PARALLEL COMPUTING

Claudio Tessa

January 16, 2026

Contents

1 PRAM (Parallel Random Access Machine)

PRAM is an abstract machine model for designing and analyzing algorithms intended for parallel computers. Is a theoretical framework to understand how multiple processors can work together to solve computational problems.

The PRAM model consists of a set of infinitely many components that function as a single system M':

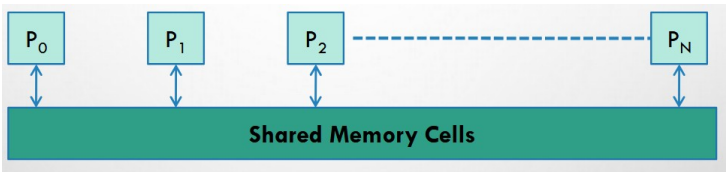
- **Processors** P_i, all of which are identical. Each processor knows it own index identifier i. These processors have an unbounded number of registers and memory cells, and can access all memory cells in unit hime. All communication between processors occurs exclusively via shared memory.
- **Memory structure** made up of three distinct types of memory cells:
  - **Input cells** $X(1), X(2), \ldots$
  - **Output cells** $Y(1), Y(2), \ldots$
  - **Shared memory cells** $A(1), A(2), \ldots$

1.1 Computation steps

The computation is **synchronous**. A single computation consists of five steps carried out in parallel by all active processors. Each processor:

1. **Read input**: read a value from an input cell X;
2. **Read shared memory**: read from a shared memory cell A;
3. **Compute**: perform some internal computation;
4. **Write output**: may write to an output cell Y;
5. **Write to shared memory**: may write to a shared memory cell A

Note that some subset of the processors can remain idle.



1.2 Memory access conflicts

A major consideration in PRAM is how the system handles multiple processors trying to read (**read conflict**) or write (**write conflict**) the same memory cell at the exact same time. We classify PRAM models by their rules regarding these conflicts:

- Exclusive Read (**ER**): processors can simultaneously read only from distinct memory locations;
- Exclusive Write (**EW**): processors can simultaneously write only to distinct memory locations;

- Concurrent Read (**CR**): processors can simultaneously read from any memory location (including the same one);
- Concurrent Write (**CW**): processors can simultaneously write to any memory location (including the same one). What value gets written in the end:
  - **Priority CW**: processors have priorities. The highest priority is allowed to complete write.
  - **Common CW**: all processors are allowed to complete write *if and only if* all the values to be written are equal. Any algorithm for this model has to make sure that this condition is satisfied (otherwise the algorithm is *illegal* and the machine state will be undefined).
  - **Arbitrary/Random CW**: one randomly chosen processor is allowed to complete write.
- A combination of both: **CREW**, **EREW**, ...

PRAM is important for designers of parallel algorithms because it is **simple**, it abstracts any communication or synchronization overhead. It can be used as a **benchmark**, because if a problem as no feasible/efficient solution on PRAM, it won't have one for any parallel machine.

1.3 Computational power

Note that not all PRAM variations possess the same theoretical capabilities.

Definition 1. *We say that model A is* ***computationally stronger*** *that model B, written $A \geq B$, if and only if any algorithm written for B will run unchanged on A in the same parallel time and with the same basic properties.*

$$PRIORITY \geq ARBITRARY \geq COMMON \geq CREW \geq EREW$$

1.4 Performance metrics

To analyze the performance of parallel algorithms, we must define specific metrics:

Symbol	**Definition**
$T^*(n)$	Time to solve problem of input size n on one processor, using best sequential algorithm
$T_p(n)$	Time to solve on p processors
$\mathrm{SU}_p(n) = \frac{T^*(n)}{T_p(n)}$	Speedup on p processors
$E_p(n) = \frac{T_1(n)}{pT_p(n)}$	Efficiency (work on 1 processor / work that could be done on p processors)
$T_\infty(n)$	Shortest run time on any p
$C(n) = P(n) \cdot T(n)$	Cost (processors and time)
$W(n)$	Work = total number of operations

1.5 Scaling lemmas

These lemmas describe how PRAM algorithms scale when hardware resources are limited.

Lemma 1 (Processor scaling). *Assume $P' < P$. Any problem that can be solved by a P-processor PRAM in T steps, can be solved in a P'-processor PRAM in $O\left(\frac{TP}{P'}\right)$ steps.*

Lemma 2 (Memory scaling). *Assume $M' < M$. Any problem that can be solved by a P-processor and M-cell PRAM in T steps, can be solved on a $\max(P, M')$-processor M'-cell PRAM in $O\left(\frac{TM}{M'}\right)$ steps.*

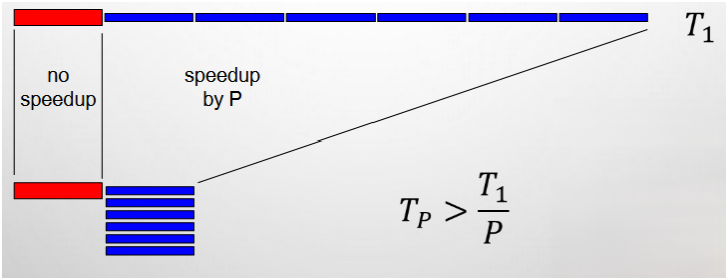
1.6 Amdahl's law

Amdahl's law is a formula used to predict the theoretical maximum speedup for a program using multiple processors.

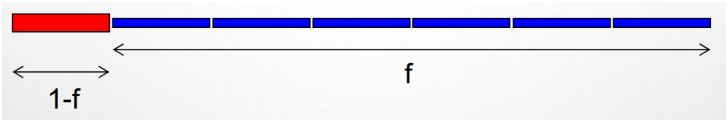
Computations consists of **interleaved segments** of two distinct types:

- **Serial segments** (in red): cannot be parallelized;
- **Parallelizable segments** (in blue): can be divided into independent tasks and processed simultaneously by multiple processors.

The following diagram shows the difference between running the workload on one processor (T_1) versus P processors (T_P).



In the serial version, the parallelizable part is a fixed fraction f of the whole program



We can derive the speedup SU by comparing the time on one processor (T_1) to the time on P processors (T_P):

$$SU(P, f) = \frac{T_1}{T_p} = \frac{1}{(1-f) + \frac{f}{P}}$$

$$\lim_{p \to \infty} SU(P, f) = \frac{1}{1-f}$$

1.7 Gustafson's law

Amdahl's law assumes a fixed problem size. Gustafson, instead, argues that when we access more powerful parallel hardware (more processors), we generally do not use it to

just solve the same small problem faster, but we use the extra power to solve larger and more complex problems in the same amount of time.

The key points are that the time spent on the serial portion of the problem remains fixed, but the serializable portion f is **not fixed**, but it is expanded to fill the available processing power.

$$SU(P) = \frac{T_1}{T_P} = s + P \cdot (1 - s)$$

where s is the serial time and $(1 - s)$ is the parallel time. Note how we have a **linear speedup**.

1.8 Scaling

1.8.1 Strong scaling

Strong scaling refers to the ability of a system to improve the performance of a parallelized program when the number of processors is increased while keeping the total problem size constant.

Amdahl's law is often used to analyze this type of scaling, as it highlights the limits imposed by the serial portion of the program.

Ideally, the time to solve the problem is proportional to $1/P$.

1.8.2 Weak scaling

Weak scaling refers to a system's ability to maintain efficiency when the number of processors is increased while the workload is also proportionally increased.

This type of scaling is useful for applications where the problem grows in proportion to the increase in available resources. Gustafson's law is often used to analyze this type of scaling.

Ideally, T is flat vs P (as P grows together with the problem size, T remains constant).

2 SIMD processing

SIMD (Single Instruction Multiple Data) is a parallel processing technique that increases compute capabilities by performing the exact same operation on multiple ALUs in parallel.

the idea is to amortize the cost and complexity of managing instructions across many ALUs.

The compiler understands if loop iterations are independent, and it maps that logic onto each ALU to make use of SIMD processing capabilities within a core.

However, the fact that SIMD makes it such that all ALUs must receive the exact same command at the same time creates a problem when the code contains an `if` statement. Let's say we have 8 ALUs in a core, and 3 loop iterations will evaluate their `if` to true while 5 to false. In this case, the hardware cannot physcally branch in two directions at once, as it has only one instruction decoder.

Instead of branching, SIMD processors use a techinique called **masking** to execude the code linearly. The processor checks the condition for all 8 elements simultaneously, then creates a **mask** of true/false values (e.g., `[T T F T F F F F]`).

When the processor broadcasts the instructions to *everyone*, the **active ALUs (True)** perform the calculations, while the **inactive ALUs (False)** receive the instruction but discard the output of the calculations. If there is an `else` block, the processor flips the maps and broadcasts the instructions again.

Terminology

Instruction stream coherence ("coherent execution") is a property of a program the same sequence of instructoins applies to many data elements.

This is **necessary** for efficient SIMD processing because SIMD hardware broadcasts one instruction to all ALUs.

It is however **not necessary** for efficient parallelization across different cores, since each core has the capability to fetch/decode a different instruction from their thread instruction stream.

2.1 Implicit SIMD

Implicit SIMD is used on modern GPUs, which means that the programmer writes standard "scalar" code, and then the compiler also generates a standard binary with scalar instructions.

The **hardware** is then responsible for simultaneously executing the same instruction from multiple program instances on different data on SIMD ALUs.

3 Accessing memory

A computer memory is organized as an array of bytes. Each byte is identified by its **address** in memory (its position in this array).

To access the content of the memory we use the **load** instruction:

```
ld R0 mem[R2] # R0 = mem[R2]
```

Terminology

Memory access latency is the amount of time it takes the memory system to provide data to the processor (e.g., 100 clock cycles, 100 nsec, ...).

3.1 Stalls

A processor **stalls** when it cannot run the next instruction stream because of a dependency on a previous instruction that is not yet complete. Accessing memory is a major source of stalls:

```
ld r0 mem[r2]
```

```
ld r1 mem[r3]
add r0, r0, r1 # dependency from previous instructions
```

Modern processors have **caches** which reduce the length of stalls, as wella s the memory access latency.

3.1.1 Prefetching

Another way to reduce stalls is **prefetching**, which "hides" latency. many modern CPUs have some logic for predicting what data will be accessed in the future, so they pre-load this data into caches. To make such predictions, the program's memory access pattern is analyzed dynamically.

With prefetching we can reduce stalls because the data is already available in the cache when needed. Note that prefetching can also reduce performance if the guess is wrong (consumes bandwidth, pollutes caches, ...).

3.1.2 Multithreading

The idea is to exploit multithreading to reduce stalls. If you can't make progress on the current thread, instead of just waiting, work on another thread.

Multithreading is a throughput-oriented optimization. Throughput-oriented systems accept to potentially increase the time it takes to complete the work for *any* single thread if this means increasing the overall system throughput when running multiple threads.

A thread is running until it hits a stall. The core then executes another thread. After the stall is resolved, the core could come back to the previous thread, but instead is executing instructions from another thread (to increase the overall system throughput).

We have 3 types of multithreading:

- **Fine-grain (interleaved) multithreading**: the processor switches threads after every cycle. If one thread stalls, others are executed immediately. Requires complex hardware to switch extremely fast.
- **Coarse-grain multithreading**: the processor only switches when a long stall occurs. This simplifies the hardware but fails to hide shorter stalls.
- **Simultaneous multithreading**: instructions from multiple threads issue to the execution units in the exact same clock cycle, using all the available functional units at once. This is useful in multiple-issue dynamically scheduled processors.

3.2 Latency and bandwidth

Terminology

Memory bandwidth is the rate at which the memory system can provide data to a processor

The instruction throughput is not impacted by memory latency, but only by memory bandwidth. The computation is bandwidth limited: if the processors request data at a rate too high, the memory system cannot keep up.

Overcoming bandwidth limits is often the most important challenge. Performant parallel programs will **organize computation** to fetch data from memory less often, by

- Reusing data previously loaded by the same thread;
- Sharing data across threads;
- Performing additional arithmetic instead of reloading values (the math is "free").

4 Programming models

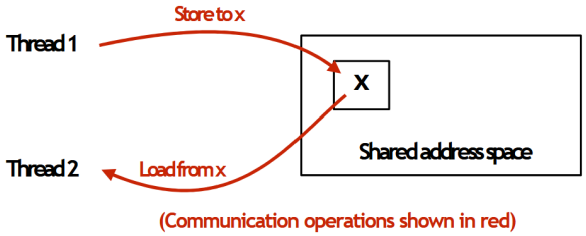
Programming models influence how programmers think when writing programs and influence the design of parallel hardware platforms designed to execute them efficiently.

They differ based on what communication abstraction they present to the programmer.

4.1 Shared address space model

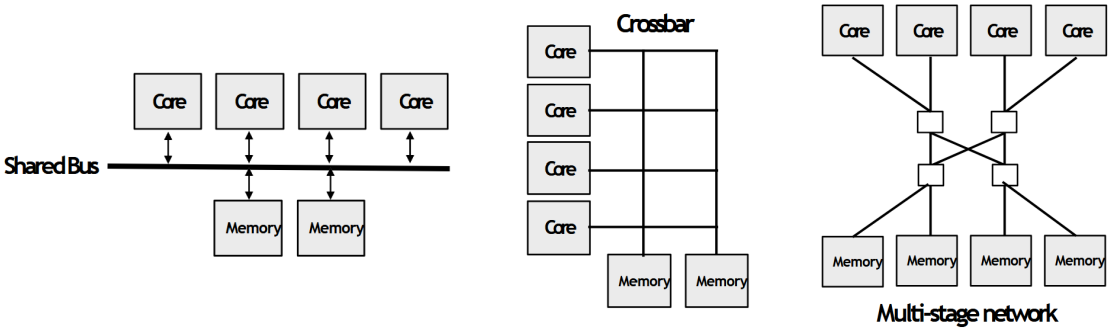
The implementation of the linear memory address space abstraction (the "array") on a modern computer is complex. The instruction "*load the value stored at address X into register R0*" might involve a complex sequence of operations by multiple data caches and access to DRAM.

In the **Shared address space model**, the threads communicate by readingwriting to locations in a shared address space (shared variables)



We coordinate the access to the shared variables with **synchronization**. To do so we use mutual exclusion.

To implement shared address space at a hardware level, the key idea is that any processor can **directly** reference contents of any memory location. Each core, is connected to an **interconnect**, which in turn is connected to the memory. Here are some examples of interconnects:



4.2 Message passing model

In the **message passing model**, threads operate within their own private address space, and communicate between each other by sendin/receiving messages (the *only* way to exchange data):

- **Send**: specifies recipient, buffer to be transmitted, and optional message identifier;
- **Receive**: sender, specifies buffer to store data, and optional message identifier.

The hardware does not need to implement system-wide loads and stores to execute message passing programs. Message passing is a programming model for clusters and supercomputers, as it can connect commodity systems together to form a large parallel machine.

4.3 Data-parallel model

The **data-parallel model** organizes computation as operations on sequences of elements (e.g., perform the same function on all elements of a sequence).

In fact, **sequences** (ordered collection of elements) are the key data type here. The program can only access elements of a sequence through sequence operators: `map`, `reduce`, `scan`, `shift`, etc.

4.3.1 Map

Map is a higher order function (i.e., a function that takes a function as an argument) that operates on sequences.

It applies a side-effect free unary function $f : a \to b$ to all elements of the input sequence and produces an output sequence of the same length.

Since $f : a \to b$ is a side-effect free function, then applying f to all elements of the sequence can be done **in any order** without changing the output of the program. This gives flexibility to parallelize the processing of the elements of the input sequence.

5 GPU architecture

While a CPU is designed for *latency* (doing one thing very quickly), a GPU is designed for **throughput** (doing a lot of things at once). GPUs are very fast processors for performing the same computation (usually shader programs) in parallel on a large collection of data.

A GPU is a multicore chip, composed of *many* simple cores. We have SIMD execution within a single core, as well as multi-threaded execution (multiple threads executed concurrently by a core).

GPUs have so many high-throughput cores because having many SIMD multi-threaded cores provides efficient execution of shader programs.

5.1 CUDA programming language

In 2007, NVIDIA introduced the "Tesla" architecture, the first non-graphic-specific ("compute mode") inteface to GPU hardware.

Let's say a user wants to run a non-graphics program on the GPU's programmable cores:

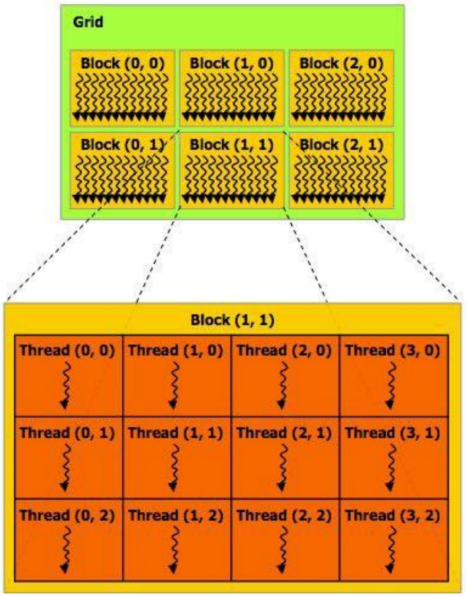
1. **Memory allocation**: the application can allocate buffers in the GPU memory and simply copies data to/from them;
2. **Kernel**: instead of a full pipeline, the application provides a single program inary called a *kernel*;
3. **Launch**: the application tells the GPU to run the kernel in an SPMD fashion (run N instances of this kernel).

Along with this architecture, NVIDIA instroduced the CUDA programming language, a C-like language to express programs that run on GPUs using this new compute-mode hardware interface. This language is intentionally low-level, as its abstractions are designed to closely match the physical capabilities and performance characteristics of the GPU, maintaining a "low abstraction distance" so programmers know exactly what the hardware is doing.

> **Note**
>
> OpenCL is an open standards version of CUDA, which runs not only on NVIDIA GPUs, but on CPUs and GPUs from many vendors. Almost everything we will see for CUDA also holds for OpenCL.

Simply launching a massive number of threads in a flat list would be cahotic and difficult to mmap to real-world problems. To solve this, CUDA instroduces a **hierarchy** of concurrent threads. We have a **two-level hierarchy**:

1. The **grid** (the global scope): the top level structure. When you launch a kernel, you create exactly one grid. This grid represents the entire problem space.
2. The **block** (the local scope): the grid is subdivided into smaller, independent groups called *thread blocks*. each block contains a specific number of threads that can cooperate with each other.



A CUDA feature is that thread IDs don't have to be just a 1D list of numbers, but they can be up to **3-dimensional**. This is very convenient for many problems (e.g., an image is 2D, a physics simulation might be 3D).

5.2 Basic CUDA syntax

The fundamental concept in the CUDA syntax is the strict separation between the two processors involved:

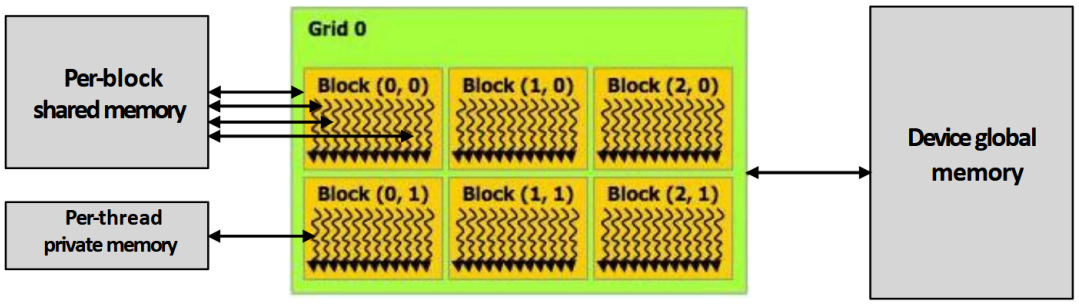
- **The host (CPU)**: runs the standsrd seiral C/C++ application. It is responsible for setting up the problem, managing memory and commanding the GPU. The CPU executes a special function call that triggers a **bulk launch** on the GPU of many CUDA threads (launches a grid of CUDA thread blocks). When this command is executed, the CPU tells the GPU exactly how many blocks to create, and how many threads to put in each block.
- **The device (GPU)**: executes the parallel compute-intensive tasks. In CUDA syntax, a specific keyword before a function definition flags the function as **device code**, telling the compiler that such code is meant for the GPU. The kernel runs in SPMD mode, so every single thread executes the exact same line of code.

There is a clear separation between host and device code. This separation is performed statically by the programmer.

5.2.1 CUDA memory model

We have distinct host and device address spaces. For the device memory, we have three distinct types of address spaces visible to kernels. CUDA organizes memory into a **hierarchy**:

- **Per-thread private memory (local scope)**: readable and writable only by the single thread that owns it;
- **Per-block shared memory (group scope)**: readable and writable by all threads within the same block;
- **Device global memory (global scope)**: readable and writable by all threads in the entire grid.



5.3 Warps

We previously saw that you can define a thread block to have any number of threads. However, the GPU hardware does not actually manage threads individually, not does it necessarily manage them as the full block you defined.

Instead, the hardware groups threads into fixed-size groups called **warps**. On modern NVIDIA GPUs, a warp consists of 32 threads.

Threads in a warp are executed in a SIMD manner, if they share the same instruction. If the 32 CUDA threads do not share the same instruction, performance can suffer due to divergent execution.

Warps are not part of CUDA, but are an important CUDA implementation detail on modern NVIDIA GPUs.

5.4 Running a CUDA program on a GPU

Running a kernel involves the following steps.

1. **Requirement definition**: the kernel is launched with specific resource requirements (e.g., X threads per block). The system calculates how many **execution contexts** (thread slots) and how much shared memory (bytes) are needed for a *single thread block*.
2. **Command**: the host (CPU) sends a command to the CUDA device (GPU) to execute the kernel, including the function to run, the arguments, and the total number of blocks to process.
3. **Blocks mapping**: the GPU work scheduler initially maps block 0 to core 0, then proceeds to map each block to available execution contexts.

Why must CUDA allocate execution contexts for all threads in a block (instead of running them in smaller batches to save space)? CUDA must allocate resources simultaneously to prevent **deadlock** caused by synchronization barriers like `__syncthreads()`. In a block, threads often reach a point where they must wait for *all* other threads in that block to arrive before proceeding.

Definition

`__syncthreads()` is a **barrier**: each thread must wait for all other threads in the block to arrive at this point.

This avoids **race conditions**.

5.5 Memory access performance

Memory bandwidth is a first-order performance factor in a massively parallel processor

DRAM is the hardware technology used to implement the device global memory. Each bit of data is stored in a tiny capacitor made of one transistor. These cells are arranged in a **core array**. Each DRAM core array has about 16M bits.

You cannot just access a single bit instantly. The access process happens in two stages:

1. **Row activation**: the *row decoder* selects an entire row of data from the core array. This row is read into the **sense amps** (which amplify the weak signals from the capacitors) and stored in the **column latches**. This entire row is now *open* or *active*.
2. **Column selection**: the column address selects the specific chunk of data we asked for from the active row (thanks to a multiplexer).

We actually have to access the whole row and then select the column we are interested in. Therefore, reading from a cell in the core array is a very slow process, also since the core array runs at only $1/N$ the speed of the interface.

5.5.1 DRAM Bursting

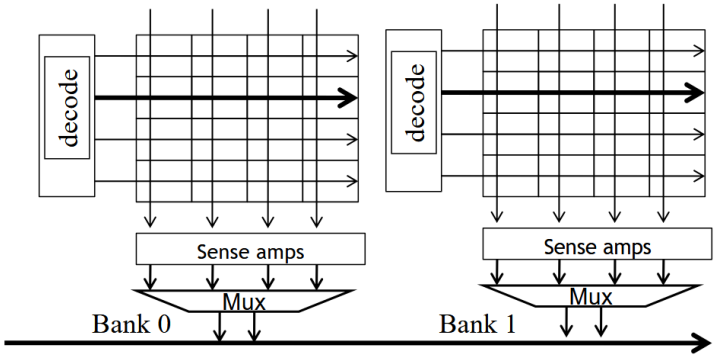
If the memory core only fetched one piece of data at a time, the interface (which is fast) would spend most of its time waiting, doing nothing.

To solve this, DRAM uses **bursting**. Instead of fetching just the single byte you asked for, it fetches a massive chunk of adjacend data all at once.

5.5.2 DRAM Banks

Even with bursting, there is a delay we haven't accounted for yet. Before you can read any data, you must use the *row decoder* to activate a row and move data into the *sense amps*. This process is slow.

To speed this up, modern DRAM chips do not contain just one core array, but they contain many independent arrays called **banks**. Each bank has its own *row decoder* and *sense amps*.



Combining bursting banking allows us to achieve maximum efficiency:

- **Bursting** ensures that when we get data from a bank, we get a large chung at high speed, filling the bus for a short duration;
- **Banking** ensures that as soon as *bank 0*'s burst finishes, **bank 1** is ready to burst immediatly.

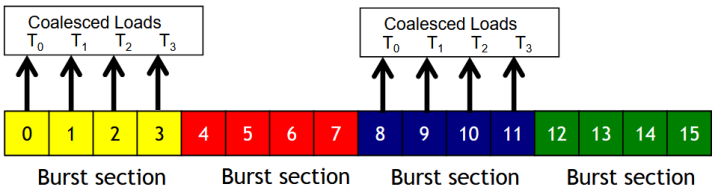
Single-bank burst timing: dead time on interface

Multi-bank burst timing: reduced dead time

5.5.3 Memory coalescing

Memory coalescing is the single most important performance optimization in CUDA programming. It describes the scenario where the hardware is able to combine (or *coalesce*) the memory requests of multiple threads into a single physical transaction.

When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only *one DRAM request* will be made:



You can tell if an access is coalesced if, when you look at a group of threads (a warp), the memory addresses they access are perfectly sequential. That is, the index in an array access is in the form:

$$A[\text{threadIdx.x} + (\text{terms independent of threadIdx.x})]$$

6 Memory consistency

While our intuition suggests that a load (read) should always return the *latest* value written, the concept of *latest* is vague in parallel systems.

Memory consistency provides the strict rules to define this. It deals with the **apparent ordering** for all location, i.e., in what order do memory operations performed by one thread become visible to other threads.

6.1 Coherence vs consistency

It's important to distinguish between these two similar terms:

- **Coherence** concerns *only one memory location*. It ensures that if you write to a variable `A`, everyone eventually sees that new value of `A`.

  Defines requirements for the observed behavior of reads and writes to the **same** memory location: all processors must agree on the order of reads/writes to an address `X`. In other words, it is possible to put all operations involving `X` on a timeline such that the observations of all processors are consistent with that timeline.

  The goal of cache coherence is to ensure that the memory system in a parallel computer behaves as if the caches were not there, just like how the memory system in a one-processor system behaves as if the cache was not there (a system without cache would have no need for cache coherence).

- **Consistency** concerns the history of *all locations* combined. It ensures that the sequence of updates across different variables makes sense to everyone.

  Defines the behavior of reads and writes to **different** locations (as observed by other processors). While coherence only guarantees that writes to an address `X` *will*

eventually propagate to other processors, **consistency** deals with **when** writes to `X` propagate to other processors, relative to reads and writes to other addresses.

6.2 Memory operation ordering

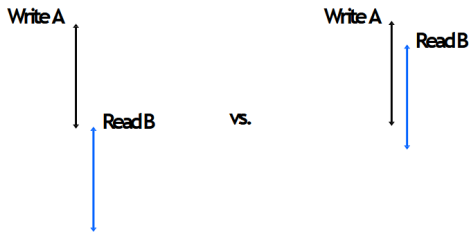
A program defines a sequence of loads and stores (this is the **program order**). There are 4 types of memory operation orderings:

- $W_X \rightarrow R_Y$: write to `X` must commit before subsequent read from `Y`.
- $R_X \rightarrow R_Y$: read from `X` must commit before subsequent read from `Y`.
- $R_X \rightarrow W_Y$: read from `X` must commit before subsequent write to `Y`.
- $W_X \rightarrow W_Y$: write to `X` must commit before subsequent write to `Y`.

Programmers should expect **sequential consistency**: it essentially promises that a parallel computer will behave like a single multitasking human. All operations from all treads are exdcuted in a single sequential order, as if they were manipulating a single shared memory. Meanwhile, the operations of any individual threads must appear in the exact order written in the code (program order).

A *sequentially consistent* memory system maintains all 4 memory operation orderings. **Relaxed memory consistency** models allow ceratin orderings to be violated.

Why are we interested in relaxed ordering requirements? To gain performance. Specifically, **hiding memory latency**: overlap memory access operations with other operations when they are independent (remember, memory access in a cache coherent system may entail much more work than simply rading bits from memory, e.g., finding data, sending invalidations, ec.).



6.3 Write buffer

To avoid stalling for many cycles while waiting for main memory, CPU designers add a small, high-speed storage are called **write buffer** to each processor.

For example, when *processor 0* executes `A = 1`,instead of sending the data all the way to the slow memory, it simply drops `A = 1` into its *local write buffer*. This action is nearly istantaneous.

Because the buffer accepts the data immediatle, *processor 0* believes the write is done. It proceeds instantly to the next instruction without waiting. Each processor reads from and writes to its own write buffer.

The write buffer allows the processors to "keep secrets" from each other (as the write buffer flushes its values to main memory at a "later time").

6.4 Allowing reads to move ahead of writes

To improve performance, modern architectures realax some of the rules of memory ordering. To do so, we can use models that deliberately break the $W_X \rightarrow R_Y$ rule:

- **Total Store Ordering (TSO)**: . A processor P can read `Y` before its write to `X` is seen by all processors (a processor can move its own reads in front of its own writes). Obviously, reads by other processors cannot return the new value of `X` until the write to `X` is observed by *all processors*.
- **Processor consistency (PC)**: any processor can read the new value of `X` before the write is observed by all processors.

With TSO and PC, only the rule $W_X \rightarrow R_Y$ is relaxed. All the other constraints still exist.

6.5 Allowing writes to be reordered

To speed thing up, we can also allow writes to be reordered, relaxing the rule $W_X \rightarrow W_Y$. This is called the **Partial Store Ordering (PSO)** model.

Example

Thread 1 on P1

```
A = 1;
flag = 1;
```

Thread 2 on P2

```
while (flag == 0)
    print A;
```

Processor P2 may observe the change to `flag` before the change to `A`.

6.6 Synchronization

Two memory accesses by differnt processors are considered to **conflict** if the access the same memory location, and at least one access is a *write*.

This leads to **data races**: the output of the program is no longer predictable, as it depends on the relative speed of the processors.

The solution is **synchronization**. Synchronized programs yield SC results on non-SC systems. There are no data races, and the reordering behavior doesn't matter. The accesses are ordered by synchronization, and *synchronization forces sequential consistency*.

In practice, most programs will be synchronized (via locks, barriers, etc., implemented in synchronization libraries).

7 Parallel algorithms and parallel programmming

7.1 Automatic parallelization

Automatic parallelization shields programmers from the complexities of multicore hardware. The programmer writes a standard sequential algorithm (high-level sequential

code), and leaves all the parallelization work to automatic tools. These tools transform the code into a parallel assembly implementation.

However, automatic parallelization often **fails** because compilers must be conservatively safe when handling memory references. the compiler doesn't always know whether some data structures are completely independent. Since the compiler does not know if a conflict exists, it must assume the worst-case scenario (that memory might overlap) and disable parallelization to guarantee the program's correctness.

Therefore, complete automatic parallelization is **not feasible**.

7.2 Parallelization by hand

In this case, the programmer needs to give **hints** to the tools. The programmer must explicitly write **parallel algorithms** implemented with high-level parallel code. The tools are only responsible for code compilation.

Here there are 3 critical aspects to keep in mind:

- Which **type** of parallelism has to be considered;
- How to **design** the parallel algorithm: trying to parallelize existing algorithms or designing one from scratch;
- How to **provide informations** about the parallelism to the tools.

7.3 Type of parallelism

There are 3 types of paralleism:

- **Bit Level Parallelism (BLP)**: the individual bits that make up a *word* are treated as representing different, independent data. A single instruction can manipulate **different data at a time** by operating on those bits simultaneously.

  This form of parallelism is critical for efficient hardware implementation, but it is also powerful in software (e.g., representing a set of elements as strings of bits, to perform massive logical operations in a single cycle).

- **Instruction Level Parallelism (ILP)**: different instructions executed at the same time on the same core. This capability is supported by multiple execution units, pipeline, vector, SIMD units, etc.

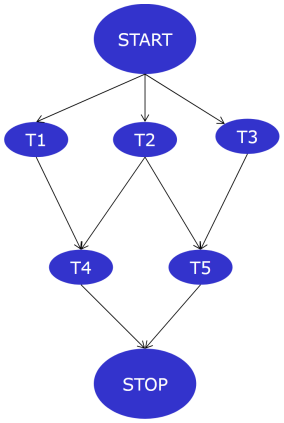
  This type of parallelism can be easily extracted by compilers.

- **Task Level Parallelism (TLP)**: instead of running single instructions simultaneously, the sysstem executes entire **tasks** in parallel. A task is a **logically discrete section** of computational work: typically a program or program-like set of instructions that is executed by a processor (much larger than a single instruction).

  A parallel program in this context is a collection of these *multiple tasks* running on multiple processors at the same time. While ILP focuses on doing more work on a single core, TLP is about using many cores.

  This technique is supported by shared memory and cache coherence mechanisms. It is usually difficult to be automatically extracted.

A fundamental model to analyze TLP is the **parallel data graph**: vertices correspond to tasks, while edges represent precedencies or data communication. Each task is executed once.



is created by the operating system

There are two main model for communication in TLP:

- **Shared memory**: all the processors (so all the tasks) share a global memory with the same address space. Modifications in a memory locatio performed by a processor are seen by all other processors.
- **Message passing**: each task has its private memory. Tasks communicate by explicitly sendind and receiving messages.

7.4 Designing parallel algorithms

Designing a good parallel algorithm by extracting all the available parallelism is *not enough*. Not all the extracted parallelism is exploitable on a real architecture.

We need to consider which parallelism is available on the considered architecture: non-suitable parallelism can introduce **overhead**. We need to *describe* the parallelism to the compilation tools to make it exploitable.

New programming languages have been introduced since parallel programming is a different paradigm. however, the **did not have much success** (mainly used for research).

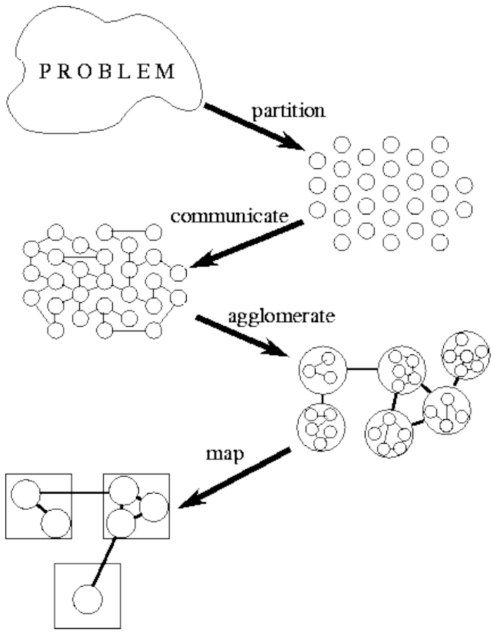
Instead, we generally use extensions to existing programming languages, as can be **easily adopted** by designers and **easily integrated** in existing compilers. However, in this way we can **describe only some types** of parallelism (e.g., pipeline parallelism is difficult to be described).

In general, designing parallel algorithms *is not an easy task*. There are however some **rules** which can help in the design. Most problems have several possible parallel solutions, a good approach is to start fromm machine-independent issues (concurrency) and delay target-specific aspects as much as possible.

7.4.1 PCAM design methodology

The **PCAM design methodology** is a pipeline for developing parallel algorithms:

1. **Partitioning**: focus on task and data decomposition, ignoring the limitations of the target hardware and focusing solely on exposing parallelism. There are two approaches:
   - *Domain decomposition*: the data associated with a problem is decomposed, and each parallel task then work on a portion of the data.
   - *Functional decomposition*: the problem is decomposed according to the work that must be done.
2. **Communication**: address task execution coordination. Define the communication structure of the algorithm, establishing which tasks must wait for data from others and which can proceed independently.
3. **Agglomeration**: evaluate the structure to optimize performance. Here we group related tasks into larger clusters to reduce the communication overhead and ensure that the task is appropriate for the system.
4. **Mapping**: handle the actual resourse assignment. Assign the agglomerated task clusters to actual physical resources.



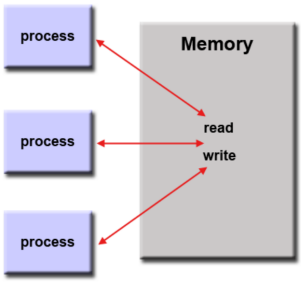
In practice, these steps are often overlapping.

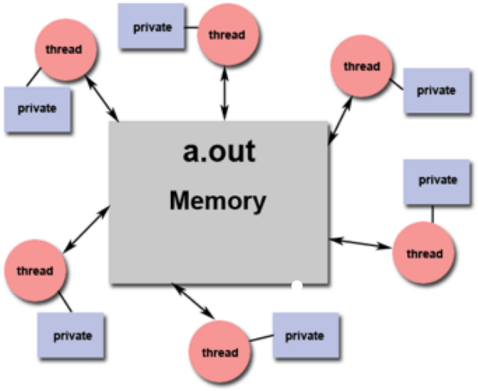
8 POSIX threads and OpenMP

POSIX threads and OpenMP are two implementations of **shared memory** parallel programming model with **threads**. Parallel programming models are an *abstraction* above the hardware. There are different implementations for each parallel programming model.

8.1 Threads model

Initially, the shared memory model was implemented using processes mapped directly to the physical memory (*no threads*), allowing data too be shared directly. However, it makes it difficult to maintain locality (ensuring the data is stored close to the processor currently using it).





The **threads model**, instead, allows to run multiple threads for a single process that can execute concurrently. This allows each thread to have local data, but can also access all resources acquired by the main proces.

A **UNIX process** is created by the operating system. Processes contain information about the program resources and execution state: process ID, process group ID, user ID, and group ID; environment; working directory; program instructions; etc.

A **thread** is an independent stream of instrucitons within a process. Threads can be scheduled to run by the operating system, meaning multiple threads can execute simultaneously (concurrently).

Threads have their own local resources and can also access the shared process resources. A thread can be seen as a **procesure** that runs independently from its main program.

Example

An example of a multi-thread program is the `main`, which calls some procedures. All the procedures run *simultaneously and/or independently* by the operating system.

Threads can execute *dynamically* during execution. Multi-thread is *lighter* than multi-processes, as they duplicate only the *bare essential* resources.

Since threads exist within a process and share most of the process resources, we have that:

- Changes made by one thread to the shared system resources (such as closing a file) will be seen by all other threads;
- Two pointers having the same value point to the same data;
- There is **implicit communication** by reading and writing shared variables;
- Reading and writing to the same memory locations requires **explicit synchronization** by the programmer.

The programmer is responsible for handling parallelism and synchronization, usually through a library of subroutines, and a set of compiler directives.

Hardware vendors have implemented their **own prorietary versions** of threads. We will see 2 different standards: POSIX threads (Pthreads) and OpenMP.

8.2 Pthreads

Pthreads is the standard that specifies the **API to explicitly manage threads**.

The Pthread API functionality is divided into two main categories:

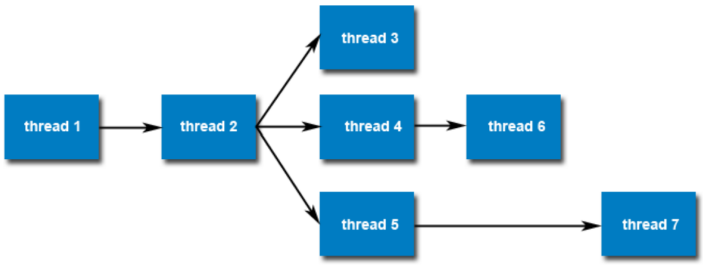
1. **Thread management**: functions to handle the *creation* of new threads, *joining* (waiting for) threads to finish, *detaching* threads to run them independently, and

setting or querying thread attributes.

2. **Synchronization**: tools to coordinate accesses and prevent conflicts: *mutexes* to control access for mutual exclusion sections, and *condition variables* to manage conditional inte-threads communications.

8.2.1 Thread management: Creation

Once created, threads are peers, and may even create other threads. there is no implied hierarchy or dependency between threads (the maximum number of threads depends on the particular implementation).



The `pthread_create` function is the primary subroutine used to create a new thread:

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

where:

- `thread` is the identifier for the new thread returned by the subroutine.
- `attr` is used to set the thread attributes: joinable/detached, scheduling, stack size.
- `start_routine` is the C routine that the thread will execute once it is created.
- `arg` is the argument passed to `start_routine`. It must be passed by address as a pointer cast of type `void`.

8.2.2 Thread management: Termination

A thread's execution can be terminated in several ways:

- The thread finishes its work and returns from its starting routine;
- The thread ends itself by callling the `pthread_exit` subroutine;
- The thread is canceled by another peer thread via the `pthread_cancel` routine;
- The entire process is canceled;

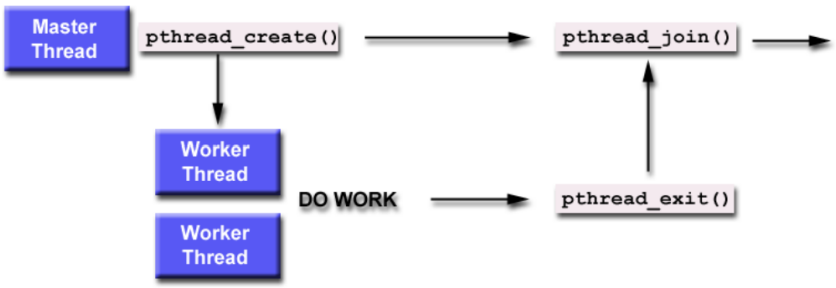
Note that if the main thread calls `pthread_exit` (rather than returning or callling `exit()`), the process remains alive, allowing the other created threads to "survive" and continue running.

8.2.3 Thread management: Joining

The `pthread_join` subroutine is used to synchronize thread execution by forcing the calling thread to **pause** until the specific target `thread` has completely **terminated**.

```
int pthread_join(pthread_t thread, void ** retval)
```

It also retrieves results from the finished thread: the `retval` argument will contain the copy of the data that the target thread passed to `pthread_exit` just before it ended.



8.2.4 Thread management: Detaching

Threads can be created as **joinable** or **detached**. A joinable thread can become detached, but not vice-versa. The joinable/detatched attribute is set explicitly.

A **detatched** thread runs independently. When it reminates, its resources are automatically released without requiring another thread to wait for it. Detached threads may consume less resources.

8.2.5 Thread management: Joining through Barriers

Unlike `pthread_join`, which forces one thread to fait for a specific single thread to terminate, **barriers** allow a **group** of threads to synchronize together. This way, all threads of the group need to wait for all other threads, then they can resume working.

The function `pthread_barrier_init` sets up the barrier. `pthread_barrier_wait` makes a thread wait for all other threads.

```
int pthread_barrier_init(
    pthread_barrier_t * barrier,
    pthread_barrierattr_t * attr,
    unsigned int count
)
```

```
int pthread_barrier_wait(pthread_barrier_t * barrier)
```

`count` is the number of threads to be waited.

8.2.6 Syncrhonization: Mutexes

Mutexes (mutual exclusion) variables are the basic method to protect shared data when multiple writes occur.

```
pthread_mutex_lock(&my_lock);
// critical section
pthread_mutex_unlock(&my_lock);
```

Only one thread can **lock** a mutex variable at any given time. If several threads try to lock a mutex, only one thread will be successful, while the other threads that could not acquire the mutex are blocked.

Note that multiple mutexes can lead to **deadlocks**.

To reduce blocking overhead, we can use `trylock`. Unlike standard lock requests that puts a thread to sleep if the resource is busy, `trylock` allows the thread to perform other work instead of waiting if the resource is busy.

8.2.7 Synchronization: Condition variables

Mutexes implement synchronization by serializing data accesses. **Condition variables**, instead, allow threads to synchronize based on the actual state of the data. One thread can pause execution until another thread explicitly signals that a specific condition has been met.

Without condition variables, the programmer would need to loop (**poll**) continuously to check if the condition is met.

8.3 OpenMP

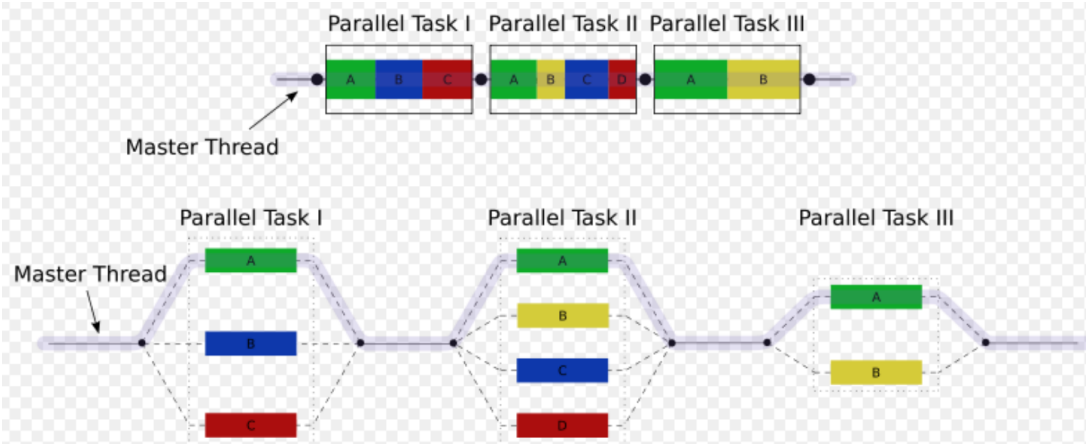
OpenMP is an API for multi-threaded shared memory programming. It mainly provides **compiler directives**, but also library routines, and environment variables. OpenMP requires specific compiler support to interpret these directives.

It is characterized by its high **ease of use**, as it relies on a *simple and limited* set of directives: 3 or 4 are enough to implement significant parallelism. This simple design facilitates **incremental parallelization**, allowing developers to gradually transform a serial program into a parallel one step-by-step, rather than rewriting the entire code base at once. Additionally, this model supports both *coarse-grained* and *fine-grained* parallelism.

OpenMP is based on the **fork-joint** paradigm, which dictates how parallel execution begins and ends:

1. **Master thread**: the execution starts with a single *master* thread running sequentially;
2. **Forking**: when a parallel region is reaches, the master thread *forks* a team of *slave* threads;
3. **Parallel execution**: the work (*tasks*) is divided among these *slave* threads, which run concurrently on different processors, as allocated by the runtime environment;

4. **Joining**: once the parallel task is complete, the thread synchronize and join back together, leaving only the master thread to continue until the next parallel region.



8.3.1 OpenMP C/C++ syntax

The OpenMP C/C++ syntax consists of 3 main components:

1. **Preprocessor directives** (`#pragma`): they are the main part of the OpenMP standard and are used to identify parallel tasks and handle synchronization. The general syntax for these directives is `#pragma omp <name> [list-of-clauses]`.
2. **Auxiliary C functions**: helper functions used to query or set runtime information (such as the number of available threads) and to manage explicit locks.
3. **Environment variables**: allow the user to configure runtime behavior without changing the code.

8.3.2 Parallel control structures

OpenMP programs execute serially until they reach a `parallel` directive:

```
#pragma omp parallel
{
    /* parallel section */
}
```

1. The thread that was executing the code spawns a group of slave threads and becomes the master (thread ID 0);
2. The code in the structured (the parallel section) is replicated, and each thread executes a copy;
3. At the each of the block there is an implied barrier: the master waits for all slaves to finish their work before it continues execution alone.

There are some **optional clauses** to the `parallel` directive:

- Conditional parallelization with `if (condition)`, to decide at runtime whether to run in parallel. If the condition is false, the overhead of creating threads is avoided.

- Force a specific number of spawned threads with `num_threads(int)`.
- Data scope clauses.

Under the hood, depending on the compiler, it might replace the directive with a Pthreads implementation.

The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Evaluation of the textttif clause. If it evaluates to false, then the region runs serially (1 thread).
2. Value of the `num_threads` clause.
3. If neither of the above is set, the system checks if the thread count was globally set within the code using the `omp_set_num_threads()` library function.
4. The system then looks outside the program to the `OMP_NUM_THREAD` environment variable.
5. Finally, if absolutely nothing is specified, it falls back to the implementation default (the number of available CPU cores).

8.3.3 Work sharing

Work sharing constructs in OpenMP are mechanisms designed to distribute the execution of a specific code region among the threads that are already active in a team. The key characteristics are:

- A work-sharing construct must be *enclosed* within a `parallel` region in order for the directive to execute in parallel;
- Work-sharing constructs do not launch new threads, they simply assign tasks to the members of the existing thread team.
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of the construct.

For example, parallelize the execution of iterations with `for`. Note that the iterations number cannot be internally modified.

```
#pragma omp parallel
{
    #pragma omp for
    /* for loop */
}
```

Other possible `for`'s clauses include:

- `schedule`: describes how the iterations of the loop are divided among the threads in the team. The most typical scheduling types for a `for` loop are:

  - `static`: loop iterations are divided into blocks of size `chunk` and then statically assigned to threads. If `chunk` is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
  - `dynamic`: loop iterations are divided into blocks of size `chunk`, and dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
  - `runtime`: depends on the environment variable `OMP_SCHEDULE`.
- `nowait`: avoid synchronizing at the end of the parallel loop;
- `reduction`: a reduction variable in a loop aggregates (accumulates) a value that depends on each iteration of the loop, and does not depend on the iteration order (e.g., a sum).

  The `reduction (operator: list)` clause helps to perform a reduction:
  - Each thread updates a private copy of each variable in the list;
  - At the end the *reduction operator* is applied to all private copies and the end result is written into a global variable.
- Data scope clauses.

Another work-sharing mechanism in OpenMP is the `section` construct. While loops are used for data parallelism (running the same code on different data), sections are used for running completely different tasks at the same time.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            /* code section 1 */
        }
        #pragma omp section
        {
            /* code section 2 */
        }
    }
}
```

Each `section` is executed once by a thread in the team.

If you are inside a parallel region but need a specific block code to run only **once**, rather than being replicated by every thread, you can use the `single` (execute only by a single thread) and `master` (execute only by the master) directives:

```
#pragma omp parallel
{
    #pragma omp single
    {
        /* code section */
    }
    #pragma omp master
    {
        /* code section */
    }
}
```

The `task` directive introduces a more dynamic method of parallelism. It specifies a work unit which may be executed or deferred (put in a queue) to be run later by another thread in the same team.

```
#pragma omp task
{
    /* code section */
}
#pragma omp taskwait
#pragma omp taskyield
```

The `taskwait` directive introduces a barrier.

The `taskyield` directive interrupts the execution of the current task. May be resumed by the same thread or by another.

8.3.4 Synchronization

The `critical` directive specifies a region of code that must be executed by only one thread at a time.

```
#pragma omp critical [name]
{
    /* code section */
}
```

The optional `name` enables multiple different `critical` regions:

- names act as global identifiers: different `critical` regions with the same name are treated as the same region;
- all `critical` sections which aare unnamed, are treated as the same section.

The `barrier` directive synchronizes all threads in the team.

```
#pragma omp barrier
```

When a `barrier` directive is reached, a thread will wait at that point untill all other threads have reached that barrier. All threads then resume executing in parallel the code taht follows the barrier.

It is not very used because of implicit synchronization of other constructs.

The `atomic` directive ensures that a specific storage location is accessed atomically.

```
#pragma omp atomic
/* statement */
```

Multiple reads and writes to an atomic operations are not allowed. Only valid for the following statement, not for a structured block.

8.3.5 Data environment

OpenMP is based upon the shared memory programming model so most variables are **shared by default**.

The OpenMP Sata Scope Attribute clauses are used to explicitly define how variables should be scoped. They include: `private`, `shared`, `default`, and `reduction`.

Data Scope Attribute clauses are used in conjunction with several directives to:

- define how and which data variables in the serial section of the program are transferred to the parallel sections;
- define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.

The `shared` clause declares variables in its list to be shared among all threads in the team.

```
#pragma omp <name> shared (list)
```

A `shared` variable exists in only one memory location and all threads can read or write to that address. It is the programmer's responsibility to ensure that multiple threads properly access `shared` variables.

8.3.6 Runtime functions

the OpenMP standard defines an API for library calls that perform a variety of functions to control execution of the program:

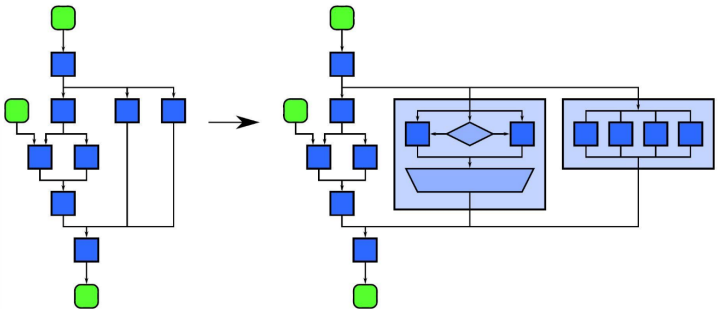
- `int omp_get_num_threads()` returns the number of threads that are currently in the team executing the parallel region from which it is called.
- `int omp_get_thread_num()` returns the identifier for the thread making this call.

- `void omp_set_num_threads(int num_threads)` sets the number of threads that will be used in the next parallel region.
- `double omp_get_wtime()` provides a wall clock timing routine.
- `double omp_get_wtick()` returns the number of seconds between two clock ticks.

9 Parallel patterns

Parallel patterns are a recurring combination of task distribution and data access that solve a specific problem in parallel algorithm design. Patterns provide us with a vocabulaty for algorithm design and are universal, they can be used in *any* parallel programming system.

We can use **nesting**, that is the ability to hierarchically compose patterns.



9.1 Dependencies

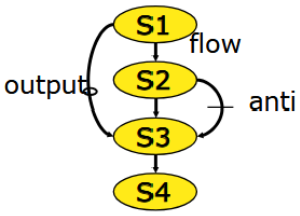
Parallel execution must address control, data, and system dependencies. A **dependency** arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed. We extend this notion of dependency also to resources.

We say that two statements are **independent** if their order of execution does not matter. Instead, when the order of their execution affects the computation outcome, they are **dependent**.

Given statements S1 and S2 (executed in this order), we have that:

- S2 has a **flow dependence** on S1 if and only if S2 reads a value written by S1;
- S2 has an **anti-dependence** on S1 if and only if S2 writes a value read by S1;
- S2 has an **output dependence** on S1 if and only if S2 writes a value written by S1;

We can use graphs to show dependence relationships, for example



Two statements can execute in parallel if and only if there are **no dependencies** between them. Some dependencies can be removed by modifying the program, rearranging and eliminating statements.

Data dependence relations can be found by comparing the IN and OUT sets of each node, defined as:

- IN(S): the set of memory locations (variables) that may be used in S;
- OUT(S): the set of memory locations (variables) that may be modified by S.

9.2 Loop-level parallelism

Significant parallelism can be identified **within** loops. We can unroll loops into separate statements to show dependencies between iterations.

A **loop-carried** dependency between two statements is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different loop iterations).

Otherwise, it is **loop-independent**, including if it has dependencies between two statements in the same loop iteration.

9.3 Parallel control patterns

Parallel control patterns extend serial control patterns. Each pattern is related to at least one serial control pattern, but it relaxes assumptions of serial control patterns. Those patterns are: fork-join, map, stencil, reduction, scan, recurrence.

9.3.1 Parallel control patterns: Fork-join

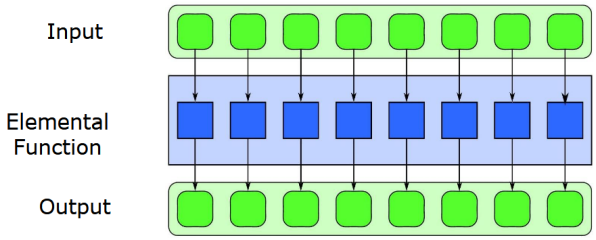
Fork-join allow control flow to fork into multiple parallel flows, then rejoin later.

A *join* is different than a *barrier*. Here only one thread continues, while with a barrier all threads continue.

9.3.2 Parallel control patterns: Map

Map performs a function over every element of a collection. It replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection.

The replicated function is referred to as an *elemental function*.



9.3.3 Parallel control patterns: Stencil

The **stencil** pattern updates an element in an elemental function by looking at the *neighbors* of each element.

Boundary conditions must be handled carefully in the stencil pattern.

9.3.4 Parallel control patterns: Reduction

Reduction combines every element in a collection using an associative *combiner function*. Thanks to the associativity of the combiner function, different orderings of the reduction are possible.

9.3.5 Parallel control patterns: Scan

Scan computes all partial reduction of a collection.

For every output in a collection, it computes a reduction of the input up to that point. If the function being used is associative, the scan can be parallelized.

Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop.

9.3.6 Parallel control patterns: Recurrence

Recurrence is a more complex version of map, where the loop iterations can depend on one another. It is similar to mup, but elements can use the outputs of adjacent elements as inputs.

For a recurrence to be computable, there *must* be a serial ordering of the recurrence elements, so that elements can be computed using previously computed outputs.

9.4 Serial data management patterns

Serial programs can manage data in many ways. Data management deals with how data is allocated, shared, read, written, and copies. Patterns are:

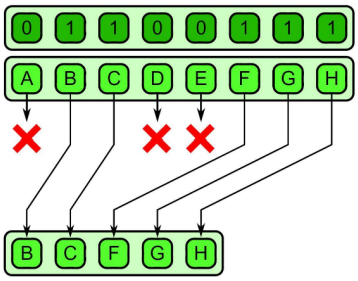
- **Random read and write**: memory locations are indexed with addressses. Pointers are typically used to refer to memory addresses.
- **Stack allocation**: useful for dynamically allocating data in a LIFO manner. It is efficient as an arbitrary amount of data can be alloated in constant time. Typically, when parallelized, each thread will get its own stack, so thread locality is preserved.

- **Heap allocation** useful when data cannot be allocated in a LIFO fashion, but is slower and more complex than stack allocation. A parallelized heap allocator will keep separate pools for each parallel worker.
- **Objects**: they are language constructs to associate data with code to manipulate and manage that data. Objects can have member functions, and they are also considered members of a class of an object.

9.5 Parallel data management patterns

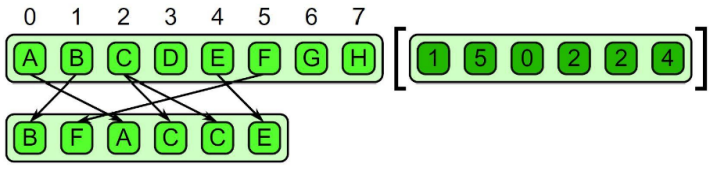
To avoid things like race conditions, it is very important to know when data is (and isn't) potentially shared by multiple parallel workers:

- **Pack**: used to eliminate unused space in a collection. Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order. It is useful when used with map.



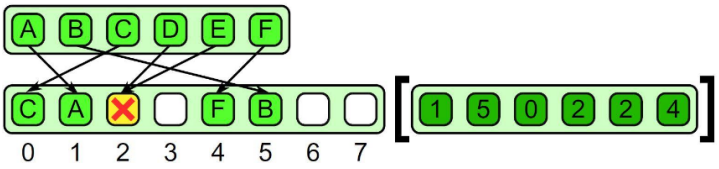
  Unpack is the inverse and is used to place elements back in their original locations.

- **Pipeline**: connects tasks in a producer-consumer manner. One task (the producer) creates data and passes it to the next task (the consumet), which processes it.
- **Geometric decomposition**: arranges data in subcollections. Overlapping and non-overlapping decompositions are possible. Note that this pattern doesn't necessarily move data, it just gives us another view of it.
- **Gather**: reads a collection of data given a collection of indices. Think of a combination of map and random serial reads.



  The output collection shares the same type as the input collection, but it shares the same shape as the indices collection

- **Scatter**: the inverse of gather. A set of input and indices is required, but each element of the input is written (instead of read) to the output at the given index. Race conditions can occur when we have two writes to the same location.



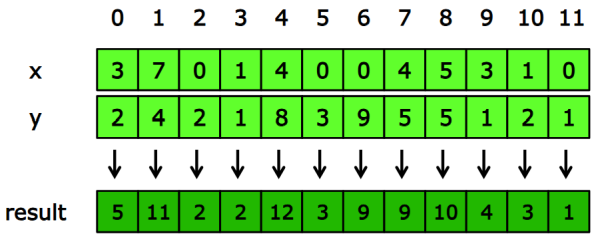
9.6 Map

Mapping involves taking a collection of data and applying the *same function* to every single element independently. An operation is a map if it can be applied to each element without knowledge of neighbors.

Since each iteration is independent, we can run map completely in parallel. Map function should be *pure* or *pure-ish* and should not modify shared states. Modifying shared states breaks perfect independence.

9.6.1 N-ary maps

So far, we have only dealt with mapping over a single collection. However, sometimes it makes sense to map over multiple collections at once.



9.6.2 Sequences of maps

Often several map operations occur in sequence. Vector math consists of many small operations such as additions and multiplications applied as maps.

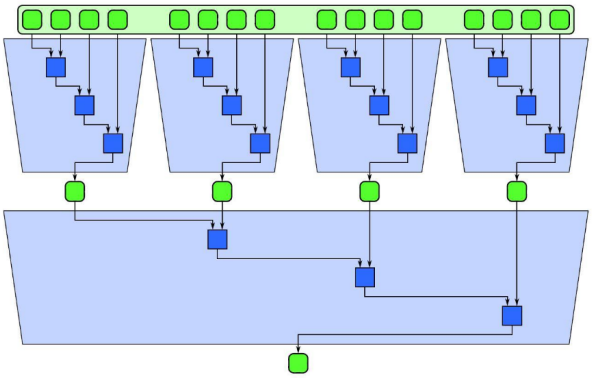
A naïve implementation may write each intermediate result to memory. However, we can sometimes **fuse** together the operations to perform them at once. This adds arithmetic intensity but reduces memory/cache usage. Ideally, operations can be performed using registers alone.

Sometimes it's impractical to fuse together the map operations. We can instead break the work into blocks, giving each CPU one block at a time.

9.7 Reduce

Reduce is used to combine a collection of elements into one summary value. A combiner function combines elements pairwise, and it only needs to be associative to be parallelizable.

We can use **tiling** to break-up chunks of work for workers to reduce serially.



Precision can become a problem with reductions on floating-point data. Different orderings of floating-point operations can change the reduction value.

Example: dot product

To perform the dot product between 2 vectors of the same length:

1. Map (*) to multiply the components one by one;
2. Reduce (+) to get the final answer.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

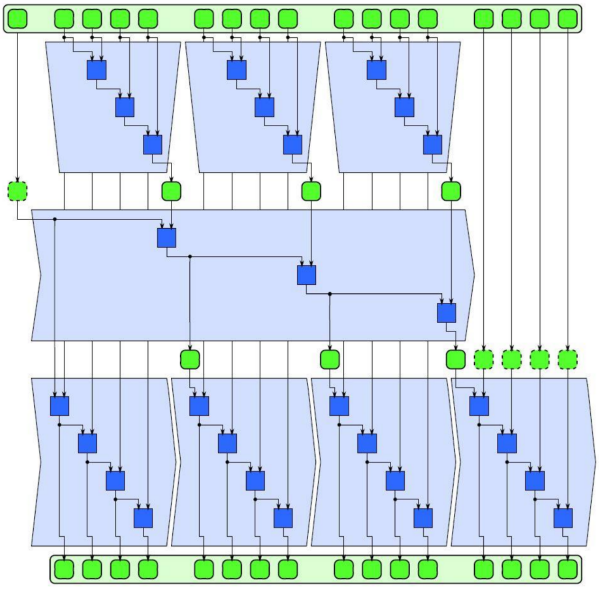
9.8 Scan

The **scan** pattern produces partial reductions of input sequence, generating a new sequence (e.g., every element in the output array will contain the sum of the elements before it). It is trickier to parallelize than reduce. There are 2 types of scans:

- **Inclusive scan**: includes current element in partial reduction;
- **Exclusive scan**: excludes current element in partial reduction. In this case, partial reduction is of all prior elements up to current element, excluded.

We can use **tiling** also on the scan pattern, which comprises 3 phases:

1. The input array is divided into independent chunks, and each thread performs a local scan on its own tile;
2. A scan is performed only on the output of phase 1;
3. The results calculated in phase 2 are fed back into the tiles.



Just like with reduce, we can also fuse the map pattern with the scan pattern.

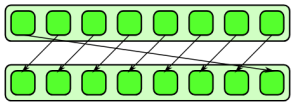
9.9 Gather

The **gather** pattern creates a collection of data by reading from another data collection.

Given a collection of ordered indices, read data from the source collection at each index, and write data to the output collection in index order. The element type of output collection is the same as the source, and the shape of the output collection is that of the index collection (same dimensionality).

9.9.1 Special case of gather: Shifts

Shifts are used to move data elements within memory, to the loft or to the right. The data accesses are offset by a fixed distance.

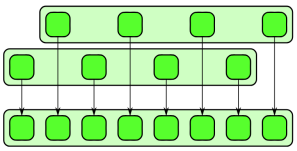


Since the movement is uniform, this pattern is highliy **regular**, which allows for specific hardware optimization. Shifts can also take advantage of good data locality.

The variants depend on how the boundary conditions are handles (e.g., rotate: when for example shifting left by 1, the 1st element becomes the last element of the new sequence).

9.9.2 Special case of gather: Zip

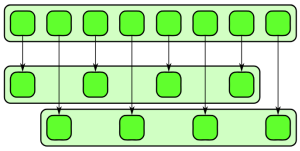
Zip takes two separate collections of data and combines them into a single collection by **interleaving** the elements (they alternate one-by-one).



It can be generalized to have more than just 2 sequences, and can also zip data of unlike types.

9.9.3 Special case of gather: Unzip

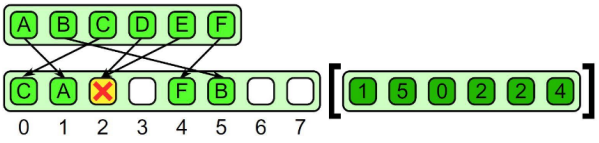
Unzip reverses a zip. It extracts sub-arrays at certain offsets and strides from an input array.



9.10 Scatter

While *gather* reads locations provided as input, the **scatter** pattern is a combination of map with random *writes*, where the write locations are provided as input.

Note that in the case of scatter, you cannot guarantee that the write destinations will be unique. This leads to **race conditions**. Parallel writes to the same location are **collisions**.



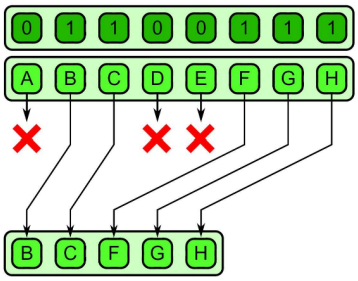
Note how the output collection does not need to be larger in size.

There exists different variations of scatter based on the way they handle **collisions resolution**:

- **Atomic scatter**: there is *no rule* to determine which one of the input items will be retained (non-deterministic).
- **Permutation scatter**: this patter simply states that collisions are *illegal*. It checks for collisions in advance.
- **Merge scatter**: associative and commutative operators are provided to merge the elements in case of a collision (e.g., sum the colliding elements). Both the associative properties are required since scatters to a particular location could occur in any order.
- **Priority scatter**: every element in the input harray has a priority based on its position. In case of a collision, the priority decides which element is written.

9.11 Pack

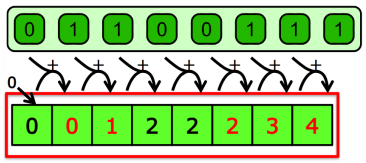
The **pack** pattern is used to eliminate unused elements from a collection. The remaining elements are then moved so that they are contiguous in memory.



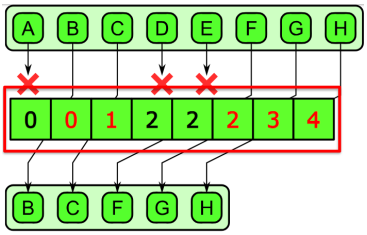
9.11.1 Pack algorithm

The **algorithm** goes as follows:

1. **Convert the type** of the input array of booleans into proper integers 0s and 1s.
2. Performe an **exclusive scan** of this integer array with the addition operation (every element will contain the sum up to and including the element itself).

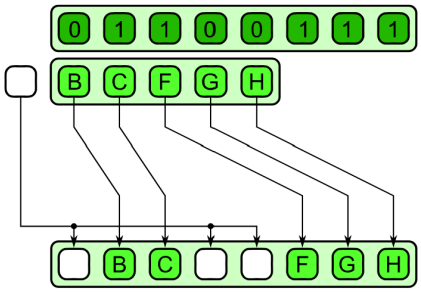


3. Each element of the input array will be written in the output array at the offset calculated in the previous step.



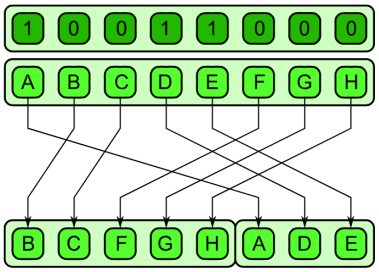
9.11.2 Unpack

Unpack is the inverse of the pack operation. Given the same data on which elements were kept and which were discarded, spread the elements back in their original locations.



9.11.3 Split

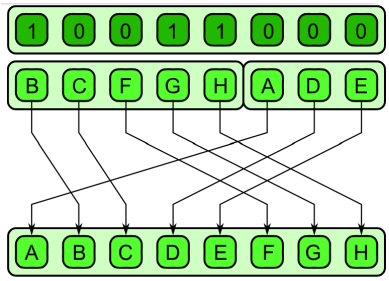
Split is a generalization of the pack pattern. Elements are moved to the upper or lower half of the output collection based on some state.



In this case we do not lose information like for pack.

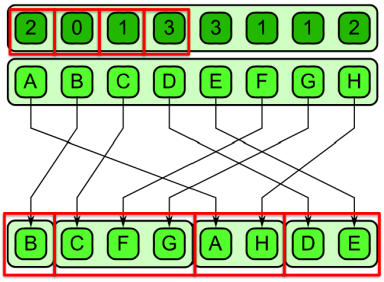
9.11.4 Unsplit

Unsplit is the inverse operation of split. It creates the ouput collection based on the original input collection.



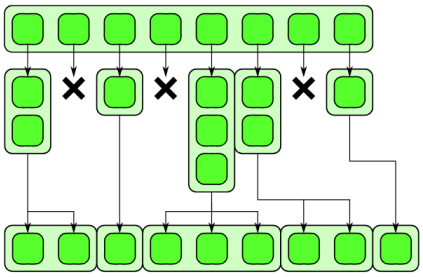
9.11.5 Bin

Bin is a generalized split taht supports more categories (instead of just upper/lower, here we can have more than 2 bins).



9.11.6 Expand

In the **expand** operation, each element can produce *any* number of elements. The results are then fused together in order.



10 Heterogeneous computing

A **heterogeneous system** is a system that uses multiple types of cores to handle the complex variety of tasks found in real-world applications.

10.1 Energy-efficient computing

Given a fixed budget, the goal is not only to increase performance, but also **energy efficiency**. Specialization (i.e., hardware specifically designed for a fixed task) can generally provide better energy efficiency.

General-purpose processors are not energy efficient, as they always need to perform the same steps to address every operation (read instruction, decode, check for dependencies, etc.), while the actual circuit that would be required to perform that specific computation can could have been simple.

We can reduce energy consumption by using specialized processors, and trying to move less data. Minimizing the communication overhead increases performance, while also reducing energy consumption.