**(Lect 1)**

**Parallel computing**: the practice of using multiple processors in parallel to solve problems more quickly than with a single processor. It implies the capability of identifying and exposing parallelism in algorithms and software systems and understanding the costs, benefits, and limitations of a given parallel implementation.

Examples:

- **Compute Cluster**: multiple independent computers connected with one or more high-speed networks.
- **SMP** (Symmetric Multi-Processor): multiple processor chips connected to a shared memory hierarchy.
- **CMP** (Chip Multi-Processor aka Multicore): multiple compute units (called cores) contained on a single chip.

Historically, the motivation for parallel computing is to **improve performance**. Today because the entire computing industry has started to shift to multiple processors (multicore) in all systems.
The shift to CMPs has certified the end of the so-called *«Free Lunch Era»*.

**Moore's law**: 2X transistors/chip every 1.5 years → Over the years, microprocessor chips have become smaller, denser, and more powerful. The cost of processor chips and memories have reduced as well. Why bother with parallel programming for performance?

The primary limits of single-chip performance improvement related to packing more and more transistors into it are **speed of light** (fundamental insurmountable limit) and **power density** (power is increasing over the years)

**Dennard scaling law**: power density was constant as transistors got smaller… **False** for power wall!

Switching to multiple core to maintain power under control.

$Power_{dynamic} \approx \frac{1}{2} \times C \times V^2 \times F$ $\quad$ $Perf = Ncores \times F$ $\quad$ V= voltage , C= capacitance, F= clock frequency

But $V \approx F$ therefore $Power_{dynamic} \approx C \times F^3$ ($Power_{dynamic}$ is for each core)

If we double $Ncores$, we double $Perf$ but also $Power$. If we double $Ncores$, and we halve $V$ and $F$ we have the same $Perf$ but the power is reduced by a factor of four.

**Heterogeneous Multicore**: heterogeneous CMPs integrate different types of processor cores on a single chip (asymmetry). For example CPU and GPU or CPU with different power efficiency → assign different workloads to the appropriate core type to minimize power consumption.

**(Lect 2)**

Example of matrix multiplication (assume size of matrix $n = 2^k$)

Machine used: Peak = $(2.9 \times 10^9) \times 2 \times 9 \times 16 = 836$ GFLOPS (Clock 2.9 GHz, 2 Processor chips, 9 Cores , 8 double-precision operations per cycle)

(First version) Nested loops in **Python**: 21042 secs Gets only 0.00075% of the peak performance.
(Second version) Nested loops in **Java**: 2738 secs (8.8x speedup w.r.t Python)
(Third version) Nested loops in **C**: 1156 secs (2x than Java)

This is because Python is interpreted., C is compiled directly to machine code, while Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled to machine code.
The interpreter reads, interprets, and performs each program statement and updates the machine state. **Interpreters** are **versatile**, but **slow**.

**JIT compilers** can recover some of the performance lost by interpretation. When code is first executed, it is interpreted. The runtime system keeps track of how often the various pieces of code are executed. Whenever some piece of code executes sufficiently frequently, it gets compiled to machine code in real time. Future executions of that code use the more efficient compiled version.

*How to **improve performance**?*

**1. Change the loop order**: between best running time and worst (changing the loop order) we get a improvement by a factor of 18 → Hardware Caches.

Each processor reads and writes main memory in contiguous blocks, called **cache lines**. Previously accessed cache lines are stored in a smaller memory, called a cache, that sits near the processor.

In this matrix-multiplication code, matrices are laid out in memory in row-major order.


Different order of loops changes the exploit of spatial locality. With row-major order pass through a column of a matrix is the worst thing possible.

**2. Compiler optimization:** *Clang* provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize ( -O0 ... -O3 from don't optimize to maximum optimization)

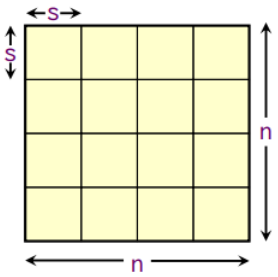
**3. Multicore Parallelism:** *We're running on just 1 of the 18 parallel-processing cores on this system*. The *cilk-for* loop (we will see) allows all iterations of the loop to execute in parallel.

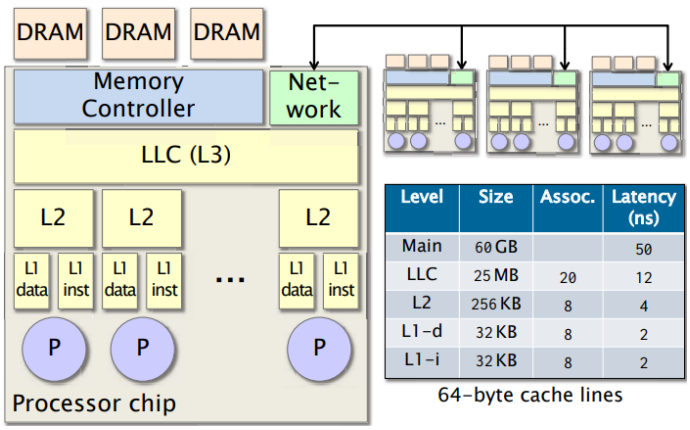
**Rule of Thumb**: parallelize out loops rather than inner loops (to avoid high overhead)

Now we are using 5% of peak performance. *How to keep improving?*

**4. Idea**: Restructure the computation to reuse data in the cache as much as possible → Instead of computing row, we compute blocks.

With this we reach 9% of peak performance




| Level | Size | Assoc. | Latency (ns) |
|---|---|---|---|
| Main | 60 GB | | 50 |
| LLC | 25 MB | 20 | 12 |
| L2 | 256 KB | 8 | 4 |
| L1-d | 32 KB | 8 | 2 |
| L1-i | 32 KB | 8 | 2 |

64-byte cache lines

**Multicore Cache Hierarchy**

Difference between L1 memory and main memory is huge.

Other idea: use **recursion**

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

8 multiplications of n/2 × n/2 matrices. 1 addition of n × n matrices.

We put a threshold, and we do recursion until we reach that and for the base case we use standard matrix multiplication. This permit to spawn less threads reducing the overhead of thread creation. With this version we reach 12.5% of peak performance.

**5. Vector Hardware:** Modern microprocessors incorporate vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion. Execute same operation on different data at the same time. *Clang/LLVM* uses vector instructions automatically when compiling at optimization level -O2 or higher, but it is conservative. Programmers can direct the compiler to use modern vector instructions using compiler flags (e.g. *-march=native*).

With **Compiler Vectorization** we reach 23.5% of peak performance.

Other option: Intel provides C-style functions, called intrinsic instructions, that provide direct access to hardware vector operations (use vectorization by ourself).

If we do a perfect manual vectorization we arrive at 41.5% of the peak performance.

**(Lect 3)**

**Classifying Parallel Architectures**

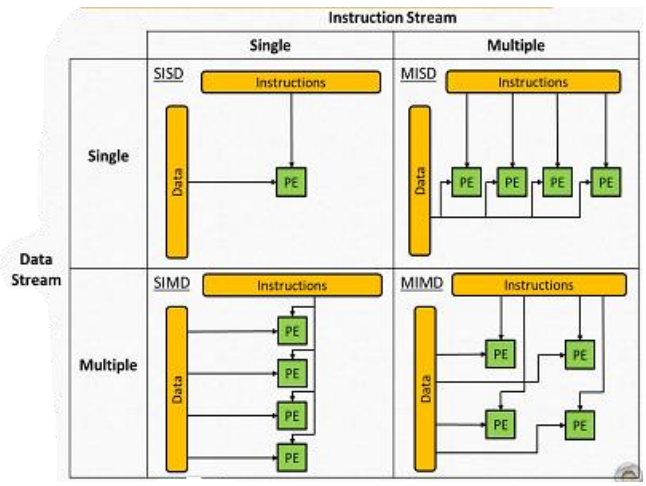
There does not exist a simple and definitive classification of parallel machines given the numerous types of systems with overlapping characteristics and features. Broadly speaking, parallel architectures can be classified using:

- **Flynn's taxonomy** (nowadays not so representative but still instructive)
- Considering the memory organization and also the core count
- Considering the interconnections among Pes *(processing elements)* and memory module

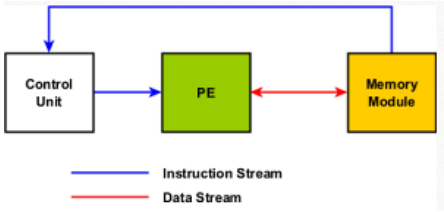
**Flynn's Taxonomy (1966)**

Classification based on the number of instructions and data streams:

- **SISD** (Single Instruction, Single Data) refers to the traditional von Neumann architecture where a single sequential processing element (PE) operates on a single stream of data.
- **SIMD** (Single Instruction, Multiple Data) performs the same operation on multiple data items simultaneously.
- **MIMD** (Multiple Instruction, Multiple Data) uses multiple PEs to execute different instructions on different data streams.
- **MISD** (Multiple Instruction, Single Data) employs multiple PEs to execute different instructions on a single stream of data (not used commercially).

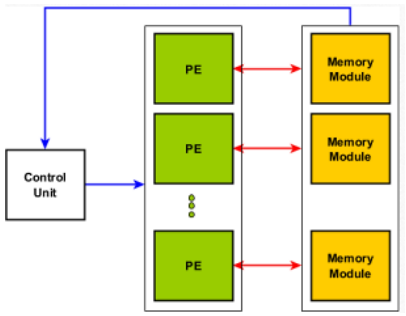


### Single Instruction, Single Data Stream (SISD)



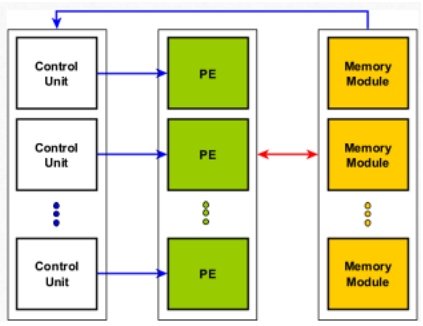

Serial (non-parallel) computer. Single stream of instructions executed serially. The processor executes a **single instruction** at a time operating on data stored in a **single memory**. Example: Uniprocessor unicore machine.

### Single Instruction, Multiple Data Stream (SIMD)




All CPUs execute the **same instruction** at any given clock cycle **on a different set of data**. Each CPU operates on different data streams (usually **each CPU** has an associated **data memory module**). The execution is **synchronous** (one instruction at a time). Modern GPUs resemble this type of parallel processing (array processors).

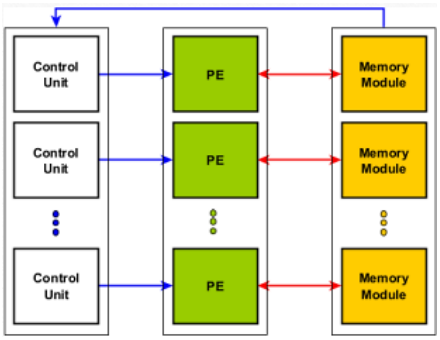
### Multiple Instruction, Single Data Stream (MISD)




All CPUs (can) execute a **different instruction** sequence on a **single data** stream.

This type of system is not commercially implemented. One application of this model is in mission-critical systems for fault tolerance reasons (select the answer with majority principle).

### Multiple Instruction Stream, Multiple Data Stream (MIMD)




A **set of processing elements** (processors) simultaneously executing **different instruction** sequences on **different data** streams (asynchronous). Each processor can execute all instructions. MIMD architectures are the most used parallel architectures.

### Classification based on the memory system

*Here we implicitly refer to MIMD parallel architectures, i.e., the most general ones.*

Considering the memory system we have: **Shared Memory architectures** and **Distributed Memory architectures**.

Shared Memory architectures are also known as **multiprocessors** while Distributed Memory architectures are also known as **multicomputers** (now we call distributed systems).

In distributed memory architectures sharing data is an IO operation. In shared memory we use *load* and *store* operation (not system calls).

The memory organization of shared-memory architectures can be:

- **uniform** (SMP – Symmetric Multiprocessor) [ Uniform Memory Access architecture (UMA) ]
- **non-uniform** (NUMA – Non-Uniform Memory Access)

In **NUMA** multiprocessors, the memory is physically distributed but logically shared and the memory access time is **asymmetric**.

Distributed Memory architectures are inherently NUMA (for a node accessing its memory is much faster than accessing memory of other nodes), and the address space of nodes is disjoint.

**Classification based on the cores count**

- $O(10^1 \div 10^2)$ cores, for a single multiprocessor chip (CMP)
- $O(10^2 \div 10^3)$ cores, for a Shared-Memory tightly-coupled multiprocessor (e.g. single node with 8 processors)
- $O(10^3 \div 10^5)$ for Distributed-Memory loosely-coupled systems, i.e., from small to large compute clusters
- $O(10^5 \div 10^6)$ top supercomputers

**Parallel architectures**

For **shared-memory architectures**, the emphasis is primarily on the **memory organization** (memory hierarchy, processor-memory interconnections). The aim is to minimize memory contention and reduce the von Neumann bottleneck.

For **distributed-memory architectures**, the emphasis is primarily on the **interconnection network** topology. The aim is to reduce communication costs (reducing latency, increasing available bandwidth).

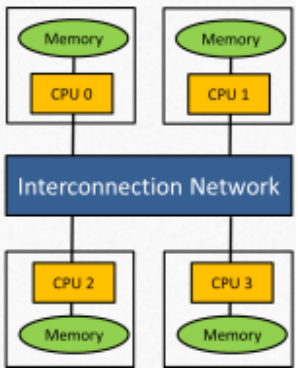
**Programming parallel architectures**

***Shared-Memory systems***: by exploiting the physical shared memory through thread-level parallelism (reference model: ***Shared Variables*** *programming model*, e.g., Pthread)

***Distributed-Memory systems***: by exploiting process-level parallelism (reference model: ***Message Passing*** *programming model* , e.g., POSIX socket) (there were attempts to provide a transparent shared-memory software abstraction atop distributed-memory system, but they were not scalable because the cost of the software abstraction is too high, it works with no more than 8 nodes).

Message passing is more general because we can share messages even in shared-memory systems, why not use only this model? Today we mixed the two models, we use threads internally and messages between processor running on different machine.

**Distributed Memory Systems**



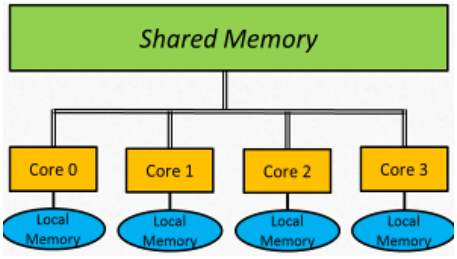

Each node is a complete computer system. Nowadays, it is a CMP (Chip Multi-processor, aka multicore).

Processes on different nodes communicate explicitly by sending messages across a network.

Depending on the network (and other aspects), we can further classify distributed systems: *Clusters, Cloud, geographically distributed systems, …*

We are interested in systems with high-performance network topologies, i.e., **compute clusters** (small distance, orders of meters, and very fast connection). Usually the nodes are **homogeneous** (same processor, same amount of memory).

**Shared Memory Systems**




Parallel architecture comprising a (modest) number of cores, all with **direct hardware access** to a **shared memory** space (SHM).

In addition to the SHM, each core contains a **smaller local memory** (e.g. L1 -cache) to reduce expensive accesses to main memory (von Neumann bottleneck)... *but they also introduce issues.*

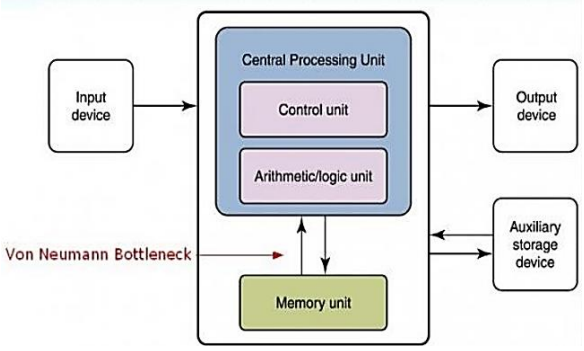
**Shared vs Distributed Memory systems**

Distributed Memory systems are **more scalable**, but **more costly** and **less energy efficient**.

From the standpoint of the **runtime system programmer**:

- For **shared memory system** the physical shared memory can be used directly for fast synchronization and communication between processes/threads. However, efficient management of **locking** and **synchronization** is generally a **critical aspect**.
- For **distributed memory system**, the most important aspect is to **reduce** as much as possible the **cost of communications** (i.e. I/O) through overlapping of I/O and computation, reduce memory copies, using fast messaging protocols.

**(Lect 4)**




In early computer systems timings for accessing main memory and for computation were reasonably well balanced. During the past few decades, computation speed **grew much faster** than main memory access speed, resulting in a significant performance **gap**.

**Von Neumann Bottleneck**: discrepancy between CPU compute speed and main memory (DRAM) speed

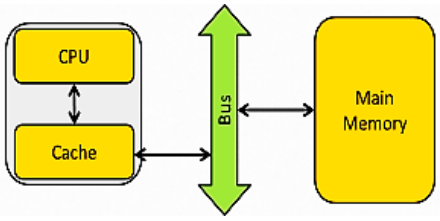
**Example**: imagine having a CPU with a peak performance of 384 Gflops/s and DRAM peak memory transfer rate equal to 51.2 GB/s. Let's consider the Dot Product, if n = $2^{30}$ we have $2 * n$ floating point operations (i.e. 2 GFlop) and $2 * n * 8\ B$ = 16 *GB (two number)* data transferred from memory, then:

$$t_{comp} = \frac{2\ GFlop}{384\ GFlop/s} = 5.2ms\ , \qquad t_{mem} = \frac{16\ GB}{51.2\ GB/s} = 312.5ms$$

If we overlap computation and memory data transfer, a lower bound of the execution time is the maximum between $t_{comp}$ and $t_{mem}$, hence 312.5 ms. So, the achievable performance is (2 GFlop / 312.5 ms) = 6.4 GFlops/s, (2 GFlop every 312.5 ms) i.e. less than 2% of peak compute performance.

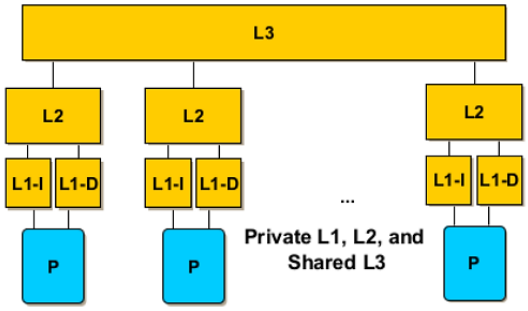
Considering our architecture, «Dot Product» is **memory bound**.

**Basic CPU structure with a single Cache**




CPUs typically contain a **hierarchy** of two or three levels of **cache** (L1, L2, L3). Caches have **higher bandwidth** and **lower latency** compared to main memory but **much smaller capacity**, it is a **trade-off** between capacity and speed. Caches could be **private** for a single core or **shared** between several cores.

The most common cache organization is the following:




Let's what happen when we add a cache with a matrix multiplication example, using the same CPU and DRAM as the example seen early. The in-chip shared cache of capacity is 512 KB (SRAM).

$W = U \times V$ square matrices with n = 128

Total size of the matrices: $128^2 \times 3 \times 8B = 384$ KB (fits in Cache)

Data transfer time (from/to cache): $t_{\text{mem}} = \frac{384\ KB}{51.2\ GB/s} = 7.5\ \mu s$

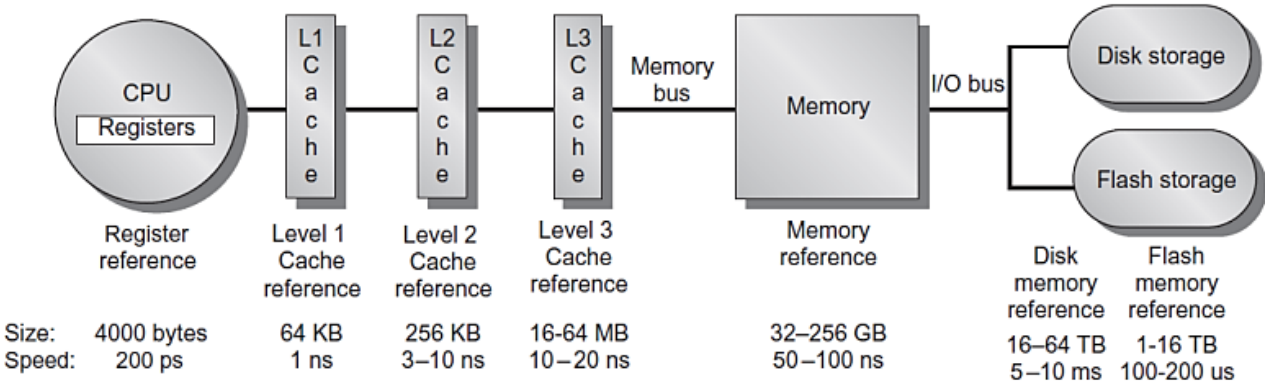
Computation time: $t_{\text{comp}} = \frac{2^{22}\ Flop}{384\ GFlop/s} = 10.4\ \mu s$ (Total operations: $2 \cdot n \cdot n^2 = 2 \cdot 128^3 = 2^{22}$ Flops)

Execution time: $t_{\text{exec}} \geq 7.5\ \mu s + 10.4\ \mu s = 17.9\ \mu s$

In this case $t_{mem} > t_{comp}$ thus this matrix multiplication is **compute bound**.

Achievable performance (upper bound): $\frac{2^{22}\ Flop}{17.9\ \mu s} = 223 GFlop/s$ (60% of peak perform.)

**Memory Hierarchy**




*The transferred data is called Cache line (for cache), RAM pages (for RAM), Disk blocks (for disk)*

**The Locality Principle**

The locality principle is the driving force that makes the memory hierarchy (caches) work properly. It increases the probability of reusing data blocks that were previously moved from level n to level n-1 (i.e., closer to the CPU), thus reducing the MissRate.

**Temporal Locality** (data reuse): refers to the property of a program to repeatedly access the same memory locations over a short period of time.
Property of the data access pattern, the cache mapping strategy (direct cache vs associative cache) and the replacement algorithm (LRU, Random) have an impact on temporal locality.

**Spatial Locality**: refers to the property of a program to access memory locations that are spatially close to each other. Enforced by moving data in blocks between levels of the memory hierarch.

**Measuring $CPU_{time}$ with caches**

$$CPU_{time} = ClockCycles * ClockCycleTime = IC * CPI * ClockCycleTime$$

IC (Instruction Count) is the number of program instructions executed.

IC can be further detailed as IC = $IC_{CPU}$ + $IC_{MEM}$, the former are ALU instructions (e.g., register – register), the latter are memory access instructions (e.g., load, store)

CPI is the average *ClockCycles Per Instruction*, and is defined as CPI = $(ClockCycle/IC)$

$$\text{CPI} = \left(\frac{IC_{CPU}}{IC}\right) * CPI_{CPU} + \left(\frac{IC_{MEM}}{IC}\right) * CPI_{MEM}$$

where $CPI_{CPU}$ are the average cycles per ALU instruction and $CPI_{MEM}$ are the average cycles per memory instruction.
Considering that each memory instruction may generate a cache hit or miss with a given probability, and given the HitRate the probability of a cache hit, we have:

$$CPI_{MEM} = CPI_{MEM-HIT} + (1 - HitRate) * CPI_{MEM-MISS}$$

*(Slide 13) Example of application of the formula*

**Cache Algorithms**

Cache hierarchy is managed by a set of caching policies (cache algorithms) that determine which data is cached during program execution.

**Cache hit**: The data is present in the cache **Cach miss**: data is **not** present in the cache

**Miss penalty**: the time spent transferring a cache line into the first level cache and the requested data to the processor.

**Hit ratio**: the percentage of data requests resulting in a cache hit.

Caches are organized into a number of **cache lines**, typically with a size of 64B.

Cache **mapping strategy** decides in which location in the cache a copy of a particular entry of main memory will be stored:

- **Direct-Mapped Cache**: Each block from main memory can be stored in **exactly one** cache line (high miss rates, thrashing problem, no temporal locality for cache line replacement)
- **n-way Set Associative Cache**: Each block from main memory can be stored in **one of n** possible cache lines (higher hit rate at increased complexity) *(the most used, with n=4)*

*When the cache is full how to decide which data do we evict?* **Least Recently Used** (**LRU**): commonly used policy to decide which of **several possible locations** to choose is based on temporal locality.

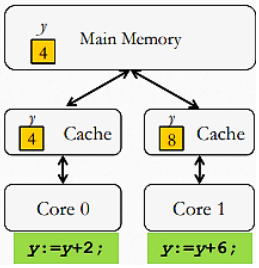
**Cache Write Policies**

When a CPU writes data to the cache, the value in the cache may be inconsistent with the value in main memory.

**Write-through policy**: caches handle this by updating the data in the main memory when it is written to the cache. Always implemented with a store write buffer.

**Write-back policy**: caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory. A store write buffer is generally used to reduce the cost of cache writes.

### Cache Coherence problem




With private caches per core, it is possible to have several copies of shared data in distinct caches, each cache stores a different value for a single address location.

Cache inconsistency: the two caches store different values for the same variable.

**Cache coherency protocols** are required. A prominent example is the **MESI** protocol, which marks a cache line using four possible states: Modified (M), Exclusive (E), Shared (S), and Invalid (I).

The granularity of the cache coherency protocol is the cache line. All modern CMPs provide automatic cache coherence protocol.

**(Lect 5)**

### Superscalar processors

Modern CPUs are highly parallel (and quite complex), mixing pipelining and superscalar technologies.

Superscalar CPUs are designed to **execute multiple instructions** from a **single process/thread simultaneously** to improve performance and CPU utilization.

The processor fetches **multiple instructions concurrently** in a single clock cycle and executes them **out-of-order** (i.e., as soon as operands are available) to keep high utilization of the execution units. Results are then re-ordered to ensure they are written back to the register file or memory in the correct program order. The **problem** is that in **sequential programs**, the **number of instructions** that are **independent are small** thus, the **exploited parallelism is low**.

To overcome such low efficiency, **SMT** (Simultaneous Multi-Threading) has been added in superscalar processors to execute **multiple instructions** from **multiple threads** of control simultaneously.

### HW Multithreading

HW multithreading enables a single core to execute multiple threads concurrently (a thread represents an independent sequence of instructions).

There are two main types of HW multithreading:

- **fine-grained multithreading**: interleaves instructions from different threads at the instruction level in a single clock cycle *(most common)*
- **coarse-grained multithreading**: switches between threads at the thread level. A switch happens only when the thread in execution causes a stall.

Each thread has its own **set of registers** and **program counter** (PC), representing its state. The processor maintains the **context** of each thread to **quickly** switch between them. This context includes the values of registers, PC, and other state information. For the OS, each context is a **logical core.** Typically, 2-4 contexts per core.

**SMT**: the superscalar processor interleaves the execution of instructions from **different threads**. Instructions are simultaneously issued from multiple threads at each clock cycle to the execution units. *Intel SMT is called Hyperthreading.* *(Slide 5-6) Example of our machine topology.*

### Programming Shared Memory Systems

We create parallelism starting threads running concurrently on the system. We prefer thread instead of process because thread creation is more **lightweight** and **faster** compared to process creation.

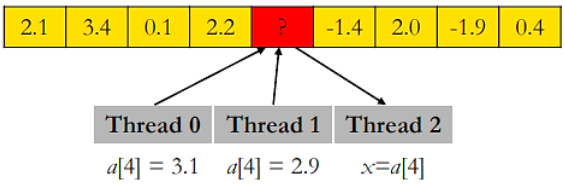
Creating a thread takes O($10^4$) clock cycles in C/C++ (from three to five times faster than a process) because, for example, fork system call requires copying more data (e.g., page table *[shared for thread]*)

Furthermore, process does not share memory unless we use POSIX command, while exchange of data is implemented by threads reading from and writing to shared memory locations.

### Sharing memory *(using load [read] and store [write])*

**Data Race** (**DR**): occurs when two (or more) threads access a shared variable simultaneously, with at least one doing a write operation, and the accesses to the shared variable are not separated by synchronization operation.

Data races produce **non-deterministic behavior**. Debugging is hard. To **avoid data races**, we need to use **synchronization mechanisms** *(mutexes, condition variables, semaphores, atomic instructions)*.



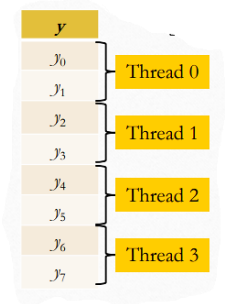

### False Sharing

Caches are organized in cache lines, where each line holds several values (typically 8-16 memory words). The **cache coherence protocol** works by invalidating/updating the **entire cache line** and not single values within the cache line.

If distinct data items (e.g., variables) present in a single cache line are written by different threads, we have the **false sharing effect** (or cache line ping-pong). Different threads are using same cache line but no true sharing actually exists since threads work on different variables. Each time the thread access to an invalidate line of cache it has to **reload it from memory** (creating an **overhead**), but this is not necessarily because the variables **are not really shared**.

The overhead introduced by the cache coherence protocol to keep the cache line consistent among all copies can be significant. This is an indirect effect produced by the coherence protocol that operates at cache line granularity.

Example:




```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (int j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

Consider m = 8, and a cache line size of 64 B
→ y is stored in a single cache line

4 threads, one per core, each thread computing 2 elements of y.

False Sharing: every write to y invalidates the cache line in the other core's cache. Most of these updates to y force main memory accesses

There are two possible solutions:

- **Use temporary variable**: use register variables *(= local variable)* and make a single store in the end when the value is fully computed (best solution)
- **Use data padding**: add space between the two variables (separating the false shared variable in different cache lines). This is a hardcoded solution, and it is difficult to extend to different machine with different cache line sizes.

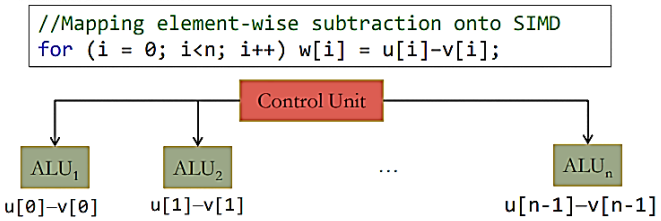
```
for (i = 0; i < m; i++) {
    float _y = 0.0;
    for (int j = 0; j < n; j++
        _y += A[i][j] * x[j];
    y[i]= _y;
}
```

```
// forcing ying and yang to lay in
// two distinct cache lines
struct pack_t {
   uint64_t  ying;
   char padding[CACHELINE_SIZE_BYTE -
                sizeof(uint64_t)];
   uint64_t  yang;
};
```

To understand if false sharing is happening, we can debug the code with Valgrind, analyzing miss rate and watching computation time. Compiling with optimization flags can remove false sharing problem automatically, but in many cases the compiler is not able to recognize false sharing.

**More on SIMD** (Single Instruction Multiple Data)

All PEs/CPUs execute the same instruction at any given clock cycle on a different set of data. The execution is synchronous (lockstep execution). All ALUs are required to execute the **same instruction or stay idle**.




If we don't have as many ALUs as data items, we divide the work and process iteratively (*[amount of data] / [number of ALU]* cycles of clock).

Modern CPU cores typically contain a vector unit that can operate a number of data items in parallel.

**SIMD with AVX registers**

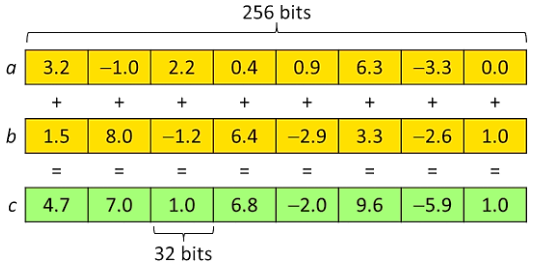
**Intrinsics** are assembly-coded functions that can be used in C/C++ code instead of assembly instructions. Not all intrinsic functions directly map to a single assembly instruction, some of them can be implemented using several assembly instructions. Intrinsics are **expanded inline** eliminating function call overhead. Intrinsics provide access to instructions that **cannot be generated using the standard constructs** of the C and C++ languages.

**Advanced Vector Extensions** are SIMD extensions to the x86 instruction set architecture. The intrinsic function allows to use the AVX instructions.

Example:

```
//AVX-Programming with C/C++ Intrinsics
__m256 a, b, c;          // declare AVX registers
...                      // initialize a and b
c = _mm256_add_ps(a, b); // c[0:8] = a[0:8] + b[0:8]
```

*We do 8 instructions in one cycle of clock, so we reduce by a factor of 8 the computational time of this simple loop.*



**SIMD with AVX2 registers**

```
//AVX2-Programming with C/C++ Intrinsics
__m512 a, b, c;          // declare AVX2 registers
...                      // initialize a and b
c = _mm512_add_pd(a, b); // c[0:8] = a[0:8] + b[0:8]
```

packed double-precision floating point

intrinsic operation: add, sub, load, set, ….

Uses 512-bit registers *(the biggest in modern architecture)*.

Compiler uses these registers to speed up computation vectorizing. Typically, the compiler is capable to autocompile the code using vectorization of data, generating intrinsics instruction.

**(Lect 6)**

*(Slide 16)* Matrix multiplication (with transposing) using SIMD instruction (single thread)

*(Slide 18)* Details of the **hsum_avx(__m256 v)** function (sum of element of an array): divide a 256 bit register in two 128 bit register so that we can apply SIMD summation to this two register. On the result (a 128-bit register) we apply **hsum_sse3(__m128 v)** function. This function moves the values in the register, creating temporal register, and does SIMD summation. At the end the first element of the result register will be the sum of all the element in the initial register.

Using this version of matrix multiplication, we obtain a speedup of 5.8 times w.r.t. the plain version of matrix multiplication using transpose.

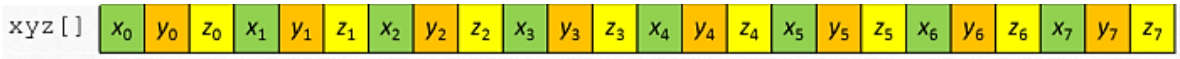
**AoS and SoA layouts**

To exploit the power of SIMD parallelism, it is often **necessary to modify the layout** of the data structures used.

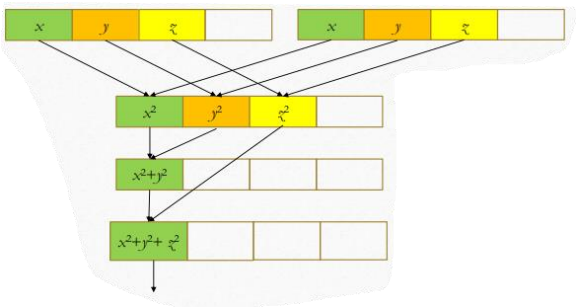
*The case study (in the pictures) is a collection of n values representing 3D coordinates and vector normalization code.*

Two different layouts:

- **AoS** (Array of Structures): stores records consecutively in a single array




- **SoA** (Structure of Arrays): uses one array per dimension. Each array only stores the values of the associated element dimension.


Vectorization of 3D vector normalization based on the **AoS format** is relatively **inefficient**. Vector registers would **not** be **fully occupied** for 128-bit registers, we use three out of four available vector lanes. Summing up the squares requires operations between neighboring horizontal lanes, resulting in only a single value for the inverse square root calculation. Scaling to longer vector registers becomes increasingly inefficient.

*(Slide 23)* Using **SoA format** we can normalize eight vectors at a time.

Moving from AoS format to SoA format can be parallelized using intrinsic operation so the cost is lower. Despite the data movement, in fact, in out example we are still gaining time.

*(Slide 25)* Snippet of code to move from AoS to SoA using vectorized **shuffling._mm256_shuffle_ps(M03, XY , _MM_SHUFFLE( 2,0,3,0))** extract element at index 0 and 3 from the lower part and the upper part of M03 (considering the 256-bit register as two register of 128-bit). Same thing with XY using index 0 and 2. The extracted element are then ordered with first the lower part of M03 and XY followed by higher part of M03 and XY, giving us the X array in SoA format.

**Auto-vectorization using compiler's options**

Auto-vectorization is a compiler optimization technique that transforms scalar into vectorized code in order to improve performance on modern CPUs leveraging SIMD instructions.

Compler flags in GCC:

- -O3: includes options to enable auto-vectorization (-O2 in recent GCC versions)
- -ftree-vectorize: enable vectorization optimizations (for example to vectorize using -O1)
- -fopt-info-<option>: enable optimization dumps from various optimization passes (a report to understand what is been optimized and what not).
  -fopt-info-vec-all: enable dumps for all vectorization optimization

-O3 -march=native and –ffast-math –mavx2 often do the trick to activate auto-vectorization.
( –ffast-math tells the compile that we are willing to lose some precision in order to speedup)

**General guidelines for automatic loop vectorization**

The compiler is conservative, to be sure it vectorize the code we need to follow some rules:

Avoid dependencies between loop iterations and avoid 'read-after-write' dependencies
A[0] = 0; for (int_i = 1; i < N; ++i) A[i]= A[i-1] + 1; **// will not be vectorized!**

The number of iterations must be known at entry of the loop, and it can't be change dynamically.

The exit of the loop must not be data-dependent. Avoid conditional 'break'.

Avoid 'switch'/'return' statements inside the loop. The 'if' statement can be vectorized if it can be implemented as a masked assignment (computes both values and select only the truth values with respect to the condition).

No function calls. If a function call can be inlined this is OK. Intrinsic math functions (sin, sqrt, ...) are allowed. Library functions inside the loop body may prevent vectorization.

Avoid pointer aliasing (i.e., two or more pointers refer to the same memory location). The type qualifier '**__restrict__**' (is not standard but GCC accept this keyword) tells the compiler that no other pointer can modify the same memory location.

**(Lect 7)**

**Interconnection Networks**

Interconnection networks for parallel systems share many technical features of WAN, but they have very different requirements.

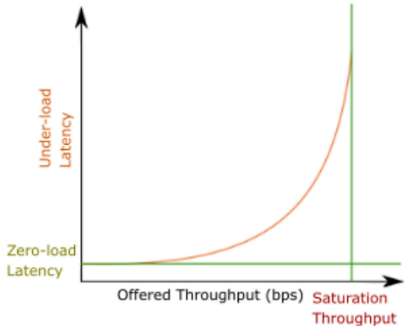
Usually, $O(10^1 \div 10^5)$ nodes connected to the same network infrastructure. Distances from $O(10^0)$ to $O(10^1)$ meters.

The general requirement is to provide the user with **low message latency** and **high bandwidth** at an acceptable cost. Usually, the cost is relatively high depending on the scale and the performance of the network.

**Network Performance Metrics**:

- **Latency**: the time lapse between when a packet starts to be transmitted from the source node and when it is wholly received at the destination node. Expressed in time units.
    - **No-load** (or **zero-load**) **latency**: latency experienced in a network when **there is not traffic**. It represents the baseline performance of the network.
    - **Under-load latency**: latency experienced when the network is utilized and the **traffic is below the saturation point** (i.e., the load is below the network capacity)
- (**Offered**) **Throughput**: the actual amount of data sent into the network per unit of time. Expressed in bits per second (bps) or multiples (Kb/s, Mb/s, Gb/s).
    - **Saturation Throughput**: the maximum amount of traffic sustained by the network. It is the point at which the network is fully utilized.
- **Bandwidth**: the theoretical maximum data transfer rate under ideal conditions across a given network path. It represents the max capacity of a communication channel.

The side figure sketches the general behavior of **latency vs. offered throughput** (i.e., injection rates). As the nodes keep injecting traffic into the network, it reaches a **saturation point** where the latency **grows exponentially** due to contention.




The latency depends on network contention and the distance between the source and the destination node. Therefore, high network contention and/or longer distances will produce more latency.

**Basic network terminology**:

- **Endpoints**: they are the sources and destinations of messages (we consider endpoints the computing nodes. In a distributed system, a computing node is equipped with a NIC [Network Interface Card] that sends data to and receives data from the network on behalf of the processor)
- **Switches**: a switch is a device connected to a set of links. It transmits received packets to one or multiple links. It doesn't have computing power.
- **Links**: it is a physical connection (a wire) used to transfer data between endpoints, endpoints and switches, and between switches.

**Direct networks**: the nodes are both endpoints and switches (the nodes are directly connected to other nodes without intermediate).

**Indirect networks**: the endpoints are connected indirectly through switches.

**Degree**: the degree (deg) of a network is the maximum number of neighbors of any node.

**Diameter**: the diameter (diam) of a network is the length of the longest of all shortest paths between any two nodes

**Bisection-width**: the bisection-width (bw) of a network is the minimum number of edges (or links) to be removed in order to disconnect the network into two halves of equal size (in case of an odd number of nodes, one of the two halves can include one more node).

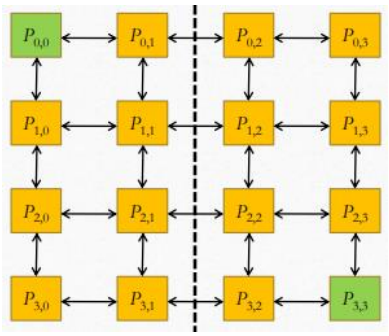
**Example of topologies**

**Linear Array**


Assume a linear array with n processors, denoted as $L_n$

*degree*($L_n$ ) = 2 *diameter*($L_n$ ) = n−1 *bw*($L_n$ ) = 1

**2D Mesh**



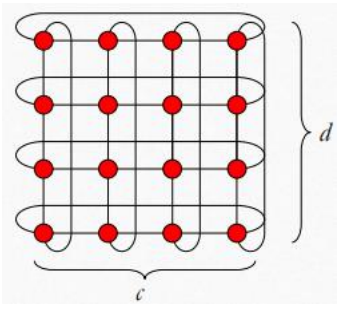

M(d, d) has $n = d^2$ processors (endpoints)

*degree*( M(d, d) ) = 4 O(1)

*diameter*( M(d, d) ) = 2(d−1) $O(\sqrt{n})$

*bw*( M(d, d) ) = d $O(\sqrt{n})$

**2D Torus** *(most common)*




A **Torus** T(c, d) is a Mesh augmented by wraparound edges at the border of the mesh.

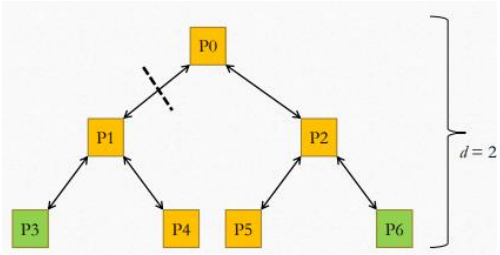
T(c, d) has $n = c \cdot d$ processors/endpoints

*degree*( T(c, d) ) = 4 O(1)

*diameter*( T(c, d) ) = d/2 + c/2 $O(\sqrt{n})$

*bw*( T(c, d) ) = min{2·c, 2·d} $O(\sqrt{n})$

**Binary tree**



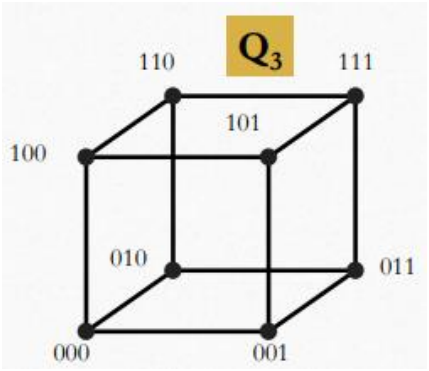

BT(d) has $n = 2^{(d+1)} - 1$ processors

*degree*( BT(d) ) = 3 O(1)

*diameter*( BT(d) ) = 2d O( log(n) )

*bw*( BT(d) ) = 1 O(1)

**Hypercube**




The **Hypercube**, denoted by $Q_d$ ($d \geq 1$), is the graph that has vertices representing the $2^d$ bit strings of length d. Two vertices are **adjacent** if and only if the bit strings that they represent **differ** in **exactly** one bit position.

HC(d) = Hypercube of degree d  Number of processors: $n = 2^d$

degree( HC(d) ) = d  O( log(n) )

diameter( HC(d) ) = d  O( log(n) )

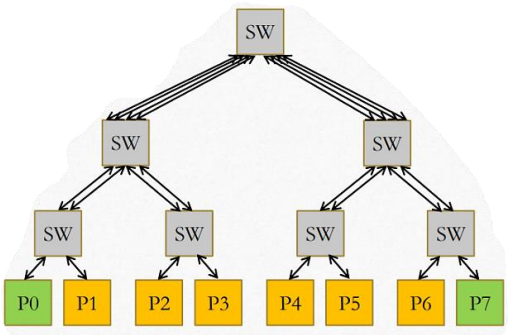
bw( HC(d) ) = n/2  O(n)

**Criteria to Evaluate Network Topologies**

We want:

- **Low Diameter**, in order to support efficient communication between any pair of processors.
- **High Bisection Width**: a low bisection width can slow down many collective communication operations and thus can severely limit the performance of applications.
- **Constant degree** (i.e. independent of network size): allows a network to scale to a large number of nodes without the need to add an excessive number of connections.

**Fat Tree**



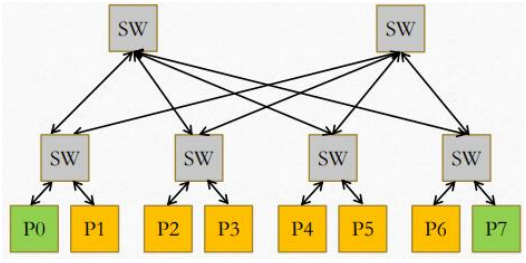

**Indirect tree topology** in which the endpoints are the leaves, and all other nodes are switches.

The characteristic is that each switch has **the same number of links going up and going down** (links are increasing, getting fatter, towards the root)

The idea is to keep the **bandwidth constant at each level** of the tree (i.e., we have the same number of links at each level)

The main **problem** is the **cost**! It increases with the depth of the tree. Top-level switches have too many links, not realistic implementation.

Implemented in a **different way**, using **switches with a limited degree** (*k-ary n-fly* network topologies and *folded clos* network topologies *[we will not see this]* ).



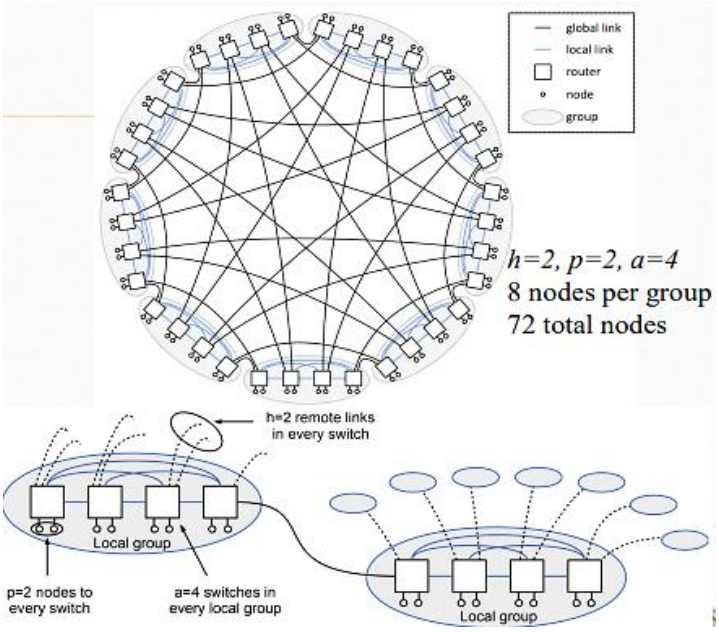

In the figure a 2-Level Fat Tree (i.e., 2 levels of switches)

Each switch has the same number of ports (4 in the picture). Low-cost solution, limited by the number of ports of the switch.

To scale out the network we can increase the number of levels.

For a 2-level Fat Tree where each switch has k ports:

$k + k / 2$ switches in total  $n = k * k / 2$ maximum number of nodes




**Dragonfly**

**Multi-level network**.
Three levels: Router, Group, System

Each router has connections to ***p*** endpoints, ***a-1*** local channels (to other routers in the same group) and ***h*** global channels (to routers in other groups).

A group consists of ***a*** routers. Each group has ***a·p*** connections to endpoints (i.e., fully connected topology) and ***a·h*** connections to global channels.

*The box is a switch (router), the circle is a processor machine (node), the grey oval is a group.*

Routing is easy, if destination node is not inside the group send it to the router connected to the destination group. It will send it, and the router of the destination node group will deliver it to the router at which the destination node is connected.

**A simple communication cost model**

The data transfer time (or **communication time**) can be described by a simple **linear model**:

$$T_{comm} = t_0 + n \times s \approx \begin{cases} t_0 & for\ small\ n \\ n \times s & for\ large\ n \end{cases}$$

- $t_0$ is a constant term corresponding to the setup time
  - It can be easily measured, if the network is empty, sending some shortest message (e.g., 1-byte payload, i.e., n=1) to from any node to any other node and computing the average time. Sometimes is called latency.
  - Its value may be different for different programming models (e.g., because of data copies, or sync vs async operations)
- $n$ is the amount of data to be transferred (i.e., number of bytes)
- $s$ is the transmission cost. Usually, s = 1/$B$ where $B$ is the available bandwidth along the transmission path.
  - s is limited by the slowest part of the path between the sender and receiver processes
  - s includes both software contributions (i.e., the rate at which the application can feed the network) and hardware contribution (i.e., the rate at which data moves along the wires, switches… of the network).

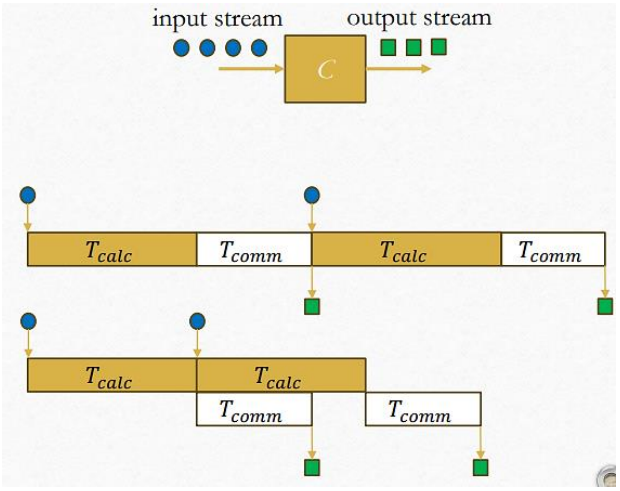
On modern HPC systems, $t_0$ is about 0.1-10 usec, whereas B is 20-200 Gb/s.

**Computation-to-communication overlap**

In a distributed system, a computing node is equipped with a **NIC** (Network Interface Card) that sends data to and receives data from the network on behalf of the processor.

The NIC can be a **specialized processor** (so-called **SmartNIC**) with multiple capabilities (also processing, e.g., compression) including DMA and RDMA transfer (Remote direct memory access)

The sender process may execute a **non-blocking send** to the destination process, the NIC executes the data transfer and then notifies the sender process about the completion of the transmission.




**While the NIC executes the data transfer**, the processor executes some other useful operations (e.g., it can execute, or partially execute, another task). There could be full or partial overlap between the computation of tasks and the communication with other processes.

The computing module C receives in input a list of tasks (data stream). For each input, it takes $\boldsymbol{T_{calc}}$ time for the computation and $\boldsymbol{T_{comm}}$ time for sending the task computed to the next module.

The **service time of a module** C ($T_C$) is the average time taken by the module between two consecutive inputs.

We may have:

- $T_C = T_{calc} + T_{comm}$ **with no overlap** *(above)*
- $T_C = \max(T_{calc}, T_{comm})$ **with overlap** *(below)*

**Synchronous vs. Asynchronous communications**

The **asynchrony degree of a channel** is the maximum number of messages ($k \geq 0$) the sender can send before it has to block waiting for the receiver to start receiving data. It depends on the memory capacity of the channel and the size of the message being sent.

**Synchronous communications**: a send/receive operation is called synchronous if the operation completes only after the message has been received/sent. Sender-receiver rendez-vous.

**Asynchronous communications**: communication operation returns without waiting for the message to be effectively sent/received. The completion/success of the communication will be tested later on. The number of asynchronous sends might be limited by the asynchrony degree of the channel.

**Linear communication model pitfalls**

The linear model seen before is **simple**, and it applies to many fields of computer architectures (e.g., to model memory access time, bus transactions, pipeline operations)

However, it **has some limitations**:

- Does not include the effect of **network distance**
- Does not model **contention** (remember that latency depends on network utilization)
- Does not model how **the data transfer is performed** (i.e., synchronously or asynchronously)

**Input non-determinism**

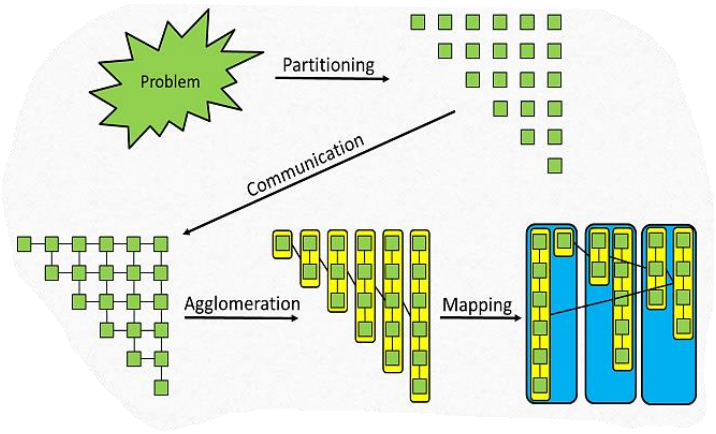
If we have a logical input channel, which multiplexes a set of independent input channels from multiple sources, **input non-determinism** refers to the property that the receive operation may return a message coming from any channels in the set. There is **no fixed order** when receiving a message from a multi-input channel.

If there is more than one message ready to be received from the set of input channels, one of them is picked up in **random order** (i.e., non-predictable order). Fixing a receiving order is usually a **non-optimal approach**.

### Foster's Parallel Algorithm Design Method

*How to parallelize a given problem?* Ian Foster proposed the **PCAM approach**:

- **Partitioning**: decompose the problem into a large amount of small (fine-grained) tasks that can be executed in parallel.
- **Communication**: determine the required communication between tasks.
- **Agglomeration**: combine identified tasks into larger (coarse-grained) tasks to reduce communication by improving data locality.
- **Mapping**: assign the agglomerated to processes/threads to minimize communication, enable concurrency, and balance workload.

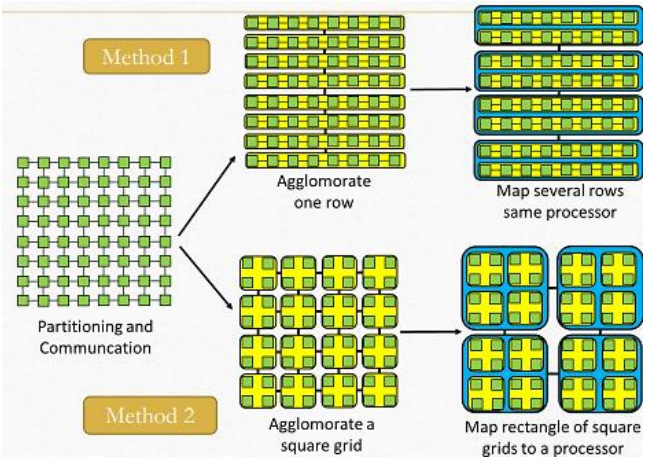



*(Example slide 27)* Jacobi iteration: each cell of the next iteration matrix is computed summing the four neighbors (left, up, right, down). Boundary values remain constant at each iteration. We compute the iteration in a new matrix and the end of the iteration we swap the updated array with the original one.

**Partitioning**: The smallest task is the computation of a single element of the Jacobi matrix

**Communication**: Within an iteration all fine-grain tasks can be computed independently. Each task needs the data of four neighbors. At the end of each iteration, there is a synchronization barrier among all p processors, and data is exchanged.

**Agglomeration**: two options proposed 1) by row (or by column) 2) by using a square grid.




**Mapping**: it follows the policy used for the agglomeration to map coarse-grain tasks to processors. By row, contiguous groups of rows are assigned to the p processors. Or by square grids, rectangles of square grids are assigned to the $p$ processors organized in a $\sqrt{p} \times \sqrt{p}$ grid.

In the top one we have maximum two communication, each row communicates with the upper and the lower row. In the bottom one we have maximum four communication, each square communicates with upper, lower, right and left square.

Method 1: $T_{comm}(n) \approx 2\,(t_0 + s \times n)$ *(communication time)*

Method 2: $T_{comm}(n) \approx 4\left(t_0 + s \times \left(\frac{n}{\sqrt{p}}\right)\right)$ Assuming: $p = \sqrt{p} \times \sqrt{p}$ (in the picture $p$=4)

Method 2 **superior for large p** since communication time decreases with $p$ while it remains constant for Method 1.

**(Lect 8)**

(Slide 32) Matrix Chain Ordering Problem: finds the most efficient order to multiply a sequence of 2D matrices using Dynamic Programming.

Given $n$ matrices $M_i$ of size $d_{i-1} \times d_i$, for $1 \le i \le j \le n$ :

$$F[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{F[i,k] + F[k+1,j] + d_{i-1} \cdot d_k \cdot d_j\} & \text{if } i < j \end{cases}$$

Value stored in F[i, j] ($i \le j$) is the number of multiplications required in the optimal ordering for computing the matrix product $M_i \times \ldots \times M_j$

Overall, the minimum cost is stored in F[1,n].

To compute the element F[i, j] we need all the previous elements in the row (< i) and all the successive element of the column (> j)

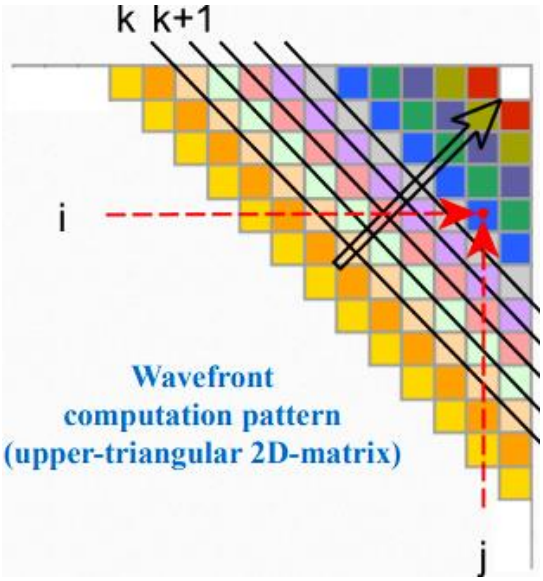
In another matrix S we store which is the value of the index k that give us the minimum for F[i, j]. This is useful to reconstruct the correct multiplication ordering.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 168 | 945 | 924 | 1721 | 2346 |
| | 0 | 504 | 819 | 1224 | 2115 |
| | | 0 | 840 | 1920 | 2765 |
| | | | 0 | 2835 | 2640 |
| | | | | 0 | 1485 |
| | | | | | 0 |

1224 = min{F[2][2] + F[3][5] + 3×8×27, F[2][3] + F[4][5] + 3×21×27, F[2][4] + F[5][5] + 3×5×27}

**Partitioning**: the smallest task is the computation of a single element of the F matrix (we have to compute all upper-triangular elements)

**Communication**: Tasks laying on the same diagonal are independent. Tasks on adjacent diagonals (e.g., k and k+1) are dependent. Specifically, to compute F[i, j] we need all F[i, k] and F[k+1, j] ($i \le k < j$). A **synchronization** between the computation of two diagonals is needed, i.e. we can compute in parallel only the elements in the same diagonal.




**Agglomeration**: Tasks can be agglomerated in different ways. One option is to agglomerate by column or by rows to save some communications *(we already have some of the necessary data)*.

**Mapping**: can be done cyclic (or block-cyclic, *depends on the agglomeration chosen*) to PEs. For example, column c is assigned to PE m such that c % n = m

**Metrics & Laws**

**Speedup**: The speedup (S) is defined as the quotient of the time taken using a single processor ( $T(1)$ ) over the time measured using $p$ processors ( $T(p)$ )

$$S = \frac{T(1)}{T(p)}$$

The best speedup one can expect is **linear speedup** varying the number of processors, i.e., the maximum speedup with $p$ processor is $p$ (there are exceptions to this, which are referred to as **super-linear speedup**).

**Efficiency**: The efficiency (E) is defined as the quotient of $S$ over $p$
$T(p) \times p$ is also called **cost of parallelization** (or simply cost)
Ideal efficiency is 1

$$E = \frac{S}{p} = \frac{T(1)}{T(p) \times p}$$

The scalability and the speedup are defined in the same way, but there is a significant difference:

- The **scalability** considers as $T(1)$ the time obtained executing the **parallel implementation of the algorithm on a single processor**. This is often also called **relative speedup**.
- The speedup, instead, consider as $T(1)$ **the best sequential version** of the algorithm solving the same problem. This is often also called **absolute speedup**.

$$Scalability = \frac{T_{par}(1)}{T_{par}(p)}$$

$$Speedup = \frac{T_{seq}}{T_{par}(p)}$$

**Strong scaling**: the problem size is **fixed** (e.g. matrix multiplication).
Linear scaling (i.e., S=p) is **hard** to achieve due to **Amdahl's law**. It depends on the amount of serial work (i.e., the part of the problem that cannot be parallelized)

**Weak scaling**: considers computing a problem whose **size is p times bigger**, in the same amount of time. The problem size remains constant per processor (but grows globally). Weak scaling **assumes that**, as the size of the problem grows the amount of **serial work** remains constant (or increases slowly) and the amount of **communication** among processors remains constant (or increases slowly).

*(Trivial example)* An array A of n number in input. The sum of the element in output.

Assumptions (unrealistic):

- **Computation**: each PE can add two numbers stored in its local memory in 1 sec
- **Communication**: a PE can send data from its local memory to the local memory of any other PE in 3 sec (independently of the data size, *don't follow the model but it is just an assumption*)
- **Input and Output**: At the beginning of the program, the whole input array A is stored in PE #0. At the end, the result must be gathered in PE #0
- **Synchronization**: All PEs operate in a lock-step manner; i.e., they can either compute, communicate, or be idle (no computation-to-communication overlap).

Establish **sequential** runtime as a baseline (p = 1, one processor): $T(n, 1) = n - 1$ seconds

Establish runtime for 2 PEs (p = 2) and 1024 numbers (n = 1024): **T(1024, 2)** = 3 + 511 + 3 + 1 = 518 sec (send half of the array to PE #1, sum 511 number [in parallel], PE #1 send the result to PE #0, sum the two results).
**Speedup**: T(1,1024)/T(2,1024) = 1023/518 = 1.975
**Efficiency**: 1.975/2 = 98.75%

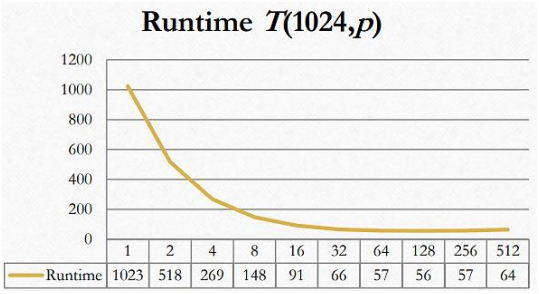
**T(1024, 4)** = 3×2 + 255 + 3×2 + 2 = 269 sec
(send two times [with a double send in parallel], every processor sum, send back results, sum results)
**Speedup**: T(1,1024)/T(4,1024) = 1023/269 = 3.803
**Efficiency**: 3.803/4 = 95.07%

In general, using $p = 2^q$ PEs and $n = 2^k$ input numbers:

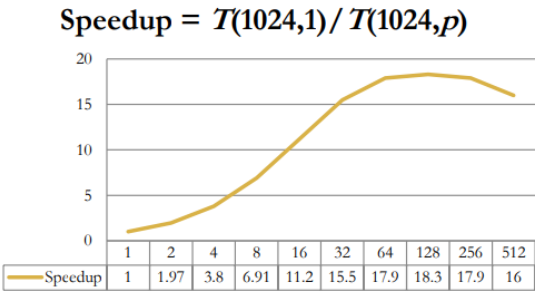
- Data distribution: $3 \cdot q$
- Computing local sums: $n/p - 1 = 2^{k-q} - 1$
- Collection partial results: $3 \cdot q$
- Adding partial results: $q$

$$T(p,n) = T(2^q,2^k) = 3q + 2^{k-q} - 1 + 3q + q = \mathbf{2^{k-q} - 1 + 7q}$$

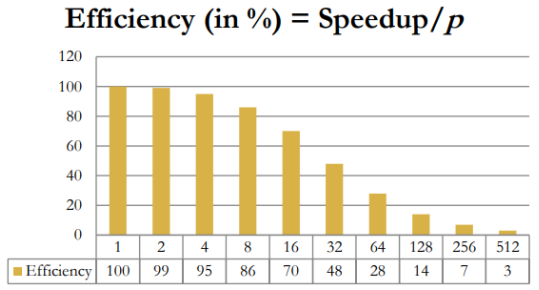
**Strong scalability Analysis (n=1024)** *(fixing the problem size)*




| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Runtime | 1023 | 518 | 269 | 148 | 91 | 66 | 57 | 56 | 57 | 64 |




| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Speedup | 1 | 1.97 | 3.8 | 6.91 | 11.2 | 15.5 | 17.9 | 18.3 | 17.9 | 16 |



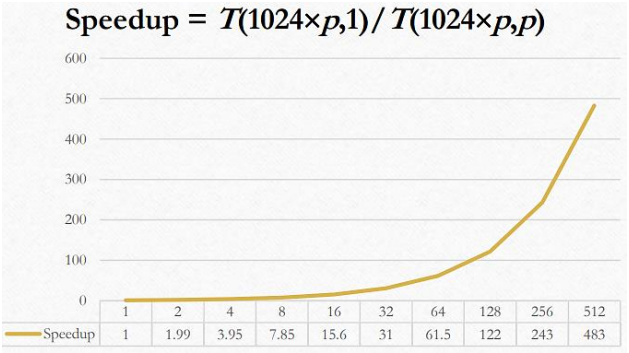

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Efficiency | 100 | 99 | 95 | 86 | 70 | 48 | 28 | 14 | 7 | 3 |

This behavior is typical of many parallel applications, we keep going better and better and at a given point we start going worse.

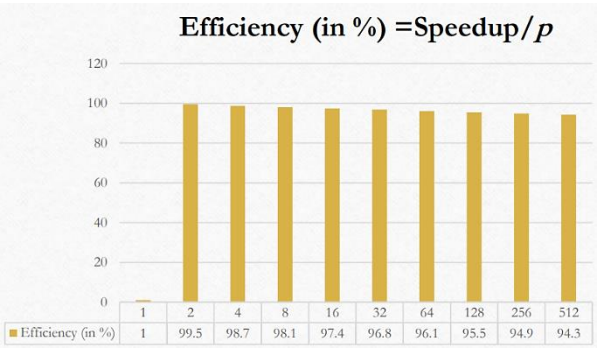
Efficiency very low with high number of the processors, we are not exploiting the machine power.

This can be seen in the formula, at the beginning when the number of processors is small (q is small), computation factor $2^k$ dominate over the communication factor 7q. When we keep increasing q (the processors) the communication factor 7q starts to dominate the cost.

**Weak scalability Analysis (n=1024×p)**




| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Speedup | 1 | 1.99 | 3.95 | 7.85 | 15.6 | 31 | 61.5 | 122 | 243 | 483 |




| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| Efficiency (in %) | 1 | 99.5 | 98.7 | 98.1 | 97.4 | 96.8 | 96.1 | 95.5 | 94.9 | 94.3 |

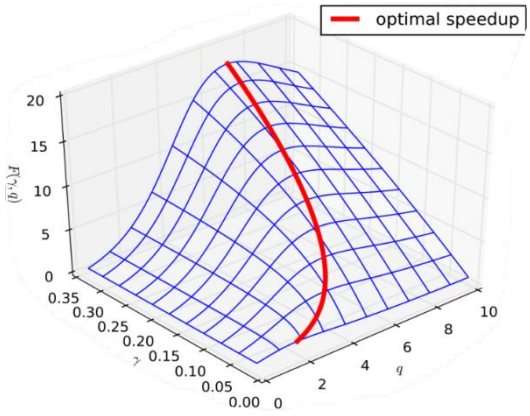
Weak scaling is much better than strong scaling. This is because the impact of the communication is kept low by the size of the problem getting bigger and bigger. Every time we have more computation than computation we obtain good scalability.

**Computation-to-Communication ratio**

Let $\alpha > 0$ and $\beta > 0$ be the time needed to perform a single addition and to communicate a set of numbers, respectively (previously $\alpha = 1$ and $\beta = 3$).

$\gamma = \alpha / \beta$ is the **computation-to-communication ratio**

Speedup and parallelization efficiency **depend** on both the **number of PEs** used and the **computation-to-communication ratio** for a fixed message size.



Speedup usually increases with the number of PEs used up to a **local maximum**, then it tends to **decrease** as more PEs are employed.

Speedup is **low** if the **communication has a significant impact**. Usually, the longer the communication takes, the fewer PEs should be used.

Efficiency is a monotonic function in both the number of PEs and the computation-to-communication ratio.

**Superlinear Speedup**

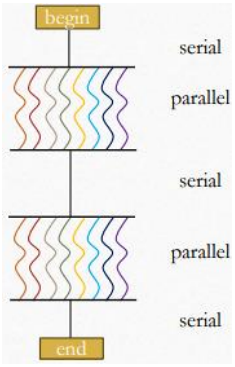
Two possible reasons:

- Unfair comparison with a naïve serial algorithm
- Cache/memory effects (more processors imply more memory and larger logical caches thus fewer cache misses and page swapping)

It is also possible to have an area of super-linear speedup and then experience a linear speedup due to the rise of communication time after a certain number of processors.

Unfortunately, quite often the speedup is **sub-linear** due to the **parallel run-time overhead**.

**Amdahl's Law**




The Amdahl's law gives an **upper bound** on the speedup that can be achieved.
It states that no matter how many processors are used in a parallel run, the program's speedup **will be limited** by its **fraction of sequential code**.
Almost every program has a fraction of the code that doesn't lend itself to parallelism. This is the fraction of code that will have to be run with just one processor, even in a parallel run.

$T(1) = Tser + Tpar$ (a part of the program is serial, and another can be parallelized)

We now assume that the best possible speedup we can achieve is linear. Then, we derive an upper bound for achievable speedup:

$$T(p) \geq T_{\text{ser}} + \frac{T_{\text{par}}}{p} \qquad S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}}$$

Instead of using absolute execution times ($Tser$ and $Tpar$), we now use their **relative fraction**, ***f*** is the **serial fraction** whereas ***(1-f)*** is the (potentially) **parallelizable fraction**. Substituting this in the previously derived upper bound, results in Amdahl's Law.

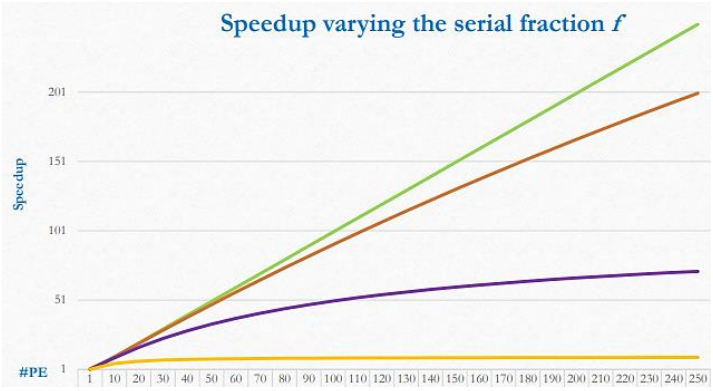
$$T_{\text{ser}} = f \cdot T(1) \qquad T_{\text{par}} = (1-f) \cdot T(1); \;\; (0 \leq f \leq 1)$$

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}} = \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

Raising *p* to the infinity the maximal speedup is 1 / *f* (if 10% of the code is not parallelizable the maximal speedup is 10)

*Example:* 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 6 CPUs?

$$S(6) \leq \frac{1}{0.05 + \frac{(1-0.05)}{6}} = 4.8$$




Green line is perfect speedup
Red is with serial fraction of 0.1%
Purple is with f = 1%
Yellow is with f = 10 %

This show that strong scaling is really hard, we have to parallelize almost all the code otherwise the line will be very flatten.

**(Lect 9)**

**Scaled Speedup**

Amdahl's law **only** applies in situations where the problem **size is constant,** and the number of processors varies (⇒ strong scalability)

However, when using more PEs, we may also use larger problem sizes (⇒ weak scalability)

**Scaled Speedup** incorporates such scenarios when calculating the achievable speedup. We derive a more general law that enables to study the scaling with respect to the problem's complexity. **Gustafson's Law** is a particular case that can be used to predict the theoretically achievable speedup using multiple processors when the parallelizable part scales linearly with the problem size while the serial part remains constant.

**Derivation of Gustafson's Law** *(parametrize even more the timing function)*

$$T_{\alpha\beta}(1) = \alpha \cdot T_{ser} + \beta \cdot T_{par} = \alpha \cdot f \cdot T(1) + \beta \cdot (1-f) \cdot T(1)$$

**α:** scaling function of the part of the program that does not benefit from parallelization with respect to the complexity of the problem size

**β:** scaling function of the part of the program that may benefit from parallelization with respect to the complexity of the problem size

α: how much serial code grow

β: how much parallel code grow w.r.t. problem size

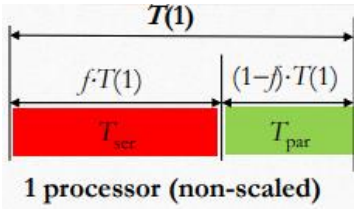
$$S_{\alpha\beta}(p) = \frac{T_{\alpha\beta}(1)}{T_{\alpha\beta}(p)} \leq \frac{\alpha \cdot f \cdot T(1) + \beta \cdot (1-f) \cdot T(1)}{\alpha \cdot f \cdot T(1) + \frac{\beta \cdot (1-f) \cdot T(1)}{p}} = \frac{\alpha \cdot f + \beta \cdot (1-f)}{\alpha \cdot f + \frac{\beta \cdot (1-f)}{p}}$$

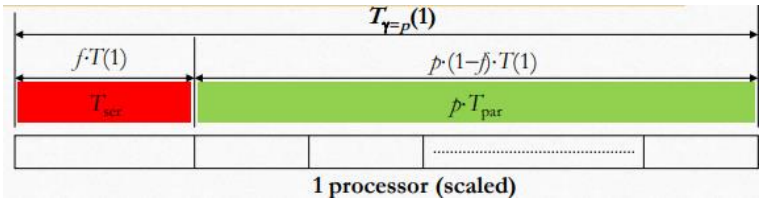
$$\gamma = \frac{\beta}{\alpha}$$ (Ratio of the problem complexity scaling between the parallelizable part and the non-parallelizable part)

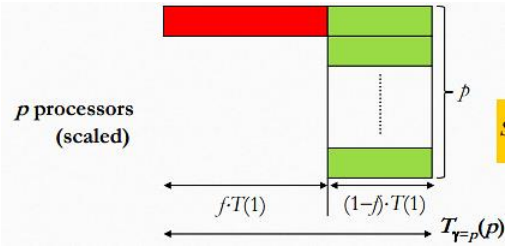
$$S_\gamma(p) \leq \frac{f + \gamma \cdot (1-f)}{f + \frac{\gamma \cdot (1-f)}{p}}$$ Scaled speedup as a function of γ.

Using different functions for γ yields to the following two notable cases:

- γ = 1 (i.e. α = β) ⇒ Amdahl's Law
- γ = p (e.g. α = 1; β = p), i.e., the parallelizable part grows linear in p while the non-parallelizable part remains constant ⇒ **Gustafson's law**: $S(p) \leq f + p \cdot (1 - f) = p + f \cdot (1 - p)$




$$S_{\gamma=p}(p) = \frac{T_{\gamma=p}(1)}{T_{\gamma=p}(p)} \leq \frac{f \cdot T(1) + (1-f) \cdot T(1) \cdot p}{f \cdot T(1) + (1-f) \cdot T(1)}$$

$$= f + (1-f) \cdot p = p + f \cdot (1-p)$$

Amdahl's law → Strong scaling  Gustafson's law → weak scaling  Scaled speedup → something in the middle (varying alpha and beta)

*Scaled speedup example:* Suppose we have a parallel program that, for a given problem size, is 15% serial and 85% linearly parallelizable. Let us assume that the (absolute) **serial time does not grow** as the problem size is scaled.

How much speedup can we achieve if we use 50 processors without scaling the problem?

$$S_{\gamma=1}(50) \leq \frac{f + \gamma \cdot (1-f)}{f + \frac{\gamma \cdot (1-f)}{p}} = \frac{1}{0.15 + \frac{0.85}{50}} = 5.99$$

(Amdahl's law)

Suppose we scale up the problem size by a factor of 100. Which is the speedup with 50 processors?

$$S_{\gamma=100}(50) \leq \frac{f + \gamma \cdot (1-f)}{f + \frac{\gamma \cdot (1-f)}{p}} = \frac{0.15 + 100 \cdot 0.85}{0.15 + \frac{100 \cdot 0.85}{50}} = 46.03$$

(Scaled speedup with γ=100, Gustafson's law)

Assume that you want to write a program that should achieve a speedup of 100 using 128 processors. What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of **strong scalability**?

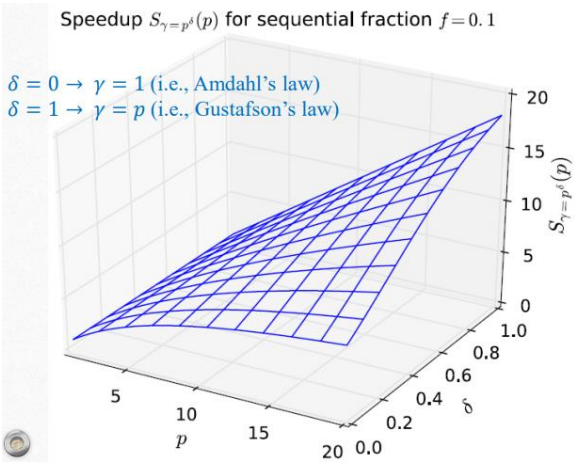
$$100 = \frac{1}{f + \frac{(1-f)}{128}} = \frac{128}{128 \cdot f + 1 - f} = \frac{128}{127 \cdot f + 1} \Longrightarrow f = \frac{0.28}{127} = 0.0022$$

(apply Amdahl's law)

What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of **weak scalability** (whereby γ scales linearly)?

$$100 = 128 + f \cdot (1 - 128) = 128 - 127 \cdot f \Longrightarrow f = \frac{28}{127} = 0.22$$

(apply Gustafson's law)




When delta is 0, we are in Amdahl's law and the curve is flattening fast, bad speedup.

When delta is 1, we are in Gustafson's law and the curve grow faster, better speedup.

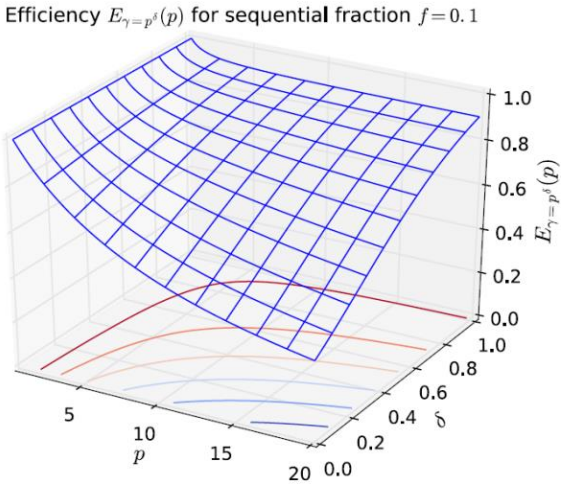
**Scaled Efficiency**

$$E_\gamma(p) \leq \frac{f+(1-f)\cdot\gamma}{p\cdot f+(1-f)\cdot\gamma} \quad \gamma = p^\delta \; \delta \in [0,1]$$

(Scaled Efficiency as a function of γ, speedup over number of processors)

$$\lim_{p\to\infty} E_{\gamma=p}(p) = (1-f) \qquad \lim_{p\to\infty} E_{\gamma=1}(p) = 0$$

(If we consider weak scaling efficiency tends to a good value if the fraction of not scalable code is low, while considering strong scaling efficiency tends to zero)




Values of p and δ that guarantee **constant efficiency** are called **iso-efficiency lines** (colored in the plot).

For example, given a certain delta how many processors we need to use to keep a certain efficiency?

*But not easy to compute…*

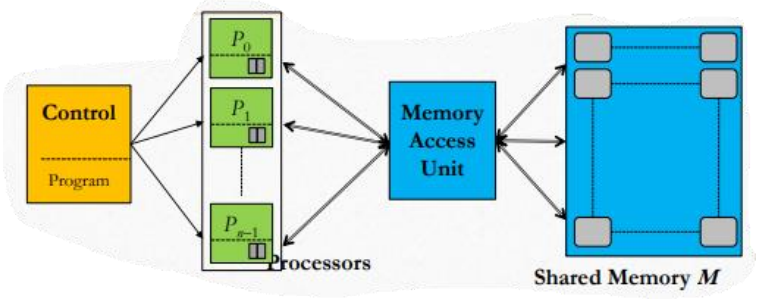
**Models of Computation**

*Sometimes, instead of implementing a complex algorithm directly on a specific parallel system, it is better to explore its possible limitations independently of a specific architecture and programming language. Theoretical parallel models of computation may be helpful for such exploration. They allow the programmer to focus first on the best algorithm and then on how to deal with specific technological limitations.*

**Parallel Random-Access Machine – PRAM model**

Idealized Shared-Memory platform: no caching, no NUMA organization (memory accessed in constant time), no synchronizations overhead.

*n* processors $P_0$, …, $P_{n-1}$ are connected to a global shared memory M.



Communication between processors can be implemented by reading and writing to the globally accessible shared memory.

*n* identical processors Pi , i = 0, ...,n−1 operate in lock-step (in a synchronous way). In every step, each processor executes an instruction cycle in three phases:

- **Read phase**: Each processor can simultaneously read a single data item from a (distinct) shared memory cell and store it in a local register.
- **Compute phase**: Each processor can perform a fundamental operation on its local data and store the result in a local register.
  **Write phase**: Each processor can simultaneously write a data item to a shared memory cell, whereby the **exclusive write PRAM** variant allows writing only distinct cells while the **concurrent write PRAM** variant also allows processors to write to the same location (possible race conditions).

Each step on the PRAM takes O(1) time (**constant time**).

Several types of PRAM variants have been defined **to solve conflicts** that arise when processors read or write to the same shared memory location.

- **Exclusive Read Exclusive Write** (EREW): two processors are not allowed to read or write to the same shared memory cell during any cycle.
- **Concurrent Read Exclusive Write** (CREW): Several processors may read data from the same shared memory cell simultaneously. Still, different processors are not allowed to write to the same shared memory cell.
- **Concurrent Read Concurrent Write** (CRCW): both simultaneous reads and writes to the same shared memory cell are allowed. In case of a simultaneous write we further specify which value will actually be stored:
  - **Priority CW**: Processors have been assigned distinct priorities, and the one with the highest priority succeeds in writing.
  - **Arbitrary CW**: A randomly chosen processor succeeds in writing its value.
  - **Common CW**: If the values are all equal, then this common value is written, otherwise, the memory location is unchanged.
  - **Combining CW**: All values to be written are combined into a single value by means of an associative binary operation (e.g. sum, product, minimum, logical OR/AND).

**Prefix Computation**

Binary **associative** operation $\circ$ on the set X $\quad \circ : X \times X \to X \quad$ [ associative: $(x_i \circ x_j) \circ x_k = x_i \circ (x_j \circ x_k)$ ]

*Examples*: Addition, Multiplication, Minimum, Maximum, String concatenation, boolean AND/OR

$X = \{x_0, \ldots, x_{n-1}\}$, $x_i \in X$ for all i = 0, ...,n−1. We want to compute $s_0 = x_0$, $s_1 = x_0 \circ x_1$ ... $s_n = x_0 \circ x_1 \circ \ldots \circ x_{n-1}$

Obtaining $S = \{s_0, \ldots, s_{n-1}\}$ from $X = \{x_0, \ldots, x_{n-1}\}$ is called **prefix computation**.

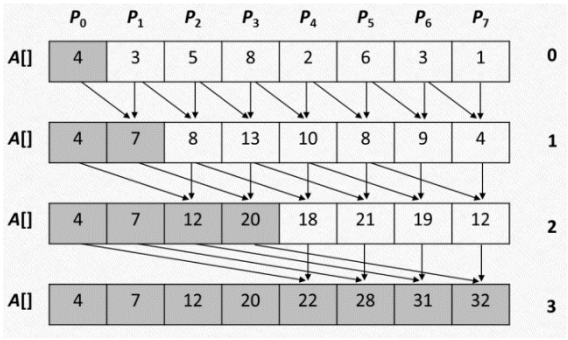
*(Example with the minimum operation)* Input: {39, 21, 20, 50, 13, 18, 2, 33, 49, 39, 47, 15, 30, 47, 24, 1}
Output: {39, 21, 20, 20, 13, 13, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1} $\quad \Omega(n)$

**Parallel Prefix ($\circ$ = '+') on a PRAM**

Goal: design a cost-optimal PRAM algorithm, i.e., C(n)=O(n)

We use the recursive doubling algorithm using p=n processors

```
for (j = 0; j<n; j++) do_in_parallel              // each processor j
  reg_j = A[j];                                   // copies one value to a local register
for (i = 0; i<ceil(log(n)); i++) do                // sequential outer loop
  for (j = pow(2, i); j<n; j++) do_in_parallel     // each proc. j
    reg_j += A[j - pow(2, i)];                    // performs computation
    A[j] = reg_j;                                 // writes result to shared memory
}
```



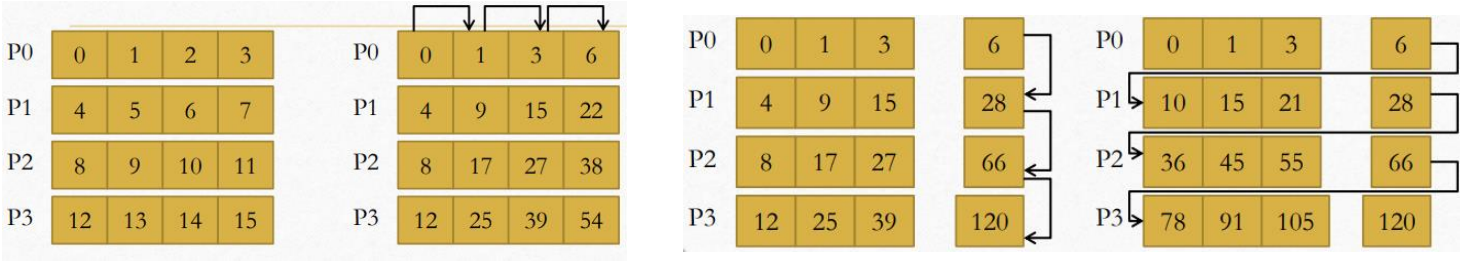

C(n) = T(n, p) × p = O(log n) × n = O(n × log n)

*[n processors doing something with logarithmic cost]*

This algorithm is NOT cost-optimal when using p=n processors.

To reduce the cost C(n) = T(n, p) × p we can reduce the number of processers used. Let's use p = n/log(n) processors and the following algorithm:

- Partition the *n* input values into chunks of size log(*n*). Each processor computes local prefix sums of the values in one chunk in parallel, takes time O( log(*n*) )
- Perform the old non-cost-optimal prefix sum algorithm on the $n$ / log(*n*) partial results, takes time O( log( *n* / log(*n*) ) )
- Each processor adds the value computed in step 2 by its left neighbor to all values of its chunk, takes time O( log(*n*) ).




16 numbers → 16/log(16) = 4 processors used
C(n) = T(n,p) × p = O(log n) × O(n/log(n)) = O(n)  This algorithm is cost-optimal.

*(Slide 14) Sparse array compaction example*

**Bulk Synchronous Parallel (BSP) Model**

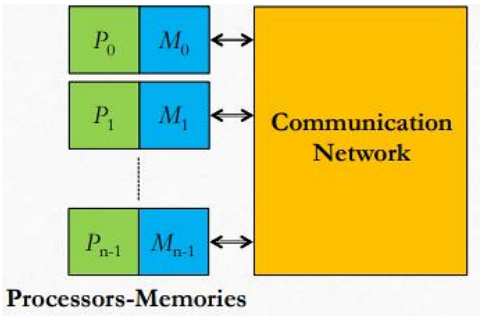
The Bulk-Synchronous Parallel (BSP) model of computation was proposed as a unified framework for the design, analysis, and programming of general-purpose parallel systems (it is not just a theoretical model, it can also serve as a paradigm for parallel programming).

*n* processors $P_0$ , ...,$P_{n-1}$ each with its **own memory** (a distributed memory multiprocessor).

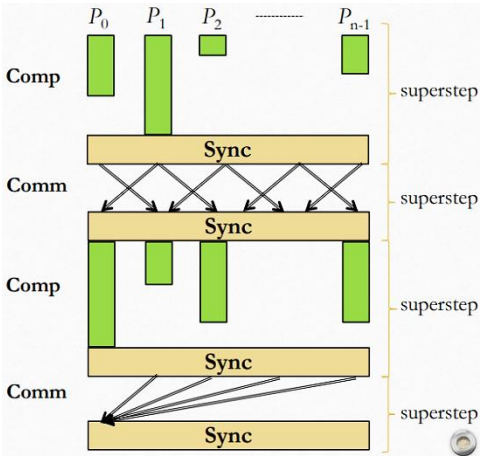
The remote memory access time is **uniform** (the access time to all non-local memory locations is the same)

Communication between processors happen via explicit messaging (message-passing model)

The communication network is a blackbox (it can be fat tree, dragonfly, ...)



**BSP Algorithm**




A BSP algorithm consists of a sequence of supersteps.
A superstep consists of:

- a number of **computation** or **communication** steps or both (called **mixed** superstep in case both are executed)
- a **global barrier synchronization** (bulk synchronization).

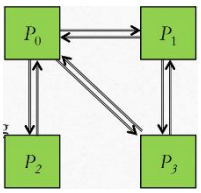
A **computation superstep** consists of many small steps executing operations (e.g., floating point operations).

A **communication superstep** consists of many basic communication operations (e.g., a data word transferring).

An **h-relation** is a communication superstep in which every processor sends and receives **at most** h data words (it is the maximum between the data words sent and received by a processor in a communication superstep, because we must synchronize *i.e. waiting the slowest*).

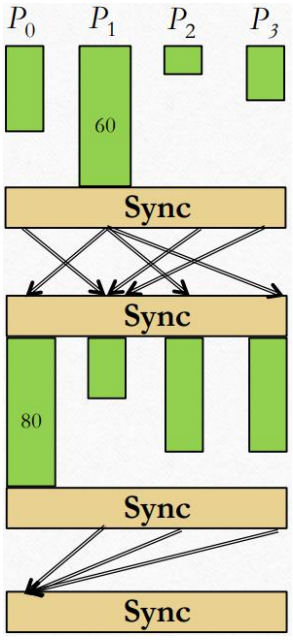
The **cost** of an h-relation is $T(h) = h \cdot g + l$ where $g$ (gap) is the time per data word and $l$ (latency) is the global synchronization time. Note that $g$ and $l$ (which are fixed) depend on the number of processors (p) and that $l > 0$ includes the costs of the global synchronization plus all fixed costs for start-up of the communications and ensuring that all data have arrived at the destination.

Example of 3-relations, i.e. the maximum number of arrows entering or exiting in a processor is 3 (in $P_0$). Potentially $P_0$ can receive/send 3 data words in a communication superstep.




The **cost** of a computation superstep is $T_{comp}(w) = w + l$ where $w$ is the amount of work defined as the maximum number of operations performed in the superstep by any processor (processors with less work than $w$ operations must wait)

*(Example)*




p=4 g=4 flops *(measured)* l=20 flops *(measured)*

In the first computation superstep $P_1$ takes 60 flops, while in the second one $P_0$ takes 80 flops.

In the first communication superstep $P_1$ sends/receives 10 words to/from **all other** processors (3 processors).

In the second communication superstep $P_0$ receives 10 words from **all other** processors (3 processors).

$$CBSP = (60 + l) + (3 \cdot 10 \cdot g + l) + (80 + l) + (3 \cdot 10 \cdot g + l)$$
$$= 140 + 60 \cdot g + 4 \cdot l = 140 + 60 \cdot 4 + 4 \cdot 20 = 460\ flops$$

*(Slide 22) Dot product example*

The major vantage of BSP model is that is very simple, the biggest drawback is that we need to follow the rules. For example, we must put a barrier after a computation step even if we think is not necessary. It limits the freedom of writing the code as we want. The payoff is that if we write the code in this way we have a precise cost for the algorithm we are developing.

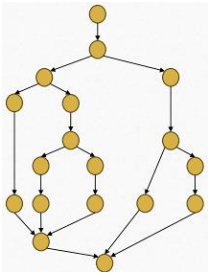
It encourages **SPMD** (Single Program, Multiple Data) model of computation, i.e., only one program needs to be written; all processors run the same program, but on different data.

Since there is a global synchronization is encouraged to balance the communication among the processors (sending all data to one processor is discouraged)

**(Lect 10)**

**Work-Span model**

The **Work-Span model** (also called Work-Depth) model is another classic performance model. It provides **more strict bounds** than those offered by the Amdahl's and Gustafson's laws.



The program's tasks form a **DAG** (Directed Acyclic Graph).

A **task** is a unit of work, i.e., arbitrary sequential code, that can be executed in parallel (using threads or processes) with other program's tasks.

A given graph's task is ready to run **if and only if** all its predecessors in the graph have been completed (edge represent data dependencies).

In this model we assume:

- *p* **identical processors**, each executing one ready task at a time
- The task **scheduling** is **greedy**, i.e., whenever there is a ready task (and an available processor), the task is executed immediately.

$T_p$ is the time when executing with a greedy scheduler with *p* processors.

$T_1$ is called **work** *(the sequential time)*

$T_\infty$ is called **span** (also called **critical path**, time taken by the longest path in the DAG)

$$S(p) = \frac{T_1}{T_p} \leq \frac{T_1}{\frac{T_1}{p}} = p$$

(Upper bound to speedup, derived by the definition of $T_p$. In this model is impossible to achieve superlinear speedup).

$$S(p) = \frac{T_1}{T_p} \leq \frac{T_1}{T_\infty}$$

(Another upper bound.
The minimum of the two is the upper bound)

To derive these upper bounds, we use:

$$T_p \geq \frac{T_1}{p}$$
$$T_p \geq T_\infty$$

**Brent's theorem**

Assume a parallel computer whose processors can perform a task in unit time with **greedy scheduling** of tasks. Assume the computer has enough processors to **exploit the maximum concurrency** in an algorithm containing $T_1$ tasks such that it completes in $T_\infty$ time steps [at each level i of the DAG, there are $m_i$ tasks. We may use $m_i$ processors to compute all results in O(1) ].

**Brent's theorem** states that a similar computer with fewer processors *p* can execute the algorithm with the following upper time limit:

$$T_p \leq \frac{(T_1 - T_\infty)}{p} + T_\infty$$

Brent's inequality establishes an **upper bound** on $T_p$

Considering the speedup, we have that $S(p) \leq \min(p, \frac{T1}{T\infty})$ *(from previous page results)*

To get good speedup, $T_1$ must be significantly larger than $T_\infty$ (i.e. $T_\infty \ll T_1$). In this case $T_1 - T_\infty \approx T1$, therefore considering Brent's inequality we have:

$$T_p \approx \frac{T_1}{p} + T_\infty \quad \text{if } T_\infty \ll T_1$$

When designing a parallel algorithm, **focus on reducing the span**, because the span is the fundamental asymptotic limit on scalability. Increase the work only if it enables a drastic decrease in span.

Overall, we have: $\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$

Additionally, it can be proved that: $S(p) = \frac{T_1}{T_p} \approx p \quad \text{if} \quad \frac{T_1}{T_\infty} \gg p$

It says that greedy scheduling achieves (almost) linear speedup if a problem is **overdecomposed** to create much more potential parallelism than the number of processors.

The excess parallelism is called the **parallel slack**, and is defined by: $slack = \frac{T1}{p \cdot T\infty}$

In practice, it has been observed that a parallel slack of at least 8-10 works well.

However, remember that these formulas assume:

- **No parallel overhead**, i.e., the parallel code does $T_1$ total operations.
- The **memory bandwidth is not a limiting resource** (i.e., we consider PRAM-like abstract machines)
- The **scheduler is greedy in scheduling tasks** (i.e., **no delays** due to lock or synchronization in general)

Very strong assumptions, the model is not concrete but can give good estimation.

**Concurrency in C++**

**Spawning and joining threads**

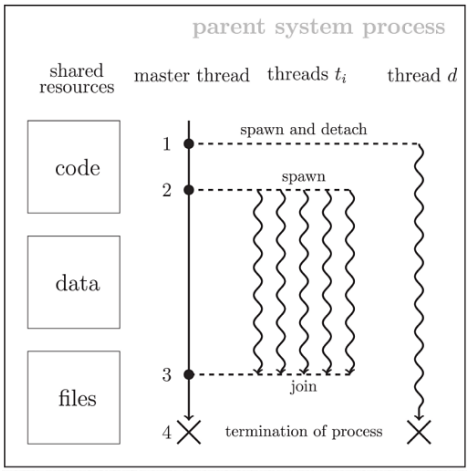
The master thread can **spawn threads**, and each thread can spawn threads as well. The number of spawned threads should be roughly the number of cores (i.e., pay attention to oversubscription).

**Oversubscription**: having much more thread than cores (or logical cores).

Threads **share process resources** (page table, process file, …), but each thread has a **separate stack**.

A thread can be **joined** (wait the termination) or **detached** once (like say "I don't want to wait its termination"). A detached thread **cannot be joined**. Joined or detached threads **cannot be reused**.
All threads must be joined or detached **within the scope** of their declaration.

We need to store the **thread handles** explicitly to be able to access them during the join phase.




```cpp
#include <cstdint>  // uint64_t
#include <iostream> // std::cout std::endl
#include <vector>   // std::vector
#include <thread>   // std::thread

// this function will be called by the threads (should be void)
void say_hello(uint64_t id) {
    std::cout << "Hello from thread: " << id << std::endl;
}

// this runs in the master thread
int main(int argc, char * argv[]) {

    const uint64_t num_threads{(argc==2)?std::stoul(argv[1]):10};
    std::vector<std::thread> threads;

    // for all threads
    for (uint64_t id = 0; id < num_threads; id++)
        // emplace the thread object in vector threads
        // using argument forwarding, this avoids unnecessary
        // move operations to the vector after thread creation
        threads.emplace_back(
                             // call say_hello with argument id
                             say_hello, id
                             );

    // join each thread at the end
    for (auto& thread: threads)
        thread.join();
}
```

In the code snippet, we use a *std::vector* container and the method *emplace_back* (it creates the thread in place adding it at the end of the vector).

Alternatively, we could have moved the thread object implicitly using the vector member function push_back:

```cpp
threads.push_back(std::thread(say_hello, id));
```

The type *std::thread* is move-only (i.e., not copyable). For this reason if you use *push_back()* you have to move, after the creation, the location of the thread into the vector on which we are pushing (more overhead).

How to compile:

```
g++ –O2 –std=c++17 hello_world.cpp –o hello_world -pthread
```

```cpp
template <typename value_t>
void fibo(value_t n,
          value_t & result) { // <- here we pass the reference
    value_t a_0 = 0, a_1 = 1;
    for (uint64_t index = 0; index < n; index++) {
        const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
    }
    result = a_0;
}

// this runs in the master thread
int main(int argc, char * argv[]) {
    const uint64_t num_threads = 32;
    std::vector<std::thread> threads;

    // allocate num_threads many result values
    std::vector<uint64_t> results(num_threads, 0);

    for (uint64_t id = 0; id < num_threads; id++)
        threads.emplace_back(
            // specify template parameters and arguments
            // fibo<uint64_t>, id, std::ref(results[id])

            // using lambda and automatic type deduction
            // pay attention to id, must be passed by value!
            [id, &results]() { fibo(id, results[id]); }
        );
    // join the threads
    for (auto& thread: threads)
        thread.join();
    // print the result
    for (const auto& result: results)
        std::cout << result << std::endl;
}
```

We can pass the arguments by values or by reference.

In the example we pass the results vector as reference to store the results in it.

In the commented call of "fibo" we instantiate the type explicitly.

In the uncommented call we use lambda function to exploit automatic type deduction.

**Possible pitfall of passing by reference**: The memory passed via a reference (or pointer) **has to be persistent** during the execution of threads. The memory used for variables or objects defined within the loop body where threads are spawned **is destroyed** when leaving the scope (the destructor is called).

**Promises and futures**

C++ provides a mechanism for return-value-passing specifically designed to fit the characteristics of asynchronous execution (the so-called **promises** that are fulfilled in the future).

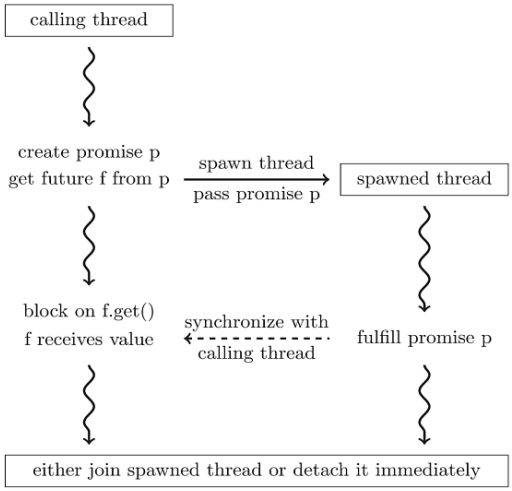
*s* = ( *p*, *f* ) where *p* is a writable view of the state *s* (the **promise**) and *f* (the **future**) is a readable view of the state *s* that **can be accessed after being signaled** (only once!) **by the promise**.

The causal dependency between the promise *p* and the future *f* **implements a synchronization mechanism** between two threads.

We **create** the state *s* = (*p*, *f* ) by initially declaring a promise p for a specific data type T via
*std::promise<T> p*;

and subsequently **assign** the associated future f with
*std::future<T> f = p.get_future();*

*p* is passed as rvalue reference in the signature of the called function via
*std::promise<T> && p;*




*Fibo example:*

(p has to be moved using *std::move()* from the master thread to the spawned thread since *std::promise* is **not copiable**).

The promise *p* is **fulfilled** within the body of the spawned thread by
*p.set_value(some_value);*

Finally, we can **read the future** *f* in the master thread using *f.get()* , blocking its execution until *f* is signaled by *p*.

Very useful if we have to synchronize only one time (for example at the end of a function).

```
template <typename value_t>
void fibo(value_t n,
          std::promise<value_t> && result) { // <- pass promise
    value_t a_0 = 0, a_1 = 1;
    for (uint64_t index = 0; index < n; index++) {
        const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
    }
    result.set_value(a_0); // fulfill promise
}

// this runs in the master thread
int main(int argc, char * argv[]) {
    const uint64_t num_threads = 32;
    std::vector<std::thread> threads;
    // allocate num_threads many result values
    std::vector<std::future<uint64_t>> results;

    for (uint64_t id = 0; id < num_threads; id++) {
        // define a promise and store the associated future
        std::promise<uint64_t> promise;
        results.emplace_back(promise.get_future());
        threads.emplace_back(
            // specify template parameters and arguments
            // fibo<uint64_t>, id, std::move(promise)

            // using lambda and automatic type deduction
            // mutable avoids p (and also id) to be const
            [id, p{std::move(promise)}]() mutable {fibo(id, std::move(p));}
        );
    }
    // read the futures resulting in synchronization of threads
    // up to the point where promises are fulfilled
    for (auto& result: results) std::cout << result.get() << std::endl;
    // join the threads
    for (auto& thread: threads) thread.join();
}
```

**Creating task objects**

The “future” C++ header provides the *std::packaged_task* function that allows you to conveniently construct task objects.

A **task object** is a **callable object** (function pointer, functor class, member function pointer, lambda, std::function wrapper, ...) with associated the corresponding future object handling the return value.

We can use task object to use promised and future without creating every promise, passing them, …

```
// create the task and assign future
std::packaged_task<bool(float, int64_t)> task(comp);
auto future = task.get_future();

// call the task with arguments
task(value, threshold); // WARNING: this is sequential!

// access future object (either true or false)
std::cout << future.get() << std::endl;
```

The problem with this approach is that the function signature is hard-coded in the template parameter of the *std::packaged_task*. (e.g. for example if a function takes one float number, an integer id and return a Boolean we have, *std::packaged_task<**bool(float, int64_t)**> task(comp);* )

It would be better if all tasks exhibit the same signature, e.g., *void task(void)*, independently of the callable object to be invoked.

This can be achieved by using a task factory function template (make_task).

*(Code example at slide 10-11, not easy)*

**Asynchronous way** *(Example at slide 13)*

The "future" C++ header provides an out-of-the-box *std::async* function for creating task objects.

It **executes a task object asynchronously** using either a thread or the calling thread.

```
auto future = std::async(fibo, id);
```

However, there are pitfalls to pay attention to:

- A call to *std::async* **does not necessarily** imply that a new thread is spawned. The calling thread might execute the task without spawning a thread! We can force the creating of a new thread using *std::launch::async*

```
auto future = std::async(std::launch::async, fibo, id);
```

- The execution of the task could be **deferred forever** if the future is not accessed (i.e., *future.get()* )
- The execution of distinct tasks **could be serialized**. If the destructor of the future is called just after the async.

**(Lect 11)**

**Mutex variables**

A **mutex** is a synchronization mechanism that guarantees **mutual exclusion** execution of a critical section. Its usage restricts the execution of a critical section to a single thread at a time.
A thread locking a mutex prevents other threads from acquiring the mutex. The other threads wait for its release (**passive** waiting).

Using this syntax the mutex is released automatically when we leave the scope.

```
#include <mutex>
std::mutex mutex;

// to be called by threads
void some_function(...) {

    {
        // here we acquire the lock
        std::lock_guard<std::mutex> lock_guard(mutex);
        // this region is locked by the mutex
    } // <- here we release the lock

    // this region is processed in parallel
}
```

**std::lock_guard** is a wrapper around std::mutex that acquires ownership of the mutex upon construction and releases it upon destruction:

- Once constructed, it cannot be explicitly released until it goes out of scope.
- It can only manage a **single** mutex.

**std::scoped_lock** manages multiple mutexes simultaneously, acquiring/releasing all of them simultaneously when entering/exiting the scope:

- It provides an unlock() method to explicitly release the locks before the end of the scope.
- Helps **avoid deadlock** when you need to acquire multiple locks together.
- Pay attention to the fact that it accepts 0 mutexes, resulting in a run-time error.

### Condition Variables (CV)

A CV enables one or more threads to **wait** (**passively**) **for an event** inside a critical section.

Conceptually, a CV is associated with an event or condition. When a thread has determined that the condition is satisfied, it can **notify** one or more of the threads waiting on the CV to wake them up.
Pay attention to **spurious wakeups**! The condition must be **checked in a loop** (typically a while loop).

**Workflow of the signaling thread**

1. The signaling thread has to acquire a mutex using either mutex.lock() or through a scoped wrapper such as *std::lock_guard* or *std::unique_lock*.
2. While holding the lock, the shared state is modified.
3. The lock is released either explicitly with *mutex.unlock()*, or implicitly by leaving the scope of *std::lock_guard* or *std::unique_lock*.
4. The actual signaling of the condition variable cv is performed using *cv.notify_one()*, or *cv.notify_all()*

Note that 3 and 4 can be swapped! **Slower but usually safer**!

**Workflow of the waiting thread(s)**

1. Acquire a ***std::unique_lock*** using the same mutex as in the signaling phase. Note that *std::lock_guard* cannot be used here.
2. While being locked call either *cv.wait()*, *cv.wait_for()*, or *wait_until()*. The lock is released automatically (and atomically).
3. In case (i) cv is notified, (ii) the timeout of *cv.wait()* or *cv.wait_for()* expires, or (iii) a spurious wake-up occurs, then the thread is awakened, and the lock is reacquired. At this point, we must check whether the globally shared state (the condition) indicates proceeding or waiting (i.e., sleeping) again.

*(Slide 17) Alarm clock example  Note: [&] in the lambda function capture everything (all the variable)*

We care about performance so we put the notify after the mutex as been released, but pay attention because on some occasions this could result in the destruction of the CV and the program not work.

*(Slide 18) Wrong alarm clock example: the CV is **memory less** so a notify is lost if nobody is waiting.*

### One-shot notifications using futures

Futures and promises can be used for the so-called one-shot synchronization scenario.

Synchronization between the issuing thread and the waiting threads is achieved since the access to a future's values via *f.get( )* blocks until the fulfilling of the corresponding promise with *p.set_value( )*.

The so-called **shared futures** can be used to broadcast a specific value to more than one thread.

*(slide 19) One-shot notification example with shared future. When the set_value( ) is called all the waiting threads on the get( ) receive the same value.*

*(Slide 20) Matrix-Vector multiplication example: we don't need synchronization, but we need thread to exploit the computing power of the machine. Each processor computes the product between a row of the matrix and the vector. Total cost is O(m x n). But if the number of rows is much higher than the number of cores of the machine, how can we aggregate tasks and assign them to the available threads to balance the workload?*

**Static Data Distribution Policies**

**Static**: all threads know which is the data that have to compute beforehand.

Example p=5 m=16 | $b_0$ | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $b_5$ | $b_6$ | $b_7$ | $b_8$ | $b_9$ | $b_{10}$ | $b_{11}$ | $b_{12}$ | $b_{13}$ | $b_{14}$ | $b_{15}$ |

*(b is the result vector, m the number of rows, p the number of processors)*

There are three options:

- **Block task distribution**:

| $p_0$ | | | | $p_1$ | | | $p_2$ | | | $p_3$ | | | $p_4$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_0$ | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $b_5$ | $b_6$ | $b_7$ | $b_8$ | $b_9$ | $b_{10}$ | $b_{11}$ | $b_{12}$ | $b_{13}$ | $b_{14}$ | $b_{15}$ |

  - The block size is at least the lower integer approximation of $m/p$
  - If $(m \bmod p) = k \neq 0$ then the first k block will have a size equal to the upper integer approximation of $m/p$ (i.e. we distribute the remaining element)
- **Cyclic task distribution**:

| $p_0$ | | | | $p_1$ | | | $p_2$ | | | $p_3$ | | | $p_4$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_0$ | $b_5$ | $b_{10}$ | $b_{15}$ | $b_1$ | $b_6$ | $b_{11}$ | $b_2$ | $b_7$ | $b_{12}$ | $b_3$ | $b_8$ | $b_{13}$ | $b_4$ | $b_9$ | $b_{14}$ |

  - The task $t_i$ is assigned to $p_{i \bmod p}$
- **Block-Cyclic task distribution**:

| $p_0$ | | | | $p_1$ | | | | $p_2$ | | | | $p_3$ | | $p_4$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_0$ | $b_1$ | $b_{10}$ | $b_{11}$ | $b_2$ | $b_3$ | $b_{12}$ | $b_{13}$ | $b_4$ | $b_5$ | $b_{14}$ | $b_{15}$ | $b_6$ | $b_7$ | $b_8$ | $b_9$ |

  - Given a block of size $c > 0$ ($p \cdot c$ is called **stride**) then $t_i$ is assigned to processor $p_{(i \, div) \, c \bmod p}$ (in this example c=2)

Note: Cyclic task distribution can bring to false sharing, for example imagine if the cache line has size 4. Block task distribution may have false sharing but only in the border, if the block is quite large the impact is negligible.

(Slide 22) Example of implementation of matrix-vector multiplication. For the block distribution version has been used an alternative proposed by the book to not use the modulo that is costly but is not very effectively at distributing homogenously the task to the processors (in this case we don't even use the processor $p_5$).

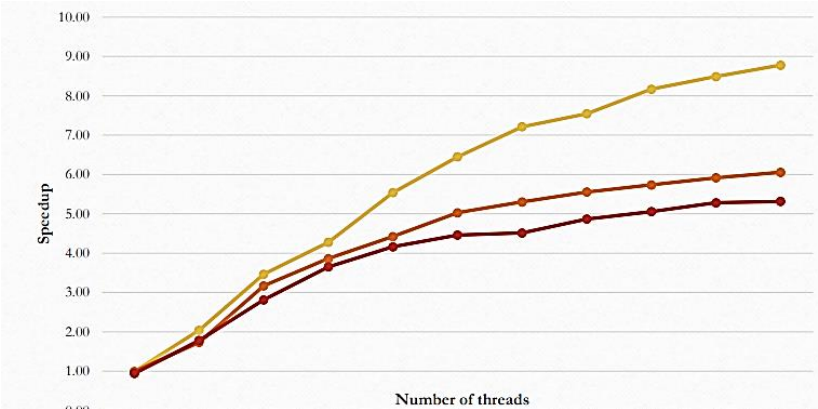
**Speedup**

$m = 2^{18}$ $n = 2^{15}$

The block-cyclic distribution (bc) uses c=8 as chunk size (to be near the size of the cache line)

Block distribution is the one that works best for this problem.

We have to try the different implementations to understand which is the best one, and we need to repeat the experiment more than 1 time to have statistical significance.




| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| block | 0.99 | 2.04 | 3.47 | 4.28 | 5.54 | 6.45 | 7.21 | 7.54 | 8.17 | 8.49 | 8.78 |
| cyclic | 0.99 | 1.73 | 3.17 | 3.86 | 4.42 | 5.03 | 5.30 | 5.55 | 5.74 | 5.92 | 6.06 |
| bc (c=8) | 0.94 | 1.78 | 2.81 | 3.65 | 4.16 | 4.46 | 4.51 | 4.87 | 5.06 | 5.28 | 5.31 |

In this example we assume to have dense matrix and consequentially compute $b_i$ is the same for all i. When we have very unbalanced computations use static distributions is very risky, we could assign the costly computations to the same thread obtaining a speedup near one. To solve: dynamic distribution.

**(Lect 12)**

**Unbalanced workloads**

In the static data distribution, the data assignment is predetermined at the program start. No extra overhead at running time for assigning tasks to threads.

This static approach works quite well if the computation-per-data element does not vary too much and we know in advance the number of tasks to compute.

Otherwise, there could be a **load imbalance** among threads.

One option is to use very small chunk sizes of tasks and block-cyclic assignment strategies. However, when tasks are heavily skewed, only a few more tasks per thread may produce load imbalance. In the worst-case scenario, all heavy tasks can be assigned to the same thread.

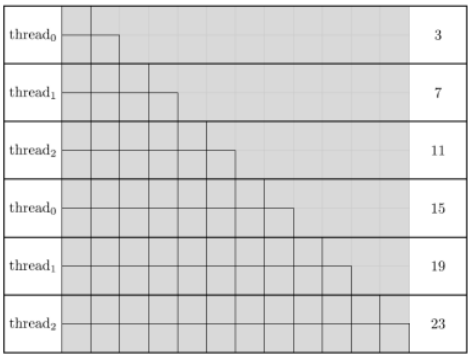
For unbalanced workloads it is better to use **dynamic assignment strategies**.

*(All-pairs distance matrix example)* We have a matrix $D_{ij}$ of shape $m \times n$ where *i* denotes one of the *m* vectors (for example, which image we are considering) and *j* enumerates all *n* elements of the vectors (for example, which pixel of the image we are considering).
We want to compute the distance (or more generally, the similarity) $d(\cdot, \cdot)$ between all pairs of vectors in $D$ (i.e. between all the images).

$\Delta_{hk} = d(x^h, x^k)$ for all $h, k \in \{0, \dots, m-1\}$

We must calculate $m^2$ distance/similarity scores between vectors. Assuming that the computation of a single score value takes $O(n)$ (for example using the Euclidean distance), we have $O(m^2 n)$ operations to compute the distance matrix Δ.

However, the matrix Δ is symmetric thus we have to calculate the lower triangular part of the matrix Δ *(compute i+1 entries in row i)* and then copy the elements in the corresponding position in the upper triangular part.




A straightforward parallelization of the problem uses a **static block-cyclic distribution** of row elements of the matrix Δ to threads for some value of the chunk c.

This an example considering a 12 x 12 distance matrix and c=2.

It can be better with c=1 but still it remains unbalanced ($thread_0$ 22 tasks, $thread_1$ 26 tasks, $thread_2$ 30 tasks)

The workload of the threads is **unbalanced** (increasing c we have even worse unbalance).

We need a different strategy that assigns elements to compute to threads dynamically.

| Static block-cyclic distribution | | | | |
|---|---|---|---|---|
| **chunk size (c)** | **1** | **8** | **32** | **128** |
| **Time (s)** | 37.28 | 41.52 | 45.31 | 45.27 |

*(Code example at slide 8) In the dynamic block cyclic we just keep the block which we have already computed (shared between threads). When a thread finishes to process its block, it can compute the next block to compute (using a mutex to read and update the shared variable). In this way there are not threads that finish their work and just wait for the thread at which was assigned a computational harder task.*

**Dynamic scheduling** guarantees better workload balance but has much **higher overhead** *(in the example getting and realizing a lock for each task we are computing is costly*). Sometime the extra overhead may have a big impact on the speedup, it **isn't' always true** than using dynamic policies bring a higher speedup. We have to choose considering the problem (if we know that using static policies, we have a not very unbalanced workload usually is better to use this).

**Producer - Consumer**

Classical concurrency pattern: two types of threads, **producers**, and **consumers**.

Producers **generate data** items and add them to a shared data structure (usually a queue). Consumers **take data** items from the **shared data structure** (usually one at a time) and consume them doing some work.

*(Slide 9) Code example.*
*std::jthread → thread with additional features*
*std::condition_variable_any → condition variable that allows to wait multiple events*
*std::stop_source → allows to cancel the execution of a thread (thanks to the stop_token)*
*Note: the condition to check for the conditional variable is inside the wait call*

**Multiple-Reader Single-Writer**

Classical concurrency pattern: two types of threads, **readers** and **writers**.

A reader may access the data concurrently with other readers. A writer needs exclusive access to the shared data.

*(Slide 10) Code example.*
*std::shared_mutex → allow the reader to access the critical section concurrently (shared_lock) (if there is no writer inside), while the writer access only with mutual exclusion (unique_lock)*

If we need different priorities between readers and writers to prevent potential starvation or readers or writers, we have to implement it by ourselves.

**Thread Pool**

Instead of creating threads on-demand for each task to compute, we can keep a pre-spawned set of Workers threads that we reuse for computing all the tasks at hand.

There is **no out-of-the-box implementation** of a Thread Pool (TP) in the C++ standard.

The idea is to insert tasks into the TP's queue, which returns a future. The threads of the pool (called Workers) execute the tasks and fulfill the associated promise to provide the results.

*(the function to compute and the data associated is enqueued)*

```
ThreadPool TP(8);  // 8 Workers in the pool

// function executed by the generic Worker
auto square = [](uint64_t x) { return x*x; };

std::vector<std::future<uint64_t>> futures;
for (uint64_t x = 0; x < N; ++x)
  futures.emplace_back( TP.enqueue(square, x) );
```

The Workers compete in accessing the concurrent task queue. Each Worker can execute whatever task (i.e., Workers are not specialized, but **generic executors**).

The **synchronization** is handled through mutex and condition variables (Producer-Consumer synchronization pattern)

In the implementation provided (threadPool.hpp) the task queue is **unbounded**. If the producers are much faster than the Workers, the tasks queue becomes huge, and the **memory blows up**.

To avoid this issue, we can:

- **increase** the **number of Worker** threads (if we have enough available cores)
- **set a limit** to the task queue and block the producers when the limit is reached.

*(Slide 14) Thread pool can be used to implement a dynamic scheduling of a number of tasks. We traverse the three and for each node we enqueue the task of computing the square root of number that label the node. The thread pool will compute the result for us and sooner or later we can read the result.*

If all workers execute the same function this is called task farm.

**(Lect 13)**

*Programming without lock (**Lock-free** programming).*
*We will do busy waiting instead of sleep waiting for a event.*
*Pros: it is more reactive.  Cons: we spend a part of time just busy waiting and other thread can't run.*

**Atomic data types**

C++11 introduces atomic data types that can be safely manipulated in a concurrent context without acquiring time-consuming locks.

*(Slide 3) Code example of shared counter with mutex or with atomic variable. The use of atomics is approximately 6 times more efficient than the traditional approach using mutexes and locks. This is because acquiring and releasing the lock introduce an overhead.*

**std::atomic<T>** :

- std::atomic is neither copyable nor movable.
- T must be copyable and movable.
- T cannot have a user-defined copy constructor.
- Operations (the methods of the std::atomic class):
    - **load**/**store**: to get and set the content of a std::atomic (but for integer some operation are already provided as adding, subtracting, …).
    - **exchange**: stores a new value and returns the old value of the variable.
    - **compare and exchange**: it does an atomic exchange only if the value is equal to the provided expected value.
    - **wait**/**notify** *(from C++20)* behavior similar to condition variables (but **actively** waiting not passively as in condition variable).

C++11 provides support for 8-, 16-, 32-, or 64-bits wide integers. Operations are guaranteed to be atomic in this case.

We can wrap structs and objects of different length with std::atomic<T>. The compiler realizes their concurrent manipulation with locks (the code you write remains correct even if the underlying HW does not natively support the corresponding data type).
We can use the method *is_lock_free( )* to check whether the object manipulation is done with atomics or mutexes.

*Example of operation guaranteed to be atomic with different type T.*

| Operation | atomic_ flag | atomic <bool> | atomic <T*> | atomic <integral -type> | atomic <other-type> |
|---|---|---|---|---|---|
| test_and_set | Y | | | | |
| clear | Y | | | | |
| is_lock_free | | Y | Y | Y | Y |
| load | | Y | Y | Y | Y |
| store | | Y | Y | Y | Y |
| exchange | | Y | Y | Y | Y |
| compare_exchange _weak, compare_ exchange_strong | | Y | Y | Y | Y |
| fetch_add, += | | | Y | Y | |
| fetch_sub, -= | | | Y | Y | |
| fetch_or, \|= | | | | Y | |
| fetch_and, &= | | | | Y | |
| fetch_xor, ^= | | | | Y | |
| ++, -- | | | Y | Y | |

**Compare-And-Swap (CAS)**

Every C++ atomic data type features a CAS (Compare-and-Swap) operation for implementing arbitrary atomic assignments.

Language provide two methods: ***compare_exchange_strong*** and ***compare_exchange_weak***. The last one is more efficient but might suffer from spurious fails (it may return false even if the comparison yields true). In any case is better to use weak as we usually put the code in a loop and we just try again.

```
atomic.compare_exchange_weak(T& expected, T desired);
```

1. Compare the value ***expected*** with the value stored in the atomic
2. If yes, then it sets the ***atomic*** to the value ***desired***, otherwise writes the **actual value** stored in atomic into ***expected*** *(to know what was contained in the atomic)*
3. Return ***true*** if the swap operation in step 2 was successful, false otherwise.

*(Slide 7) Example of code. Compute the maximum of a given array.*
*- **false_max**: occasionally computes the incorrect results since the condition tested in the 'if' and the assignment (line 17) are two independent operations (two or more threads might have read the same value before executing the assignment in a random order). We are using atomic in a wrong way.*
*- **correct_max**: uses a CAS in which the read and write operations are atomically executed. We read the current value with a load and then we use compare_exchange_weak, repeating until it succeeds.*

**Memory Consistency Models**

The intuition says that a memory read should return the latest value written in a memory location.

What does *«the latest value written»* mean in a CMP system when executing multiple threads?
It depends on the **ordering** of memory operations, precisely the order in which memory operations performed by one thread become visible to other threads. Modern CMPs **reorder memory operations** in different (strange) ways for performance reasons**.**

**Memory consistency** defines the allowed behavior of reads and writes (i.e., loads and stores) to different addresses in a parallel system.

*Why should we care of memory ordering?* It is fundamental if we want to implement lock-free data structures (if we use lock the ordering is guaranteed by the synchronization).

E.g.

```
// initially, A = B = flag = 0
Thread1                 Thread2
Store A = 1;            while (Load flag==0); // spin
Store B = 1;            Load A;
Store flag = 1;         Load B;
                        print A and B;
```

We expect Thread2 should print A=B=1

But If there are caches in the system, from the viewpoint of the cache coherence protocol, the printing of A=B=0 can be perfectly fine.

**Memory consistency** defines correct program behavior. Consistency is part of the specification of architectures (i.e., its model is visible to system software)

**Cache coherence** is a different concept, it deals with the objective of ensuring that the memory system in a parallel computer behaves as if the caches were not present. Coherence is not visible at the software level (we may see it as a side effect, e.g., false sharing, superlinear speedup, ...).

### Memory operation ordering

**Program order**: a program defines a sequence of reads and writes.

Four types of memory operation ordering:

- $W_X \rightarrow R_Y$: write to X must commit before the subsequent read of Y (i.e., when a write comes before a read in program order, the write must commit (i.e., its result must be visible) by the time the read occurs)
- $R_X \rightarrow R_Y$: read from X must commit before the subsequent read from Y.
- $R_X \rightarrow W_Y$: read from X must commit before the subsequent write to Y.
- $W_X \rightarrow W_Y$: write to X must commit before the subsequent write to Y.

### Sequential Consistency (SC)

Leslie Lamport (1979): *"A multiprocessor is **sequentially consistent** if the result of any execution (of the program) is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*.

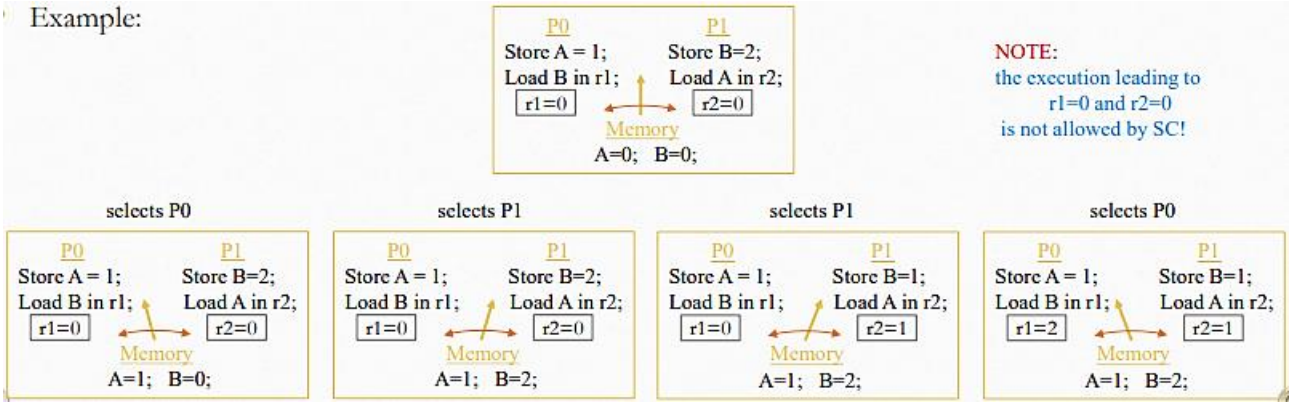
First precise definition of consistency. SC is the **most restrictive** memory consistency model.

Another equivalent definition for SC:

- There exists a **total order** of all loads and stores across all threads such that the value returned by each load is equal to the value of the most recent store to that location.
- There is a chronology of all memory operations that is consistent with observed values.

SC maintains all four memory operation orderings ($W_X \rightarrow R_Y$, $R_X \rightarrow R_Y$, $R_X \rightarrow W_Y$, $W_X \rightarrow W_Y$)

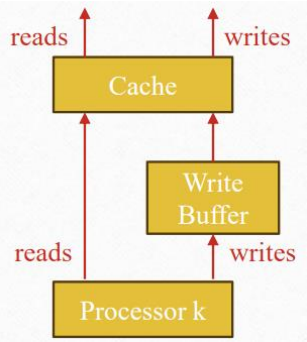
Switching metaphor: all processors issue loads/stores in program order, the memory chooses a processor at random, performs a memory operation to completion, then selects another processor, and so on.




A and B at the begging are zero. In this example we suppose to choose P0 as first memory operation. When the operation is completed, we switch to P1 and so on (choosing randomly). With this order we obtain A=1 and B=2. This is sequential consistency. Using sequential consistency, it cannot happen that we obtain A=0 and B=0.

### Motivation for Relaxed Consistency

**Relaxed memory consistency** models allow certain orderings to be violated. This is allowed to gain performance by hiding memory latency (overlapping memory access operations with other operations when they are independent).




Every modern processor uses Write Buffers for performance reasons.

The $W_X \rightarrow R_Y$ ordering is relaxed to hide write latency (because we don't read the write buffer but the cache or the memory).

This is perfectly fine from the viewpoint of a single-thread control flow.

If we relax $W_X \rightarrow R_Y$ ordering, in the previous example we can obtain A=0, B=0 (that probably is not what we expect).

**Total Store Order** (TSO): guaranteed every ordering relaxing only write before read ($W_X \rightarrow R_Y$). This the memory consistency model used by x86.

### Allowing reordering of writes

**Partial Store Ordering** (PSO) relaxes also $W_X \rightarrow W_Y$

Because the processor might reorder write operations in the Write Buffer for performance reasons (e.g., one write might be a cache miss, while the other might be a cache hit whose management costs less).

| Thread1<br>on P0 | Thread2<br>on P1 |
|---|---|
| A = 1;<br>flag=1; | while (flag==0) ; //spin<br>print A; |

Under PSO, P1 may observe the change to *flag* before the change to *A*, thus printing 0 (*A* initial value)

*Not allowed by SC and TSO.*

Why it might be useful to allow even more aggressive reordering?

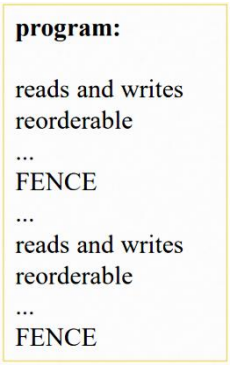
- Overlap multiple reads and writes in the memory system.
- Execute reads as early as possible and writes as late as possible to hide memory latency.
- An example of a fully relaxed model is **Weak Ordering** (all orderings are relaxed).

Intel memory model is not fully defined. It is a decision of the architecture.

***How can we do?***

Every architecture provides **synchronization primitives** to make memory ordering stricter:

- **Memory barrier instructions** (also called fences) prevent reordering, but they are expensive (all memory operations must be completed before any memory operation after the barrier can begin).
- Other synchronization primitives enable ordering on a specific memory address (e.g. *test_and_set*, *compare-and-swap*, and all *read-modify-write* (RMW) operations.




However, if a program is **Data Race Free** (DRF), then its **behavior is sequentially consistent**. If a program is DRF, reordering of instructions does not have an impact on program correctness.

The **compiler** may reorder instructions for performance reasons independently from the low-level **architectures** reordering. It is generally accepted that a compiler can reorder ordinary reads from and writes to memory almost arbitrarily, provided the reordering cannot change the observed single-threaded execution of the code.

Modern C++ (starting from C++11) and also the Java language **guarantee SC for DRF programs**, the compiler will insert proper synchronization to cope with the HW memory model. There aren't guarantees if your program contains data races.

**(Lect 14)**

**C++ memory model basics**

C++ provides the user with six memory ordering options for atomic types.
*Load*, *Store*, and *Read-Modify-Write* (RMW) accept **explicit ordering values**.
The **default** memory ordering is the stricter one: sequential consistency (*std::memory_order_seq_cst*)

- **Store** operations can have *std::memory_order_relaxed, std::memory_order_release*, or *std::memory_order_seq_cst* ordering values
- **Load** operations can have *std::memory_order_relaxed*, *std::memory_order_acquire*, or *std::memory_order_seq_cst* ordering values
- **RMW** operations can have *std::memory_order_relaxed*, *std::memory_order_acquire*, *std::memory_order_release*, *std::memory_order_acq_rel* or *std::memory_order_seq_cst* ordering values

Chosen the order we want is the compiler that provide to put the correct synchronization mechanism (memory fences, ...) to let the program respect that ordering.

**Types** of memory ordering:

- **Sequential consistency**, loads/stores atomic operations happen in the order specified by the program.
- **Acquire**, only valid for atomic load operations. It prevents all loads/stores in the current thread from being moved to a position **before** the considered atomic load operation.
- **Release**, only valid for atomic store operations. It prevents all loads/stores in the current thread from being moved to a position **after** the considered atomic store operation.
- **Relaxed**, no restrictions in memory reordering of surrounding load/store operations. Only the atomicity of the actual operation is guaranteed.
- **Acquire-Release**, only valid for atomic exchange and compare-exchange operations (i.e. RMW operations). It is a combination of the Acquire and Release modes, simultaneously loads the current value in Acquire mode and stores a new value in the atomic variable in Release mode.

***Note****: in this next examples the x and y atomic variable are initialized at false and the store put them to true, so the condition on the load is equivalent to check if the store has been executed*

The **SC semantics** requires **a single total ordering** over all atomic operations with ordering *std::memory_order_seq_cst*

*(Example slide 21)* In this program assert at line 37 can never fire *(z always > 0 at the end)*, because either the store to x (line 9) or the store to y *(line 12)* must happen before (we don't know the order) of one of the loads on x or y.
If the load on y at *line 16* returns false, it means that the store to x at *line 9* must occur before the store to y at *line 12* and thus the load on x at *line 21* must return true (other way around is also valid, if the load on x at *line 21* returns false, the load to y at *line 16* must return true).

SC is the **most expensive memory ordering**, it requires global synchronization between all threads (enforced by the C++ compiler).

***Note****: writing a program using the memory orderings that provide the language we can then rely on the compiler that guarantees the ordering is preserved on all platforms*.

With **relaxed ordering** there is no synchronization among atomic operations.

*(Example slide 22)* In this code, the assert at *line 27* can fire, because the load at *line 15* can return false even if the load at *line 14* returns true (i.e. store to y is done before store to x).

***Note:*** *this program in x86 never fire the assert because it implements a TSO at architectural level, but this program is not correct because in another platform the assert can fire*.

**Acquire and Release Ordering**

If an atomic store in thread A is tagged "*memory_order_release"*, an atomic load in thread B from the same variable is tagged *"memory_order_acquire"*, and the load in thread B reads a value written by the store in thread A, then the store in thread A synchronizes-with the load in thread B.

All memory writes (including non-atomic and relaxed atomic) that happened-before the atomic store from the point of view of thread A, become visible side-effects in thread B. That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.

The synchronization is established **only** between the threads releasing and acquiring the same atomic variable. Other threads can see different ordering.

*(Example slide 23)* In this code, the assert at *line 37* can fire (z not incremented). This is because after thread C exits from *line 15,* we are sure it sees every write of thread A (they are synchronized), but we are not sure that the load at *line 16* is not executed before *line 15* as we use *memory_order_acquire* on different variables. *Same thing for thread D.*

*(Example slide 24)* In this code we obtained the same effect obtained with SC *(assert cannot fire)* but with less costly synchronizations.

The store to x at *line 9* happens before the store to y at *line 10* (release). The load on x at *line 14* happens after the load on y at *line 13* (acquire). *Line 10* and *line 13* are synchronized thanks to the while at *line 13* that waits the store at *line 10* to be completed. So, the load on x at *line 14* is always executed after the store on x at *line 9* and z is therefore incremented.

**Fences**

**Fences** are global operations that affect the ordering of other atomic operations without modifying any data. Fences are typically used for ordering *memory_order_relaxed* atomic operations.

C++11 provides *std::atomic_thread_fence(memory order)*

Fences can also be used to order non-atomic operation.

*(Example slide 25)* Store at *line 10* can't be done after the release fence at *line 11*. Load at *line 17* can't be done before the acquire fence at *line 16*. The two fences are synchronized thanks to the while at *line 15* that waits the execution of *line 12*. So, z is always incremented.

**Spin-lock** *(busy waiting approach)*

*(Slide 14)* Simple implementation of **spin lock** using C++20 features.

It uses an *atomic_flag* (a boolean flag). Operations on this type are required to be lock-free, we can use *test_and_set* and *clear*. C++20 provided also *test*, *wait*, *notify_one*/*all*.

It tries to **acquire the lock** at *line 11* using *test_and_set(...)*. If it fails, the lock method spins using the test (*line 12*) until it sees a change, and then it retries to acquire the lock. The while at *line 12* is not strictly necessary, but it uses *test(...),* that do only a read, and therefore is more power friendly than *test_and_set(...)*.

The *clear(...)* method (*line 16*) sets the *atomic_flag* to false (**release the lock**).

**Note**: implementing a lock makes the program **slower** because the kernel does it better. The standard mutex implements first a spin-lock, waiting for the lock without send to sleep the thread, and only after a while the lock is not realized it save the context and put the thread to sleep. Furthermore, if we have lots of thread spinning is not a good idea because there is high contention.

*(Slide 27)* Experiment on all-pairs distance matrix. With high-concurrency atomics behaves better that mutex, in the others case the is no difference. In this program the lock is not very much contented

because the threads are doing useful work and only time to time the try to acquire the lock. That's why the mutex works as well as atomic when there are not a lot of thread.

(Slide 28) Experiment on Max-Reduction code. Atomic-based version is consistently faster and reasonably stable. Our Spin-Lock implementation (almost) always worse than mutex lock.

**The ABA problem**

Atomics-based data structures might be faster to use than locks-based data structures. However, atomics-based code features the so called "ABA problem". This is a typical problem of pointer-based lock-free data structure.

I.e. it is possible that the behavior will not be correct due to the "hidden" modification in shared memory.

1. Thread 1 reads an atomic variable, `x`, and finds it has value `A`.
2. Thread 1 performs some operation based on this value, such as dereferencing it (if it's a pointer) or doing a lookup, or something.
3. Thread 1 is stalled by the operating system.
4. Another thread performs some operations on `x` that change its value to `B`.
5. A thread then changes the data associated with the value `A` such that the value held by thread 1 is no longer valid. This may be as drastic as freeing the pointed-to memory or changing an associated value.
6. A thread then changes `x` back to `A` based on this new data. If this is a pointer, it may be a new object that happens to share the same address as the old one.
7. Thread 1 resumes and performs a compare/exchange on `x`, comparing against `A`. The compare/exchange succeeds (because the value is indeed `A`), but this is the wrong `A` value. The data originally read at step 2 is no longer valid, but thread 1 has no way of telling and will corrupt the data structure.

The common approach to avoid the ABA problem is to add an extra tag to the data being considered. The tag is incremented each time the data is used. This way the CAS (Compare_And_Swap) will fail even if the address is the same.

**To be or not to be Lock-free**

Lock-free (non-blocking) algorithms are guaranteed to make progress even if one or more threads fault or are indefinitely delayed (**lock-free progress property**). This is generally not the case for lock-based (blocking) algorithms.

Apart from simple cases lock-free concurrency is **significantly more complicated** than blocking lock-based concurrency (ABA problem, memory instruction order, ...)

The relative **performance** of blocking vs. non-blocking algorithms depends significantly on the amount of concurrency (and the number of cores) in which they run. It also depends on the specific algorithm used. On dedicated environments with high concurrency and thus high contention on shared variables, lock-free algorithms are generally **more performant**.

**Work-sharing Thread Pool**

(Slide 31) Implementation of an extension of the Thread Pool class (*threadPoolWS.hpp* file)

The new version leverages some atomic variables (i.e., *active_threads*, a counter of the active thread in the pool) and provides two new methods: *wait_and_stop*, and *spawn*.

Instead of pushing all the tasks to compute in the queue of the pool in this version we may start a big task and then split it until there are available workers. We create new task dynamically and it is useful when we don't know in advance how many tasks we must compute.

*Spawn* method: it takes the function to compute and its arguments. If there are idle thread then we enqueue the new task in the queue of the thread pool, otherwise the function is computed sequential inside the method.

*Wait_and_stop* method: it uses a condition variable to waits that there are no more active threads, and the queue of the thread pool is empty. At this point the thread pool can be stopped.

This new version works only with the spawn method, if we just enqueue tasks as in the previous version it doesn't work.

*(Slide 32)* Example of tree traversal with Work-sharing thread pool: we start the task giving the root as starting point. The thread will, after doing something, call another spawn on a subtree and then call recursively the function to traverse the other subtree (i.e. we split the task). When all the thread are busy the spawn will compute the function sequentially (no splitting).

*(Slide 34)* Binary Knapsack problem: this problem exhibits superlinear speedup when the optimal solution is determined using a branch-and-bound algorithm that recursively explores the complete state using a depth-first search. The superlinear speedup comes from the more aggressive pruning of worse solutions when two threads visit the two sub-trees and update the solution found so far (i.e. the cooperation of the treads allow to reduce the number of the computation to perform).
But if in the sequential version we reverse the order of the tree traversal we get much faster sequential version, and the speedup is no more superlinear.

The point is that superlinear speedup is often a kind of cheating, we must be sure that the sequential algorithm is a good one.

**(Lect 16)**

**OpenMP**

**OpenMP** in an API for platform-independent shared-memory parallel programming in C, C++, and Fortran providing high-level parallel abstractions atop low-level multithreading features.

OpenMP extends C, C++ and Fortran programming languages with directives, a few library routines, and environmental variables.

It is natively supported by almost all compilers (GCC, Intel, …). It is used to exploit shared-memory parallelism on small CMPs and on supercomputer nodes with hundreds of cores. The compiler generates threaded programs and all needed synchronizations.

OpenMP is **not** a parallelizing compiler (i.e., it does not automatically parallelize sequential code). Parallelism is explicit. Avoiding data races and obtaining good speedup is programmer's responsibility.

OpenMP does not require that single-threaded code is changed for threading. It preserves **sequential equivalence** (the idea is that we compile code without OpenMP we have the original [working] sequential version). We can implement parallelization incrementally on different part of the code.

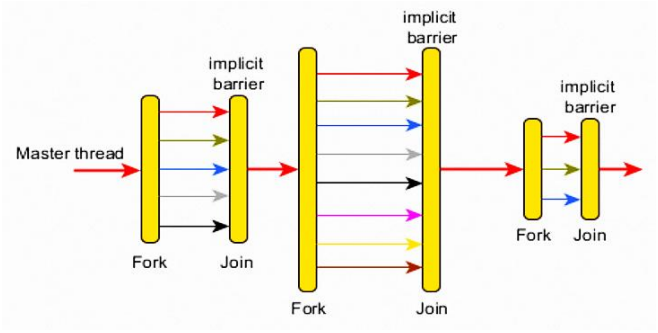
OpenMP relies mainly on compiler directives (directives increase the power if a language). If the compiler does not recognize a directive, it ignores it, and code remains sequentially.

### OpenMP execution model (Fork-join)

The execution starts with a single thread called **Master thread**.

The Master thread creates a **team** (or pool) of Worker threads to execute **parallel regions** where tasks are computed in parallel by Workers.

At the end of a parallel region there is an **implicit barrier synchronization**; after the barrier only the Master thread continues the execution.




### OpenMP Memory Model

OpenMP provides a **relaxed consistency model** that is similar to weak ordering, it allows the reordering of accesses within a thread to different variables.

Each thread has its temporary view of the memory (induced by machine registers, cache, etc.). Additionally, each thread has its ***threadprivate*** memory that cannot be accessed by other threads.

The **flush** directive is used to **enforce consistency** between a thread's temporary view of memory and the primary memory (or between multiple threads' views of memory).

Flush operations are implied by all synchronization operations in OpenMP (e.g., barrier, at entering/exiting to/from a parallel region, at the exit from workshare regions, ...).

### OpenMP directives

C and C++ compilers use *#pragma omp prefix*

Pragmas are preprocessor directives enabling behaviors that are not part of the language.

Structure: *prefix directive [clause list]*

Example: *#pragma omp parallel num_threads(8)* (a given region is executed in parallel with 8 threads)

Most OpenMP constructs apply to the **structured block** following the directive.

```
#include <omp.h> // needed only for using library routines

int main() {
#pragma omp parallel
 {  // <- spawning of threads (Workers)
    int i  = omp_get_thread_num() ;
    int n = omp_get_num_threads();
    std::printf("Hello from thread %d of %d\n", i,n);
 } // <- joining of threads
```

All the threads will execute the code between the { }

*omp_get_thread_num( ):* returns the ID of the Worker thread.
*omp_get_num_threads( ):* returns the size of the thread pool.

The *-fopenmp* **compiler flag** enables OpenMP support.

```
[torquati@spmln spmcode5]$ g++ omp_hello.cpp -o omp_hello -fopenmp
[torquati@spmln spmcode5]$
[torquati@spmln spmcode5]$ OMP_NUM_THREADS=6 ./omp_hello
```

The **environment variable** *OMP_NUM_THREADS* can be used to set the default number of threads in the teams that will be created for the parallel regions (if not already specified with *num_threads(num))*. If not set it will the number of logical cores of the machine.

*(Slide 10)* A structured block can be a single function call (in the example a C++ lambda).

The -fopenmp compiler flag defines the preprocessor variable *_OPENMP* that can be useful to include/exclude code at compile time.

*(Slide 11) if* clause is evaluated and if its expression evaluates to true, the parallel construct is enabled with num_threads thread, otherwise is ignored. A way to **conditionally** enable parallelism or keep sequential of a specified part of code.

To measure the time, OpenMP provides a library routine *omp_get_wtime( )*, which returns wall clock time in seconds.

**OpenMP parallel directive** *#pragma omp parallel [clause list]*

When a thread reaches a parallel directive, it creates a pool of threads and becomes the Master (the master has ID 0).
Each thread of the pool computes the code of the parallel region (i.e., the structured block following).
At the end of the parallel region there is an implicit barrier

Some commonly used clauses:

- **if** (scalar expression): determines whether the directive creates threads.
- **num_threads** (integer expression): number of threads to create.
- **private** (variable list): specifies variable local to each thread.
- **firstprivate** (variable list): similar to private, the variables are initialized to variable value before the parallel directive.
- **shared** (variable list): specifies that variables are shared among all threads in the region. This is the default visibility scope of variables!
- **default** (data scoping specifier): default data scoping specifier for the given region.
- **reduction** (operator : variable list): specifies how to combine local copies of a variable in different threads into a single copy at the master at region exit.

**OpenMP worksharing directives**

Within the scope of a parallel directive, a worksharing directive specifies that the created threads should execute statements, blocks, loop iterations, and tasks cooperatively (i.e., in parallel).

There are four worksharing directives:

- **for**: threads cooperatively execute loop iterations.
- **sections**: threads cooperatively execute tasks.
- **single**: one thread executes the code, other threads wait.
- **workshare**: divide the execution of enclosed structured block into separate units of work *(Fortran only)*.

E.g. *for* directive: *#pragma omp for [clause list]* it partitions loop iterations across threads.

```
#pragma omp parallel
{  // <- spawning of threads
   #pragma omp for
   for (uint64_t i = 0; i < num_entries; i++) {
     x[i] = i;
     y[i] = num_entries - i;
    }
   // <- implicit barrier
   #pragma omp for
   for (uint64_t i = 0; i < num_entries; i++)
     z[i] = x[i] + y[i];
   // <- another implicit barrier
   #pragma omp for
   for (uint64_t i = 0; i < num_entries; i++)
     if (z[i] - num_entries)
       std::cout << "error at position " << i << std::endl;
} // <- joining of threads (final barrier)
```

Spawning threads once at the beginning of the parallel region and joining at exit.

Loop is executed in parallel between the threads (the iterations are divided by the threads).

At the end of the loop there is an implicit barrier.

In the left example we need the implicit barrier but if it is not needed and we don't want to wait other threads:

Avoiding implicit barriers when not needed (**nowait** clause):

```
#pragma omp parallel
{  // <- spawning of threads
   #pragma omp for nowait
   for (...) { ... }

    // <- here there is no barrier

   #pragma omp for
   for (...) { ... }
} // <- joining of threads (final barrier)
```

Any thread can begin the second loop immediately without waiting for other threads to finish the first loop

**Variable sharing and privatization**

Variables scope:

- **shared**(x): all threads access the memory location where x is stored (this is the default).
- **private**(x): each thread has its private copy of x.
  - All local copies are **not** initialized.
  - Local updates to x are lost when exiting the parallel region. At the end of the region, x retains its original value.
- **firstprivate**(x): each thread has its private copy of x
  - All local copies of x are initialized with the value that x has before starting the parallel region.
  - Local updates to x are lost when exiting the parallel region. At the end of the region, x retains its original value.
- **lastprivate**(x): used in *for* and *sections* directives. It has the same semantics as private(x), but instead of dropping the local updates, at the end of the region x retains the thread local copy value of the thread that executes the last loop iteration or the last section.
- **default**(shared) or **default**(none): affects all variables not specified in other clauses
  - default(none): ensures that you must specify the scope of each variable in the parallel block.

*(Slide 16)* Example of use of variable scope.
A is private and will be a local copy of non-initialized (compiler warning).
B is shared but for all threads start from b=1 so the increments in parallel overwrites the other threads increments (b++ is not atomic).
C is first private, each thread copy the initial values of C (=2) and then increments its local value (all thread will get value equal to 3).
D is default so shared, same behavior of B.

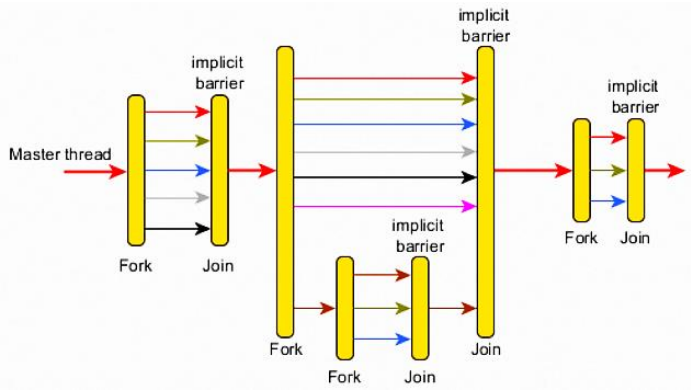
(Slide 17) Example of parallel loop. *i* is lastprivate so at the end of the loop its value will be 15.

**Nested parallelism**

Nested parallelism enabled using the OMP_NESTED environment variable.

Nested parallelism is not enabled by default because if not used with care will produce a lot of threads.

Note: *omp_get_num_threads( )* return the size of the innermost pool to which the calling thread is part of *(1 is no thread pool is active)*.




*omp_get_max_threads( )* return the maximum number of threads that can be created, i.e. the values of the environment variable OMP_NUM_THREADS.

**Reduction clause for the parallel directive** *#pragma omp parallel reduction (op : variable list)*

It specifies the **binary associative operator** to use to combine local copies of a variable in different threads into a single copy at the master when threads exit.

One private copy of each variable (compiler privatizes each variable by default when we use reduction) in the variable list is created for each thread. Each copy is initialized with the neutral element of the reduction operator (e.g., 0 for +, 1 for * , largest positive number for min), then each thread executes the parallel region. At the end of the region execution, the reduction operator is applied to the last value of each local reduction variable, and the initial value of the reduction variable it had before entering the parallel region.

Example: x is initialized by 1 (because reduction is done with *). Then we sum 3 and each thread now has x=4. Reduction for x is * so we get 4^3 * 5 (4^3 is the reduction for the 3 threads and 5 is for the initial value of x).

Reduction operators: +, * , -, |, ^ , &&, ||, min, max

*(Slide 20) Example of reduction code*

```
int x=5;
int y=1;
#pragma omp parallel reduction(* : x) \
                     reduction(+ : y) num_threads(3)
{ // <- spawning 3 threads, x set to 1, y set to 0
  x  += 3;
  y  += 3;
} // <- joining threads
 // the value of x here is 320 ( (1 + 3)^3 × 5 )
 // the value of y here is 10 (3 × 3 + 1 )
```

**(Lect 17)**

**Mapping iterations to threads** *#pragma omp parallel for schedule(scheluling_class [, chunksize] )*

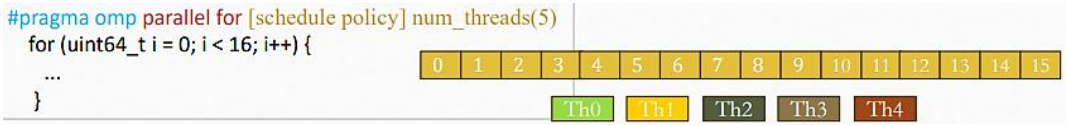
The schedule clause of the for directive enables different loop iterations **distribution policies** (iterations scheduling). There are four classes:

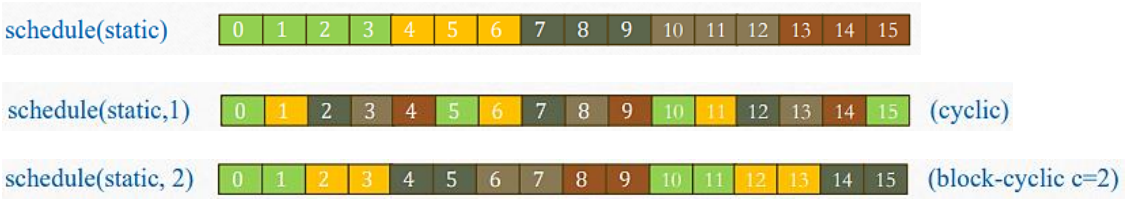
- **Static**: work assigned to threads statically at compile time in a cyclic fashion. If chunksize is not set, then $\left\lceil \frac{\#iterations}{nthreads} \right\rceil$ The iteration space is divided into pieces of size chunk.
- **Dynamic**: iterations assigned dynamically at run time at a granularity of chunksize (default chunksize=1). When a thread completes one chunk, another one (if available) is assigned to it.
- **Guided**: iterations assigned dynamically at run time at a granularity that is exponentially reduced with each dispatched piece of work up to chunksize (the default minimum chunksize is 1). Large chunk at the start and then in reduce them in an exponential way. (The actual implementation depends on the compiler).

- **Runtime**: the scheduling policy depends on the OMP_SCHEDULE environment variable (e.g., OMP_SCHEDULE="dynamic,10" ./myprog). Useful to decide later what is schedule to use.
- **Auto**: delegates the decision of the scheduling to the compiler and the runtime system

The default schedule is implementation dependent. In GCC it is static.

*Examples:*




### Dynamic schedule

For the **dynamic** and **guided** schedule clauses the iterations are split into chunks of chunksize consecutive iterations.

In the **guided** schedule, the chunksize is a **lower bound**. Bigger chunks are assigned at the beginning, and as chunks are completed, the size of the new chunks decreases up to chunksize.

When a thread finish executing a chunk, a new chunk is assigned by the runtime system:


*(a hypothesis)*

Logically, it is a master-worker computation paradigm where the master dispatches tasks to Workers.

The dynamic schedule requires **more work at runtime** (thus more overhead than the static schedule), but it guarantees **better workload balancing** among Workers in the presence of **unpredictable** or **highly variable** work per iteration. The chunksize is another critical factor, the optimal value is system- and application-dependent.

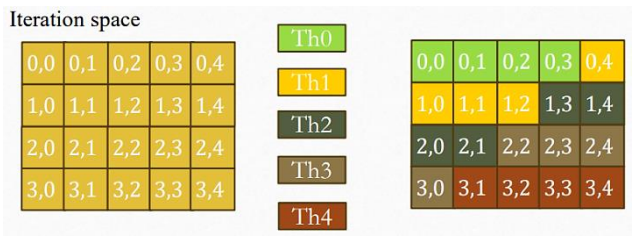
If we know that the work in the iteration is well-balanced using static is a good idea.

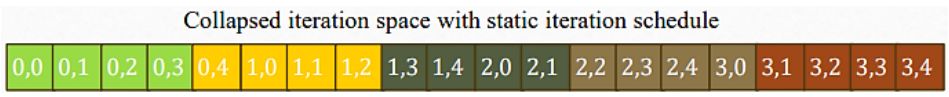
*(Slide 25) Example of different timing for All-pairs distance Matrix computation*

### Collapse clause

It specifies how many loops in a loop nest should be collapsed into one large iteration space and then divided according to the schedule clause.

```
#pragma omp parallel for schedule(static) \
                num_threads(5) collapse(2)
  for (int i = 0; i < 4; ++i)
    for(int j=0; j < 5; ++j)
      do_something(i,j);
```




*Note: we need an iteration space that can be flatten*

**Worksharing sections directive** *(not very used)*

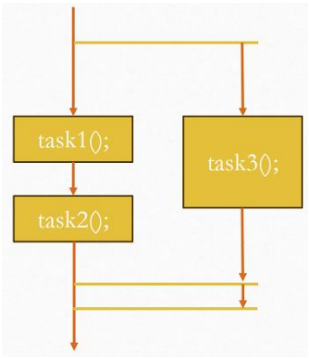
```
#pragma omp sections [clause list] {
      [#pragma omp section
        // structured block
      ] // square brackets means optional!
}
```

The sections directive enables specification of task parallelism.

It specifies that the enclosed section(s) of the structured block have to be divided among team threads.

Each **section** is executed **once** by one thread. Different section may be executed by **different threads**. There is an **implicit barrier** at the of a sections directive, unless the nowait clause is used.

```
#pragma omp parallel num_threads(2)
{
  #pragma omp sections
  {
    #pragma omp section
    {
      task1();
    }
   #pragma omp section
    {
      task2();
    }
   #pragma omp section
    {
      task3();
    }
  } // <- implicit barrier
} // <- implicit barrier
```




The execution order of tasks is not known.

The assignment of tasks to thread is non-deterministic.

(Slide 29) Example of worksharing sections directive

**OpenMP synchronization constructs**

***#pragma omp barrier***: all threads in the active team must reach this point before they can proceed

***#pragma omp single [clause list]***: mark a parallel region to be executed only by one thread (the first one reaching it, others skip the region). At the end of the block there is an **implicit barrier**.

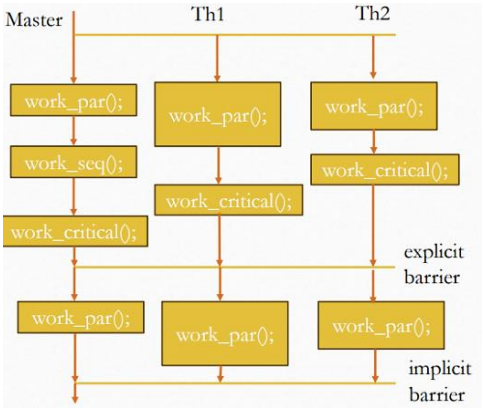
***#pragma omp master***: mark a parallel region to be executed only by the Master thread (the one with ID 0). There is **no implicit barrier** at the end of the block.

**#pragma omp critical**: mark a parallel region to be a critical section. All threads will execute the critical section one at a time.

**#pragma omp ordered**: in loops with dependencies, ensures that carried dependencies do not cause data race. It states that the structured block must be executed in sequential order (preserve the iteration ordering). This reduces a lot the parallelism but in some cases we still benefit of a little bit of parallelism.

**#pragma omp atomic**: only one thread at a time updates a shared variable

```
#pragma omp parallel
{
  work_par();

 #pragma omp master
  work_seq();
  #pragma omp critical
  work_critical();
  #pragma omp barrier

  work_par();
} // <- implicit barrier
```




**OpenMP tasks**

Even if loops are the main sources of parallelism, not all programs have easily parallelizable loops. For example, loops in which the iteration count is not known in advance, containing complex dependencies, Divide and Conquer algorithms, data streaming computations (producer-consumer).

A task is an **independent unit of work** enclosing a function code to execute, data and in general its data environment, control variables.

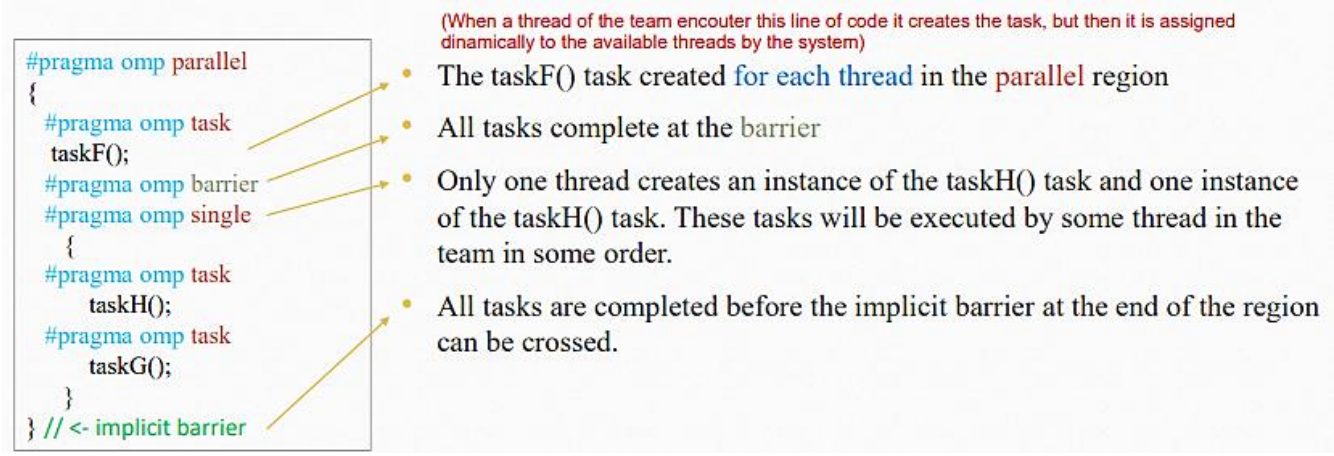
The execution of a task by a thread of the team can begin immediately or can be deferred (executed later, this depends on the scheduling of the OpenMP library).

A task can be tied to a thread (i.e., task stealing is not allowed). By default, a task is tied to the first thread that executes it.

Creating tasks with the task directive is more flexible than sections

*#pragma omp task [clause list]* Possible clauses in [clause list]:

- **if (scalar expression)**: determines whether the directive creates a task.
- **untied**: no scheduling restrictions for the task (can be stopped and restarted multiple time and can be moved between threads). A task, by default, is tied to the first thread that executes it.
- **private (variable list)**: specifies variable local to the child task.
- **firstprivate (variable list)**: similar to private, the variables are initialized to variable value before the parent directive.
- **shared (variable list)**: specifies that variables are shared with the parent task.
- **default (data scoping specifier)**: default data handling specifier may be shared or none.




**Data scoping for tasks**

If no default clause is specified, then:

- Static and global variables are **shared**.
- Local variables are **private**.
- Variables for **orphaned** tasks (thread that spawn the tasks have finished) are **firstprivate**. This is because the execution of a task can be deferred, and when the task will be executed, variables may have gone out of scope.
- Variables for **non-orphaned** tasks are **firstprivate** by default unless the **shared clause** is specified in the enclosing context.

```
int main() {
  int x =1;
  int y = 2;
  #pragma omp parallel private(y)
  {
    int z = 2;
    #pragma omp task
    {
      int w = 3;
      // x shared, y and z firstprivate, and w private
      // Note: y is not initialized
      F(x,y,z,w);
    }
  }
}
```

In the example, *w* is a local variable for the task so it is private. *z* is firstprivate (by default because is declared inside the parallel region and is consequently private), inside F will start with value 2 but after F its value will be back to 2 (updates inside F are lost). *x* is shared because is global (declared before parallel region). *y* is declared private by the parallel directive (so it is not initialized, and each thread

will have its own copy) and will be firstprivate inside the task (by default).

**Wait for the completion of tasks**

The ***#pragma omp taskwait [clause list]*** directive can be used to wait for the completion of child tasks defined in the current task. The taskwait suspends the parent task until all children's tasks are finished.

Note that this is less strictly than a barrier because we wait only for the generated tasks and not for all descendant tasks.

*(Slide 38) Fibonacci example*

**Task scheduling** *Important point: thread ≠ task*

**Tied** tasks:

- Only the thread that the task is tied to may execute the task.
- A task can be suspended at specific scheduling points (creation, completion, taskwait, barrier, ...)
- If a task is not suspended at a barrier, the thread executing the task can only switch to a descendant of any task tied to the thread.

**Untied** tasks:

- The task can be executed by different threads.
- There are no scheduling restrictions. The tasks can be suspended at any point, and the thread can switch to any task.

**(Lect 18-26)** On tablet

**(Lect 27)**

**Fastflow**

C++ parallel programming library promoting structured parallel programming both at the RTS level and at the higher level by using parallel patterns.

FastFlow is a class library, i.e. a set of header files. Since this, when compiling is important to use -O3 to optimize the code of FastFlow (in a real library code has been precompiled with optimization).

**Interface**

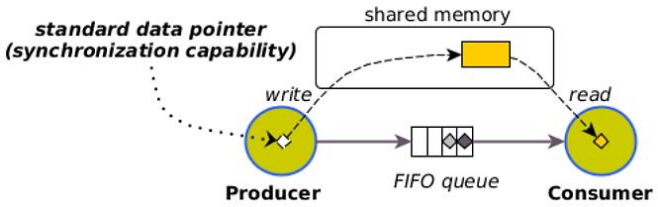
Two layers: building blocks (lower level) and parallel patterns (higher layer). Parallel patterns have higher abstraction (much less code required), while the lower level is much more flexible.

**Programming model**

A FastFlow application is a **directed graph** whose nodes are computing entities (either sequential or parallel building blocks) and edges are channels carrying **data references** (pointers)

To the buffer we can write only buffer, not plain data. This is done for efficiency reason (not using locks or atomic instructions).

Graph's nodes can be stateless or stateful, and they synchronize through message-passing for accessing shared data.




Writing a data pointer into the FIFO queue enforces write memory ordering.

To ensure to avoid data race when a node pass a pointer into the FIFO queue, it can't reuse it to write again. But the same pointer can be used by the next node do write and pass it again to the next step.

**Communication channels**

A channel connecting two FastFlow nodes is implemented by using an SPSC (Single-Producer Single-Consumer) FIFO queue.

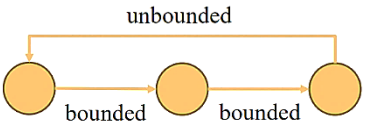
Distinct input/output channels to a single FastFlow node are implemented by using **independent** SPSC FIFO queues to enable input **non-determinism** without any lock/atomics (node receiving from multiple nodes without any assumption on order/source. It can see from who is receiving but can't ask to receive from a specific node) and **output selectivity** (if we want, we can choose where to send data)

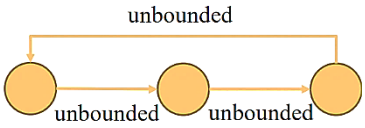
**Concurrency control**:

- **Non-blocking/lock-free** *(default implementation)*: node uses active-waiting loops with some back-off before trying again if the operation fails (e.g., push/pop operations in an SPSC queue). No atomics nor locks. Everything is non-blocking: lot of resources used but very reactive, i.e. we have slightly higher throughput but also higher power consumption.
- **Blocking**: standard protocol based on condition variables (waiting for a signal/event)

**Channel capacity**:

- **Bounded**: fixed predetermined size
- **Unbounded** *(default)*: the capacity is not limited.




Feedback channels have always unbounded capacity to avoid deadlock.

To send a pointer the latency is order of tens of nanoseconds.

**Data streaming**

Data streaming is a **first-class concept** in the FastFlow library. FastFlow is born for data streaming computation, then the same concepts have been used for non-stream-based problems. The stream is a general concept that can be used also for non-stream-based problem (but not always with the best performance).

A **stream** is a sequence of data values (possibly infinite), coming from one or more sources. The stream items coming from the same source usually have the same data type.

**Terminology**: stream source is the first stage of a FastFlow pipeline (sink is the last one). There is no specialized node for sources/sinks in FastFlow (it is always the same node).

**Basic Sequential node**

The node has an input queue to receive data and an output queue to send data.

**svc_init( )** is called once at the beginning, each time the thread associated with the node is started (or restarted).

**svc( )** is called for each input item present in one of the input queues (the only mandatory method). Its return value can be:

- A memory pointer to the data result
- EOS (end-of-stream, used to terminate), GO_ON (usually returned by sink nodes) and few other "special values".

ff_send_out can be used to produce more than one output in one activation (used instead of return).

```
class myClass: public ff_node_t<IN_t, OUT_t>{
public:
  OUT_t* svc(IN_t * in) {   // mandatory
     <business logic code of the node
      producing the output task (out)>;

     return out;  // ... also GO_ON, EOS
  }
  int svc_init() { .....; return 0; }   // optional

  void svc_end() { ..... }              // optional

  void eosnotify(ssize_t ch) { ... }  // optional

protected/private:
   <local-state, if any>
};
```

**svc_end( )** is called before terminating the node (or when the node is put to sleep). It is not possible to send any "final" result into the output queues.

**eosnotify( )** is called each time an EOS message is received by the node. It is possible to send data out into the output queue (or queues).

**Multi-Input/Output Sequential node**

**Single-Input Multi-Output node**: ff_monode_t<IN_t, OUT_t>
ff_send_out(ptrdata) or ff_send_out_to(ptrdata, channelID) [the second can be used to send the data in a **specific output channel** (channelID in [0,N-1]) ]

**Multi-Input Single-Output node**: ff_minode_t<IN_t, OUT_t>
ff_get_channel_id() to know from which channel the node received the input (from 0 to N-1)

**Multi-Input Multiple-Ouput** nodes do not exist.

FastFlow nodes cannot be spawned dynamically. They are used within BB (building blocks) parallel components. Usually, a node is **executed by a single thread** of control for all its lifetime.

**Sequential BB life cycle**

Runtime system spawn one thread for each node in the network.

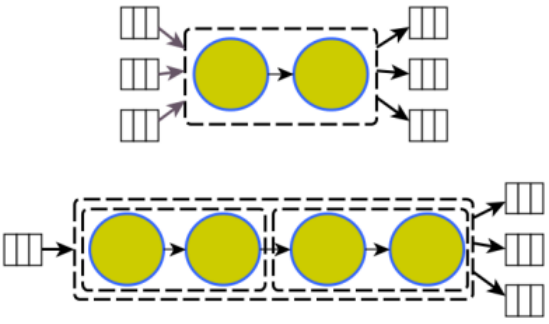
When a node input is EOS, it sends EOS and terminate.

If the input is something else it produces something with svc and send it to the output channel (unless the return valued by svc is GO_ON, in this case it just goes to the next input).

Note: in the svc_end( ) we can't do any communication with the others node.

```
do {
   if (svc_init() < 0) break;
   do {
      in = input_channel.pop();
      if (in == EOS) {
        eosnotify();
        output_channel.push(EOS);
        break;
      }
      out = svc(in);
      if (out == GO_ON) continue;
      output_channel.push(out);
   } while (out != EOS);
   svc_end();
} while(!shouldIrestart());
```

**Combining sequential BBs**

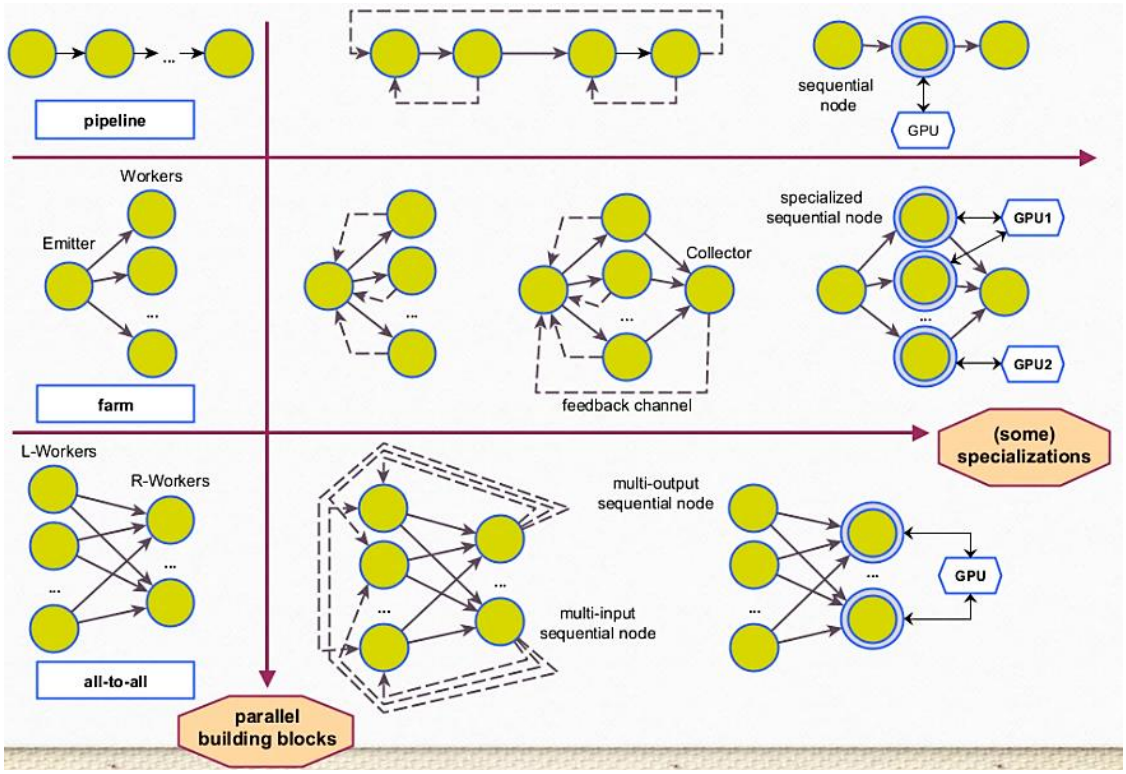


Sequential nodes can be combined like function composition (ff_comb BB). A combined node is executed by a single thread, thus decoupling nodes from thread.

$$out = svc_4(svc_3(svc_2(svc_1(in)))$$

We reduce the number of nodes without changing much the code.

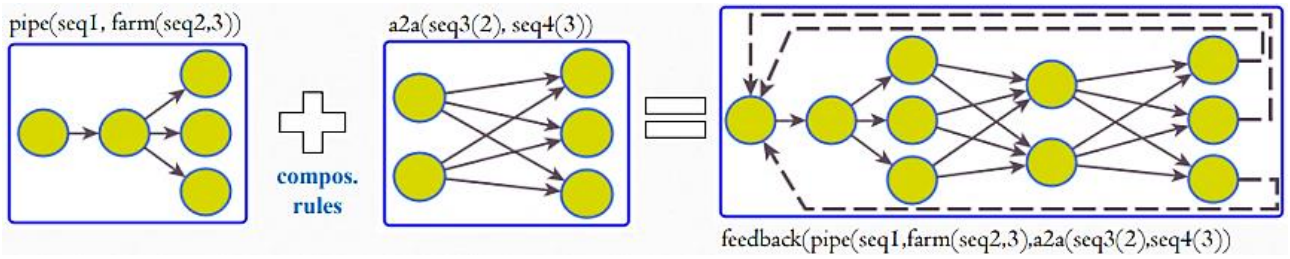
**Parallel BB**




All-to-all is a generalization of the farm. Instead of having a single master (that is a potential bottleneck) we have a list of emitters (L-Workers).

**Structured composition of BBs**

Building blocks are bricks that can be composed using rules. With simple building blocks we can build complex structures.




(Slide 21-22) Example of combination application

(Slide 23) Example of adding all-to-all

### BBs and basic High-Level patterns

| | BBs | High-Level Pattern |
|---|---|---|
| pipeline | ff_pipeline pipe;<br>pipe.add_stage(S1);<br>pipe.add_stage(S2); | ff_Pipe pipe(S1, S2); |
| farm | ff_farm farm;<br>farm.add_emitter(…);<br>farm.add_workers(…);<br>farm.add_collector(…);<br>farm.set_ordered(); // if needed | ff_Farm farm(Workers, Emitter,Collector);<br><br><br><br>ff_Ofarm ofarm() |
| all-to-all | ff_a2a a2a;<br>a2a.add_firstset(….); | no high-level pattern available, yet |

There is 1-to-1 mapping between BBs and **basic** High-Level patterns. The difference is only syntax.

farm.set_ordered( ) is used to preserve the order of the input stream. Ofarm is the high-level equivalent.

### Pipeline


Pipeline stages are sequential nodes. A pipeline itself is a node (from the FastFlow point of view).

It is possible to build pipe of pipes.

The first stage is the source node.

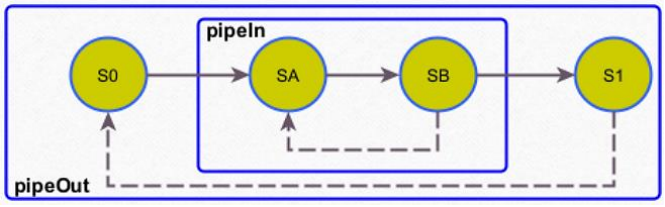
A node, for each input, may generate zero, one, or many outputs.

```
#include <ff/ff.hpp>
using namespace ff;
struct Source: ff_node_t<mytask_t>{
  mytask_t* svc(mytask_t* in) {
    for(long i=0;i<N;++i)
      ff_send_out(new mytask_t(i));
    return EOS;
} S0;
struct Stage: ff_node_t<mytask_t>{
  mytask_t* svc(mytask_t* in) { return in; }
} S1;
struct Sink: ff_node_t<mytask_t>{
  mytask_t* svc(mytask_t* in) {
    foo(in);
    return GO_ON;
} S2;
ff_Pipe pipe(S0, S1, S2);
pipe.run_and_wait_end();
```

### Feedback channels

**Feedback channel**: its data flow is directed in the opposite direction than the standard data flow.

Used to route back results, or to provide control information to a previous node. They always have an unbounded capacity to avoid deadlock.




In the picture, SA is multi-input, SB is multi-output, S0 and S1 are standard sequential nodes (ff_node_t). For nodes with feedback, the feedback channel has always the lower id (zero).

```
#include <ff/ff.hpp>
using namespace ff;

ff_Pipe pipeIn(SA, SB);
pipeIn.wrap_around();

ff_Pipe pipeOut(S0, pipeIn, S1);
pipeOut.wrap_around();

If (pipeOut.run_and_wait_end() < 0)
    error(''running pipeOut'');
```

After creating the pipe, the pipe itself is a node that can be used. Method wrap_around( ) creates the feedback channel.

**Handling termination with feedback channels**

Feedback channels **complicate** program termination.

**Standard termination**: a node terminates when it receives an EOS message from all its input channels. Then it forwards the EOS into all its output channels and then calls svc_end( ).

**With cyclic graph**: the previous condition does not hold. We need to use the eosnotify( ) method to be notified each time an EOS is received and depending on some application-specific conditions to properly forward the EOS at the right moment.

In previous example, SA can terminate only when receive EOS from S0 and SB. But to receive EOS from SB, SB must receive EOS from SA… This doesn't work.

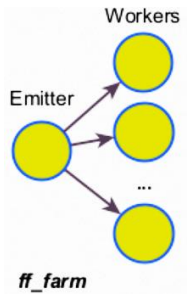
To solve this problem, we count the number of on-the-fly message we have in the loop. The counter is incremented every time a message from the previous stage is received. The counter is decremented when a message is received from the feedback channel.

```
struct SA: ff_node_t<T>{
  T* svc(T* in) {
    if (!this->fromInput()) {  // from feedback
     if ((--on_the_fly == 0) && eosreceived)
        return this->EOS;
      return this->GO_ON;
    } // from input
    on_the_fly++;
    return in;
  }

  void eosnotify(ssize_t id) {
    eosreceived = true;
     if (on_the_fly == 0)
       this->ff_send_out(this->EOS);
  }

  bool eosreceive=false;
  long on_the_fly=0;
}
```

When the EOS from the previous stage is received and there are no on the fly message we can terminate. To know if we received the EOS from the previous stage, we use method *eosnotify( )*. This is a virtual function called by the runtime (if it is redefined in the class) each time we receive an EOS by any of the input channel. In the code there is *"if(on_the_fly == 0)"* because can happen that all the messages have been received back before getting the EOS from the previous stage (Note: in the eosnotify we must use *ff_send_out( )* to send messages).

**(Lect 28)**

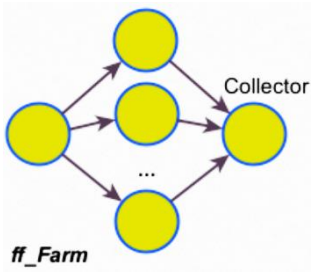
**Farm**




Farm Workers can be any BB. The number of replicas (parallelism degree) is given by the size of the vector of the node objects.

By default, the **ff_Farm** (high level pattern) has an Emitter and a Collector. The Collector can be removed (myFarm.remove_collector( ) )

The **ff_farm** (building block) has only the Emitter. The Collector can be added explicitly.



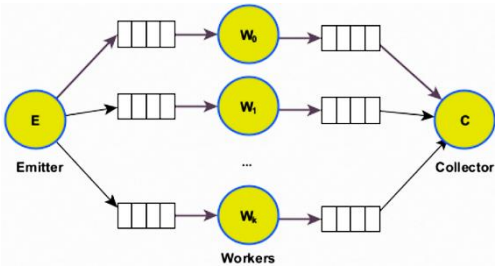

A difference with OpenMP is that there the master thread (the one who emits) is also a worker, here we have a dedicated thread (Emitter) which job is only to emit and a pool of workers.

*Note: in general thing with Uppercase are high level pattern, things with lowercase are building blocks*

Default task scheduling is **round-robin**. However, If the channels have bounded capacity the scheduling is "loose" round-robin.

Another option is **on-demand**, the emitter send data to one of the workers that is free.

It is possible to implement **user's specific scheduling strategies** by adding a *ff_monode_t* object (multi-output node) as Emitter and using the *ff_send_out_to* method to explicitly schedule tasks to specific Workers.




(Slide 29) Code to create a Farm within a pipe. We create a vector of workers (one entry for each worker, in this case 2). We pass the vector to the farm constructor. We create a pipe with the farm as second stage.

Pipe and farm BBs can be nested with almost any restrictions. If the farm used are without collector all the workers will be connected to the next stage of the pipe if its node is multi-input (third image slide 30), otherwise an error is raised.

**Farm task scheduling options**

The default policy is **round-robin** assignment of tasks starting from Worker with ID 0. This holds true if the channels between the Emitter and Workers have **unbounded** buffer capacity (communication channels have unbounded capacity by default).

By compiling the program with -DFF_BOUNDED_BUFFER, all channels (but the feedback ones) have bounded buffer capacity. The capacity of the channel buffer can be controlled (systemwide) with the compilation flag –DDEFAULT_BARRER_CAPACITY=<N>.

With **bounded buffers**, the task scheduling policy is "**loose round-robin**". If the next output channel is full, the Emitter does not wait for free-space availability and moves to the next channel (i.e., next Worker) until the task can be placed in one of the next channels.

A pure round-robin policy with bounded channel capacity can be implemented as a user specific scheduling using *ff_send_out_to* in the emitter and try to send to a worker until there is free space in its queue.

A scheduling option is the "**on-demand**" policy, which **emulates** the auto-scheduling policy in which Workers "ask for" a new task rather than passively accepting tasks sent by the Emitter. This policy is enabled by calling the method *set_scheduling_ondemand( )* on the farm object.

To implement this, it uses a bounded queue of size n (called *asynchrony level*, default n=1) between the Emitter and Workers (that will be full after the first round-robin scheduling because the worker is slower than the emitter). If the queue has free space, it means the Worker is "ready" (this is the trick to emulate the "I'm free" of the worker).

(Slide 33) Example of different implementations to compute the square root of the sum of dot product of a set of random vectors.

**(Lect 29)**

**Implementing a «real» on-demand policy**

We need to customize the farm's Emitter, and we need feedback channels to send control requests ("ready") from Workers to the Emitter.

```
struct Emitter: ff_monode_t<task_t, float> {
    float* svc(task_t* in) {
        ssize_t wid = get_channel_id();
        if (wid<0) { // tasks coming from input

            if (nready>0) {
                int victim = selectReadyWorker(); // get a ready worker
                ff_send_out_to(in, victim);
                ready[victim]=false;
                --nready;
            } else
                data.push_back(in);  // no one ready, buffer task
            return GO_ON;
        } // coming from Workers...
        if (data.size()>0) {
            ff_send_out_to(data.back(), wid);
            data.pop_back();
        } else {
            ready[wid] = true;
            ++nready;
            if (eos_received && (nready == ready.size()))
                return EOS;
        }
        return GO_ON;
    }
}
```

**[First if]** The message to the Emitter is not a ready message from one worker but a task coming from input
(victim is the selected ready worker, must be set to false in the ready vector)

**[Second if]** Send the data to the worker who sent the ready message (data is the data structure for the buffer).

**[Else]** If we don't have work to be done, we remember which worker have sent us the ready message.

To terminate check is we received the EOS (with *eosnotify( )* ) from the input and all workers are ready.

**Farm ordering**

The default implementation of the farm pattern **does not preserve the input ordering** of stream items. But in some applications preserving input ordering is mandatory (e.g. video/audio streaming).

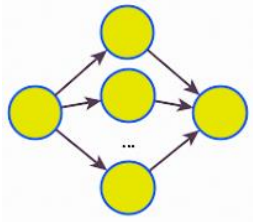
FastFlow provides the **ff_OFarm** pattern, which preserves input ordering when producing results.

There are some **limitations**:

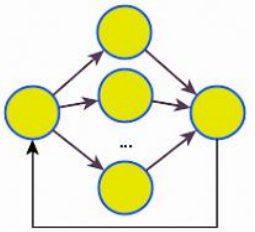
- Workers cannot change the length of the stream (e.g. a worker taking one input and producing 10 output items or not producing nothing outputting GO-ON using ff_send_out)
- Workers can only be standard sequential nodes.

It is possible to use on-demand scheduling and also to define user-defined scheduling policies by defining the Emitter and the Collector

**Rule-of-Thumb**: If you do not need a strict input/output ordering, it is **generally better** (from the performance standpoint) to use a standard farm and to implement your own policy by redefining the Emitter scheduling and the Collector gathering policies.



farm with Collector



farm + feedback

Is not possible to remove the Collector because it gathers the item in a certain order to maintain the input ordering.

Only standard sequential nodes can be used as farm Workers.

**Farm ordering implementation**

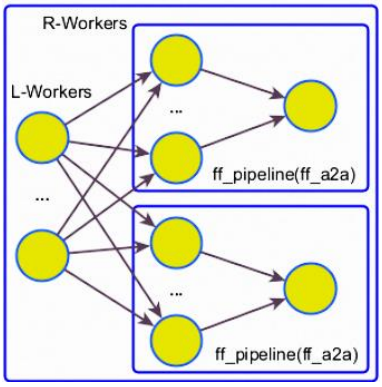
Custom Emitter and Collector used in ff_OFarm:

- scheduling of data elements is **strict round-robin** (even with bounded queues)
- gathering follows the **same ordering** used in data distribution.

This simple (and effective) policy does not work well if the workload is unbalanced.

For **non-deterministic scheduling** (e.g., on-demand, random) it is necessary to **tag** input data. Data tagging is automatically implemented in the ff_OFarm if set_scheduling_ondemand is used. The Collector buffer the data to recreate the correct order in output (in current implementation the Collector has a fixed size buffer that if is not enough must be reallocated → extra overhead).

**All-to-All (A2A)**

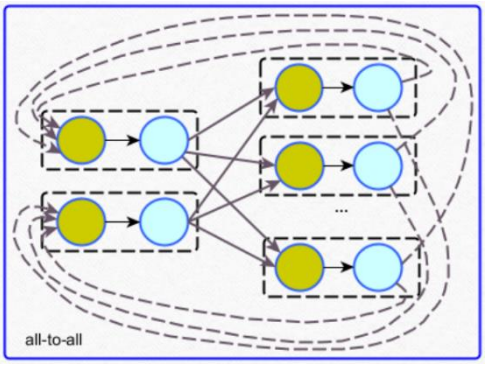



Multi-function replication BB called ff_a2a *(a generalization of the farm without a Collector and with multiple Emitters)*.
It gets two arrays of L-Workers and R-Workers.
Each L-Worker has a channel to every R-Workers.

*(In the example an All-To-All BB where the R-Workers are pipeline that embeds others All-To-All BB).*

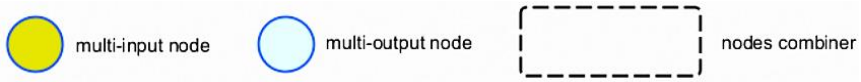
Scheduling and gathering policies are the same of the farm BB ("loose" round-robin, on-demand). No data ordering guarantees.

L-Workers must be **multi-output**, R-Workers must be **multi-input**.

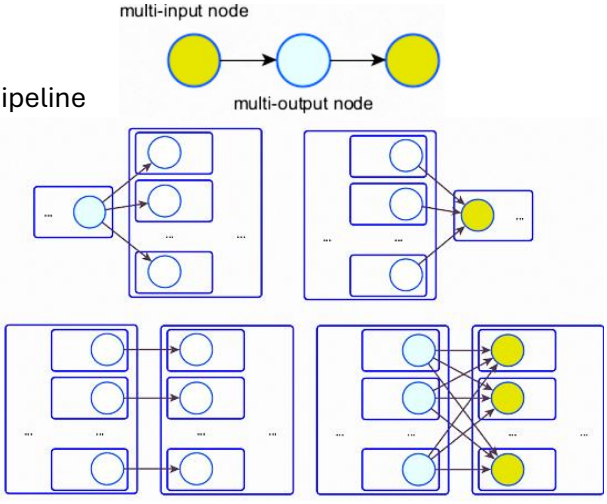



Because the nodes between the two nested All-to-All must be multi-input and multi-output (they are L-Workers for the inner All-to-All BB and R-Workers for the outer All-to-All BB) we use an helper node composed of a multi-input node and a multi-output node.

The exact same problem as to be tackled if we use feedback channels in All-to-All *(in the image)*.




**Composition rules of BBs**



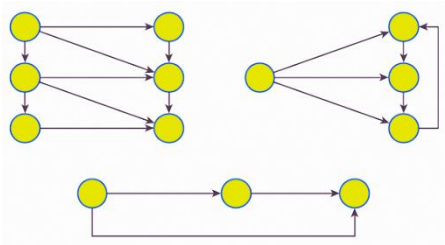

**Rule 1**: two sequential building blocks can be connected in pipeline regardless of their input/output cardinality.

**Rule 2**: a parallel building block can be connected to a sequential building block through a multi-output and multi-input node.

**Rule 3**: two parallel building blocks can be connected either if they have the same number of standard nodes or through multi-input/multi-output nodes.

**Graphs that cannot be built with FastFlow BBs**

Some examples of graphs that cannot be built by using FastFlow BBs. Only a subset of all possible concurrency graphs can be built … but all the most used ones can be built.



**(Lect 30)**

**CPU affinity**

It is a feature provided by several OS to **control threads** (processes) **placement** onto CPU cores. The placement of threads into cores is also called **thread mapping** (also **thread pinning**).

Deciding the **optimal** mapping of application threads to cores is **NP-hard**. There are several heuristics that work well in practice, which are implemented by the OS.

If the platform and the application structure are known in advance, it is possible to derive good mapping strategies to boost performance (e.g. caches are less flushed, load balancing, …)

In Linux, the ***taskset –c*** command allows users to bind the process execution to a given set of cores.

**CPU affinity in OpenMP**

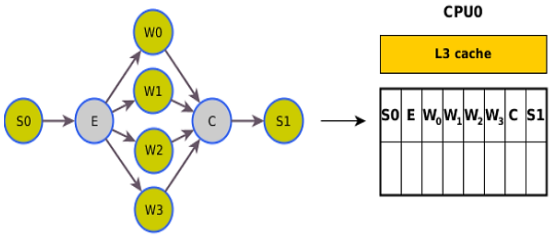
- OMP_PROC_BIND="true" ./my_omp_program (the threads spawned by OpenMP will be sticked to a given core and not moved)
- OMP_DISPLAY_AFFINITY="true" OMP_PROC_BIND="true" ./my_omp_program (allow to print the mapping between thread and core)
- OMP_PLACES="0,1,2,3,4,5,6,7,8" OMP_PROC_BIND="true" ./my_omp_program (provide a list of cored on which run the threads)

Other commands are available to have a fine grained control.

**CPU affinity in FastFlow**

**Default**: threads-to-core mapping following a depth-first graph (?) visit and assumes a linear numbering of cores. The cores ID are extracted from the FF_MAPPING_STRING (a string with the IDs separated by a comma) in the ff/config.hpp file.

Following this approach, threads are pinned to **sibling cores** (cores that are near. First thread to first number of the string, second thread to second number of the string, …). This is a reasonable heuristic in the Producer-Consumer model because we can exploit the cache to store the shared queue.




If FF_MAPPING_STRING is **empty** (e.g. when we have just downloaded FF) the runtime assumes a linear numbering of the cores. In some machine the numbering used by the OS is not linear, in this case we can use the script *mapping-string.sh* that inspect the cores and propose a string that can be added to the config file.

The NO_DEFAULT_MAPPING compilation flag **disables** the default thread mapping.

It is possible to map threads at **runtime**:

- in the svc_init method by calling the ff_mapThreadToCpu function
- by using the optimize_static function or the no_mapping() (disable thread mapping at run time) method of farm, A2A, and pipeline BBs.

If we have a lot of thread (more than the number of cores) usually disabling the thread mapping is the best option (let do the job to the OS).

*(Code example in SPMCode8, farmpinning.cpp)*

Note that in OpenMP there is a pool of threads that execute, while in FastFlow there is a structure (each node is a thread), so in FastFlow we have more chances to configure the affinity and having a performance boost.

**Concurrency control**

**Non-blocking** is the **default** concurrency control mode (all threads perform busy waiting). If the number of threads used is greater than the number of cores, the busy-waiting loop executed by the threads might introduce too much noise for other threads. In these cases, a blocking concurrency control mode for accessing the queues might be more appropriate even though it is intrinsically less reactive (it has more overhead).

In FastFlow it is possible to **change to the blocking mode** system-wide by compiling the code with - DBLOCKING_MODE.
It is also possible (as for thread mapping) to switch to blocking mode at **run-time** by using optimize-static or the blocking_mode() method of farm, A2A, and pipeline BBs.

The best configuration w.r.t CPU affinity and concurrency control mode depends on the application.

**Programming with High-Level Patterns** (pattern built atop BBs)

*We already seen a high-level pattern that is the ordered farm ff_OFarm*

**ParallelFor** *(Data-parallel pattern)*

Sequential code (π approx.):

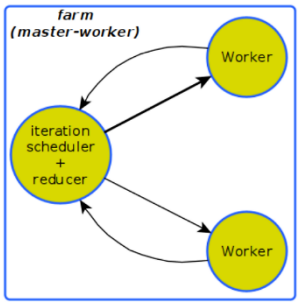
```
double step,x,pi,sum=0.0;
step = 1.0/num_steps;
for(long i=0;i<num_steps; ++i) {
  x = (i+0.5)*step;
  sum += 4.0/(1.0+x*x);
}
pi= step*sum
```

```
double step = 1.0/num_steps, sum=0.0;
ff::parallel_reduce( // static function
    sum, 0.0, 0, num_steps, 1, 0,
    [&](const long i, double& sum) {
          double x= (i+05)*step;
          sum += 4.0/(1.0+x*x);
    },
    [](double& s, const double& d) { s+=d;});
pi= step*sum;
```

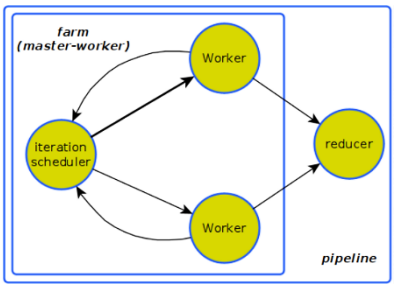
To the static **parallel_reduce** function we must provide two lambda functions. The first is the **map function**, the second is the **reduction function**. We also must declare which is the **reduction variable** (sum), which is the **neutral value** for the operator (0.0), the **range of iterations** (from 0 to num_steps), the **increment** between one step to another (1, *i.e. i*++), and the **chunks size** (how many iteration a single work have to compute at each assignment from the emitter, 0, *see next page*). In this example the **number of workers** is not provided, and it will be by default equal to the number of cores.

Class objects (ParallelFor/ParallelForReduce/ParallelForPipeReduce) and some static functions (e.g., parallel_reduce_idx.

Under the hood the BB used to implement the ParallelFor is a farm with a special emitter that schedule the iterations to the workers. Every time a worker has a partial sum computed sent it to the emitter. At the end the emitter executes the reduce lambda function.

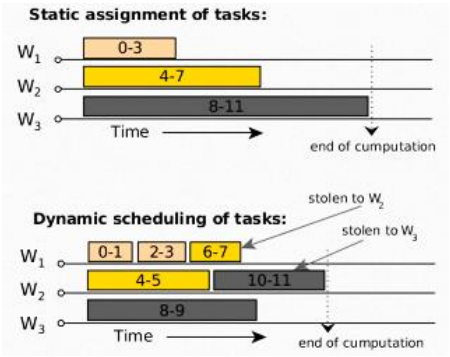



There is also another version where the reduce function is computed not in the emitter but in another thread. This allow to have a pipeline parallelism between the map and the reduction.




**Iteration scheduling in the ParallelFor**

Three options for the chunksize:




- **chunksize = 0**: static scheduling. Each worker gets a contiguous chunk of iterations, i.e. ~(#iteration / #workers)
- **chunksize > 0**: dynamic scheduling with task granularity equal to chunksize.
- **chunksize < 0**: static scheduling with task granularity equal to chunksize, chunks are assigned to Workers in a round-robin fashion (allow to create smaller chunks than ~(#iteration / #workers) and statically assign them to worker with round-robin). Less overhead than dynamic scheduling and a (sometimes) little bit better than chunksize=0.

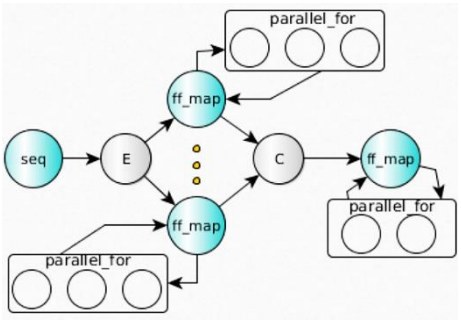
*(Code example, primes_parallelfor.cpp)* parallel_for_**idx** provide an extra parameter to the lambda function so that we know at every single time who is the worker (the thread id) that execute the lambda function. This is useful, for example, to store the result in a private vector for each thread.

*(Code, ff_pi.cpp)* It is also possible to create an object (e.g. ParallelForReduce<T> pf(nw) ). In this case the threads are spawned when we create the object and each call of *pf.parallel_reduce(...)* will use the same threads without spawning and killing them every time (not a good idea if we have only a single call of parallel_reduce).

**Combining Data- and Stream-Parallelism**

If one of the stages of a pipeline is a bottleneck, its service time can be reduced by parallelizing the stage. If the stage computes a Map or a Map+Reduce it can be parallelized by using a ParallelFor pattern.

To simplify this, we can use the FastFlow Map Pattern (**ff_Map**) that is just a single-input single-output node that wraps a ParallelForReduce pattern. *ff_Map<IN_t, OUT_t, reduce-variable-type>*




```
double step,x,pi,sum=0.0;
#include<ff/map.hpp>
using namespace ff;
struct myMap:ff_Map<Task,Task,float> {
   using map=ff_Map<Task,Task,float>;
   Task* svc(Task* in) {
      map::parallel_for(...);
      float sum=0.0;
      map::parallel_reduce(sum, 0.0, ...);
      return in;
   }
};
```

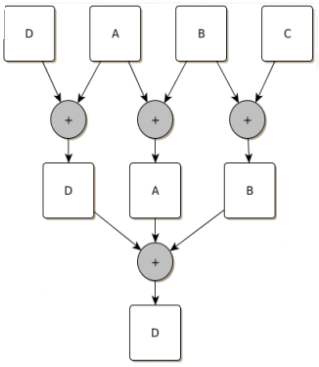
Inside a pipeline or a farm, it is generally better to use the ff_Map than a plain ParallelForReduce because some optimizations are automatically introduced by the Map (mapping of worker threads, scheduler disabled, etc..).

To use it we need to create our node (deriving it from ff_Map) and inside call the parallel_for(...) associated with the map.

**Macro Data-Flow**

The Data-Flow programming model is a general approach to parallelism based upon data dependencies among the program's operations expressed in the data-flow graph. In the data-flow graph representing a given program, nodes are instructions while edges are true data dependencies (read-after-write dependencies).

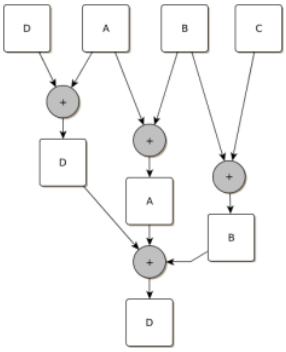
**Macro** Data-Flow (MDF): the same concepts as Data-Flow but instructions are "fat" instructions, i.e., entire function(s) or block(s) of code.




To reduce memory consumption, in-place computation is generally used (i.e. we directly modify the variable value, e.g. A= A+B) but anti-dependencies are possible (i.e., write-after-read).

For example, is not possible to execute in parallel this data-flow because if we compute A = A+B then we don't have anymore the correct value of A to compute D = A+D.

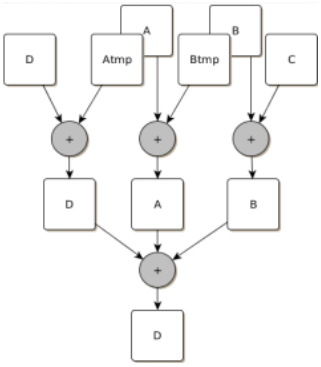
To solve anti-dependencies, either instruction reordering, synchronizations, or data copies are needed (e.g. firstprivate in OpenMP).



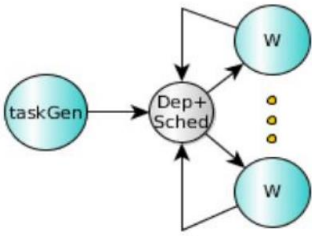

First solution (left): use synchronization and wait to avoid problem with the dependencies. This reduce the concurrency.

Second solution (right): use temporary variables.

Assuming 1 second for each sum we expect 4 second for first solution and 2 second for second solution.




The Macro Data-Flow pattern is implemented in FastFlow using a pipeline of two stages (ff_mdf pattern). The first stage generate the data according to the dependencies of the program (specialized node called task generator). The second stage is a farm with feedback that execute ready instructions (instructions that have all data dependencies satisfied).



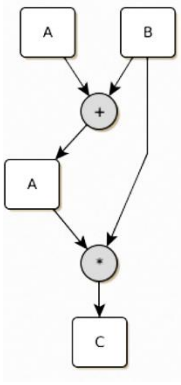

The user has to explicitly declare INPUT and OUTPUT data dependencies for each macro task and provides pointers to input and output data (pointers are used as unique identifiers)

The AddTask method (executed by the taskGen node) creates a macro task.

The run-time takes care of the data dependencies and of the scheduling of the ready task.

*(Code)* We create two tasks using the AddTask. In the first one, to compute the sum function, we say to the run time that A and B are taken in input and we produce A in output. Then we create the vector of dependencies. We pass the vector of dependencies, the function and the pointer to data to the AddTask function.

```
// X = X+Y
void SUM(long *X, long *Y, size_t size);
// Z = X*Y
void MUL(long *X, long *Y, long *Z, size_t size);

{ // A = A+B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&A, OUTPUT};
  std::vector<param_info> P={_1, _2, _3};
  mdf->AddTask(P, SUM, &A, &B, size);
}
{ // C = A*B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&C, OUTPUT};
  std::vector<param_info> P={_1, _2, _3};
  mdf->AddTask(P, MUL, &A, &B, &C, size);
}
```



*(Slide 20, strassen_mdf.cpp)* Example of Strassen's Algorithm for matrix multiplication

*(Slide 21, ff_chol_mdf.cpp)* Cholesky factorization example. Very complex dependencies. *[Tricky code]*

**Divide & Conquer** *ff_DC pattern*

In the Divide & Conquer pattern we split data until we reach the base case and when we reach the base case, we execute the sequential base function.

We need a function that split the data, a function that combine the data, a function that checks if we are in the base case and a function that give the result for the base case.

The D&C pattern is implemented in FastFlow using a master-worker skeleton.

Each worker executes the DC algorithm in parallel.

If we are not in the base case we split the problem (with the divide function givern by the user) in smaller problems. For each smaller problem we call DC is a recursive way. When we reach the base case we compute the sequential function provided by the user. When we return form the base case we combine the result togheters.

Data dependencies are managed by the farm's Emitter. The Emitter takes care of the scheduling of the macro instructions.

```
// functions aliases
using divide_f_t=std::function<void(const ProblemType&,
                                    std::vector<ProblemType>&)>;
using combine_f_t=std::function<void(std::vector<ResultType>&,
                                     ResultType&)>;
using base_f_t=std::function<void(const ProblemType&, ResultType&)>
using cond_f_t=std::function<bool(const ProblemType&)>;
// D&C pattern constructor
template <typename ProblemType, typename ResultType>
ff_DC(const divide_f_t& divide, const combine_f_t& combine,
      const base_f_t& base, const cond_f_t& cond,
      const ProblemType& p, ResultType& res, int par_degree)


void DC(const ProblemType &p,ResultType &ret) {
 if(!cond(op)) { //not the base case
   //divide
   std::vector<ProblemType> ps;
   divide(p,ps);
   std::vector<ResultType> res(ps.size());
   //conquer, recursive phase
   for(size_t i=0;i<ps.size();i++)
     DC(ps[i],res[i]);
   combine(res,ret);    //combine results
   return;
 }
 seq(p,ret); //base case
}
```

*(Code example, mergesort_dac_ptr.cpp)* We use a cutoff value to stop recurring before having an array of size 2 (otherwise we have a too deep tree), huge difference in term of performance.

**(Lect 31) Resource Management System** *(command not asked at exam)*

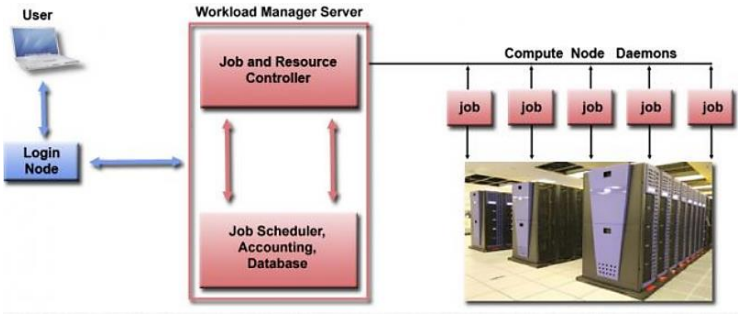
Managing the resources of a supercomputer with thousands of nodes is **not** trivial. **Resource** (or Workload) **management systems** are required. The two most largely used Workload Managers are **SLURM** and PBS.

The nodes of a cluster are divided into login (entry) nodes (used as login node, where do you login and then ask for resources) and computational nodes (usually, there is more than one login nodes).

Workload managers are used to manage the Cluster computing resources shared between users. They mainly perform the following functions:

- **Resource Allocation**: assign nodes to users
- **Manage the schedule**: decide when a given job starts on the assigned nodes and relinquish the resources at the end of program execution.
- **Distribute** the total workload by balancing the work over machine resources.
- **Monitoring and Accounting**

Usually, the most important element is the scheduler because we have many users and how to assign resources is a critical problem.




**SLURM (Simple Linux Utility for Resource Management)**

Computational nodes are divided into **partitions**. Partitions can be **configured in different ways**: limiting the number of hours and the number of nodes that can be used per jobs.

The command *sinfo* can be used to know which partitions are available on the cluster.

**SLURM policy**

SLURM assigns the available nodes of a partition until they are exhausted to the submissions with the **highest priority**.

SLURM decides the priority of jobs that are scheduled through configurable policies, two choices are:

- **backfill algorithm**: backfill scheduling will start lower-priority jobs if doing so does not delay the expected start time of any higher-priority jobs.
- **priority queue algorithm**: it schedules jobs in strict priority order within each partition/queue.

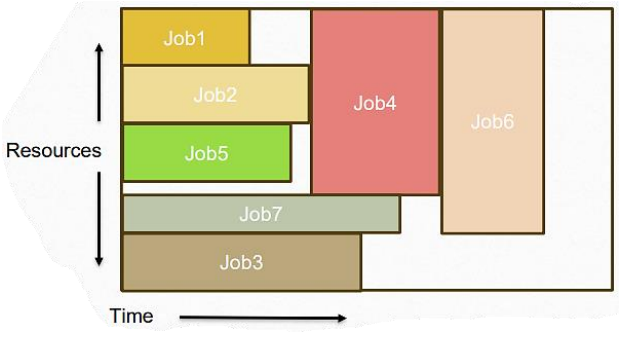
**Backfill algorithm**

The default algorithm attempts to schedule low-priority jobs if they do not prevent higher-priority jobs from starting at the scheduled time.

**Disadvantage**: depends on the time parameter set by the users when submitting a job, who often do not know how long their work will take and **overestimate** it, causing slowdowns.

Example:

- Jobs 1,2 and 3 start immediately.
- Job 4 cannot start. It waits for Job 2 termination.
- Job 5 can start immediately if it finishes no later than Job 2 (**not to delay** Job 4)
- Job 6 waits for the completion of Job 4
- Job 7 can start immediately on the available resources provided it finishes before the start of Job 6




**Priority queue algorithm**

It uses a global queue, ordered according to a certain priority assigned to each job, with all jobs that have been submitted. SLURM decides how to allocate resources to the job at the top.

**Disadvantage**: It can cause SLURM blocking when there are not enough resources to execute the job at the head of the queue.

The spmcluster has a SLURM configuration with one single login node and 8 compute nodes. One SLURM queue (i.e. only one partition). Scheduler uses backfill algorithm. Only one job per node with a time limit of 3 hours per job. Users' homes exported on all slave nodes via NFS (same mounted disk partition in all the nodes).

It is not possible to obtain ssh login to internal nodes. An interactive shell on a node can be started with: *srun –n 1 --time=01:00:00 --pty bash –i* or alternatively *salloc -w node07 --time=01:00:00* followed by *ssh node07*

**Main SLURM commands**

**srun**: executes a parallel job *srun [options] [executable] [argument]* (Slide 10) Possible options

**sbatch:** submits a batch script to SLURM allocating resources. Used to execute a job file in a non-interactive way *sbatch [options] [scripts]* (Example slide 13)

**salloc**: allocates a set of nodes/resources (without run) *salloc [options] [command] [arguments]*

**scancel**: forces the termination of a job subjugated by SLURM *scancel job-id*

**scontrol**: provides information on the system (**scontrol show nodes** provides information on nodes; **scontrol show jobs** provides information on jobs)

**Working with MPI in a SLURM cluster**

It is possible to run MPI application directly using srun.

*salloc --nodes N --time 00:01:00* allocating resources and then using
*mpirun mpirun -n N myprog*

*srun --mpi=pmix --nodes N myprog* using PMIx plugin (Process Management Interface for Exascale)

**MPI (Message Passing Interface)**

A standard for the message passing programming model.

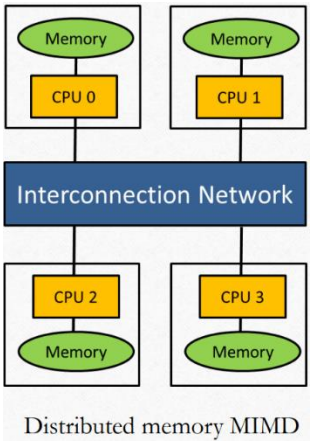
**Message Passing Logical View**

P processes, each with its exclusive address space.

All data must be explicitly partitioned and distributed.

All interactions among processes are **two-sided**:

- Process that has the data **sends** it.
- Process that wants the data **receives** it.



Distributed memory MIMD

**Strengths**:

- Relatively simple performance model.
- Offers high-performance by co-locating data with computation.
- General model (portability) that can be used in all systems.

**Weakness**: the two-sided model can be complex to program (e.g. can bring to deadlock with cyclic communications)

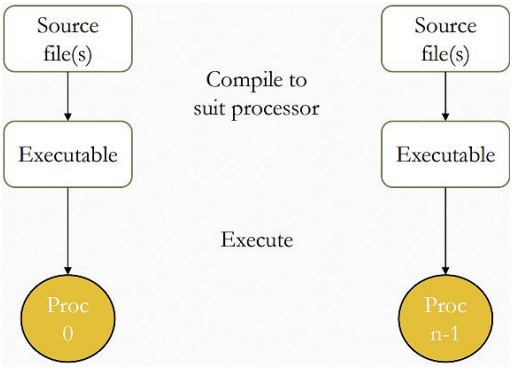
### User-level Message-Passing Programming

Two primary mechanisms are needed:

- A method of **creating separate processes** for execution on different computers
  - **MPMD**: Multiple-Program Multiple-Data
  - **SPMD**: Single-Program Multiple-Data (the one commonly used by MPI)
- A method of **sending** and **receiving messages** (both synchronously and asynchronously)

### MPMD model

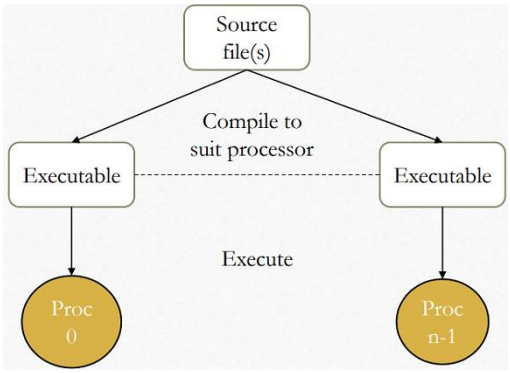
Separate programs for each executable. Processes cooperate typically through a connection-oriented library (e.g., POSIX socket API).




### SPMD model

Different logical flows merged into one program. Control statements select different parts for each processor to execute. All executables started together.

SPMD is the **default** MPI model (but MPI also supports the MPMD model)




### Message Passing in MPI

MPI is a **message-passing interface specification**. It is a library interface and not a language. It has multiple implementations (OpenMPI, Intel MPI, MPICH, MVAPICH, ...)

MPI goals are portability, easy-of-use, and efficiency. MPI is supported by (binded with) C, Fortran, Python, Rust, ... C++ is not supported but we can use the C++ compiler with the C binding.

No shared variables. **Communication** and **Synchronization** operations are MPI library functions.

Two classes of **communication**:

- **Point-to-Point**
- **Collectives**: optimized implementation of group communications (e.g. broadcast)

Two types of **Synchronization**:

- **Explicit barrier**
- **Implicit synchronization** bound to communication collective execution.

**SPMD model in MPI**

The same program is executed by all P processes. Each process chooses a different execution path depending on its ID (called rank in MPI jargon), the ID is unique for each process in the range 0…P-1.

```
...
MPI_Init(...);                  // no MPI call before this point
myFunction1();                  // executed by all processes
switch(myID) {
  case 0:  foo1(); break;       // executed by process 0 only
  case 1:  foo2(); break;       // executed by process 1 only
  default: foo3(); break;       // executed by all other processes
}
myFunction2();                  // executed by all processes
MPI_Finalize();                 // no MPI call after this point
...
```

**Starting and Terminating MPI program**

- **MPI_Init(int* argc, char** argv):** initialize MPI environment. Not thread safe (MPI_Init_Thread is thread safe). No MPI calls before this call. Each instance of the spawned have the same argc and argv in which are removed the extra argument added by srun (stripping off MPI arguments).
- **MPI_Finalize( )**: terminate MPI. No MPI calls after this call.

**Communicators**

Communicator (MPI_Comm) define the scope of communications, i.e., the **communication domain**.

A communicator is a group of processes and a context. It is supplied as an argument to all MPI message transfer routines.

Processes can belong to multiple communication domains (i.e., domain can overlap)

Initially, all processes enrolled in MPI_COMM_WORLD. Its group contains all application processes. Other communications can be created for grouping processes.

- **MPI_Comm_size(MPI_Comm comm, int* size)**: determine the number of processes in the *comm* communicator (result of the call is inside *size*).
- **int MPI_Comm_rank(MPI_Comm comm, int* rank)**: determine the ID of the calling process in the group (called rank).

**(Lect 32)**

(Slide 15/16) Example of MPI program with compilation and execution. Another option not showed in the slide is to use an hostfile with the list of the node to use (*man mpirun* to see the documentation). The important point is that we can have a fine-grained control of the resources if needed.

**Sending and Receiving Messages**

- **int MPI_Send(void* buf, int count, MPI_Datatype dt, int dest, int tag, MPI_Comm comm)**
- **int MPI_Recv (void* buf, int count, MPI_Datatype dt, int source, int tag, MPI_Comm comm, MPI_Status* status)**

**Buffer** is the data we want to send or where we want to put the data received. Datatype is the type of data we are sending or receiving (mpi_int, mpi_char, mpi_long, … full list at slide 19).

**Source** and **destination** are the ID of the processes (rank) in the communicator comm. Receiver source wildcard is MPI_ANY_SOURCE (any process in **comm** can be a source).

The message **tag** is an integer value, 0 ≤ tag < MPI_TAG_UB. It is an additional information attached to the messages (like giving a color to the messages, is a way to personalize the message). It allows to better specify the destination. Receiver tag wildcard is MPI_ANY_TAG. If the receiver gets a message

with a different tag than that expected by the MPI_Recv() call, the message is kept on hold and will be matched by a future MPI_Recv() with a matching tag.

Is not possible to have partial reads (or we receive the full message or none). But it is possible to read less than the buffer length specified if the message is shorter. If instead the message size is bigger than the buffer length, we will get an MPI_ERR_TRUNCATE error.

**MPI_Status** stores information about an MPI_Recv operation. It contains the source (MPI_SOURCE), tag (MPI_TAG), and error (MPI_ERROR) of the communication. Therefore, a process can check the actual source and tag of a message received with MPI_ANY_SOURCE and/or MPI_ANY_TAG. It is possible to use MPI_STATUS_IGNORE if no information is required.

**Count** is the number of data item contained in the buffer. If we are sending data of type mpi_int and we want to send just one single integer, the count is 1 (not the number of byte, the number of elements).

**int MPI_Get_count(MPI_Status* status, MPI_Datatype dt, int* count)**: returns the count of data items received given the status variable.

(Slide 20) Example of a single program sending and receiving strings.

### Communication Taxonomy

Two kinds of communication: **Symmetric** (point-to-point or 1:1) or **Asymmetric** (collective).

Communication can also be divided in: **Synchronous** or **Asynchronous** (that can be **blocking** is the capacity of the communication channel is limited or **non-blocking** if the channel is unbounded).

### Symmetric Synchronous communication

Synchronous send (**ssend**): waits until the complete message can be accepted by the receiving process before sending the message. Return when the message transfer is completed.

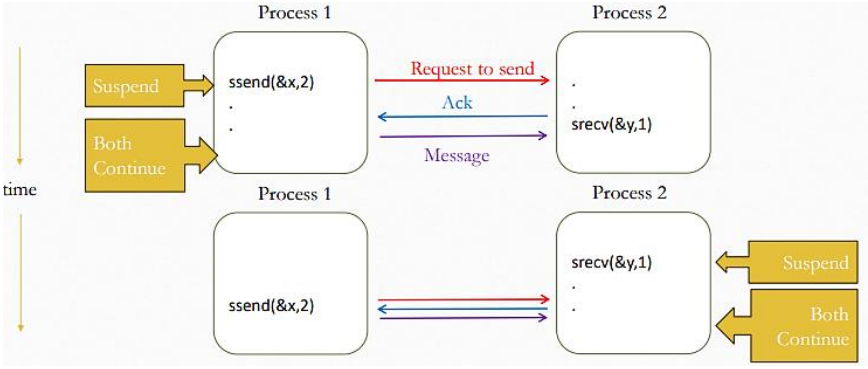
Synchronous receive (**srecv**): waits until the message arrives. Return when the transfer is completed.

In MPI, ssend implemented by **MPI_Ssend**, srecv implemented by MPI_Recv.

Synchronous calls perform two actions: transfer data and synchronize processes.

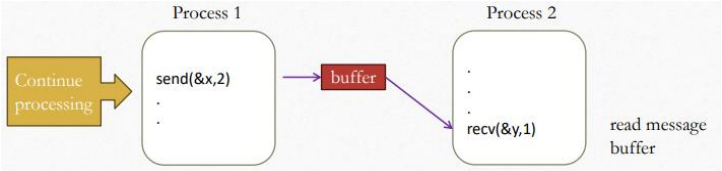
Usually, the synchronous protocol is implemented using a handshake:

The important thing is that we are moving data and also synchronize.




### Symmetric Asynchronous communication

Communication routines that do not wait for actions to complete before returning. Usually require local storage for message buffering.
Clearly recv still blocking to wait data.



**MPI definitions of Blocking and Non-blocking**

**Blocking**: a blocking call returns after its local actions are complete (i.e. the buffer has been copied in the internal buffer of MPI or transferred so we can safely modify the buffer after the call return), though the message transfer may not have been completed. With limited buffer capacity, a blocking send turns out to behave as a synchronous routine.

**Non-blocking**: a non-blocking call returns immediately. It assumes that data storage used for transfer is not modified by subsequent calls before being used for transfer. It is left to the programmer to ensure this.

int MPI_Send(...) is a blocking send. After MPI_Send returns, the buffer can be reused, but the message may not have been received by the destination process yet.

int MPI_Recv(...) is blocking, it waits until a matching (both source and tag) message is entirely received. No partial read of the message is possible. If the sender sends K items, a matching receive will return when all K items have been received. After return is safe to access the buffer.

**Blocking communication and deadlock**

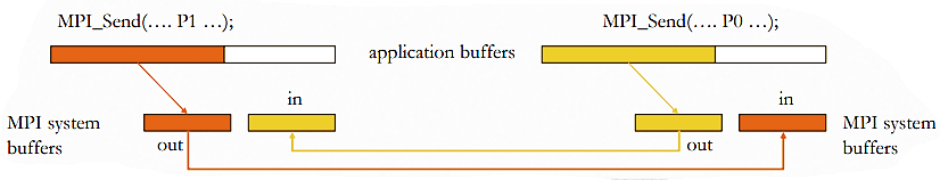
```
if (myrank == 0) {
   MPI_Send(buf, count, MPI_INT, 1, 100, MPI_COMM_WORLD);
   MPI_Send(buf, count, MPI_INT, 1, 200, MPI_COMM_WORLD);
} else if (myrank == 1) {
   MPI_Recv(buf, count, MPI_INT, 0, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
   MPI_Send(buf, count, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

The MPI_Send may block until the message is received by the destination process. If it blocks, deadlock!

Another case: sending data to the right-hand side neighbor on a ring. The same problem as Dijkstra's "Dining Philosopher problem". If MPI_Send is blocking, deadlock! This can be solved by breaking the circular wait (odd ID send and then receive, even ID receive and then send)

Why should MPI_Send get block? Suppose two processes want to send one big message each (0 to 1 and 1 to 0).

The system buffers are full and the two send are blocked because the local operation is not completed (it is not safe to reuse the application buffer).




We have now a deadlock. Use blocking send can be dangerous. One solution is to use non-blocking.

**Non-blocking send/receive**

**int MPI_Isend(void* buf, int count, MPI_Datatype dt, int dest, int tag, MPI_Comm comm, MPI_Request* req)**

**int MPI_Irecv (void* buf, int count, MPI_Datatype dt, int source, int tag, MPI_Comm comm, MPI_Request* req)**

Processing continues immediately without waiting for the completion of the operation. A communication request handle (req) is returned for handling pending message status.

It is okay to pair MPI_Isend() and MPI_Recv() and vice-versa.

### Waiting and checking non-blocking calls

**int MPI_Wait(MPI_Request* req, MPI_Status* status)**: blocks until a non-blocking send or receive operation has completed (there are also other methods to wait multiple non-blocking operation). The status can be used to check possible error in the communication or to look for other information.

**int MPI_Test(MPI_Request* req, int* flag, MPI_Status* status)**: check the status of a non-blocking send or receive. The flag parameter is set to 1 if the operation has completed, 0 otherwise.

### Message exchange

**int MPI_Sendrecv(void* sndbuf, int sndcount, MPI_Datatype snddt, int dest, int sndtag, void* rcvbuf, int rcvcount, MPI_Datatype rcvdt, int source, int rcvtag, MPI_Comm comm, MPI_Status* status)**

Exchanges messages in a single call (both send and receive), useful to avoid deadlock (MPI handles deadlock issues).

Besides avoiding deadlocks, non-blocking communication is usually employed to **overlap computation and communication** (performances reasons).

(Slide 34-36) Codes examples.

**(Lect 33)**

### Collective Communication in MPI

Collective Communication are not necessary, they can be simulated with point-to-point communications, but MPI implements different algorithms that are very efficient for the specific collective we are using (point-to-point communication can be suboptimal in some cases).

All processes in a communicator must call the same collective operation.

### Barrier

**int MPI_Barrier(MPI_Comm comm)**: wait until all processes in the communicator reach the barrier.

**int MPI_Ibarrier(MPI_Request* req)**: non-blocking version.

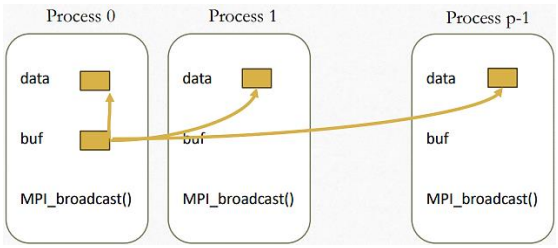
All collectives have a non-blocking version (that is not reported for the next operations)

(Slide 4) Three possible implementations of a barrier. Centralized (a master receives and send messages to all the slaves), tree-based (process interact in a tree fashion, more scalable than centralized), butterfly-based (could be better for some networks). It's the job of MPI decide which is the best implementation for our setup.

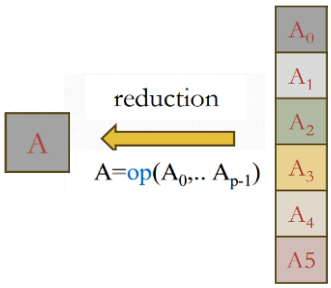
### Broadcast

**int MPI_Bcast(void* buf, int count, MPI_Datatype dt, int source, MPI_Comm comm)**: sends the same message to all processes in the communicator (one-to-all).

Each participant in a one-to-all broadcast calls the broadcast primitive (even though all but the root are just receivers).



### Reduction

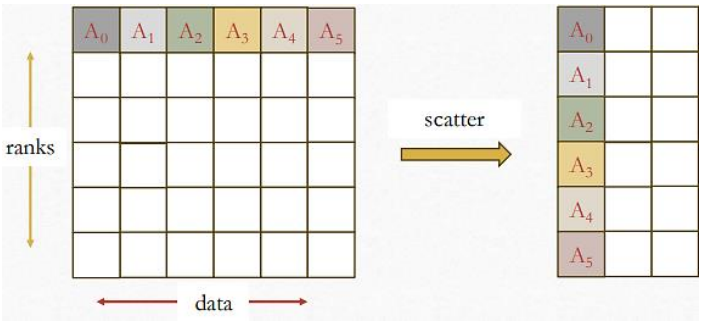



**int MPI_Reduce(void* sndbuf, void* rcvbuf, int count, MPI_Datatype dt, MPI_Op op, int target , MPI_Comm comm)**: performs a reduction and place the result in one target process (a transferring of data + an operation)

(Slide 8) MPI_Op predefined reduction operations.

(Slide 9) Example of MAXLOC and MINLOC reduction operations. Note: it's a good idea to compile in one of the target machines so that the code is optimized for that machine and not for the master that is not used to run the code.

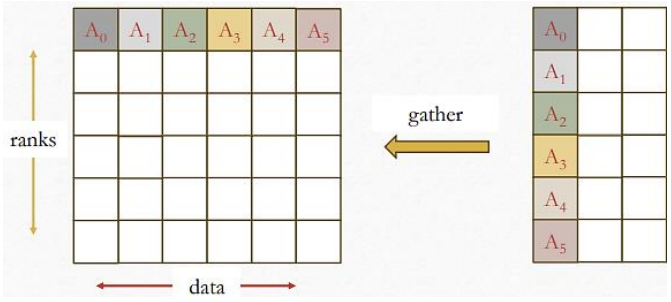
### Scatter




**int MPI_Scatter(void* sndbuf, int sndcount, MPI_Datatype snddt, void* rcvbuf, int rcvcount, MPI_Datatype rcvdt, int source , MPI_Comm comm)**: sends p-1 data blocks from the source process to all other processes in the communicator (one-to-all). Is a way to distribute data between processes of the group.

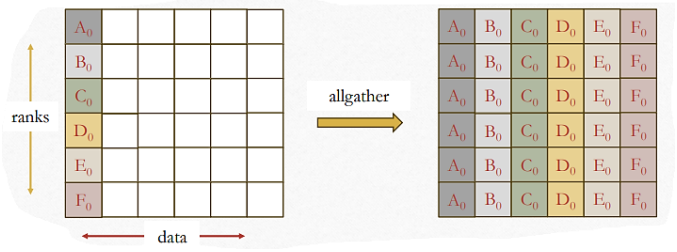
Usually, scatter is used when the number of the elements is divisible for the number of processes in the group, because if there are some remaining after the equal division of the data in part they will not be assigned.

### Gather




**int MPI_Gather(void* sndbuf, int sndcount, MPI_Datatype snddt, void* rcvbuf, int rcvcount, MPI_Datatype rcvdt, int target , MPI_Comm comm)**: the target receives p-1 data blocks from all processes in the communicator (all-to-one). The opposite of scatter.

### Allgather

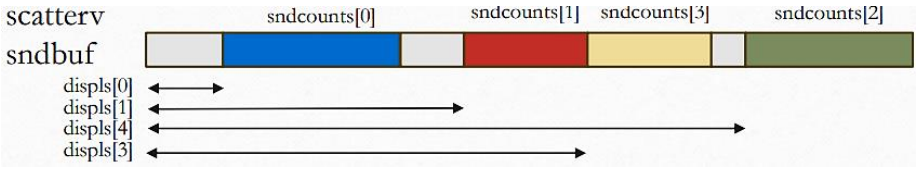



**int MPI_Allgather(void* sndbuf, int sndcount, MPI_Datatype snddt, void* rcvbuf, int rcvcount, MPI_Datatype rcvdt, int target , MPI_Comm comm)**: the target gathers data from all processes and distributes to all. Logically equivalent to MPI_Gather + MPI_Bcast.

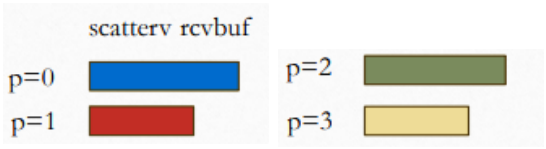
(Slide 14) Vector sum example using Scatter and Gather.

(Slide 15) Alternative version using Scatterv and Gatherv. The v versions (available for a lot of collectives) allow to completely define what to sent and to who (we are not obliged to send partition of equal size).

This is the most general way to send (or collect) data. In this case we have a vector **sndcount[]** specifying the number of elements

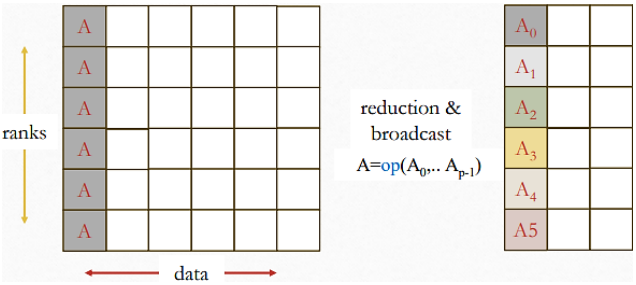


to send to each processor (**rcvcounts[]** to indicate the number of elements that are received from each process)
and a vector **displs[]** that specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (or the displacement relative to recvbuf at which to place the incoming data from process i).



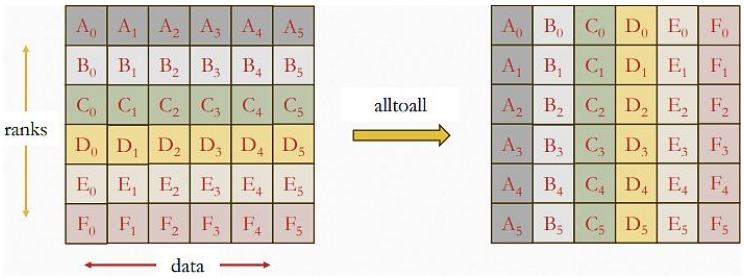

**Allreduce**

**int MPI_Allreduce(void* sndbuf, void* rcvbuf, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm)**: logically equivalent to MPI_Reduce + MPI_Bcast (the result is sent to all the processes)




**Alltoall**

**int MPI_Alltoall(void* sndbuf, int sndcount, MPI_Datatype snddt, void* rcvbuf, int rcvcount, MPI_Datatype rcvdt, MPI_Comm comm)**: each process performs a scatter operation (analogous to a matrix transpose) (all-to-all)




**Parallel prefix operations**

**int MPI_Scan(void* sndbuf, void* rcvbuf, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm)**: inclusive scan operation [process $i$ compute $result = op(v_0, \dots, v_i)$]

**int MPI_Exscan(void* sndbuf, void* rcvbuf, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm)**: exclusive scan operation [process $i$ compute $result = op(v_0, \dots, v_{i-1})$]

| Input vector | -2 | 4 | 2 | -1 | 0 | 1 | -3 | 2 | 0 | 4 | 1 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op = sum | | | | | | | | | | | | |
| scan | -2 | 2 | 4 | 3 | 3 | 4 | 1 | 3 | 3 | 7 | 8 | 13 |
| exscan | 0 | -2 | 2 | 4 | 3 | 3 | 4 | 1 | 3 | 3 | 7 | 8 |

(Slide 9) All the processes call the Scan, it is a collective operation.
On the right we have the scan using count=3 (i.e. we have 3 elements in the input buffer for each process)

**(Lect 34)** *(It's not important remembering the syntax of the command but just the semantics)*

**Groups**

A **group** is an ordered set of processes. Each process in a group is associated with a **unique rank** (from 0 to *groupsize – 1* ).

A group is typically associated with a communicator object (group of a communicator can be obtained by calling: *int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)*

MPI provides operations to construct new process groups based on existing groups (MPI_Group_union(), MPI_Group_intersection(), MPI_Group_difference(), MPI_Group_incl(), MPI_Group_excl(), the last two include or exclude in the group single elements)

(Slide 3) Code snippet to create two group (even and odd) splitting p processes.

**Communicators**

A **communicator** encompasses a group of processes that can communicate. A communicator binds a process group and a context (**intracommunicator**: used for communication within a group, **intercommunicator**: used for cross groups communication).

The purpose of this is to have a more regular way to communicate. It is possible to build user-defined virtual topologies (e.g. hypercube, mesh, …, already cooked method to create a regular topology).

MPI provides more than 40 functions/routines related to groups, communicators and virtual topologies. Some of them related to communicator management are:

- int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm* new_comm): create a new communicator starting from an old communicator and a group of processes.
- int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int* res)
- int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* new_comm): duplicate the communicator. Useful when we implement a library using MPI (e.g. if the library uses MPI_COMM_WORLD there is a conflict with the user part that also uses it)
- int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* new_comm): create new subcommunicators splitting an existing communicator with some parameters. The color define which processes will be part of which subcommunicator while the key defines the rank that the processes will have in the subcommunicator.

**Splitting Communicators**

It is a **collective** operation (all the processes in the communicator must call the function, otherwise it doesn't work). It partitions comm into disjoint subgroups, one for each value of color. Within each group, the MPI processes are ranked in the order defined by the value of the key, with ties broken according to their rank in the parent group. If all keys are the same, then all processes will have the relative rank order as they have in the parent group.

(Slide 5) Code example to split processes in different rows in different communicators. Four colors (color = rank/4, the rank is the one of the process that is calling the method, *it is collective so all processes will execute this code*), the process from 0 to 3 will have the first color, the process from 4 to 7 the second color … The rank provided is the one that the process has in the MPI_COMM_WORLD, MPI gives rank in an ascending order (4 < 5 < 6 < 7 (in comm_world) → 0, 1, 2, 3, but this can be changed providing different rank parameter)

**Virtual Topologies (hints)**

**Cartesian topologies**: for regular problems, MPI provides a convenient multidimensional mesh organization (2D, 3D, … nD regular topologies) *int MPI_Cart_create(…)*

**Graph Topologies**: allow to create a graph topologies *int MPI_Graph_create(…)*

**MPI Derived Data Types**

MPI allows users to build new, user-defined datatypes on the basis of primitive MPI datatypes.
A **general datatype** is an opaque object that specifies two things:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements

Order of items in the datatype may not coincide with their order in memory. An item may appear more than once.

Derived datatypes are useful in situations where:

- the data to send is non-contiguous data (e.g., sparse vector elements)
- the data is contiguous in memory but made of mixed types (e.g., a struct containing an integer and an array of floats)

In both cases we don't want to copy the elements in a contiguous set of bytes because is too costly. They enable more performance by reducing the number of memory copies (also improve readability).

**Building a datatype**

The primary steps are:

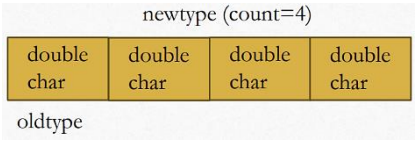
1. Construct the datatype using a constructor MPI_Type_*xxx*(...). The new datatype has type MPI_Datatype (xxx must be replaced with one of the options)
2. Allocate the datatype using MPI_Type_commit(...)
3. Once the datatype is not used anymore, free it MPI_Type_free(...)

Construction and allocation are mandatory, the release of the datatype is optional but recommended.

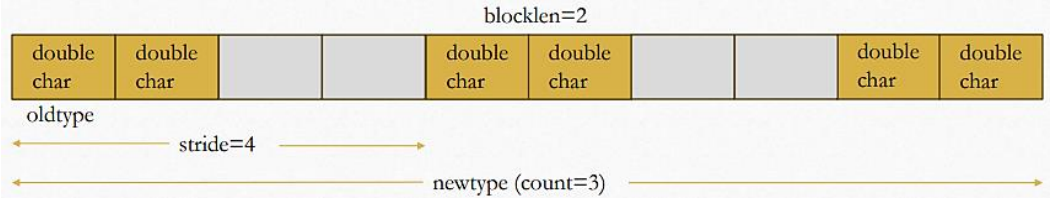
**Datatype constructors**

The most used datatypes constructors are:

- **int MPI_Type_contiguous**(int count, MPI_Datatype oldtype, MPI_Datatype* newtype): create a continuous object. The ***newtype*** is the datatype obtained by concatenating ***count*** copies of oldtype. E.g.



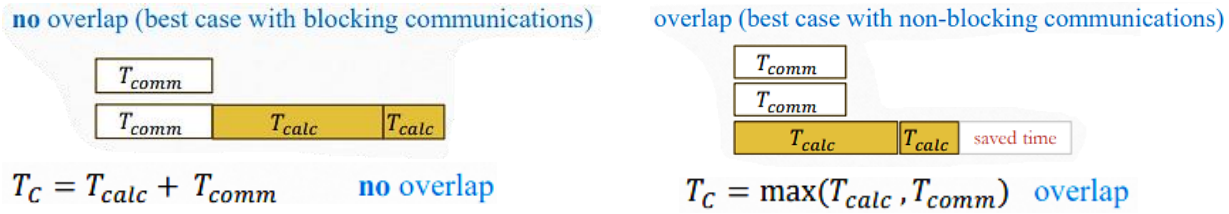

- **int MPI_Type_vector**(int count, int blocklen, int stride, MPI_Datatype oldtype, MPI_Datatype* newtype): vector that contains pointers to non-contiguous memory.
***count*** is the number of blocks, ***blocklen*** is the number of elements in each block, ***stride*** is the number of elements between the start of each block (to use bytes instead of the number of elements, the function *MPI_Type_hvector(...)* can be used)




- **MPI_Type_create_struct**(int nblocks, int blocklen, MPI_Aint displacements, MPI_Datatype oldtype, MPI_Datatype* newtype): create a datatype from a set of datatypes. Each block is a collection of data of the same type.
***nblocks*** is the number of blocks, ***blocklen*** is an array of integers setting the size of each block, ***displacements*** is an array setting the relative offset for each block (expressed in bytes). Set displacements manually is unsafe and not portable (different compilers use different padding

to align the data in the struct, e.g. a char and an integer), better use MPI_Get_address(...)

*(Jacobi iterations example)* Stencil code, we repeat iterations until we converge to the expected value. An idea to parallelize is to split the matrix in blocks assigned to different processes.
**Problem**: processes must communicate because the element on the border needs the elements of the border of another process.
**Possible solution**: use blocking communication and start computing only after all the information have been exchanged.
**Another solution**: overlap computation and communication using non-blocking communication and meanwhile compute the internal values for which we have all the necessary data.




We could use threads to communicate but the best option in MPI is to use asynchronous communications.

**(Lect 35)**

*(SUMMA example)* Elegant way of implementing matrix multiplication (A x B = C) combining MPI datatype and communicators. The matrices A, B, and C are distributed among processes without redundancy (send to each process just the data that is strictly needed for the computation). Not all data to compute the partial product are stored in the local memory of processes, Rows and Columns group communicators are used to obtain all data needed locally (through broadcast phases, slide 17). We don't do point-to-point communications, instead we define proper datatype and proper communicator in order to broadcast the data using this pattern.

Note: --bynode to spread processes among all the nodes allocated in the cluster. If we care about which process is spawned in which machine (e.g. if we want to put processes that communicate a lot with each other in the same node) we need to use the hostfile or the rankfile.

**Threds and MPI**

MPI describes parallelism between processes with separate address spaces. **Threads** are **not supported by default** by MPI (e.g. OpenMPI needs to be compiled with --enable-mpi-threads).

Threads in MPI are not addressable, it is not possible to send a message to a specific thread of a process.

To init MPI to support threads **MPI_Init_thread** instead of MPI_Init must be used (otherwise 1 thread).

The MPI process can use threads for computation and/or communication. The paradigm MPI + "X" (OpenMP, C++ threads, CUDA) is known as "**hybrid programming**" (mixing message passing and shared memory). From MPI-3 also shared memory within MPI was introduced so X could also be MPI Shared Memory.

**Initialize MPI for threading**

int MPI_Init_thread(int* argc, char*** argv, int required, int* provided)

The argument *required* is used to request the desired level of thread support. It can be:

- **MPI_THREAD_SINGLE**: only one thread calling MPI functions will execute (all other threads of the process must sleep during MPI function calls). MPI_Init is equivalent to MPI_Init_thread(..., MPI_THREAD_SINGLE, ...).
- **MPI_THREAD_FUNNELED**: only the thread that called MPI_Init_thread will make MPI calls (e.g., if OpenMP is used, only the Master thread executes MPI function calls). One thread can be for communication, the others can be used for computation,
- **MPI_THREAD_SERIALIZED**: only one thread will make MPI function calls at one time. In this case, multiple threads may execute MPI function calls with the restriction that calls are serialized.
- **MPI_THREAD_MULTIPLE**: multiple threads may call MPI functions with no restrictions.

The MPI runtime fills the provided *argument* with the actual thread support offered by the implementation. Note that **could be less** than required! (OpenMP support all the level).

When multiple threads make MPI calls concurrently, the outcome will be as if the calls are executed sequentially in some order:

- User **must ensure** that collective operations are correctly ordered among threads (cannot call broadcast in one thread and scatter in another thread on the same communicator).
- User **must ensure** data race free MPI programs.

With MPI_THREAD_MULTIPLE, an implementation must ensure that this example (with a loop in communication) never deadlocks for any ordering of threads. This does not come for free. The implementation must protect certain code and data structures with mutexes, and this introduces **overhead** (so use the lower level of thread support as possible).

| | P0 | P1 |
|---|---|---|
| Thread 0 | MPI_Recv(... 1...); | MPI_Recv(... 0 ...); |
| Thread 1 | MPI_Send(... 1...); | MPI_Send(... 0 ...); |

(Slide 23) Example of codes with MPI+OpenMP. To run the code, we can use ***-x*** to use an environmental variable that all the process that we spawn will have in their local environment (e.g. OMP_NUM_THREADS = 16).
***--bind-to none*** to tell that the process can execute in any core of the nodes, if we don't use this the default is *--bind-to core* (the process will execute only in one core) this is a problem because all the thread of OpenMP will also have to execute on that core, no parallelism!

(Not very important) Implementation of Distributed FastFlow is done using one thread that receive data, one thread that send data and two extra thread that are used as routers. This mean that the receiver and the sender are two thread that communicate concurrently using MPI communication functions, MPI_THREAD_MULTIPLE support is mandatory for the MPI backend in distributed FastFlow.