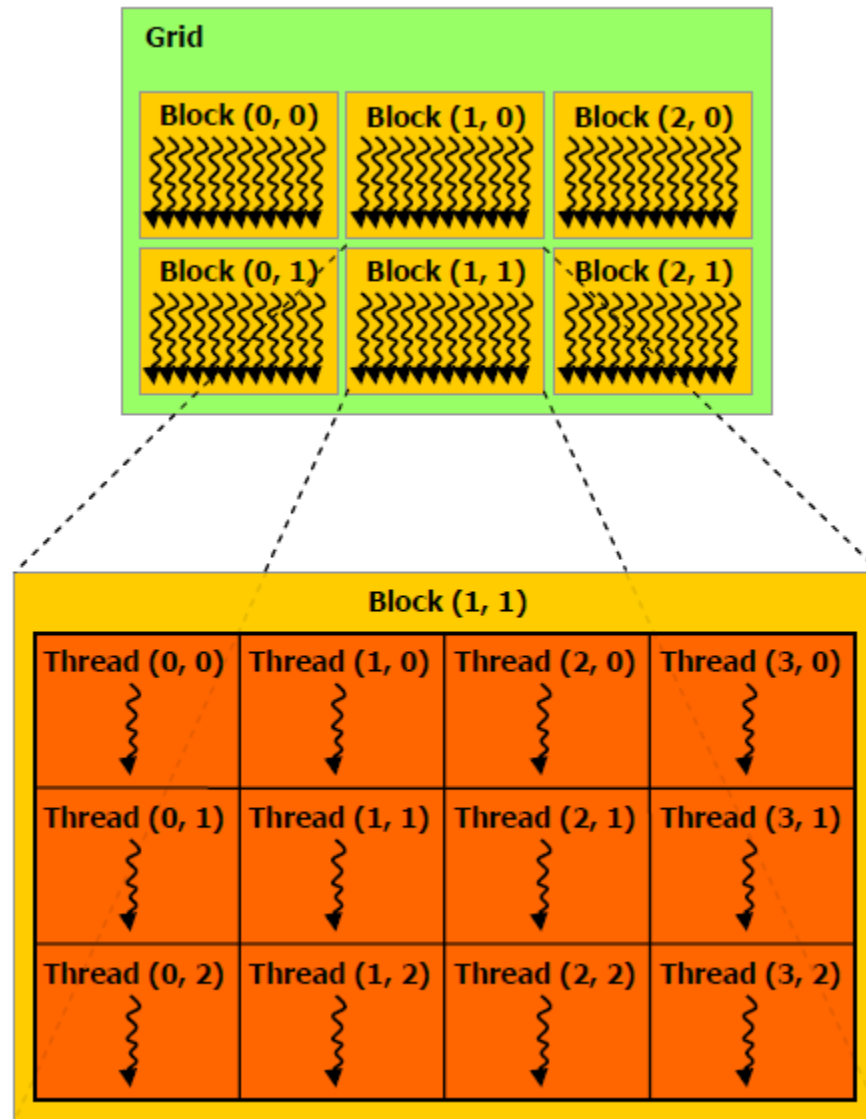
# CUDA Automatic Scalability



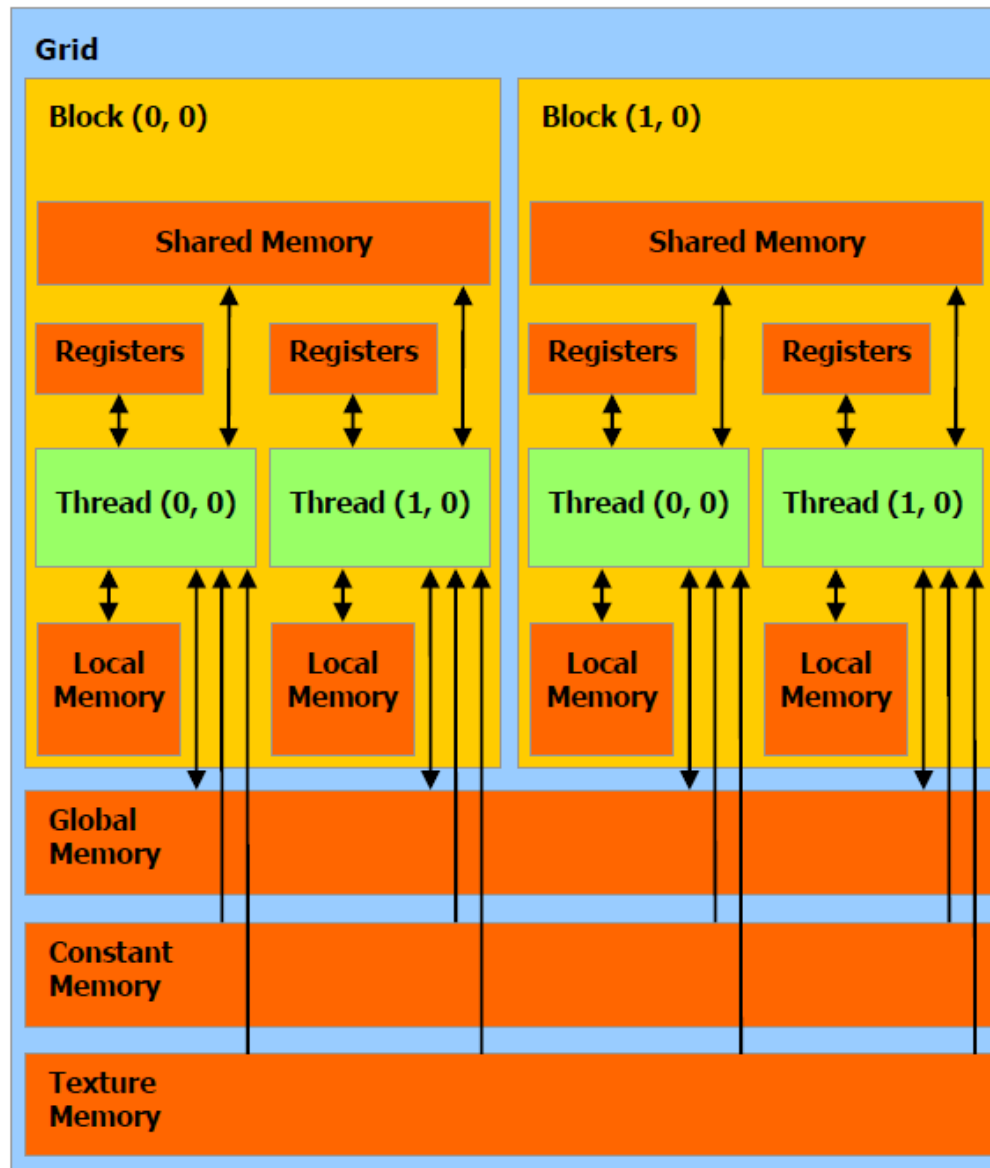
# CUDA Kernel, Grid, Block, Thread



# CUDA Grid of Threads



# CUDA memory



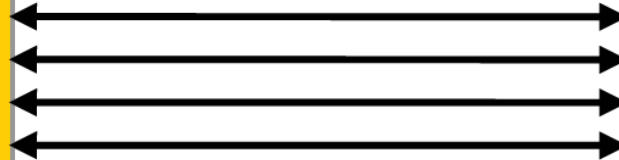
# CUDA memory

Thread



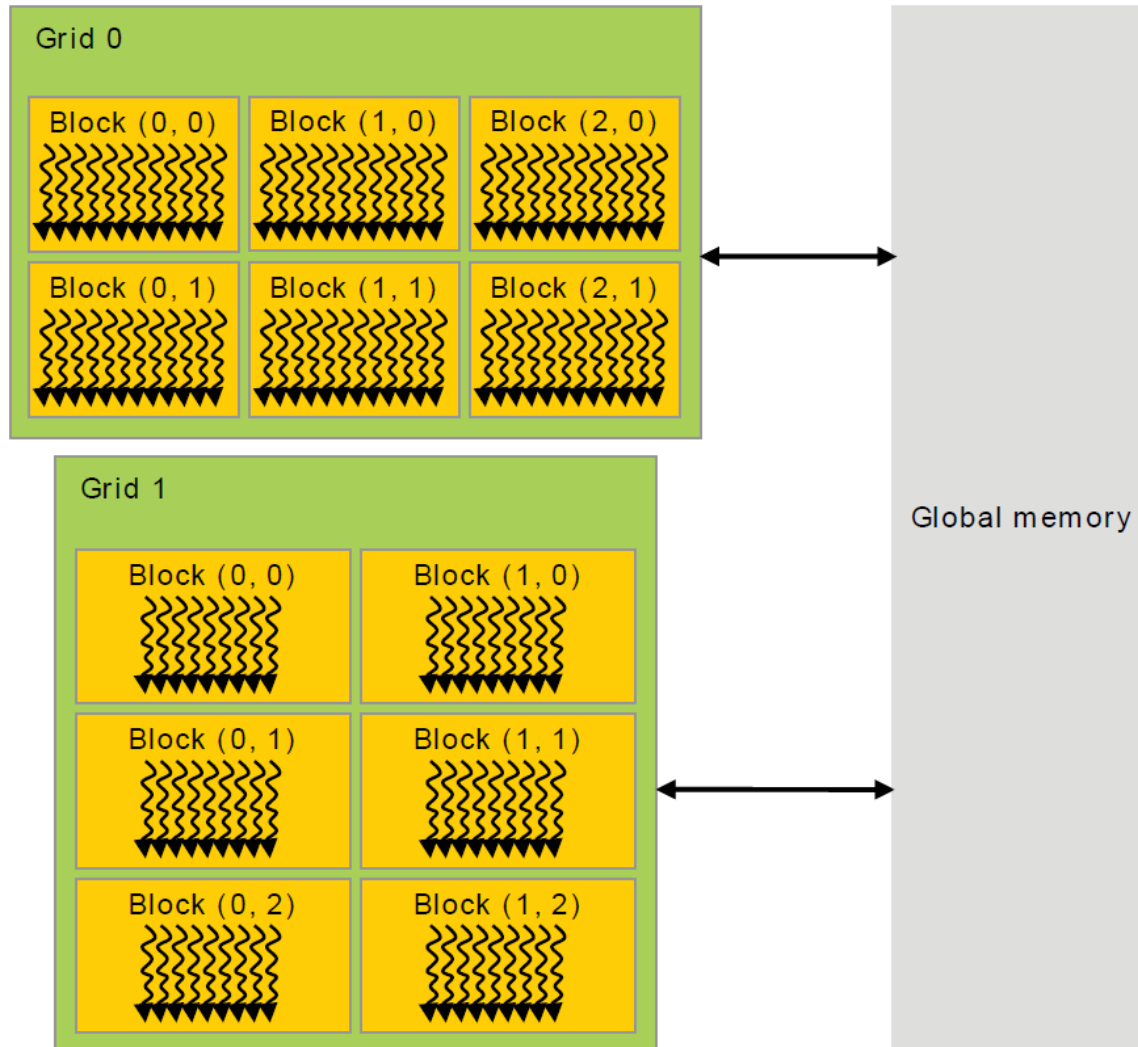
Per-thread local  
memory

Thread Block



Per-block shared  
memory

# CUDA memory



## Compute Capability 1.x

- Global memory (read and write)
  - Slow and uncached
  - Requires sequential and aligned 16 byte reads and writes to be fast
- Texture memory (read only)
  - Cache optimized for 2D spatial access pattern
- Constant memory
  - This is where constants and kernel arguments are stored
  - Slow, but with small cache
- Shared memory (16 kb per MP)
  - Fast, but take care of bank conflicts
  - Can be used to exchange data between threads in a block
- Local memory (used for data that does not fit into registers)
  - Slow and uncached
- Registers
  - Fastest, scope is thread local

## Compute Capability 2.x

- Global memory (read and write)
  - Slow, but now with cache
- Texture memory (read only)
  - Cache optimized for 2D spatial access pattern
- Constant memory
  - Slow, but with cache (8 kb)
- Shared memory
  - Fast, but slightly different rules for bank conflicts now
- Local memory
  - Slow, but now with cache
- Registers (32768 32-bit registers per MP)



## Compute Capability 3.x

- Global memory (read and write)
  - Generally cached in L2, but not in L1
- Constant memory
  - Cache shared by all functional units
- Shared memory
  - 32 banks with two addressing modes
- Local memory
  - Cached in L1

## Compute Capability 5.x

- Global memory
  - Identical to cp 3.x
- Shared memory
  - 32 banks organized such that successive 32-bit words map to successive banks

## Compute Capability 6.x

## Compute Capability 7.x

## Compute Capability 8.x

# CUDA Heterogeneous Programming

C Program  
Sequential  
Execution

Serial code

Parallel kernel  
Kernel0<<<>>>()

Serial code

Parallel kernel  
Kernel1<<<>>>()

Host

Device

Grid 0

Block (0, 0)

Block (1, 0)

Block (2, 0)

Block (0, 1)

Block (1, 1)

Block (2, 1)

Host

Device

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Block (1, 1)

Block (0, 2)

Block (1, 2)

- Kernel invocation

**kernel<<< Dg, Db, Ns, S >>>**

- **Dg** is the grid dimension (dim3 type)
- **Db** is the block dimension (dim3 type)
- **Ns** is the number of shared memory bytes
- **S** is the stream

- Block
  - `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
    - Identify the thread in the block
  - `blockDim.x`, `blockDim.y`, `blockDim.z`
  - `threadID = x+y*Dx+z*Dx*Dy`
  - Threads are executed in *warps*
    - 32 threads of the same block with consecutive ids
    - Blocks should have more than 32 threads

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

- Grid
  - `blockIdx.x, blockIdx.y, blockIdx.z`
    - Identify the block in the grid
  - `gridDim.x, gridDim.y, gridDim.z`

- **\_\_global\_\_**
  - Used to define kernel functions
  - Called by CPU; Run in GPU
- **\_\_device\_\_**
  - Used to define device functions
  - Called by GPU; Run in GPU
- **\_\_shared\_\_**
  - Used to declare variables that reside in shared memory

# CUDA main()

```
int main(void)
{
    float A[SIZE], B[SIZE], C[SIZE];
    float *devPtrA; float *devPtrB; float *devPtrC;
    int memsize = SIZE * sizeof(float);

    // Initialize arrays
    srand (time(NULL));
    for(int i=0; i < SIZE; i++) {
        A[i]=rand() % 100; B[i]=rand() % 100;
    }

    cudaSetDevice(0);    // Select GPU device (can be 0 to 1)

    // Allocate device memory for A, B and C arrays
    cudaMalloc((void**)&devPtrA, memsize);
    cudaMalloc((void**)&devPtrB, memsize);
    cudaMalloc((void**)&devPtrC, memsize);

    // Copy data (data to process) from host to device (from CPU to GPU)
    cudaMemcpy(devPtrA, A, memsize, cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, memsize, cudaMemcpyHostToDevice);

    // Execute the Kernel
    arrAdd <<<1, SIZE>>> (devPtrA, devPtrB, devPtrC); // launch 1 block with SIZE threads

    // Copy data from device (results) back to host
    cudaMemcpy(C, devPtrC, memsize, cudaMemcpyDeviceToHost);

    // Show results
    printf("      A      B      C\n");
    for (int i=0; i < SIZE; i++) {
        printf("%2d: %4.1f + %4.1f = %5.1f\n", i, A[i], B[i], C[i]);
    }

    // Free device memory
    cudaFree(devPtrA); cudaFree(devPtrB); cudaFree(devPtrC);
}
```



```
// Kernel definition, see also section 2.1 of NVIDIA CUDA Programming Guide
__global__ void arrAdd(float *A, float *B, float *C)
{
    // threadIdx.x is a built-in variable provided by CUDA at runtime
    // It represents the thread index inside the block

    int id = threadIdx.x; // id: unique thread identifier

    C[id] = A[id] + B[id];
}
```