Introduction to shared memory application programming

Credits to
(Victor Alessandrini
Paulo Santos)

- **Multithreaded** programming is today a core technology, at the basis of any software development project in any branch of applied computer science
- Today, the only way to increase performance is to increase the number of processors, and for this reason multithreading
- Which was initially a question of choice and opportunity, has become a mandatory software technology.
- Threads are definitely a part of the life of any software developer concerned
  - with issues of efficient network connectivity and I/O, interoperability, computational performance and scalability,
  - or efficient interactive access to an application.

- An overview of different libraries and programming environments operating on Unix-Linux or Windows platforms
  - underlining their often complementary capabilities, as well as
  - the fact that there is sometimes benefit in taking advantage of their interoperability.
- The first chapters of the book develop an overview of the native multithreading libraries like Pthreads (for Unix-Linux systems) or Windows threads, as well as the recent C++11 threads standard,
  - with the purpose of grasping the fundamental issues on thread management and synchronization discussed in the following chapters.
  - a quick introduction to the most basic OpenMP thread management interfaces is also introduced.
- In the second part of the book, a detailed discussion of the two high-level, standard programming environments
  - OpenMP and Intel Threading Building Blocks (TBB) is given.

- Reviews the **basic concepts and features** of current computing platforms needed to set the stage for concurrent and parallel programming.
  - Particular attention is paid to the impact of the **multicore evolution** of computing architectures, as well as other basic issues having an impact on efficient software development,
  - such as the hierarchical memory organization, or the increased impact of rapidly evolving heterogeneous architectures incorporating specialized computing platforms like Intel Xeon Phi and GPU accelerators.

- Introduces a number of **basic concepts and facts** on multithreading processing that are totally generic and independent of the different programming environments.
  - They summarize the very general features that determine the way multithreading is implemented in existing computing platforms.

- A broad view of the utilities and the strategic options proposed by the different multithreading libraries and environments seems to be an important asset for software developers.
  - The programming interfaces proposed by **Pthreads**, and **C++11** to create and run a team of worker threads
  - A first look at **OpenMP** is taken to show how the same thread management features are declined in this environment.
  - The presentation adopts the traditional ***thread-centric*** *programming style* in which the activity of each thread in the application is perfectly identified.

- Multithreading is today a **mandatory software** technology for taking full advantage of the capabilities of modern computing platforms of any kind

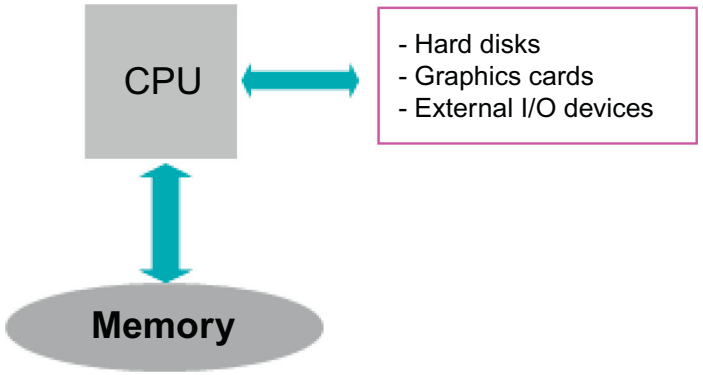
- All the data processing devices available in our environment have a master piece of software — the **operating system** (OS)
  - that controls and manages the device, running different applications at the user's demands.
  - A standalone application corresponds to a *process* run by the operating system.
- A **process** can be seen as a black box
  - owns a number of protected hardware and software resources; namely,
    - a block of memory / files to read and write data,
    - network connections to communicate with other processes or devices, and
    - code in the form of a list of instructions to be executed

- Process resources are protected in the sense that no other process running on the same platform can access and use them.
  - A process has also access to one or several central processing units (cores)
  - providing the computing cycles needed to execute the application code.

- In computing platforms,
  - we speak of concurrency when several different activities can advance and make progress at the same time
    - rather than one after the other.
  - this is possible because CPU cycles are not necessarily exclusively owned by one individual process.
  - OS can allocate CPU cycles to several different processes at the same time (a very common situation in general).
    - It allocates, successive time slices on a given CPU to different processes
    - thereby providing the illusion that the processes are running simultaneously.
- This is called **multitasking**.
  - In this case, processes are not really running simultaneously but,
  - at a human time scale,
    - they advance together,
    - which is why it is appropriate to speak about concurrent execution.

- One special case of concurrency happens
  - when there are sufficient CPU resources so that they do not need to be shared:
    - each activity has exclusive ownership of the CPU resource it needs.
- In this optimal case,
  - we may speak of ***parallel*** *execution*.
    - But remember that
    - parallel execution is just a special case of concurrency.

- A computing platform
  - consists essentially of one or more CPUs and a memory block where
    - the application data and code are stored.
  - There are also peripheral devices to communicate with the external world,
    - like DVD drives, hard disks, graphic cards, or network interfaces.



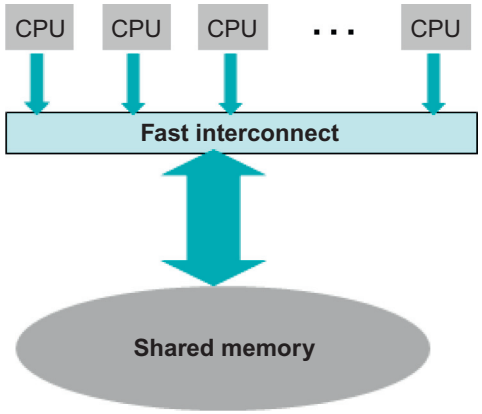
- *Peak performance*,
  - directly related to clock frequency
  - in an ideal context in which processor cycles are not wasted.
- The original Intel 8086 processor
  - in the late 70's ran at 5 MHz
  - today, processors run at about 3 GHz
    - 600 × increase in frequency
- Memory performance is a completely different issue.

$$T = L + N/B$$

- The time **T** needed to move to or from memory a data block of **N** bytes
  - where L is the **latency**,
    - the access time needed to trigger the transfer,
    - an intrinsic property of the memory device
  - B is the **bandwidth**
    - the speed at which data is moved once the transfer has started.
  - in the case we are considering, moving data from memory to the processor or vice versa:
  - a property of the network connecting the two devices. controlled by network technologies.

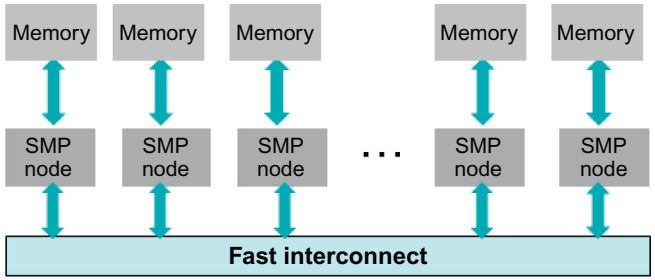
- In the 70's - i8086
  - **memory latencies** were close to processor clock
  - It was possible to feed the processor at a rate guaranteeing execution of one instruction per clock cycle
- But memory latencies have not decreased as dramatically as processor frequencies have increased.
- Today, it takes a few hundreds of clock cycles to trigger a memory access.
  - Therefore, in ordinary memory bound applications the processors tend to **starve**,
  - and typically their *sustained performances*
  - are a small fraction (about 10%) of the theoretical peak performance
    - promised by the processor frequency.

- *Symmetric Multiprocessor* (SMP) systems,
  - a network interconnect links a number of CPUs to a common, shared memory block.
  - memory is shared, and all CPUs can access the whole common memory address space.
- **coherency** protocols prevents SMP from scaling to very large numbers of CPUs



- SMP platforms are not perfectly symmetric.
  - a huge, logically shared memory block is normally constructed by using, say, *M* identical memory devices.
  - according to the topology and the quality of the network, a given CPU may not be at the same “effective distance”
  - and the access times may be non-uniform with mild differences in memory access performance.
  - this kind of computing platform is called a NUMA (Non-Uniform Memory Access) architecture.
- Performance may be optimized by placing data items as close as possible to the CPU running the thread that is accessing them.
  - placing threads as near as possible to the data they access—is called ***memory affinity***.
  - It plays a significant role in multithreaded programming.

- Moving beyond SMPs to large massive parallel computers implies:
  - to abandon the shared memory architecture and
  - move to a ***distributed*** *memory* systems,
  - essentially clusters of SMPs linked by yet another
    - higher level network interconnect.
    - each SMP has its own private memory address space.



- Large parallel applications of independent processes
  - running on different SMPs and communicating with one another
    - via an **explicit communication** protocol integrated in a programming model,
- *Message Passing Interface* (**MPI**).
  - Each process acts on its own private data set, and
  - a number of synchronization and data transfer primitives allow them
    - to **communicate** to other processes the updated data values they need to proceed
- Programming with MPI is more **difficult** than programming ordinary SMP
  - programmer lacks a global view of the application's data set
    - data is spread across the different SMP nodes, and
    - each autonomous process accesses only a section of the complete data set.

- Since 70's the sustained evolution of silicon technologies has followed a trend under the name of **Moore's law**
  - *the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years*
- Dennard and colleagues scaling rules [IBM 1974]
  - reducing by a half the transistor size multiplies by 4 the number of transistors in a given area *at constant power dissipation*, and
  - in addition each transistor is operating twice as fast.

- For more than three decades we enjoyed the benefits of automatic performance enhancements:
  - processor working frequencies (and, therefore, their peak performance) doubled every 18 months.
  - it was sometimes sufficient to wait for the next generation of CPU technology to benefit from an increased applications performance.
- Parallel processing was mainly an issue of choice and opportunity,
  - mostly limited to supercomputing or mission critical applications.

- This golden age came to an end about 2004,
  - when a power dissipation wall was hit.
  - as transistors reach smaller sizes,
  - new previously negligible physical effects emerge
    - mainly quantum effects - that invalidate Dennard's scaling.
  - Working voltages essentially stopped shrinking at this time
    - and today clock speeds are stuck because it is no longer possible to make them work faster.
- Moore's law still holds:
  - it is always possible to double the transistor count about every 2 years,
  - but shrinking the transistor size does not make them better, as was the case when Dennard scaling applied.

- Last decade has witnessed the multicore evolution.
  - Processor chips are no longer identified with a single CPU,
    - as they contain now multiple processing units called cores.
  - Since clock speeds can no longer be increased,
    - the only road for enhanced performance is the **cooperative operation** of multiple processing units,
    - at the price of making parallel processing mandatory.
- Virtually all processor vendors have adopted this technology,
  - and even the simplest laptop is today a SMP platform.
  - multicore processor chips are now called ***sockets***.
- The present situation can be summarized as follows:
  - the number of **cores per socket** doubles every 2 years.
  - clock speeds tend to mildly decrease.
    - In any case, they are stabilized around a couple of GHz.

- *Instruction Level Parallelism* (ILP)

  Os processadores incluem mecanismos para **executar várias instruções do mesmo programa** em paralelo

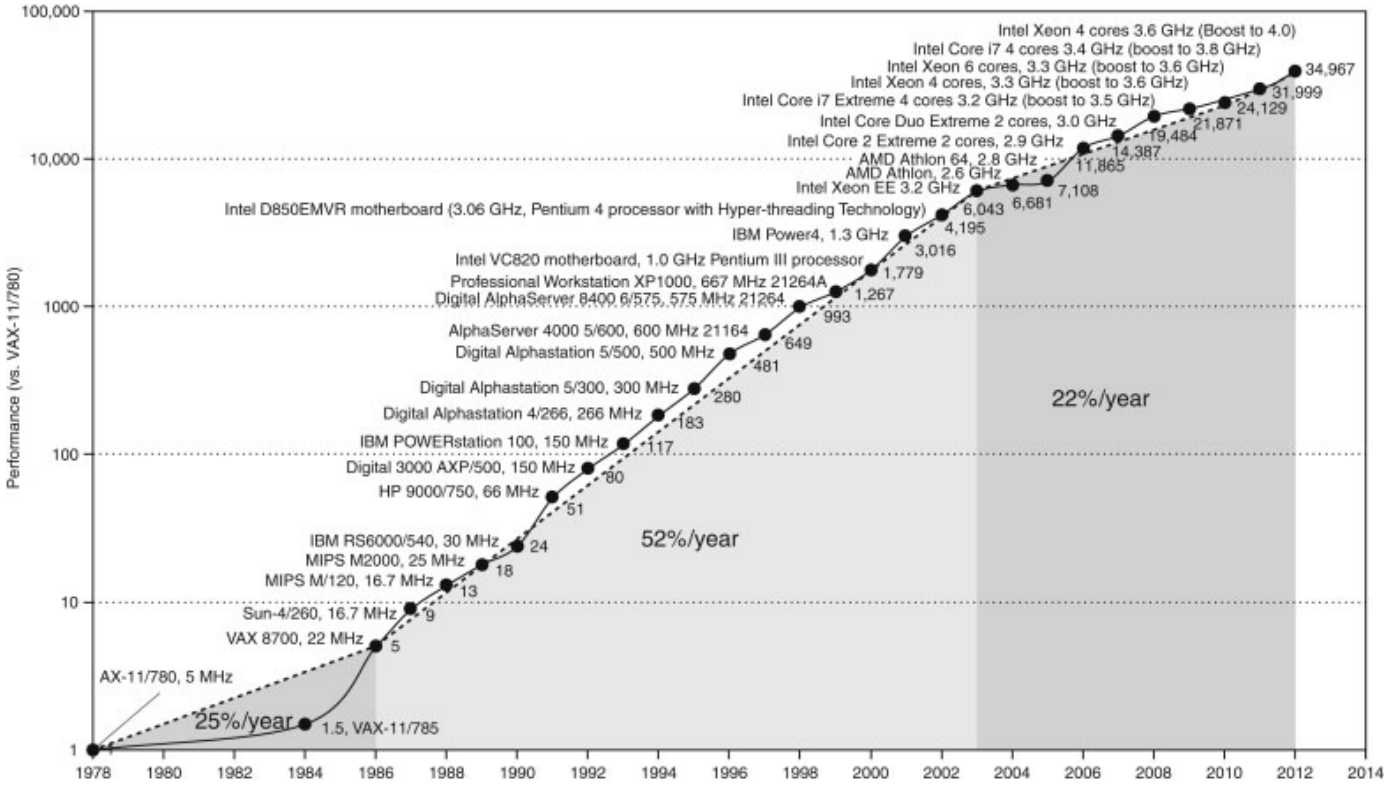
  – ***Encadeamento*** (até 32 estágios)
  profundidade óptima (potência vs. desempenho) **~ 7 estágios**

  – **Superescalaridade**
  Múltiplas *unidades de encadeamento* permitem a execução simultânea de múltiplas instruções

  – **Execução fora de ordem**
  Maximização do número de instruções em execução

  – **Execução especulativa**
  Resolução de dependências de controlo

Evolução: frequência

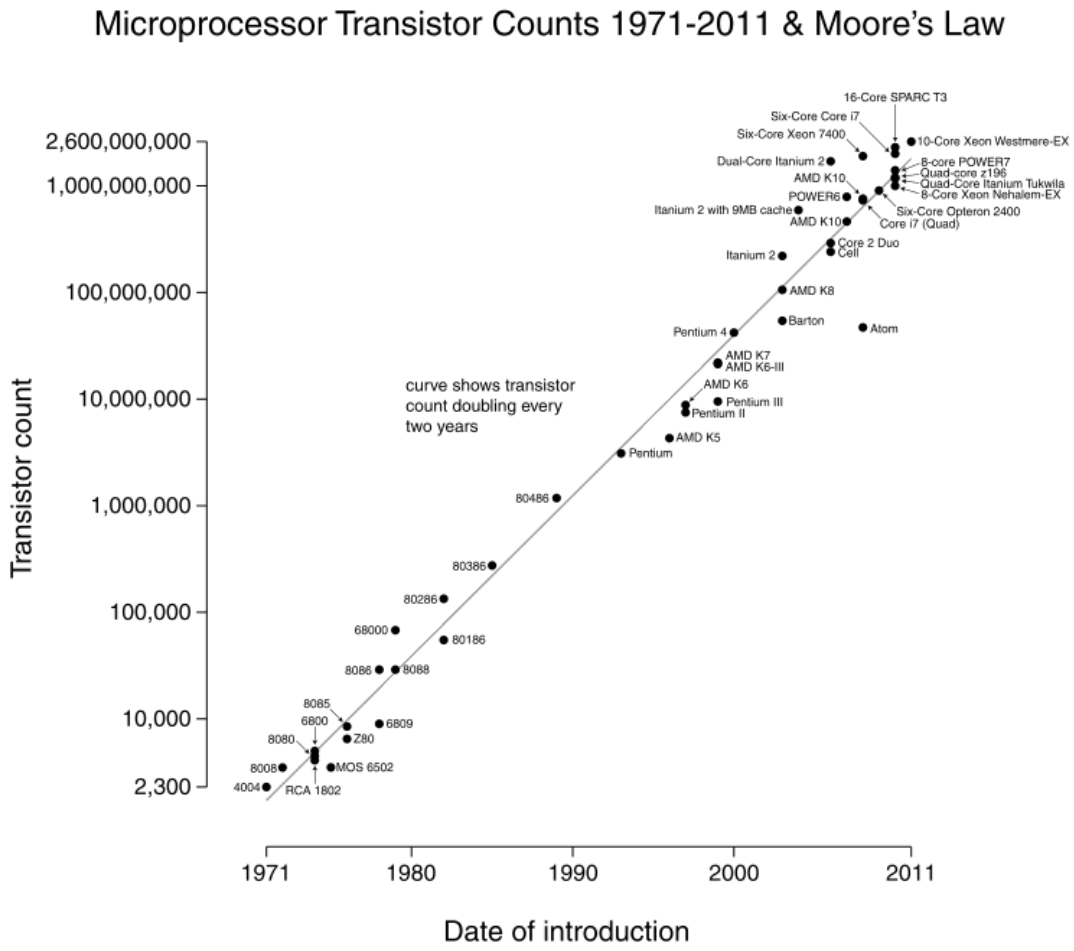


[Computer Organization Design: The Hardware / Software Interface ; Patterson & Henessey]

FIGURE 1.17 Growth in processor performance since the **mid-1980**s. This chart plots performance relative to the VAX 11/780 as measured by the **SPECint** benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about **25**% per year. The increase in growth to about **52**% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven higher in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about **22**% per year.

- Lei de Moore:
  "The number of transistors on a chip doubles every two years"
  [Adaptado de G. Moore, 1965]

  Também formulada como:
  "O desempenho dos processadores duplica cada 18 meses"
  [David House, Intel]



O aumento da **frequência** do relógio faz crescer a **potência** dissipada

$$P = CV^2 f$$

P – potência dinâmica
C – capacidade
V - tensão
f - frequência

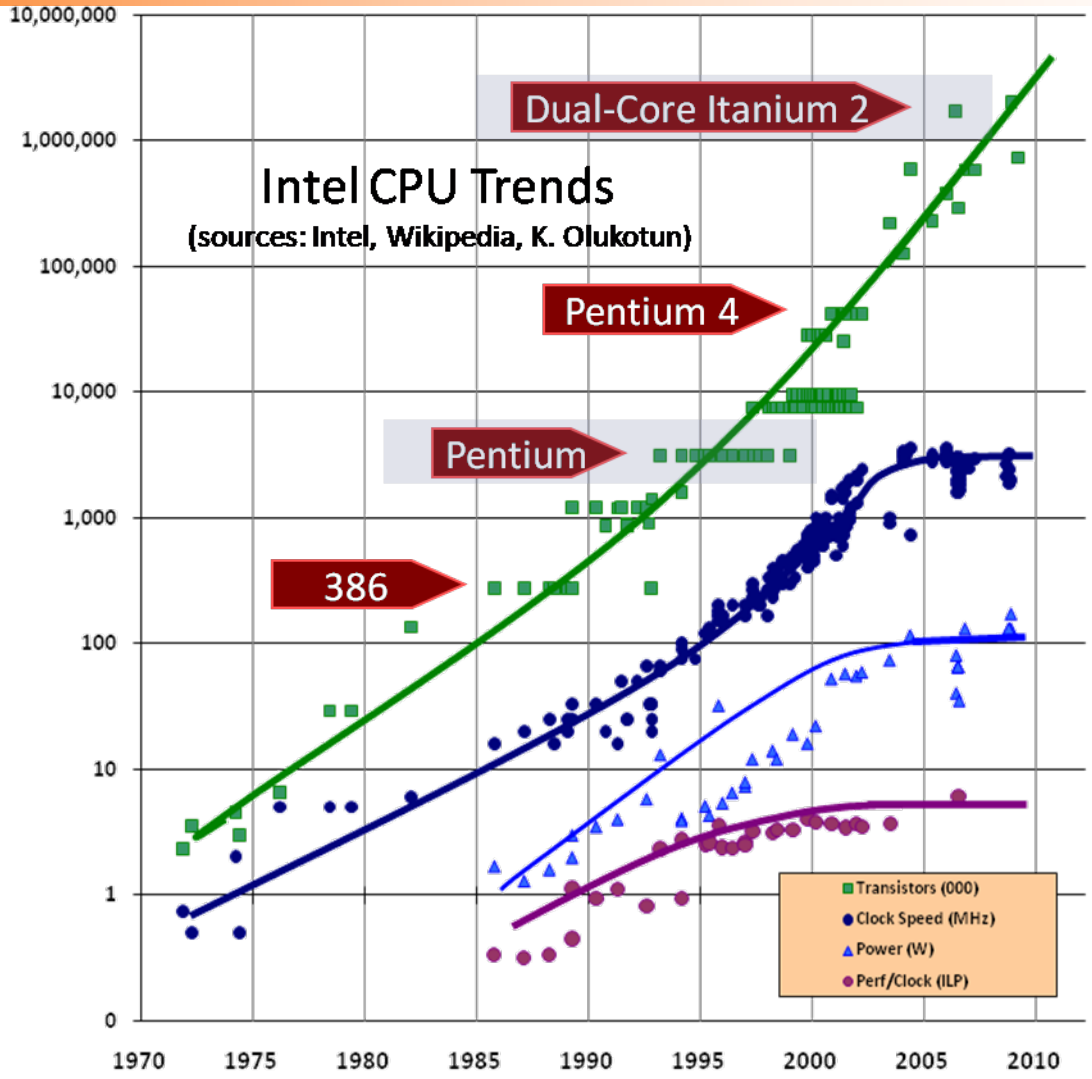
É necessário *hardware* adicional para explorar/controlar o ILP

- *o encadeamento* profundo /execução fora de ordem
  - é muito complexo,
    - elevado consumo de energia associado
  - desenho lógico cuja correcção é difícil de verificar

O grau de ILP disponível nos programas é limitado

- sendo o caso comum cerca de **3 instruções** por ciclo de relógio

Barreira de Potência



A Intel, em 2004 anuncia o processador Prescott, com um só núcleo (***single core***)

- com encadeamentos muito profundos e
- frequência até aos 10 GHz

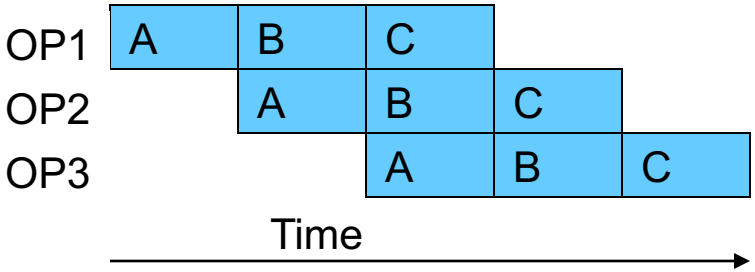
No entanto, o consumo de potência e o calor dissipado

- nunca permitiram frequências além dos **3.8** GHz

Em 2005 surge o Prescott 2M

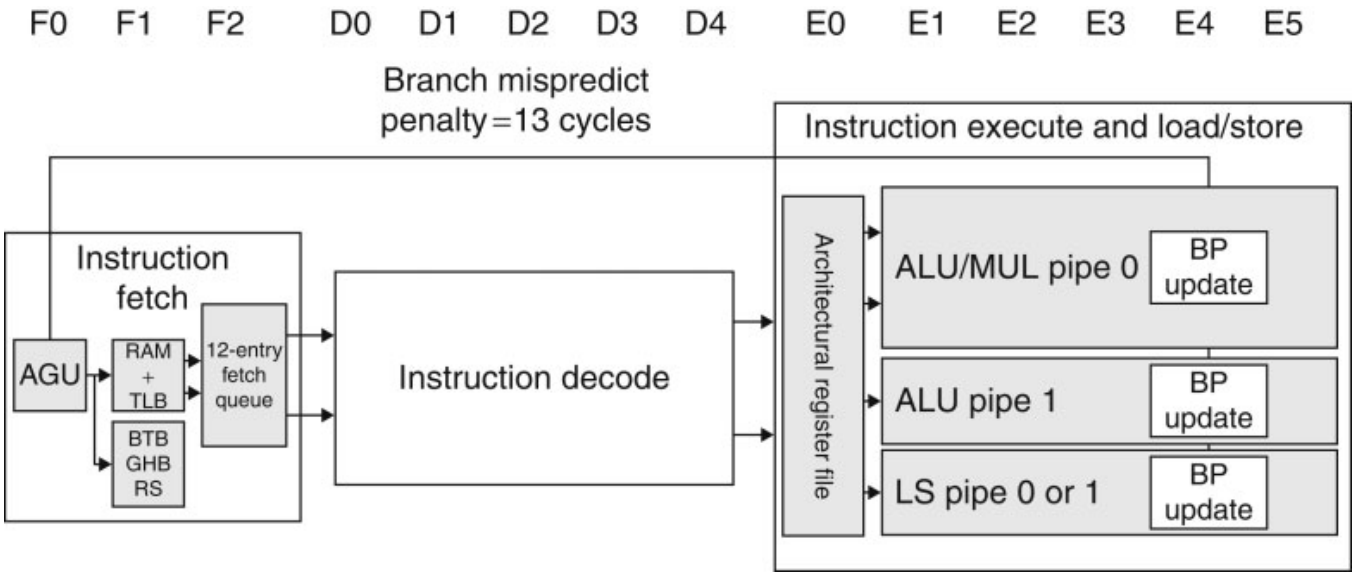
- com *hyperthreading*
- Prenúncio de uma previsível **evolução dos processadores**
  - na senda de outros fabricantes desde 2001

- Explora ILP (*Instruction Level Parallelism*) por permitir a execução simultânea de múltiplas instruções em diferentes estágios



- Permite aumentar a frequência, relativamente a organizações de ciclo único
- O CPI ideal é 1, mas difícil de manter devido a
  - dependências de dados;
  - dependências de controlo;
  - atrasos nos acessos à memória

- Uma abordagem complementar ao encadeamento consiste em
  - ter múltiplos *pipelines* (múltiplas unidades funcionais),
  - permitindo a execução simultânea de múltiplas instruções -> *multiple issue*



[FIGURE 4.75 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- The A8 pipeline]

Paralelismo a um nível mais grosso do que as instruções de um programa:

- explorar o paralelismo entre diferentes **fios de execução (*threads*)**
  - de um mesmo processo ou de processos diferentes
- cada *fio* tem uma **execução independente**
  - estado próprio
    - instruções, dados, contexto: registos, pilha, etc...

```
fio 0                                   fio 1

for (i=0; i < S/2; i++)                 for (i=S/2; i< S ; i++)
{ processa a[i]; }                      { processa a[i]; }
```

Explorar paralelismo

- **ILP / implícito** numa **única sequência de instruções**
- **TLP /explícito** entre **múltiplas sequências de instruções**

O código deve ser escrito explicitamente para expor **TLP"** "*the free lunch is over!" i.e. " não há almoços grátis!"*

- Objetivo:
  - aumentar o débito em computadores que executam vários programas
  - multi-fio *de execução*
    - diminuir o tempo de execução de programas paralelos

Multi-Fio Simultâneo (*Simultaneous multithreading)*

- Execução simultânea de múltiplos fios no mesmo núcleo (SMT)
- **Ideia base:**
  - diferentes unidades funcionais de um núcleo
  - partilhadas por diferentes fios de execução

- Exemplo:
  - Se um fio está à espera que uma unidade complete uma operação de vírgula flutuante
  - O outro fio pode, por exemplo, usar a unidade de operações inteiras

O principal critério que motiva o SMT

- Os processadores de *multiple-issue*
  - têm mais unidade funcionais do que
  - um fio individual pode efetivamente usar
- Além de que
  - com a renomeação de registos e o escalonamento dinâmico
  - múltiplas instruções de fios independentes podem ser lançadas
  - sem levar em consideração as dependências entre elas

- Existem no entanto limitações ao número de instruções que podem executar em simultâneo:
  - dependências de dados e/ou controlo
  - disponibilidade de recursos: número e tipo de unidades funcionais, latência de diferentes instruções, atrasos no acesso à memória

- Quem determina quais as instruções que podem ser lançadas simultaneamente?

static issue	***dynamic issue***
compilador	processador
	Embora o compilador possa ordenar as instruções por forma a facilitar o *multiple issue*

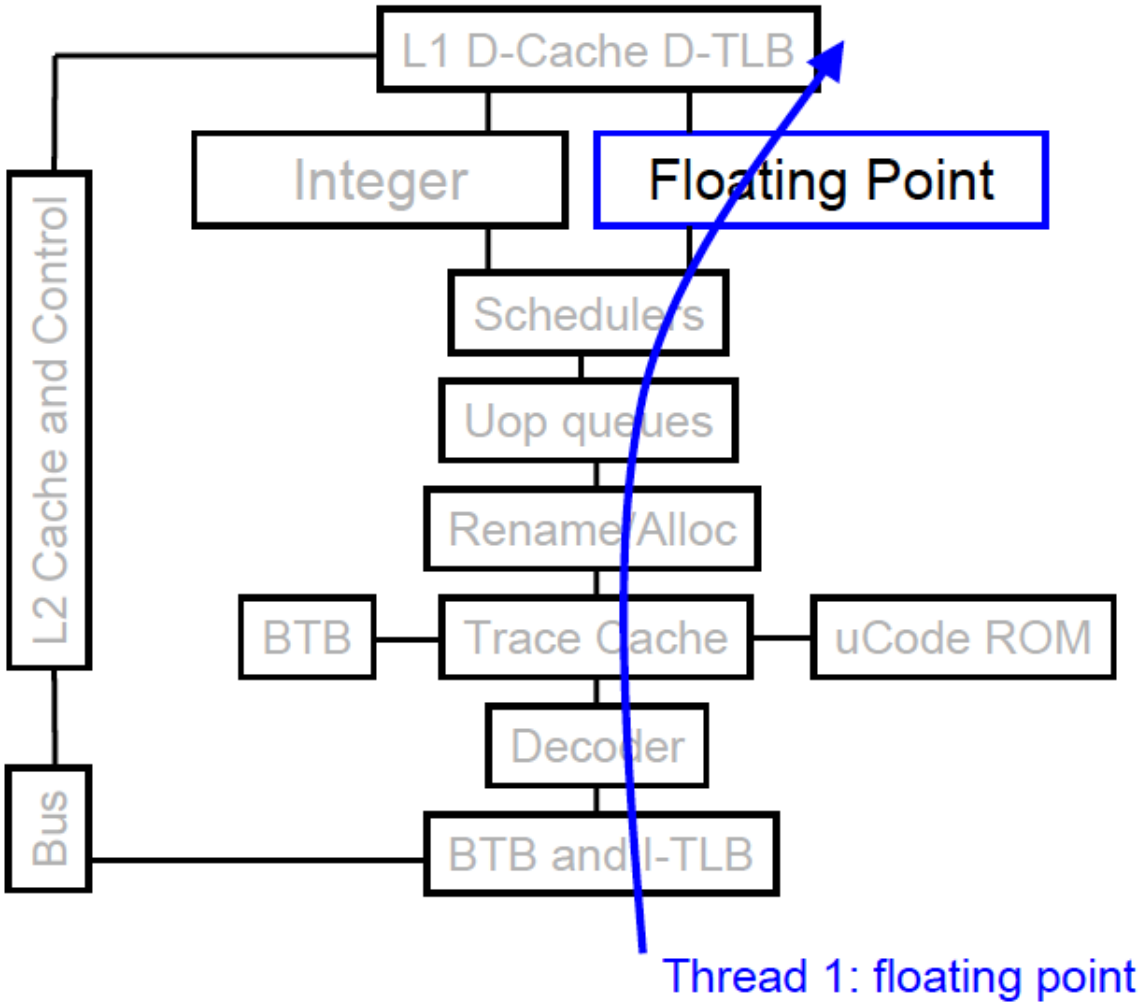
Com SMT

- a resolução das dependências
  - é tratada pelo escalonamento dinâmico

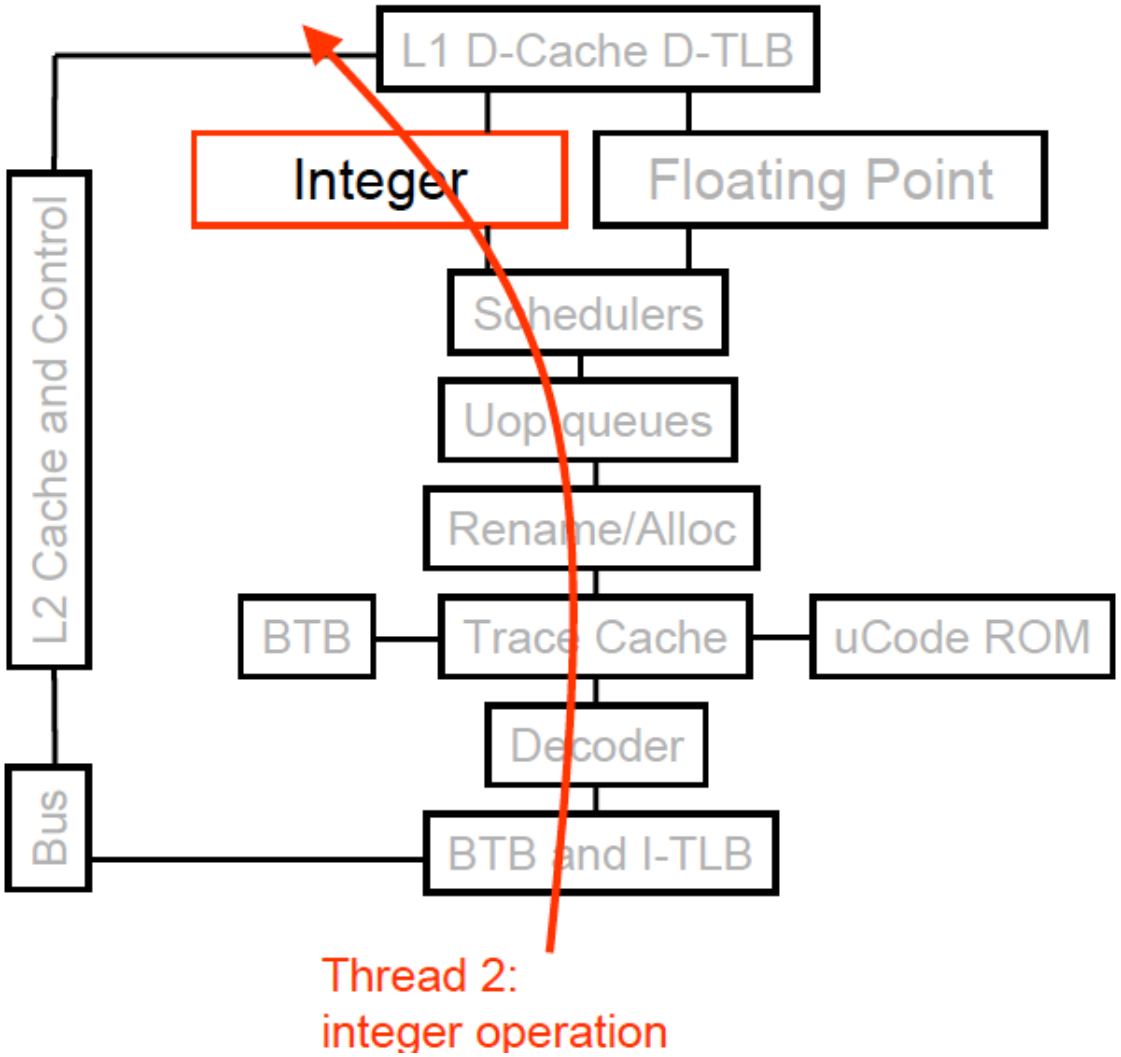
Como o SMT

- depende dos mecanismos dinâmicos existentes
- não alterna recursos a cada ciclo (ver MT de grão fino)
  - está sempre a executar instruções de vários fios
- deixando ao hardware o trabalho de
  - associar aos respetivos fios
    - as instruções e os registos renomeados

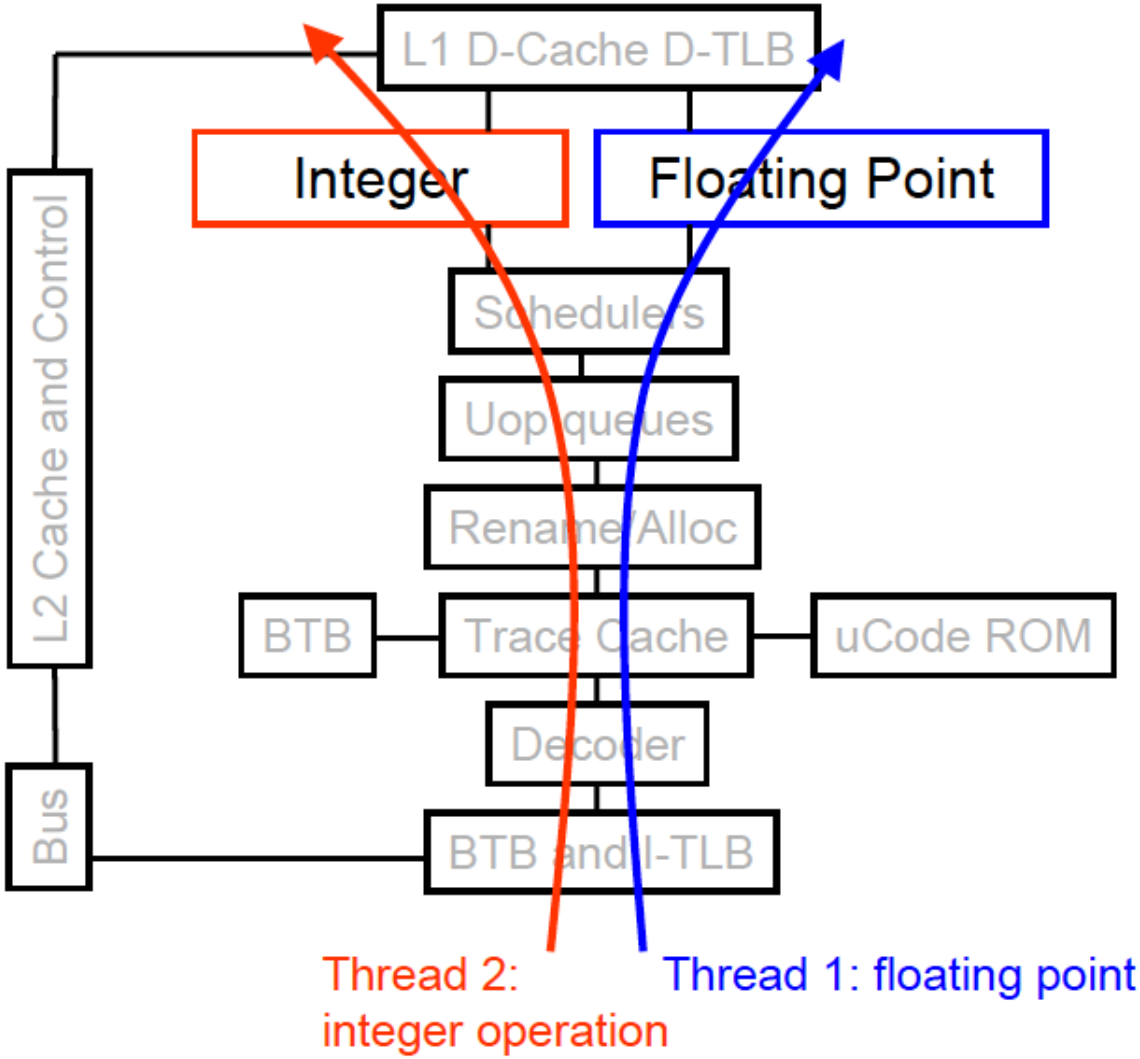
- Sem SMT
  - apenas um fio usa o núcleo em cada instante



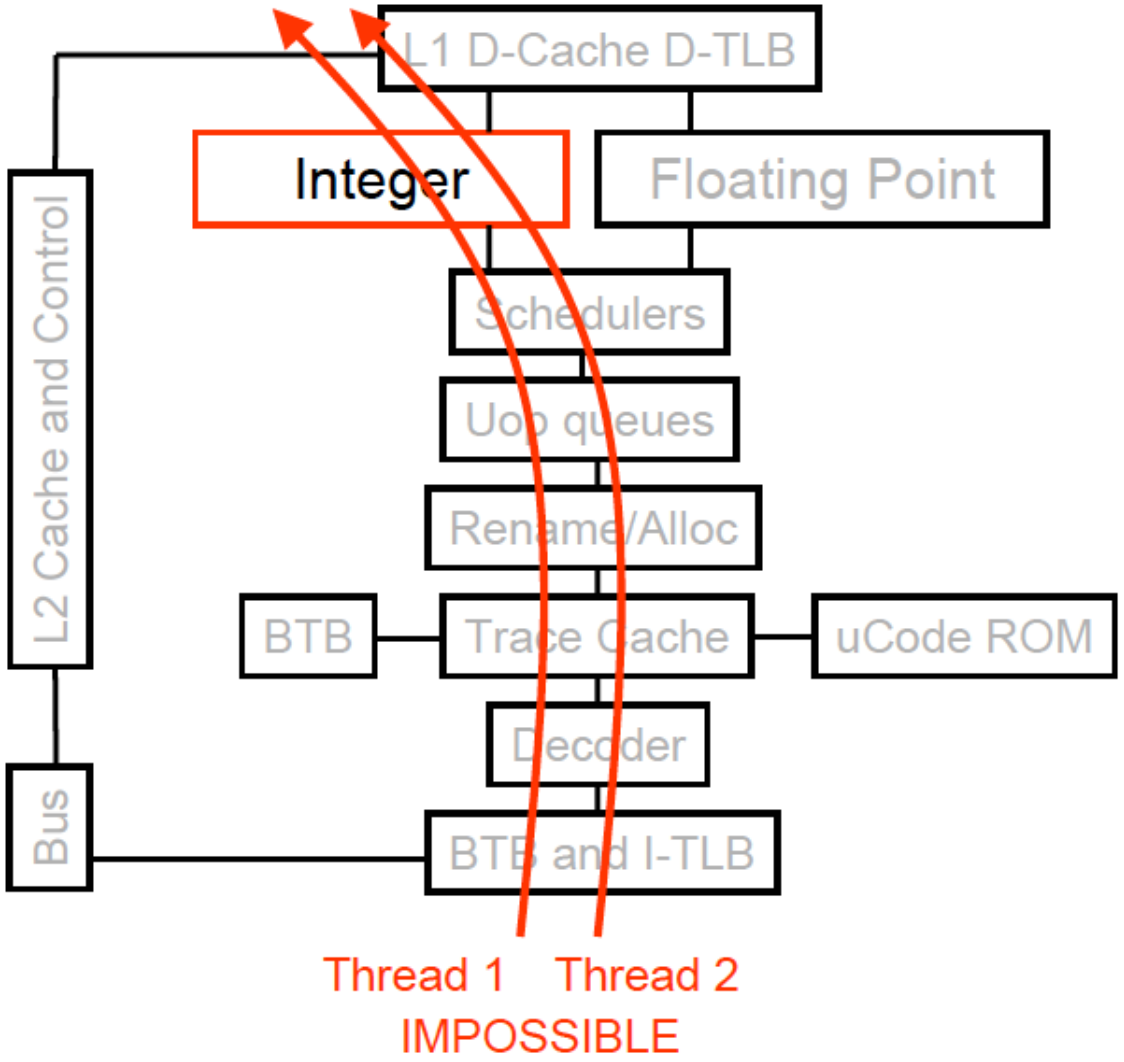
- Sem SMT
  - apenas um fio usa o núcleo em cada instante



- Com SMT
  - ambos os fios podem executar simultaneamente

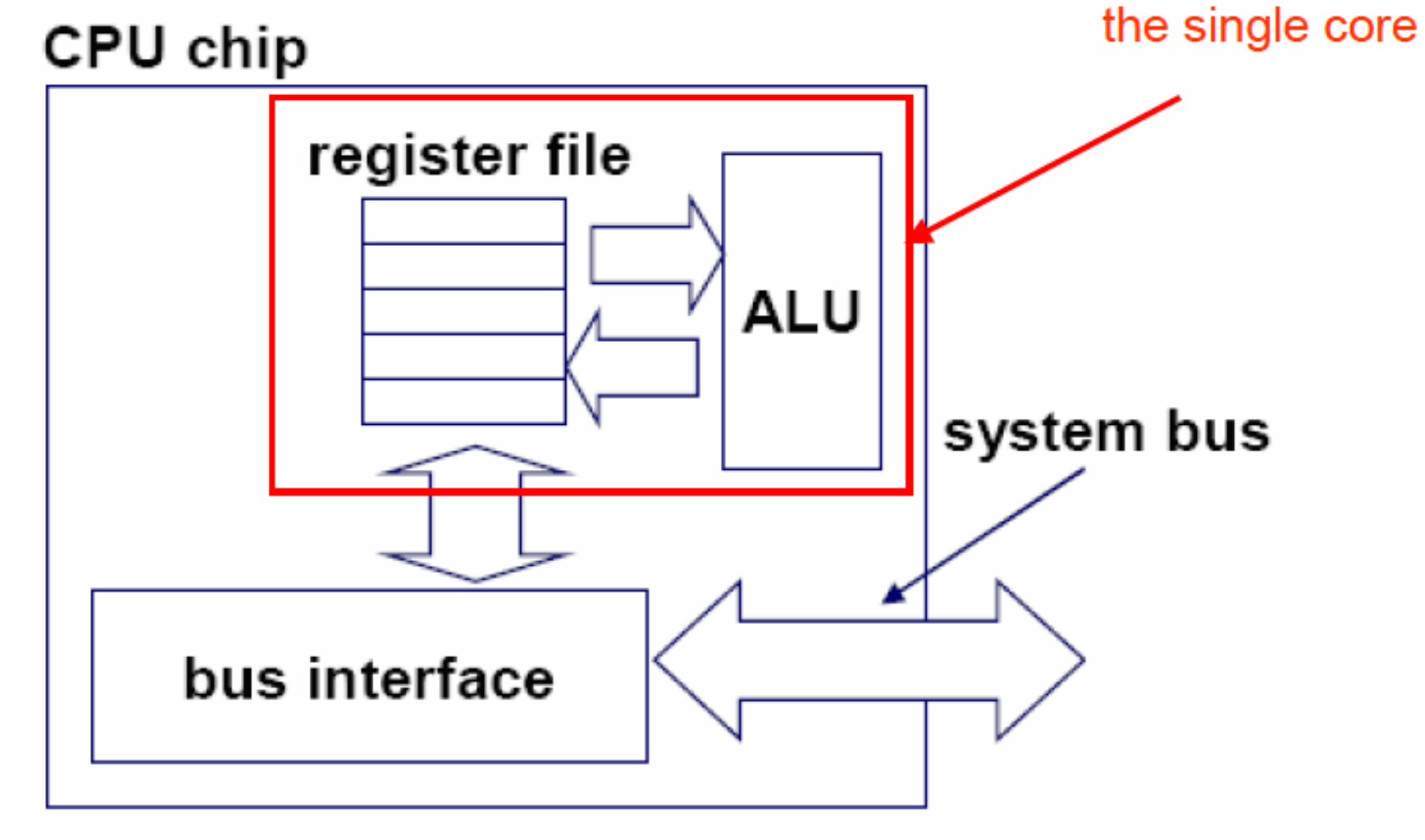


- Com SMT os recursos são partilhados
  - mas não podem ser usados simultaneamente por diferentes fios

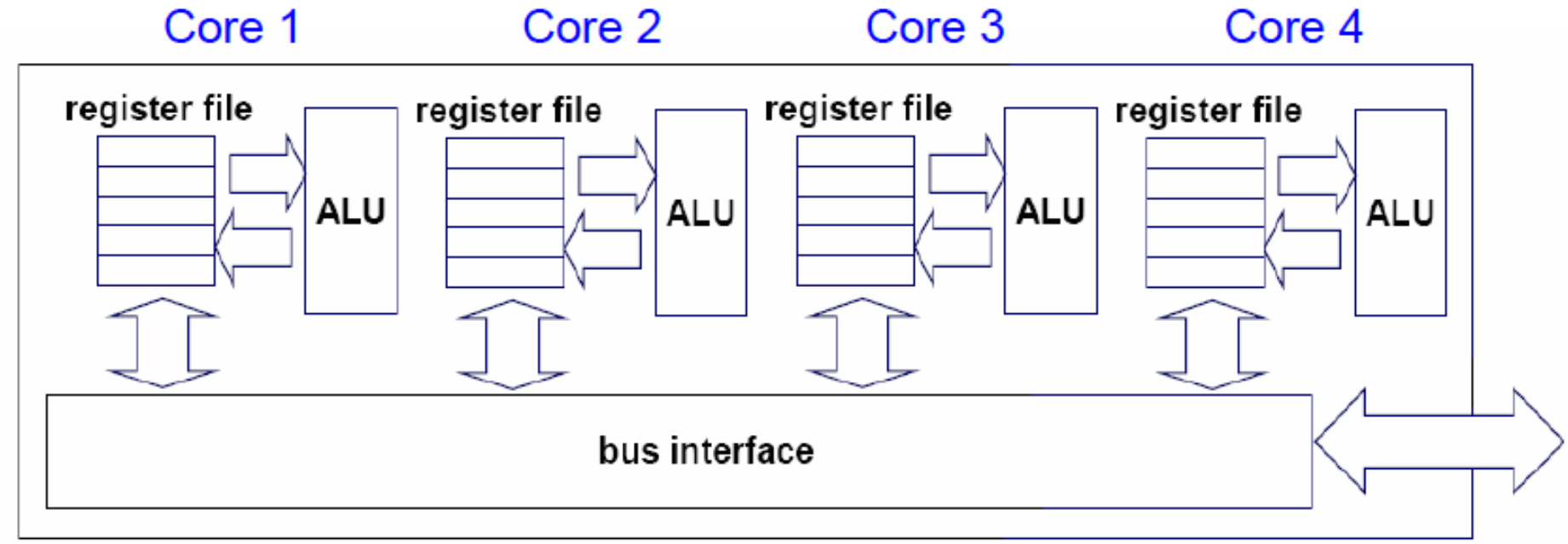


Com SMT ou HyperThreading (**HT**)

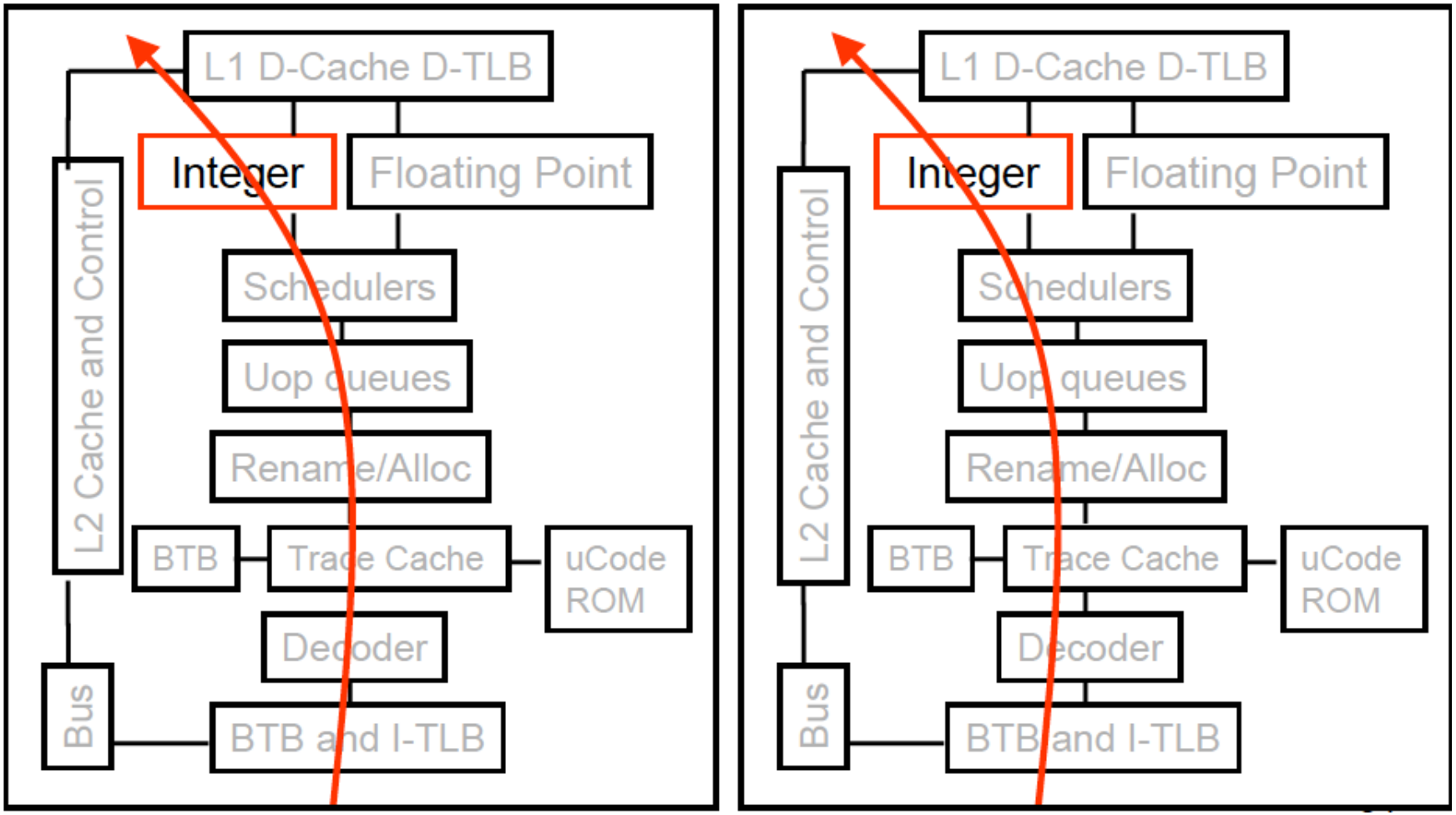
- Intel: um processador HT tem dois núcleos lógicos (cores)
- dois fios podem usar diferentes recursos do mesmo núcleo
- ganho marginal de 30% em termos de tempo de execução


CPU chip
the single core
register file
ALU
system bus
bus interface

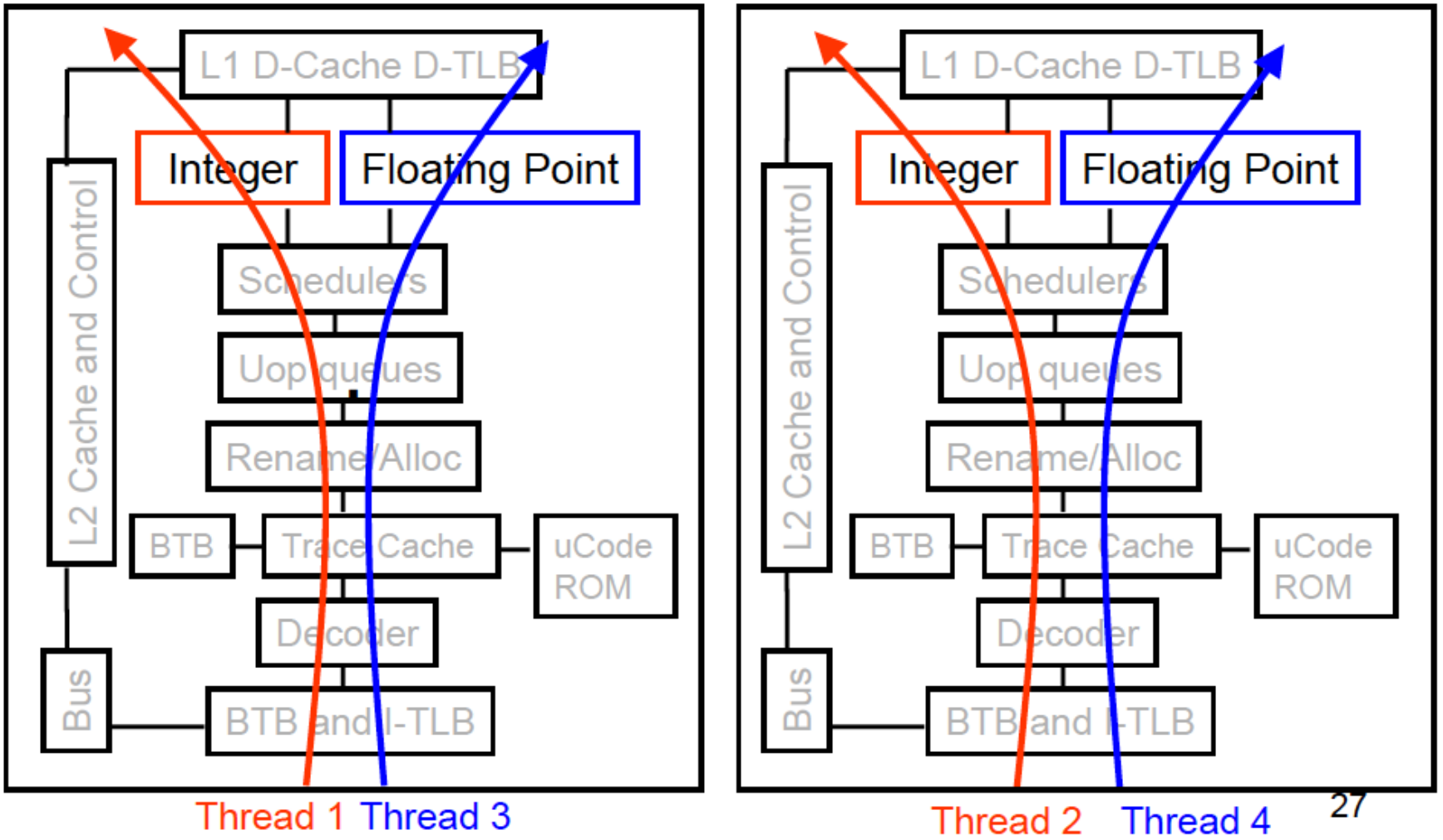
- Replicar múltiplos núcleos num único *Integrado*



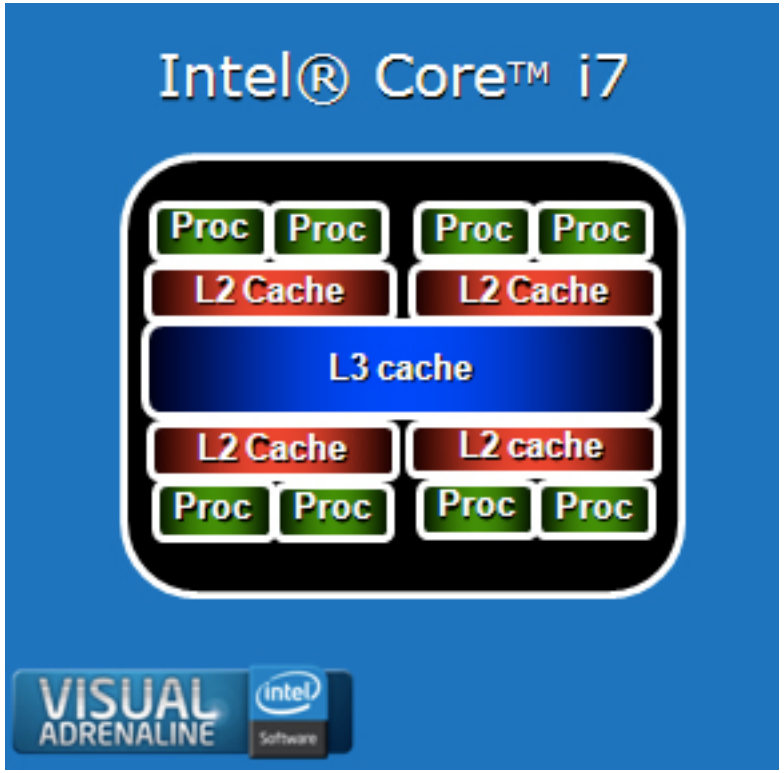
- Os fios podem correr em diferentes núcleos



- Os núcleos podem ou não suportar SMT

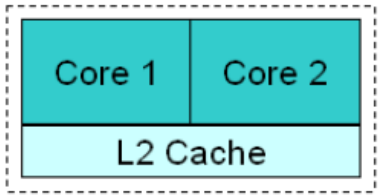


- Actualmente:
  - computadores domésticos : processadores com 2 a 4 núcleos
  - servidores : processadores com 8 a 32 núcleos
- Para o Sistema Operativo
  - cada núcleo é somo se fosse um processador independente
- O ganho com cada núcleo adicional diminui e os núcleos competem por recursos tais como barramentos e memória
- A hierarquia de memória é um aspeto fundamental no desempenho do sistema
- Para tirar partido de arquiteturas multi-núcleo
  - obriga ao desenvolvimento de programas específicos

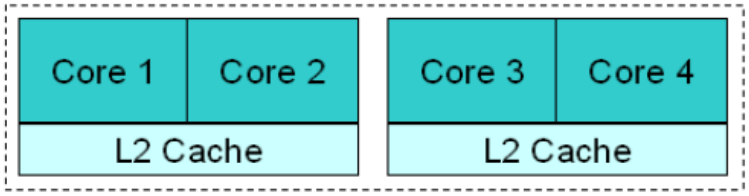


- 4 núcleos
  - (8 lógicos com *HyperThreading*)
- cache L1 e L2
  - privada
  - por núcleo físico
- cache L3
  - partilhada

Multi-Núcleo: Exemplos



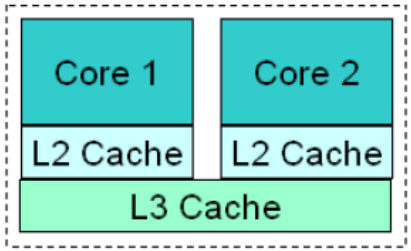
Shared Cache
- Core Duo
- Core 2 Duo



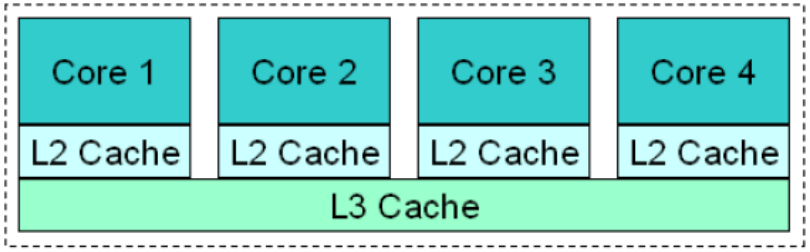
Current Intel Quad-Core CPUs
- Core 2 Quad
- Core 2 Extreme QX

L1 = 32KB+32KB (/Core)

L2 = 2MB/4MB/6MB



K10-based dual-core CPU



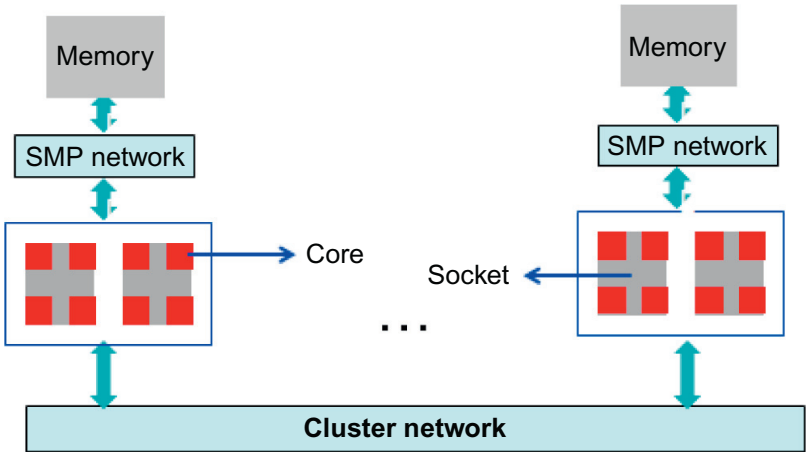
K10-based quad-core CPU

L1 = 64KB+64KB (/Core)

L2 = 512K (/Core)

L3 = 2MB/6MB

- Has the following hierarchical structure
  - **several cores** inside a socket
- A few sockets interconnected around a shared memory block implements a SMP
  - a substantial number of SMP nodes interconnected in a distributed memory cluster



- From a programmer's point of view, the logical view of a SMP
  - is just a number of **virtual CPUs**—the cores—sharing a common memory address space.
  - It does not matter whether the different cores are in the same or in different sockets.

- Flat **MPI** distributed memory programming across cores
  - each core in the system runs a full MPI process, and
  - they all communicate via MPI message passing primitives.
  - It does not matter whether the MPI processes are all in the same or in different SMP nodes.

- A **hybrid MPI-Threads** model in which each MPI process is internally multithreaded, running on several cores.
  - These MPI processes communicate with one another
  - via the MPI message passing protocol.

PARALLEL PROCESSING INSIDE CORES

- The essence of multithreaded programming is the capability of coordinating the activity of several CPUs in the execution of a given application.
- In principle, whatever happens inside a single core is not directly relevant to multithreaded programming.
  - however, it is extremely relevant to the overall performance of the application
  - there are some capabilities of current CPU architectures that must be integrated into the developer's options.

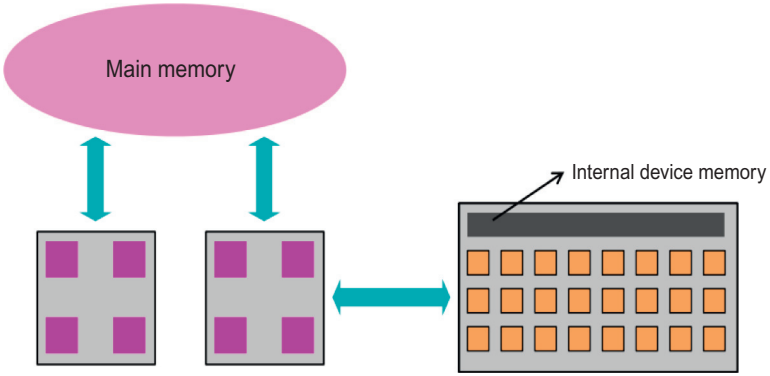
- There is an **amazing amount of parallel processing** inside a core.
  - Millions of transistors operate concurrently most of the time.
    - after translating the application code into basic assembler language,
    - hardware, system, and compiler software cooperate to take as much as possible advantage of *instruction level parallelism*.
    - this means running the basic instructions in parallel whenever it is possible to do so while respecting the program integrity.

- Instruction-level parallelism is too low level to constitute a direct concern to programmers.
  - But there are other parallel structures that are definitely worth their attention:
  - Hyperthreading and vectorization.

- **HT** is a CPU capability of *simultaneously* running several threads (**SMT**).
  - the core has the capability of **interleaving** the execution of instructions arising from different execution streams, while maintaining the program integrity.
  - these different execution streams interleaved in HT in general targeting different data sets,
- A core can service several threads by running them one at a time, in a round robin fashion, allocating CPU time slices to all of them.
  - a general operating system feature that has always existed in multithreading enabling the possibility of over-committing threads on a given core.
  - In this case, different threads access *successively* the CPU cycles.
- HT capabilities in modern architectures aims to make better use of the CPU cycles,
  - e.g., in accumulating efforts to **beat the memory wall**.
  - If a thread needs to wait for memory data, there may be tens or hundreds of cycles wasted doing nothing.
  - these cycles can then be used to run other thread instructions.
- In general-purpose CPUs,
  - HT hides occasional latencies, and its impact on performance strongly depends on the code profile.
- However, there are more specialized architectures - IBM BlueGene/ Intel Xeon Phi **coprocessor**
  - where for different reasons hyperthreading is required to benefit from the full single-core performance.
- The name ***hardware threads*** is used to refer to
  - the number of threads that a core can **simultaneously** execute.

EXTERNAL COMPUTATIONAL DEVICES

- The last few years have witnessed impressive development of external computational devices
  - that connect to a socket via standard interfaces for external devices.
  - acting as a co-processor executing code blocks offloaded from the CPU cores
  - boosting the execution performance for suitable computationally intensive code blocks—called *kernels*—in the application.
  - they are seen from the host CPU as another computational engine available in the network.
- Today two very different kinds of external devices:
  - GPUs (Graphical Processing Units) and Intel's Xeon Phi coprocessor.
- External computational
  - an internal device memory hierarchy
  - a large number of cores for computation.



- Initially, GPUs, Graphical Processing Units,
  - used for rendering visualization images and occasionally perform other computations rephrased in terms of the graphical API.
- In 2007 NVIDIA realized the interest in allowing to think of a GPU as a processor
- Today, GPUs are very powerful computational engines,
  - part of their success is due to the fact that the computing cores, being very lightweight
  - beat the power dissipation wall providing an excellent ratio of computing power (flops) per dissipated watt.

- When discussing GPU accelerators, the concept of *heterogeneous computing* is important,
  - because the GPU computing engine's design requirements are different from those guiding CPU design.
  - GPUs, instead, are ***throughput driven***. They have today tens of thousands of *very lightweight* cores,
- The CPU design is ***latency driven***:
  - making a single thread of execution as efficient as possible.
  - CPUs have a limited number of cores.

- Performance-limiting factor in GPUs is, again,the memory wall.
  - GPUs have no direct access to the main memory, and the target data set must be offloaded to the GPU together with the code to be run.
- GPU architectures are nevertheless evolving very fast,
  - and the possibility of directly exchanging data between different GPUs is available today.
- GPU programming requires specific programming environments
  - capable of producing and offloading code and data to the GPU
  - these programming environments have matured greatly in the last few years
  - CUDA, OpenCL, OpenACL
  - OpenMP 4.0 extensions incorporating directives for offloading data and code blocks to accelerators

- *Intel Xeon Phi is really a complete SMP node*
  - It must be connected to a CPU host and can communicate with it.
  - Today, incorporates 60 normal CPU cores
    - all execute a full and improved x86 instruction set
    - run a complete Linux operating system.
- As an ordinary SMP node,
  - standard programming models are used to compile and run applications
  - in all the distributed memory (MPI) and shared memory programming environments (Pthreads, OpenMP, TBB)

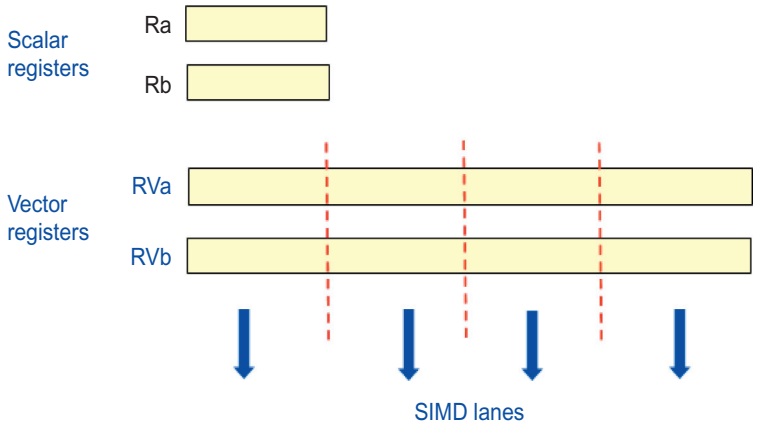
- In fact, the coprocessor has three modes of execution:
  - The main application runs on the master CPU,
    - and specific code blocks are offloaded for execution in the coprocessor
  - The main application runs on the coprocessor,
    - and specific code blocks are offloaded for execution in the master CPU.
    - to perform computationally intensive operations on the Intel Xeon Phi,
    - while offloading I/O operations for execution on the master CPU.
  - Standalone SMP node:
    - the whole application is run on the Intel Xeon Phi.

- Vectorization enhances the single core performance.
  - parallel pattern called **SIMD** Single Instruction, Multiple Data
  - a single instruction operates on several data items in one shot
- Cores with SIMD capabilities have wide vector registers that can hold several vector components.
  - in the Intel Sandy Bridge processor, **vector regist**ers are 256 bits wide,
    - holding either four doubles or eight floats, and
  - we speak in this case of four or eight SIMD lanes, respectively.

- Vector **instructions** can act simultaneously in one shot on all the SIMD lanes,
  - boosting the floating-point performance.

- Implementing wide vector registers in the core architecture is not sufficient.
  - The capability of **loading** the wide vector registers as fast as the scalar registers is also required,
  - which in turn demands an enhanced communication bandwidth between the core and the L2 cache.
  - When the code profile is well adapted, vectorization can provide significant performance enhancements in computationally bound applications.

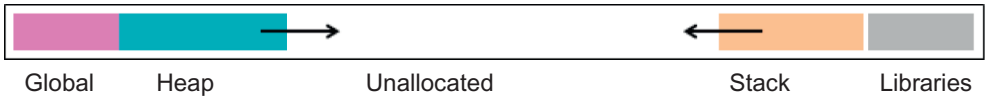
```
double a[N], b[N];
...
for(int n=0; n<N; n++)
   {
   a[n] += b[n];
   }
```



- In the default **scalar** mode of operation,
  - vector addition is computed by adding one component at a time.
  - a[n] / b[n] are loaded into Ra / Rb registers, respectively.
  - Ra = Ra+Rb is performed, and Ra is copied to a[n]
- In **vector** mode,
  - loop is computed by loading in RVa and RBv a block of four a[] and b[] components,
  - and acting simultaneously on all of them.

- Applications run as an operating system process
  - *the basic unit of resource allocation to an application*.
- when application starts execution the OS system allocates a number of resources
  - a protected fraction of the available memory.
    - no other process can access it.
  - file handles required by the application.
  - access to communication ports and I/O devices.
  - a share of the CPU cycles available in the executing platform (multitasking).

- The process executes
  - **startup** code to allocate the resources and set up the environment.
  - then the **main**() function provided by the programmer
- There are three distinct memory domains:
  - **global** variables allocated by the startup code
    - Variables declared outside the function main(),
  - the ***heap***, reserved for dynamic memory allocations
    - by the library functions: malloc() / C or new / C++.
  - ***stack***, to allocate the local variables
    - defined inside main() and all possible nested functions



- Multitasking is the capability of the operating system of running several applications at the same time.
  - In a multitasking environment, the totality of the computer's resources (memory, files, CPU time) are allocated to different applications,
  - and they are managed in such a way that each one of them gets a share according to specific priority policies.
- The operating system provides **inter-process communication** mechanisms, like pipes or signals.
  - They are meant to enable communication *across* applications.
- Switching among different processes naturally induces an execution overhead.
  - Whenever a process **switch** occurs, all the process resources must be saved.
  - If, for example, there are several open files, the file handles that identify each file, as well as the file pointers that identify the current positions inside each file, must be saved.
  - The operating system is also forced to save all other information related to the state of the process (instruction pointer, stack pointer, processor registers, etc.) needed to reconstruct its state at a later time slice.

- Processes are, the basic units of resource allocation,
  - but *they are not the basic units of dispatching*.
  - Indeed, they are not doomed to live with only the initial execution stream encoded in the main() function:
  - additional internal asynchronous **execution streams**, or *threads*, can be launched.
  - This is done by the native multithreading libraries that come with the OS:
    - the Windows thread API, or the
    - in Unix-Linux systems - Pthreads
  - A thread is therefore *a single sequential* ***flow of control*** *within a program*.
    - When mapped to different cores allocated to an application, these independent, asynchronous flows of control implement parallel processing.

- Parallel processing relies on exploitable concurrency.
  - There are a number of concepts that will be systematically referred to, and it is convenient to be precise about their meaning:
- Concurrency exists in a computational problem when the problem can be decomposed into sub-problems—called **tasks**—that can safely execute at the same time.

- Tasks are fundamental units of sequential computation.
  - They are units of work that express the underlying concurrency in the application, providing the opportunity for parallel execution.
  - Tasks that can be active at the same time are considered to be concurrent.
- Parallel execution occurs when several tasks are actually doing some work at the same point in time.
  - This happens when they are simultaneously executed by several threads.
- Concurrency can be beneficial to an application, even if there is no parallel execution.

- The application performance can naturally be enhanced if concurrent tasks can be executed in parallel, overlapped in time as much as possible.
  - Threads provide the way to implement this fact.
- One of the first things a programmer learns about programming is organizing and isolating tasks by subroutines or function calls.

- *in order to launch a new thread, a function must be provided that encapsulates the sequence of instructions to be executed by the thread*,
  - in the same way the main() function displays the sequence of instructions to be executed by the initial thread in the process, called the *main thread*.

- Threads are nevertheless more than functions.
  - **Function** calls correspond to a ***synchronous* transfer of control** in the program.
  - A standard function call is a very convenient way of *inserting* a set of instructions into an existing execution flow.
  - **Threads**, on the other hand, correspond to ***asynchronous* transfers of control** to a new execution flow.
  - The initial execution flow that launches the new thread continues its own progress, and it is not forced to wait until the new thread finishes and returns

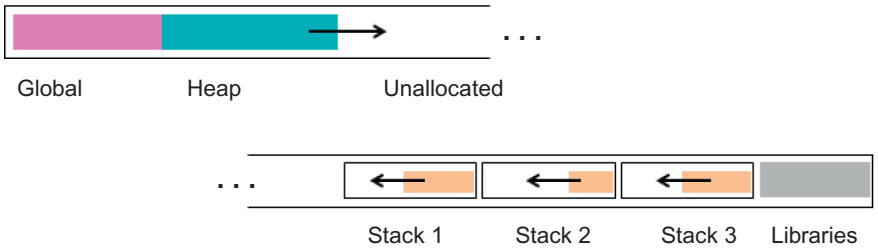
- Initially, a process consists of only one thread, the ***main*** *thread*.
  - Next, the main thread can launch other threads by calling the appropriate library functions provided by the native libraries.
- From here on, there is absolutely **no difference** between the main thread and the other threads.
  - Any thread can call again the appropriate library function and launch new threads.
- All the threads, no matter how they are created, are treated as equal by the operating system and **scheduled** in principle in a fair way.

- All these new execution streams are totally equivalent to the initial main() function.
  - Threads can be thought of as asynchronous functions that become totally autonomous and take off *with their own stack*, where they manage their own local variables as well as their own nested function calls.

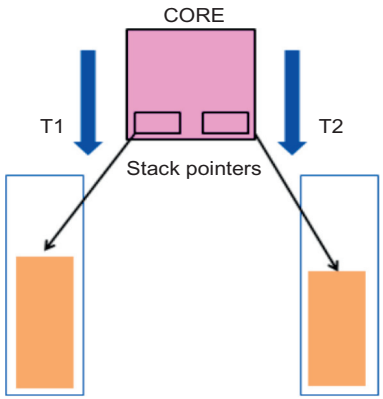
- Threads share all the process resources ***except the stack***.
  - Each thread creation involves the creation of a new stack where the new thread function will allocate its local variables or the local variables of the functions it calls.

MEMORY ORGANIZATION OF A MULTITHREADED PROCESS

- The memory organization in a multithreaded process is shown in Figure.
  - The main difference with a sequential process is that there are now several **stack buffers**—one per thread—with a definite size.
  - In fact, the stack size is one of the fundamental attributes of a thread. Default values are provided by the operating system,
- It is important to notice that:
  - Global variables are ***shared*** *variables* that can be accessed by any thread.
  - Local variables on the thread's stack are ***private*** *variables* that are only accessed in principle by the owner thread.
- Notice also that dynamically allocated memory can be shared by all threads if it is referenced by a global pointer.
  - Otherwise, dynamically allocated memory is only accessible by the thread that owns the pointer.



- As already said, hyperthreading is the capability of a core *simultaneously* execute two or more threads.
- The figure shows a core running two hardware threads
  - The two instruction sets are interleaved, and the core takes advantage of eventual delays in the execution of one of the threads to execute instructions of the other.
  - We have a core operating on two data sets
    - when hyperthreading is enabled, the core dedicates **two registers** to hold the stack pointer of the two stacks. In this way, each instruction hits its correct data target.



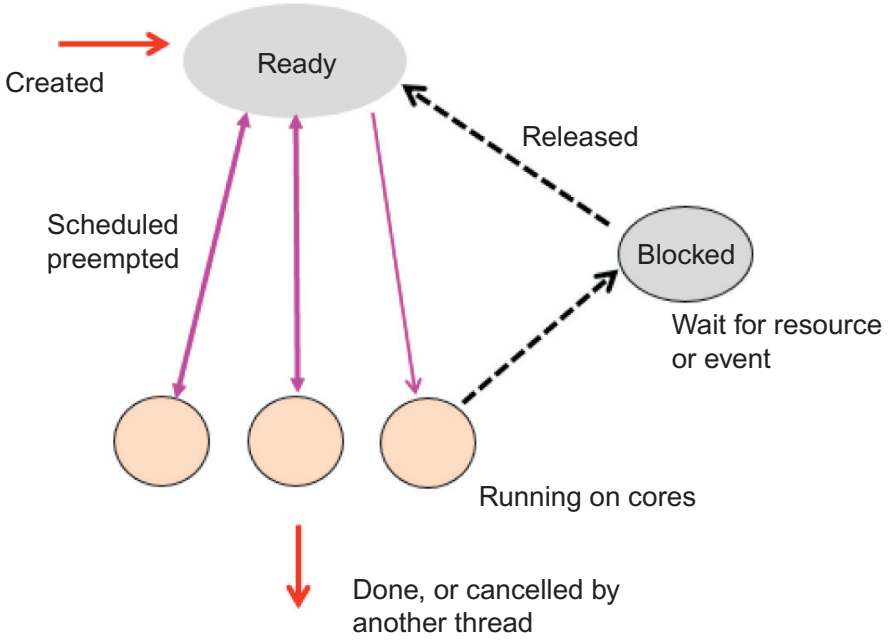
- It is useful to keep in mind that:
  - A programming model is the **logical view** that the programmer has of the application.
  - An execution model is the precise way in which the application is processed by the target platform.
- Programming models—like OpenMP, TBB, or the Pthreads library—tend to be as generic and **universal** as possible,
  - to guarantee the **portability** of the application across different platforms.
- Execution models, in contrast, **depend** on the specific hardware and software environments of the target platform.
  - For portable code, the same programming model must be mapped to different execution contexts.
  - Here is the place where the **compiler optimization** options play a fundamental role in performance.

- When mapping a programming model to an executing platform, ideally
  - each virtual CPU should be mapped to a unique core.
  - but it is possible to have in an application more threads than cores allocated to the process.
- When a multithreaded application is executed two extreme cases can be distinguished:
  - *Parallel execution:* the different threads are executed by different cores in a multicore system.
  - *Concurrent execution:* the different threads are executed by only one core, or by a number of cores smaller than the number of threads.
- Because of the way the life cycle of a thread is conceived,
  - concurrent programming with **overcommitted** cores can in many circumstances
  - contribute to improve the performance or the efficiency of an application.
- One important observation is that *the transition to a **blocked state** can be controlled by the programmer*.
  - In all the native or basic libraries there are logical constructs that allow a programmer to put a thread out of the way for some time,
    - until, for example, some data item takes a predefined value.

THREADS AS LIGHTWEIGHT PROCESSES

- Each thread created inside the process has its own stack and stack pointer, as well as its own instruction list.
  - However, all the other resources (global memory, files, I/O ports, . . .) allocated to the process are shared by all the threads.
- A process may have more threads than available cores, and in this case CPU resources must be shared among threads: each one disposes of time slices of the available CPU time.
  - However, switching among threads is much more efficient than switching among different processes.
  - Indeed, since most of the global resources are shared by all threads, they do not need to be saved when the thread switch occurs.
- To sum up:
  - Threads are slightly more complex than ordinary method functions. Each thread has its own stack.
  - Threads are simpler than processes.
  - Indeed, a thread can be seen as a sort of a "**lightweight**" process with simpler resource management, since the only data structure linked to the thread is the stack.
  - It can also be seen as a function that has grown up and acquired its own execution context, i.e., its own stack.

- The diagram shows the thread life cycle in an application.



- ***Ready****:* a pool that contains the set of threads that are waiting to be scheduled on a core.
- ***Running*** *on cores:* the thread is in progress. Two things can happen to a running thread:
  - If threads are over-committed, it will at some point be preempted by the operating system and placed back in the ready pool, in order to give other threads the opportunity to run.
  - The operating system ***does not preempts*** threads if there are as many CPUs available as active threads.
  - It may be moved to a ***blocked state***, releasing the core, waiting for an event.
  - At some point, another thread will produce the expected event and release the waiting thread, which goes back to the ready pool to be rescheduled.
    - This is an ***event synchronization*** *mechanism*, controlled by the programmer.
  - ***Termination*** occurs when the thread is done, or when it is canceled by other threads.

- In high-level programming environments—like OpenMP or TBB—threads are **implicit**,
  - and the basic thread management operations are silently performed by the **runtime** library.
- Nevertheless, understanding the way basic libraries set up a multithreaded environment is useful knowledge,
  - which definitely contributes to learning to think about threads.
- It may be useful, in a specific parallel context
  - to launch one or more threads to manage a special issue, like a long network connection,
  - and it is good to know how this can be done.

- Next, we presente an overview of the Pthreads, and C++11 interfaces for the creation of a team of worker threads
- Then, a first look is taken at OpenMP, focusing on basic features
  - to underline the universality of concepts and similarities between different approaches

- Understanding the way basic libraries set up a multithreaded environment is useful knowledge.
- Thread utility can be classified as follows:
  - **management** functions, dealing with the issues related to thread creation, termination, and return.
  - **synchronization** functions, dealing with synchronization mechanisms for threads
  - **miscellaneous** services, specialized services, dealing with thread identities, and with initialization issues
    - when the number of threads engaged in its execution is not known at compile time.
  - **thread-local** storage, providing mechanism for allowing threads to keep a private copy of dynamically allocated local data
  - **cancel** functions for enabling a thread to terminate (kill) another thread
  - **scheduling** functions allowing the programmer to determine the priority level of a new thread.
    - by default, threads are democratically scheduled with equal priorities,
    - this option is very rarely modified in application programming.
  - **signal** handling functions. deal with the interaction of threads and signals in inter-process communications.

- The Pthreads library has been available since the early days of the Unix operating system,
  - and there is a large number of bibliographical references, mainly on the Internet.
- Two classic books on this subject are
  - Butenhof D., “Programming with POSIX threads” Addison-Wesley;
  - Nichols B, Buttlar D, Proulx Farrel J. “Pthreads programming”; O’Reilly.
- **Chapter IV,** Pacheco P., “An Introduction to Parallel Programming”, Morgan Kaufmann.

- **POSIX® Threads**
  - Also known as Pthreads
  - A standard for Unix-like operating systems
  - A library that can be linked with C/C++ programs
- Specifies an application programming interface (API)
  - for multi-threaded programming.
  - only available on POSIX systems — Linux, MacOS X, Solaris, HPUX, ...

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));

  for (thread = 0; thread < thread_count; thread++)
     pthread_create(&thread_handles[thread], NULL,
         Hello, (void*) thread);

  printf("Hello from the main thread\n");

  for (thread = 0; thread < thread_count; thread++)
     pthread_join(thread_handles[thread], NULL);

  free(thread_handles);
  return 0;
} /* main */
```

Declare the various pthreads, functions, types, constants, etc

HelloWorld (2)

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

gcc −g −Wall −o pth_hello pth_hello . c −lpthread

. / pth_hello 4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

- pthread_t objects are **Opaque**
  - The actual data that they store is system-specific
  - Their data members aren't directly accessible to user code
  - However,
    - Pthreads standard guarantees that a pthread_t object does store enough information
    - to uniquely **identify** the thread with which it's associated.

```
int pthread_create (
      pthread_t*  thread_p /* out */ ,
      const pthread_attr_t*  attr_p /* in */ ,
      void*  (*start_routine ) ( void ) /* in */ ,
      void*  arg_p /* in */ ) ;
```

We won't be using, so we just pass NULL.

Allocate before calling.

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

Pointer to the argument that should be passed to the function *start_routine*.

The function that the thread is to run.

- Prototype:

  void* thread_function (void* args_p) ;

  – Void* can be cast to any pointer type in C.

  – So args_p can point to a list containing one or more values needed by thread_function.

  – Similarly, the return value of thread_function can point to a list of one or more values.

- We call the function pthread_join once for each thread.
  - A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

```
pthread_exit(void *)
pthread_join(t, (void**)&ret)
```

- The C++11 standard has incorporated a complete set of multithreading facilities into the C++ standard library.
  - This evolution is a significant step in software engineering:
    - the possibility of disposing of a **portable**, **low-level** programming environment
    - providing practically all the services and features of the native libraries.
  - Developers can use C++11 threads to implement sophisticated system programming
    - with full portability as an extra bonus.

- in Pthreads, the only header file required to be included is
  - **pthread.h**

- The C++11 thread library, instead, requires a number of specific header files corresponding to the different proposed services,
  - <**thread**>: facilities for managing and identifying threads, or for putting a thread to sleep for a predetermined time duration.
  - <**mutex**>: facilities implementing mutual exclusion, in order to enforce thread safety in shared data accesses by different threads.
  - <**condition** variable>: basic level synchronization mechanism allowing a thread to block until notified that some condition is true or a timeout is elapsed.
  - <**future**>: C++11 specific facilities for handling asynchronous results from operations performed by another threads,
  - <**atomic**>: Classes for atomic types and operations.
  - <**chrono**>: classes that represent points in time, durations, and clocks.

- C++11 std::thread class
  - Its role is very clear:
    - *when an object instance of this class is created, a new thread is launched*.
    - This **std::thread** object can be seen as representing the new thread's identity,
  - the thread function can be any function
    - that returns **void**,
    - with an arbitrary number of arguments.

```
void thread_fct(arg1, arg2, ...);        // definition of thread function
...
std::thread T(thread_fct, arg1, arg2);   // launch the thread T
```

- A worker thread waits in a blocked state for a given number of seconds to simulate a long computation.
  - the main thread launches the worker thread and waits for its termination by calling join()

```
#include <iostream>
#include <thread>
#include <chrono>
void ThreadFunc(unsigned mSecs) {
    std::chrono::milliseconds workTime(mSecs);        // create duration
    std::this_thread::sleep_for(workTime); // sleep }

int main(int argc, char* argv[]) {
    std::thread T(ThreadFunc, 3000);                   // thread created
    std::cout << "main: waiting for thread " << std::endl;
     T.join();
    std::cout << "main: done" << std::endl;
return 0; }
```

```
$ make cpp1s
g++ -std=c++11 -g -O3  -c -o Cpp1_S.o src/Cpp1_S. cxx
g++ -ocpp1s Cpp1_S.o –lpthread

$./cpp1s

main: startup
main: waiting for thread
Worker running. Sleeping for 3000
worker: finished
main: done
```

- A thread function update an external variable passed by reference

```
void ThreadFunc(int& N) { N += 10; }
int main(int argc, char* argv[]) {
     int N = 10;
     std::cout << "\n Initial value of N = " << N << std::endl;
     std::thread T1(ThreadFunc, N);
     T1.join();
    std::cout << "First value of N = " << N << std::endl;
    std::thread T2(ThreadFunc, std::ref(N));
    T2.join();
    std::cout << "Second value of N = " << N << std::endl;
    return 0;
```

```
$ ./cpp4s
 Initial value of N = 10
Second value of N = 20
```

NEED FOR MUTUAL EXCLUSION AMONG THREADS

- In a multithreaded application all worker threads occasionally increment a shared counter.
  - this increment operation looks very **innocent**, but in fact it is not.
  - current microprocessors implement ***load/store*** *architectures*
  - where direct memory operations are not supported.

```
int counter;
void *thread_function(void *arg)
        {
        ...
        counter++;
        ...
        }
```

- To increment the counter,
  - the processor ***reads*** its value from memory to an internal register,
  - increments it, and finally ***writes*** the new value back to memory
  - the increment operation is therefore **not *atomic***,

- Nothing prevents two threads running on two **different** cores
  - from reading the same value at almost the same time, before the first reader had time to increment the counter
    - and write it back to memory.
    - In this case, both threads will increment the same initial value.
  - The final value of the counter is increased by 1,
    - in spite of the fact that it has been incremented twice.

- This is a classical example of a ***race*** *condition*:
  - the outcome of the computation **depends** on the way threads are **scheduled** by the operating system.

- Things can also go wrong even
  - if the two threads are sharing cycles on the **same** core.
  - indeed, nothing prevents thread A from being **preempted** by the operating system
    - precisely in the middle of the increment operation,
    - after the read and before the write
  - then thread B is scheduled and increments the counter, after reading the same, old value not yet updated by thread A
  - when thread A is **rescheduled**, it will complete its ongoing update
  - the end result is the same as before:
    - the counter is increased by 1, in spite of the fact that both threads have increased it.

- In concurrent or in parallel programming, threads
  - require a mechanism that **locks** the access to **shared** global variables,
  - allowing them to safely complete some operations, avoiding race conditions.
  - a mutual exclusion mechanism is needed, in which
    - a well-identified code **block**
    - **cannot** be **executed** by more than one thread at the same time.

- Critical Section, is a **code block** that must be executed by one thread at a time
  - a thread cannot enter a critical section if another thread is inside
  - this mechanism is used to exclude possible race conditions.
- Very simple to implement critical sections
  - threads entering a critical section are forced to acquire ownership of a shared global resource that can only be **owned** by **one thread** at a time.
  - such a resource is called a ***mutex***, a short name for *mutual exclusion*.

- Before entering the critical section,
  - a thread **acquires** ownership of a mutex by ***locking*** it
  - when the mutex is locked, other threads that reach the critical section code are forced to wait for it to be released
  - a thread exiting the critical section **unlocks** the protecting mutex
  - if there are other threads waiting for it
    - the operating system **reschedules** one of them
    - which then locks and takes ownership of the mutex

DIFFERENT KINDS OF MUTEX FLAVORS

- Libraries implements a mutual exclusion protocol
  - mutexes can be locked and unlocked and, when locked, they are owned by only one thread.
  - introducing mutexes of different types to fine-tune the application performance.
- Standard versus **spin** mutexes:
  - When a thread acquires a mutex, the function does not return until the mutex is locked.
  - This is typical for event synchronization: a thread waiting for an event
- There are **two ways** in which this can happen:
  - An ***idle*** *wait*: the thread waiting to lock the mutex is blocked in a wait state.
    - It releases the CPU, which can then be used to run another thread.
    - When the mutex becomes available, the runtime system wakes up and reschedules the waiting thread,
    - which can then lock the now available mutex.
  - A ***busy*** *wait*, also called a ***spin*** *wait*, in which a thread waiting to lock the mutex *does not release the CPU*.
    - It remains scheduled, executing some trivial do nothing instruction until the mutex is released.
- **Standard** mutexes normally subscribe to the first strategy, and perform an idle wait.
  - But some libraries also provide mutexes that subscribe to the spin wait strategy
    - The best one depends on the application context.
  - For very short waits spinning in user space is **more efficient**
    - because putting a thread to sleep in a blocked state takes cycles
  - But for long waits, a sleeping thread releases its CPU
    - making cycles available to other threads

DIFFERENT KINDS OF MUTEX FLAVORS (2)

- A **fair** mutex lets threads acquire the mutex in the order they requested the lock
  - avoids starving threads, because each one of them will get in due time its turn.
- **Unfair** mutexes do not respect the order in which the mutex lock is requested
  - faster, allows threads ready to run to jump over the waiting queue
- **Recursive** mutexes:
  - *the owner thread may* lock them recursively several times

- **Try**_lock() functions: enables a thread to *try to lock a mutex*.
  - This function *never waits*. If the mutex is available, it returns 1 after locking the mutex. Otherwise, it returns 0 meaning *mutex not available*.
  - The intention is to allow programmers to optimize code by testing the mutex availability. If the mutex is not available, the caller thread can proceed to do something else and come back later on to try again.
- **Timed** mutexes:
  - In this case the lock function does not return immediately, and performs a timed wait for a number of milliseconds passed as argument in the function call.
  - Then, it proceeds as before: it returns 1 (or 0) if it succeeds (or fails) in locking the mutex.

DIFFERENT KINDS OF MUTEX FLAVORS (3)

- **Shared** mutexes:
  - has **two lock** modes: shared and exclusive.
  - In shared mode, several threads can take simultaneous ownership.
    - introduced for optimization purposes,
    - seems to contradict the very nature of the mutex operation.
- when several threads are accessing a data set,
  - **write** operations need **exclusive** access
  - **read** operations, instead, only need the guarantee that the data set will not be modified while they are reading,
    - so they need to exclude simultaneous writes.
  - but they do not need to exclude simultaneous reads, and can share the mutex with other readers.

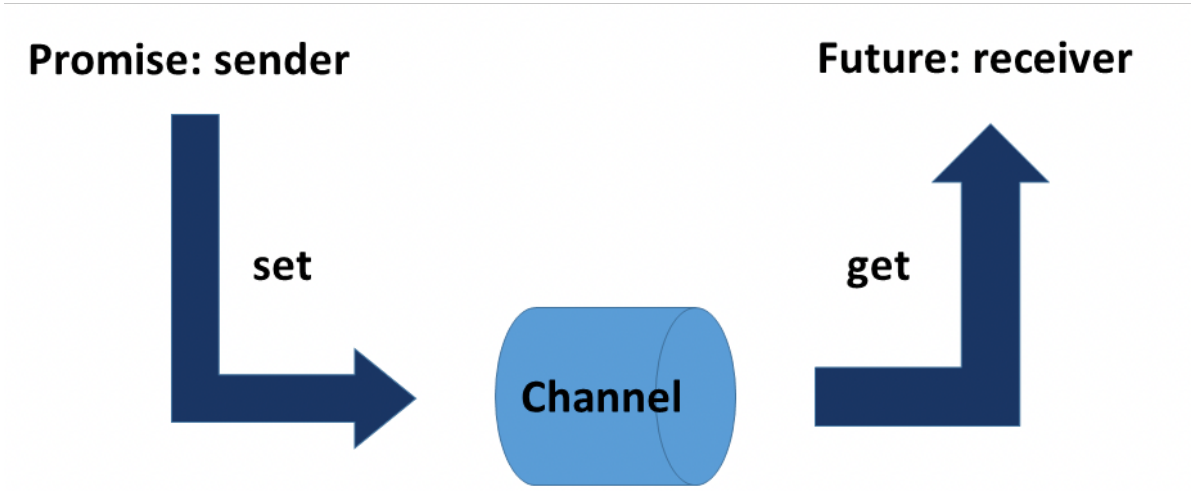
Table 5.1 Mutex Flavors in Different Programming Environments

	Mutex Flavors				
	Pthreads	**Windows**	**C++11**	**OpenMP**	**TBB**
mutex	X	X	X	X	X
recursive	X		X	X	X
timed	X	X	X		
timed recursive			X		
spin fair	X				X
spin unfair					X
shared	X	X			X

C++ has **tasks** to perform work asynchronously

- A task is parameterised with a work package and consists of the two associated components, a **promise** and a **future**.
  - Both are connected via a **data channel**.
- The promise executes the work packages and puts the result in the data channel;
  - the associated future picks up the result.
  - Both communication endpoints can run in separate threads.
- What's special is that the future can pick up the result at a later time.
  - Therefore the calculation of the result by the promise is **independent** of the query of the result by the associated future.

- Tasks behave like data channels.
  - The promise **puts** its result in the data channel.
  - The future **waits** for it and picks it up.



Threads are very different from tasks.

- For the communication between the creator thread and the created thread, you have to use a **shared variable**.
- The task communicates via its data channel, which is implicitly protected.
  - Therefore a task must not use a protection mechanism like a **mutex**.

- The creator thread is waiting for its child with the **join** call.
- The future fut is using the fut.**get**() call
  - which is blocking if no result is there.

- If an **exception** happens in the created thread, the created thread terminates and therefore the creator and the whole process.
- On the contrary, the promise can send exceptions to the future, which has to handle the exception.

- A promise can serve **one or many** futures.
  - It can send a value, an exception or only a notification.
- You can use a task as a safe **replacement** for a condition variable.

The child thread t and the asynchronous function call **std::async** calculates the sum of 2000 and 11.

- The creator thread gets the result from its child thread t via the **shared** variable res.
- The call std::async creates the data **channel** between the sender (promise) and the receiver (future).
- The future asks the data channel with fut.get() for the result of the calculation. The fut.get call is **blocking**.

```
#include <future>
#include <thread>
...

int res;
std::thread t([&]{ res= 2000+11;});
t.join();
std::cout << res << std::endl;              // 2011

auto fut= std::async([]{ return 2000+11; });
std::cout << fut.get() << std::endl;        // 2011
```

std::async behaves like an **asynchronous** function call.

- This function call takes a callable and its arguments.
  - std::async is a **variadic** template and can,
  - therefore, take an arbitrary number of **arguments**.
- The call of std::async returns a future object fut.
  - That's your handle for getting the result via fut.get(). Optionally you can specify a start policy for std::async.

- You can explicitly determine with the start **policy**
  - if the asynchronous call should be executed
    - std::launch::**deferred** - in the same thread or
    - std::launch::**async** - in another thread
- What's special auto fut= std::async(std::launch::deferred, ...)
  - is that the promise will not immediately be executed.
  - The call fut.get() **lazy starts** the promise.

- Promise associated with async**Lazy** is executed 1sec later than the associated with async**Eager**

```
#include <future>
...
using std::chrono::duration;
using std::chrono::system_clock;
using std::launch;

auto begin= system_clock::now();

auto asyncLazy= std::async(launch::deferred, []{ return system_clock::now(); });
auto asyncEager= std::async(launch::async, []{ return system_clock::now(); });

std::this_thread::sleep_for(std::chrono::seconds(1));

auto lazyStart= asyncLazy.get() - begin;
auto eagerStart= asyncEager.get() - begin;

auto lazyDuration= duration<double>(lazyStart).count();
auto eagerDuration= duration<double>(eagerStart).count();

std::cout << lazyDuration << " sec";                // 1.00018 sec.
std::cout << eagerDuration << " sec";               // 0.00015489 sec.
```

std::async should be your **first choice**

- The C++ runtime decides if std::async is executed in a separated thread.
- The decision of the C++ runtime may be dependent
  - on the number of cores,
  - the utilisation of the system or
  - the size of the work package.
- std::promise and std::future
- The **pair** std::promise and std::future give the **full control** over the task.

- Here is the usage of promise and future
  - prodPromise is moved into a separate thread and performs its calculation.
  - the future gets the result by prodResult.get().

```
...
#include <future>
...

void product(std::promise<int>&& intPromise, int a, int b){
  intPromise.set_value(a*b);
}

int a= 20;
int b= 10;

std::promise<int> prodPromise;
std::future<int> prodResult= prodPromise.get_future();

std::thread prodThread(product, std::move(prodPromise), a, b);
std::cout << "20*10= " << prodResult.get();                // 20*10= 200
```

- A future fut can be **synchronised** with is associated promise
  - by the call fut.wait().
- Contrary to condition variables,
  - no need of locks and mutexes,
  - spurious and lost wakeups are not possible.

- std::**packaged_task** enables you to build a simple wrapper for a callable,
  - which can later be executed on a separate thread.
  - Therefore four steps are necessary.

I. Wrap your work:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b)
{ return a+b; });
```

II. Create a future:

```
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```
sumTask(2000, 11);
```

IV. IV. Query the result:

```
sumResult.get();
```

Execution Policies

- The policy tag specifies whether an algorithm should run sequentially, in parallel, or in parallel with vectorisation.
  - **std::execution::seq**: runs the algorithm sequentially
  - **std::execution::par**: runs the algorithm in parallel on multiple threads
  - **std::execution::par_unseq**: runs the algorithm in parallel on multiple threads and allows the
    - interleaving of individual loops;
    - permits a vectorised version with SIMD3 extensions.

Execution policy in C++17 (2)

The execution policy

```
1  std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3  // standard sequential sort
4  std::sort(v.begin(), v.end());
5
6  // sequential execution
7  std::sort(std::execution::seq, v.begin(), v.end());
8
9  // permitting parallel execution
10 std::sort(std::execution::par, v.begin(), v.end());
11
12 // permitting parallel and vectorised execution
13 std::sort(std::execution::par_unseq, v.begin(), v.end());
```

Execution policy in C++17 (3)

```
#include <vector>
#include <algorithm>

#include <execution>

int main () {
int numComp= 0;

std::vector<int> vec={1,3,8,9,10};

//std::sort(std::parallel::vec, vec.begin(), vec.end(),
std::sort(std::execution::par, vec.begin(), vec.end(),
          [&numComp](int fir, int sec){ numComp++; return fir < sec; });
}
```

Open Multi Processing

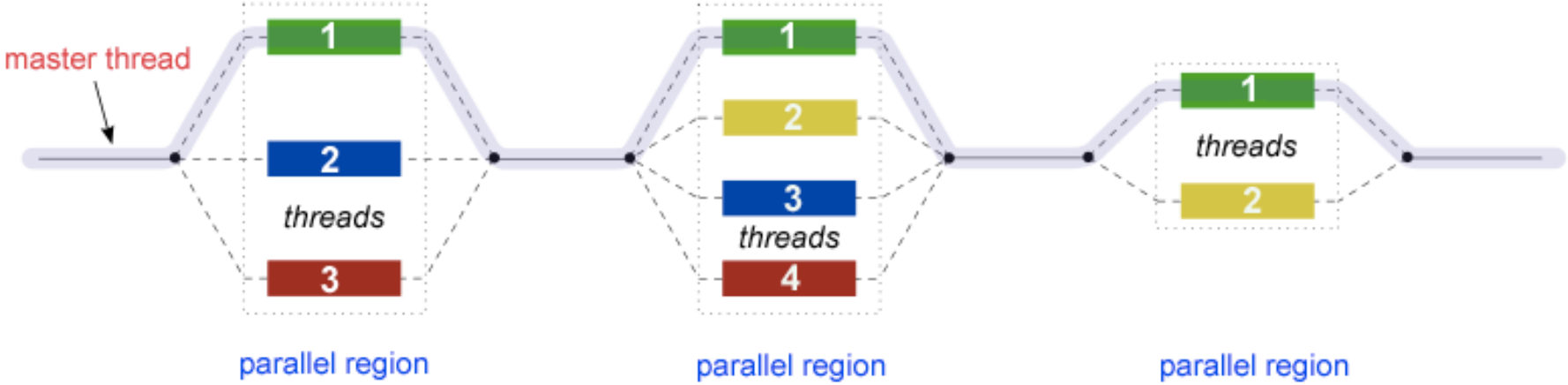
- Norma mantida pelo *OpenMP Architecture Review Board*
  - OpenMP 5.0 Specification (PDF) – Nov 2018
- **API** para expressar **paralelismo**
  - múltiplos fios de execução *(multi-threaded)*
  - memória partilhada
- Objetivos:
  - **normalização**
  - portabilidade
  - facilitar a utilização

- We need a complete discussion of OpenMP
  - But it is quite instructive to take a first look of the **fork-join** mechanism in OpenMP,
- Thread management in OpenMP is implemented with ***directives***,
  - followed by a code block to which the directive applies.
  - A directive together with its associated code block is called an OpenMP *construct*.

```
int main(int argc, char **argv)
   {
   ...
   #pragma omp parallel nthreads=2
      {
      // code to be executed by the
      // worker threads
      }
   ...
   }
```

- The listing above shows the parallel construct
  - When the main thread reaches the parallel construct, it suspends its ongoing task
    - joins the worker team and
    - all together executes the code inside the block
  - On exit from the block, when all workers have finished
    - the main thread resumes the suspended execution.
- No need to define a task function inside the code block
  - The task **function** will be **constructed** by the compiler,
  - A number of ***clauses*** added to the directive that inform the compiler how a parallel task must be constructed from the code block that follows
  - The statement nthreads=2 is a clause used to select the number of worker threads (including the master thread)
  - Other clauses have to do with the nature of the data items in the code block (shared? local to a thread?...)

- Criação **explícita** de blocos paralelos de código
  - executados por um grupo (*team*) de fios de execução



- No final de cada bloco:
  - todos os fios sincronizam (**barreira** implícita)
  - todos os fios exceto o principal deixam de existir

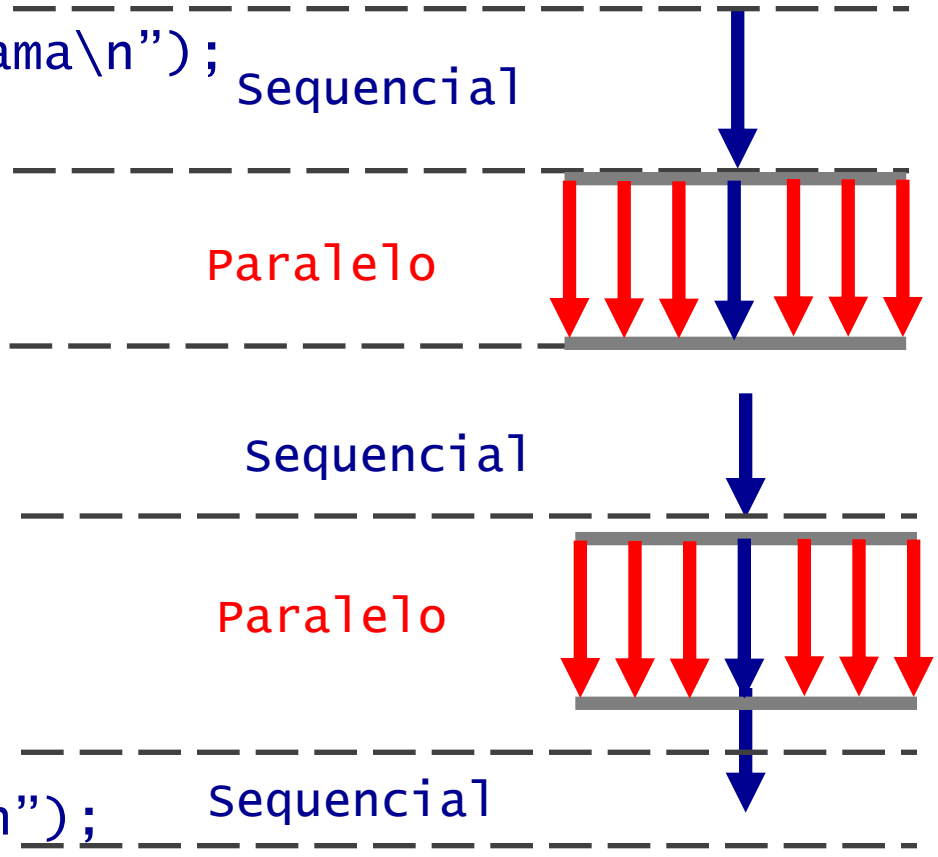
```
printf("Início  de programa\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("Fim de programa\n");
```



- Paralelismo especificado usando diretivas embebidas no código

```
#pragma omp <nome diretiva> [cláusula,...]
```

- Cada diretiva é aplicada ao bloco de instruções que se lhe segue

```
#pragma omp parallel
    {
      ... // bloco paralelo
    }
... // bloco sequencial
```

- Opções para que o compilador reconheça as diretivas
  - `gcc -fopenmp fichCodigo.c`
  - `icc -openmp fichCodigo.c`
- Se na compilação não for ativado o OpenMP
  - os ***pragmas*** são ignorados

```
#pragma omp parallel num_threads(2) {
#pragma omp for
        for (i=0; i<N; i++) A[i] = B[i] + C[i]; }
```

- deve estar dentro de um bloco `parallel`
- distribui as iterações do ciclo pelos fios ativos no grupo (*team)* atual:
  - o espaço de iterações (`i=0 .. N-1`, no exemplo)
  - é decomposto em fatias (*chunks*) consecutivas;
  - as fatias são distribuídos pelos fios
  - sem informação adicional
    - nada se pode assumir sobre o número ou tamanho das *fatias*,
    - nem sobre a sua distribuição pelos fios,
    - exceto que cada fio será responsável pela execução de pelo menos 1 fatia

- As diretivas `parallel` e `for` podem ser combinadas num só #pragma

```
#pragma omp parallel for num_threads(2)
        for (i=0; i<N; i++) A[i] = B[i] + C[i];
```

Seções (*Sections)*

- usada para dividir tarefas entre fios
- cada seção (*section*) executada por 1 e só 1 fio

```
#pragma omp sections [cláusulas, ...] {
#pragma omp section
        Bloco instruções 1
#pragma omp section
        Bloco instruções ..
```

- Um fio pode executar várias tarefas se houver mais tarefas que fios
- há uma **barreira** implícita no final da diretiva *sections*
  - É possível usar a cláusula **nowait**

- Por omissão os dados são **partilhados**, i.e.
  - as variáveis globais de um bloco paralelo
    - são acessíveis a todos os fios

- Dados **privados**:
  - as variáveis locais dentro de um bloco paralelo
  - as variáveis definidas no #pragma como privadas
  - os índices das iterações no bloco de uma diretiva `for`

Cláusula `private`:

- cada *fio* tem a sua cópia local de **`tid`**

```
main () {
 int tid;


 #pragma omp parallel private (tid)
 {
  tid = omp_get_thread_num();
  printf("Thread %d\n",tid);
 }
 printf("Fim de programa\n");
}
```

Por omissão as diretivas `critical` não têm nome

```
int x=0, y=0;
#pragma omp parallel
{
 #pragma omp critical
  x = x+1;
 #pragma omp critical
  y = y+1; }
```

As regiões **críticas** sem nome são consideradas a mesma região!
Se um fio está em `x = x+1`,
então nenhum outro fio
entra em `y = y+1`
e vice-versa

Distinguem-se regiões críticas

- dando-lhes **nomes**!

Duas regiões críticas distintas.
`x = x+1` e `y = y+1`
podem **co-existir** em paralelo

```
int x=0, y=0;
#pragma omp parallel
{
 #pragma omp critical C1
  x = x+1;
 #pragma omp critical C2
  y = y+1; }
```

- Uma operação de **redução** processa um conjunto de dados para a partir dele gerar um único valor,
- Exemplos
  - soma/máximo/produto de todos os elementos de um vector

```
int a[SIZE];
... inicializar a[]
int max=a[0];
#pragma omp parallel for
{
 for (i=0; i< SIZE ; i++)
   if (a[i]>max) max=a[i]; }
```

max é uma variável partilhada

muito ineficiente
Na verdade a execução é sequencial

```
int a[SIZE];
... inicializa   []
int max=a[0];
#pragma omp pa   llel for
{
 for (i=0; i< SIZE ; i++)
   #pragma omp critical
   if (a[i]>max) max=a[i];
}
```

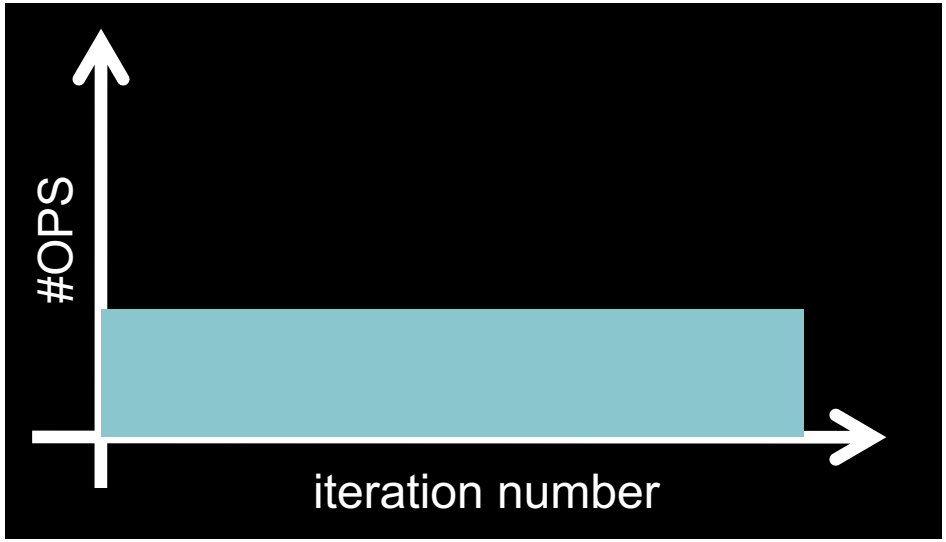
```
int a[SIZE];
... inicializar a[]
int max=a[0];
#pragma omp parallel
{int maxl = a[0];
 #pragma omp for
 for (i=0; i< SIZE ; i++)
   if (a[i]>maxl) maxl=a[i];
 #pragma omp critical
 if (maxl > max) max = maxl;
}
```

- A redução é tão comum que o OpenMP inclui uma cláusula específica.

```
int a[SIZE];
... inicializar a[]
int sum=0;
#pragma omp parallel for reduction (+:sum)
{
 for (i=0; i< SIZE ; i++)
   sum += a[i]; }
```

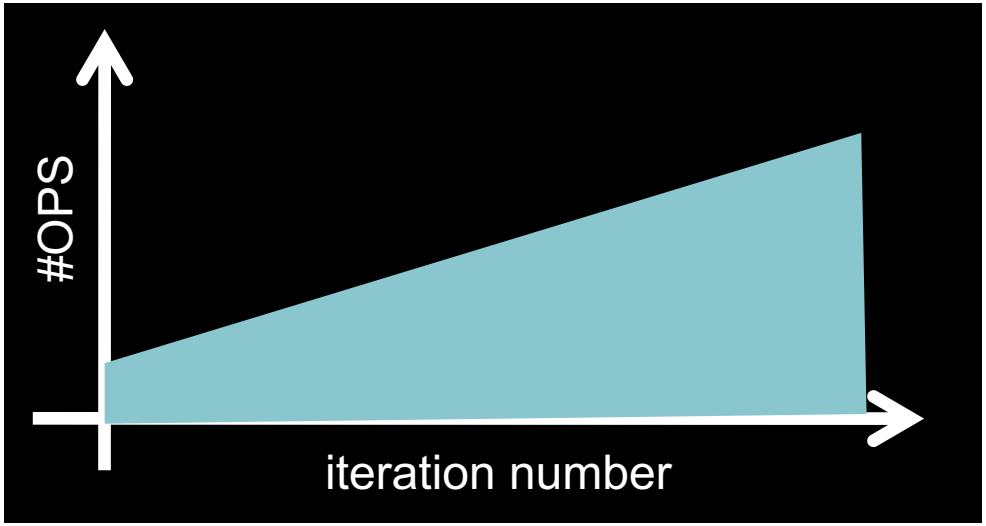
- Caso 1 - A quantidade de trabalho a realizar é igual para todas as iterações. Exemplo:

```
#pragma omp parallel for
{
 for (i=0; i< SIZE ; i++)
   b[i] = a[i]*a[i] + 10. / a[i]; }
```



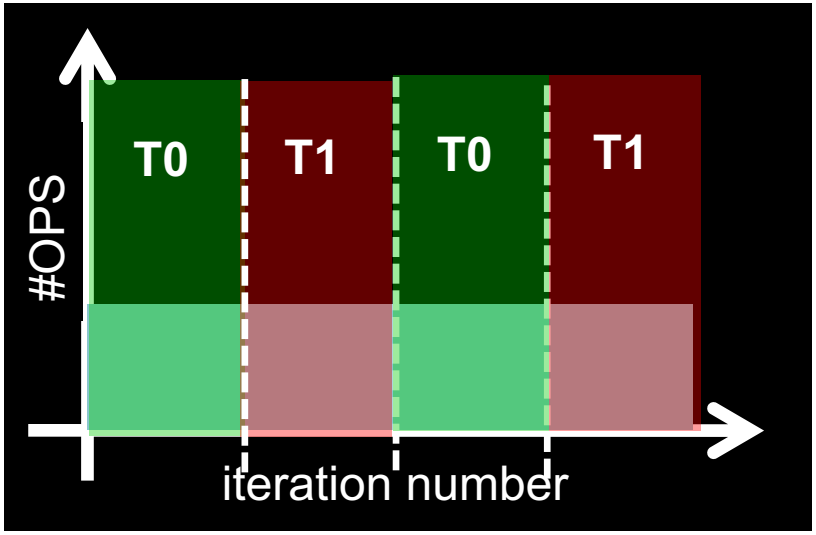
- Caso 2 - A quantidade de trabalho a realizar varia entre iterações. Exemplo:

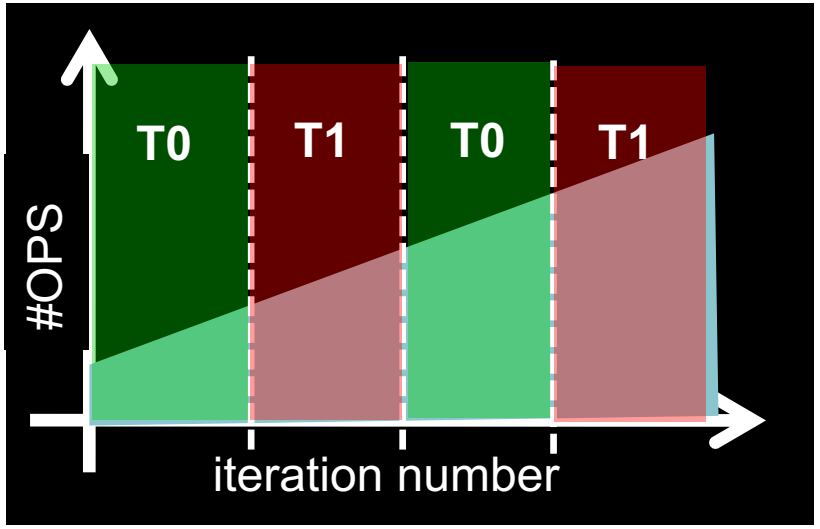
```
#pragma omp parallel for private (j)
{
 for (i=0; i< SIZE ; i++) {
   b[i] = 1.F;
   for (j=2 ; j<= i ; j++)
     b[i] *= j; } }
```



`#pragma omp parallel for schedule(static, <chunksize>)`

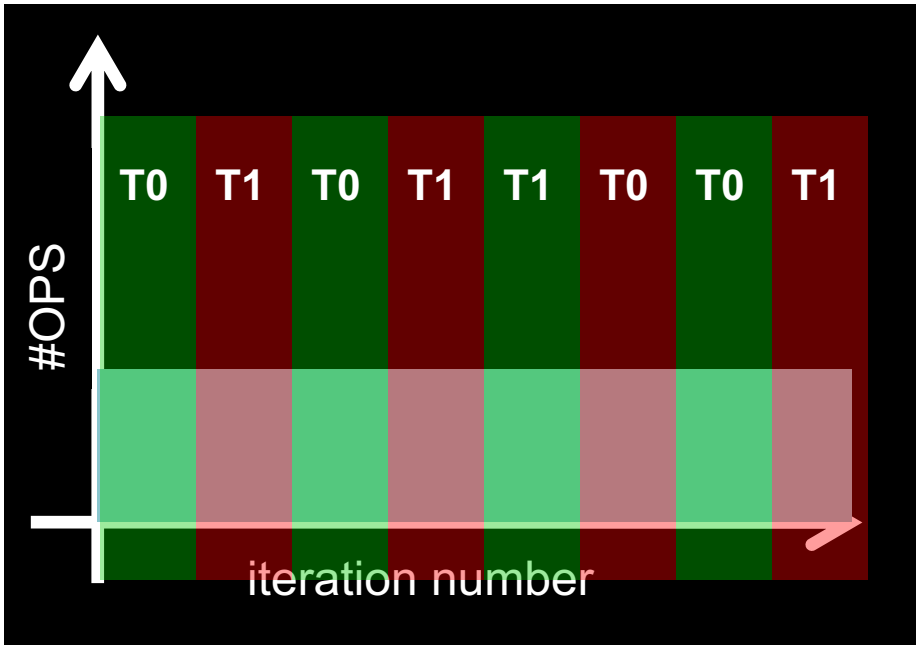
- O espaço de iterações é dividido em *chunks*, com *chunksize* iterações cada
- Os *chunks* são atribuídos às *threads* de forma **estática** usando ***round robin***, antes da execução do ciclo se iniciar
- Pode resultar em desbalanceamento de carga se a quantidade de trabalho variar entre *chunks*



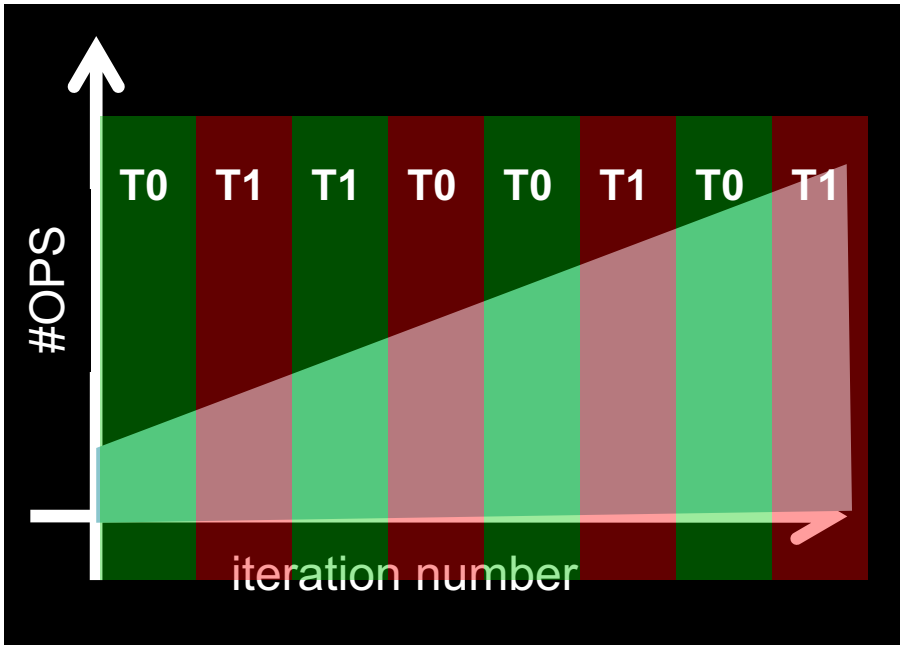


```
#pragma omp parallel for schedule(dynamic)
```

- O ciclo é dividido em muitos segmentos (*chunks*), todos com o mesmo número de iterações, e distribuídos pelas *threads* a pedido



static



dynamic

$$S_p = T_1 / T_p$$

p – número de processadores

T_1 – tempo de execução $p=1$

T_p – tempo de execução com p processadores

- indica quantas vezes é mais rápida a versão paralela com *p* processadores face à versão sequencial
- O desafio está na escolha de T_1:
  - deve-se usar o mesmo algoritmo mas apenas 1 processador?
  - deve-se usar o melhor algoritmo sequencial conhecido para aquele problema?

  A resposta depende claramente do que se pretende avaliar com este ganho!

- Linear

  $T(p) = T(1) / p \Rightarrow \lim (p \to \infty)\ T(p) = 0$

  $S_p = p \Rightarrow \lim (p \to \infty)\ S_p = \infty$

- Caso A – *no overheads*

  $T(1) = T_{seq} + T_{par} = 100\% = 1$

  $T(p) = T_{seq} + T_{par} / p \Rightarrow \lim (p \to \infty)\ T(p) = T_{seq}$

  $S_p = 1 / (T_{seq} + T_{par} / p) \Rightarrow \lim (p \to \infty)\ S_p = 1 / T_{seq}$

- Caso B – *overheads*

  $T(p) = T_{seq} + T_{par} / p + T_o(p) \Rightarrow \lim (p \to \infty)\ T(p) = T_{seq} + T_o(\infty)$

  $S_p = 1 / (T_{seq} + T_{par} / p + T_o(p)) \Rightarrow \lim (p \to \infty)\ S_p = 1 / (T_{seq} + T_o(\infty))$

$$E_p = S_p / p$$

p – número de processadores

S_p – *speed up* com p processadores

- indica em que medida os p **processadores** estão a ser bem rentabilizados
- Razão entre o *ganho* observado e o ideal (=p)
- A utilização total efectiva dos processadores resultaria numa eficiência de 100%