Parallel Computing - Notes - v0.8.0

260236

November 2024

Preface

Every theory section in these notes has been taken from the sources:

- Validity of the single processor approach to achieving large scale computing capabilities. [1]
- Introduction to parallel algorithms, Carnegie Mellon University. [2]
- Course slides. [3]
- Reevaluating Amdahl's law. [4]
- OpenMP by Example, Johnston Hans. [5]
- Introduction to parallel computing, volume 110. [6]
- NVIDIA Tesla: A unified graphics and computing architecture. [7]
- Structured Parallel Programming: Patterns for Efficient Computation. [8]
- The critical directive, OpenMP, Microsoft. [9]
- Multithreading Architecture. [10]
- Cache, Wikipedia. [11]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on parallel computing. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 PRAM

1.1 Prerequisites

Before we introduce the PRAM model, we need to cover some useful topics.

- A **Machine Model** describes a "machine". It gives a value to the operations on the machine. It is necessary because: it makes it easy to deal with algorithms; it achieves complexity bounds; it analyses maximum parallelism.
- A **Random Access Machine (RAM)** is a model of computation that describes an abstract machine in the general class of register machines. Some features are:
  - **Unbounded** number of local memory cells;
  - Each memory cell can hold an integer of **unbounded** size;
  - Instruction set includes simple operations, data operations, comparator, branches;
  - All operations take **unit time**;
  - The definition of **time complexity** is the number of instructions executed;
  - The definition of **space complexity** is the number of memory cells used.

1.2 Definition

Definition 1: PRAM

A **parallel random-access machine (parallel RAM or PRAM)** is a **shared-memory abstract machine**. As its name indicates, the PRAM is intended as the parallel-computing analogy to the random-access machine (RAM) (not to be confused with random-access memory). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the **PRAM is used by parallel-algorithm designers to model parallel algorithmic performance** (such as time complexity, where the number of processors assumed is typically also stated).

The PRAM model has many interesting features:

- **Unbounded collection of RAM processors** (P_0, P_1, and so on);
- Processors don't have tape;
- Each processor has **unbounded registers**;
- **Unbounded** collection of **share memory cells**;
- All **processors** can **access all memory cells in unit time**;
- All **communication via shared memory**.

1.3 How it works

1.3.1 Computation

A single **processor** of the PRAM, at each computation, is **composed of 5 phases** (carried out in parallel by all the processors):

1. **Reads a value from one of the cells** $X(1), \ldots, X(N)$
2. Reads one of the shared memory cells $A(1), A(2), \ldots$
3. Performs some internal computation
4. **May write into one of the output cells** $Y(1), Y(2), \ldots$
5. May write into one of the shared memory cells $A(1), A(2), \ldots$

1.3.2 PRAM Classificiation

During execution, a subset of processors may remain idle. Also, some processors can read from the same cell at the same time (not really a problem), but they could also try to write to the same cell at the same time (**write conflict**). For these reasons, PRAMs are classified according to their read/write capabilities (realistic and useful):

- **Exclusive Read (ER)**. All processors can simultaneously read from distinct memory locations.
- **Exclusive Write (EW)**. All processors can simultaneously write to distinct memory locations.
- **Concurrent Read (CR)**. All processors can simultaneously read from any memory location.
- **Concurrent Write (CW)**. All processors can write to any memory location.

  But what value is ultimately written?

  It depends on the mode we choose:

  - **Priority Concurrent Write**. Processors have priority based on which value is decided, the **highest priority is allowed to complete write**.
  - **Common Concurrent Write**. All processors are allowed to complete write **if and only if all the value 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 Concurrent Write**. One **randomly** chosen **processor is allowed to complete write**.

1.3.3 Strengths of PRAM

PRAM is attractive and important model for designers of parallel algorithms because:

- It is **natural**. The number of operations executed per one cycle on P processors is at most P (equal to P is the ideal case).
- It is **strong**. Any processor can read/write any shared memory cell in unit time.
- It is **simple**. It abstracts from any communication or synchronization overhead, which makes the complexity and correctness of PRAM algorithm easier.
- It can be used as a **benchmark**. If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution for any parallel machine.

1.3.4 How to compare PRAM models

Consider two generic PRAMs, models A and B. Model A is **computationally stronger** than model B ($A \geq B$) **if and only if any algorithm** written for model B will **run unchanged** on model A in the **same parallel time** and with the **same basic properties**.

However, there are some useful metrics that can be used to compare models:

- **Time to solve problem** of **input size** n on **one processor**, using **best sequential algorithm**:
$$T^* (n) \tag{1}$$
- **Time** to solve problem of input size n on **p processors**:
$$T_p (n) \tag{2}$$
- **Speedup on p processors**:
$$\mathrm{SU}_p (n) = \frac{T^* (n)}{T_p (n)} \tag{3}$$
- **Efficiency**, which is the work done by a processor to solve a problem of input size n divided by the work done by p processors:
$$E_p (n) = \frac{T_1 (n)}{pT_p (n)} \tag{4}$$
- **Shortest run time** on any process p:
$$T_\infty (n) \tag{5}$$

- **Cost**, equal to processors and time:

$$C(n) = P(n) \cdot T(n) \tag{6}$$

- **Work**, equal to the total **number of operations**:

$$W(n) \tag{7}$$

Some properties on the metrics:

- The time to solve a problem of input n on a single processor using the best sequential algorithm *is not equal to* the time to solve a problem of input n in parallel using one of the p processors available. In other words, **the problem should not be solvable on a single processor on a parallel machine** (otherwise, what would be the point of using a parallel model?)

$$T^* \neq T_1$$

- $\mathrm{SU}_P \leq P$
- $\mathrm{SU}_P \leq \dfrac{T_1}{T_\infty}$
- $E_p \leq 1$
- $T_1 \geq T^* \geq T_p \geq T_\infty$
- $T^* \approx T_1 \Rightarrow E_p \approx \dfrac{T^*}{pT_p} = \dfrac{\mathrm{SU}_p}{p}$
- $E_p = \dfrac{T_1}{pT_p} \leq \dfrac{T_1}{pT_\infty}$
- $T_1 \in O(C), T_p \in O\left(\dfrac{C}{p}\right)$
- $W \leq C$
- $p \approx \text{AREA} \quad W \approx \text{ENERGY} \quad \dfrac{W}{T_p} \approx \text{POWER}$

1.4 MVM algorithm

The **Matrix-Vector Multiply (MVM) algorithm** consists of four steps:

1. **Concurrent read of vector** $X\,(1:n)$ (transfer N elements);
2. **Simultaneous reads of different sections of the general matrix** A (transfer $\frac{n^2}{p}$ elements to each processor);
3. **Compute** $\frac{n^2}{p}$ operations per processor;
4. **Simultaneous writes** (transfer $\frac{n}{p}$ elements from each processor).

Let i be the processor index, so the MVM algorithm is simply written as:

```
1 GLOBAL READ (Z ← X)
2 GLOBAL READ (B ← A_i)
3 COMPUTE (W := BZ)
4 GLOBAL WRITE (W → y_i)
```

Algorithm 1: Matrix-Vector Multiply (MVM)

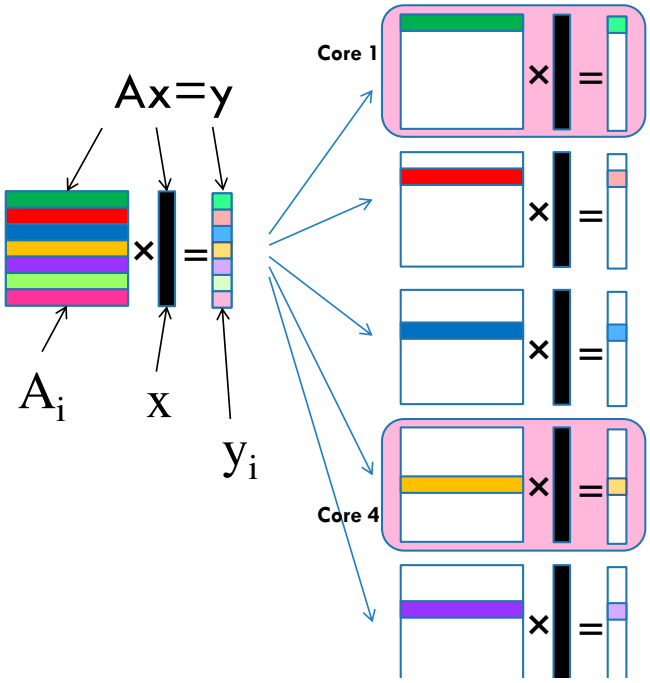


Figure 1: Example of MVM algorithm.

The performance of the MVM algorithm is as follows:

- The **time to solve** a problem of size n^2 is equal to the big O of the squared size of the problem as input divided by the number of processors available:
$$T_p\left(n^2\right) = O\left(\frac{n^2}{p}\right)$$
- The **cost** is equal to the number of processors and the time it takes to solve the problem. So it is quite trivial:
$$C = O\left(\cancel{P} \cdot \frac{n^2}{\cancel{p}}\right) = O\left(n^2\right)$$
- The **work** is equal to the cost, and the **linear power** P is equal to the ratio of work and time to solve the problem on p processors:
$$W = C \qquad \frac{W}{T_p} = P$$
- The **perfect efficiency** is equal to:
$$E_p = \frac{T_1}{pT_p} = \frac{n^2}{p\frac{n^2}{p}} = 1$$

1.5 SPMD sum

The **Single Program Multiple Data (SPMD)** is a term that has been used to **describe computational models** for exploiting parallelism, where **multiple processors work together to execute a program to get results faster**.

In this section, we will see an SPMD approach on a Parallel Random Access Machine (PRAM). We will introduce one of the most common and simple mathematical operations: the sum.

The following pseudocode takes as **input an array** of size $n = 2^k$. In this case, n is a power of 2 because it ensures that the array can be evenly divided at each step of the computation. The value k is the number of iterations or levels of the summation process.

```
1  BEGIN
2      GLOBAL READ (A ← A(I))
3      GLOBAL WRITE (A → B(I))
4      FOR H = 1 : K
5          IF i ≤ n ÷ 2^h THEN BEGIN
6              GLOBAL READ (X ← B(2i - 1))
7              GLOBAL READ (Y ← B(2i))
8              Z := X + Y
9              GLOBAL WRITE (Z → B(i))
10         END
11     IF I = 1 THEN
12         GLOBAL WRITE (Z → S)
13 END
```

Algorithm 2: Single Program Multiple Data (SPMD) sum

- First, read the entire input array `A` and copy the read data to another array `B`.
- Loop over `h` (`1 to k`). In each iteration, for each index i less than or equal to $n \div 2^h$, read values from array `B` at positions $2i - 1$ and $2i$; sum these values (and store the result in `Z`) and store the result (`Z`) back into `B(`i`)`.
- Once all iterations are complete, the final sum is stored in a variable `S`.

For example, if $n = 8$, then k would be 3, meaning that the algorithm will run for 3 iterations to sum all the elements in parallel.

h	i	adding
1	1	1,2
	2	3,4
	3	5,6
	4	7,8
2	1	1,2
	2	3,4
3	1	1,2

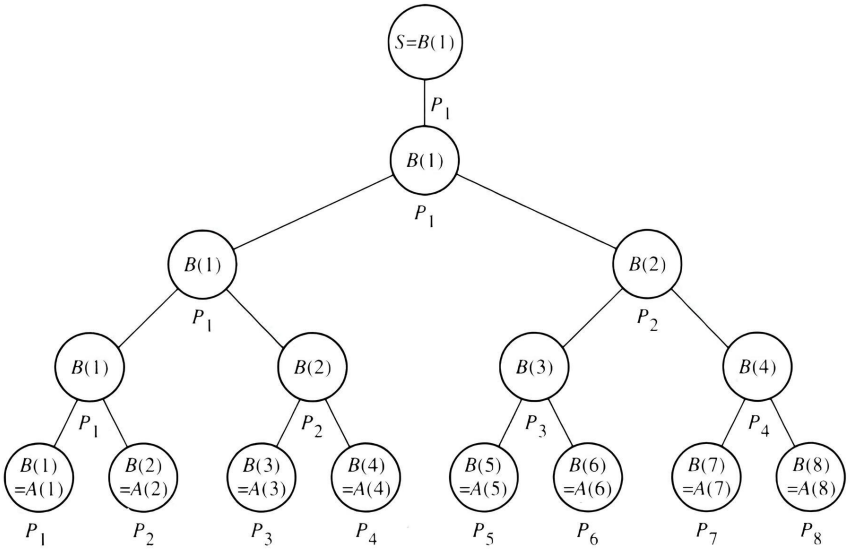
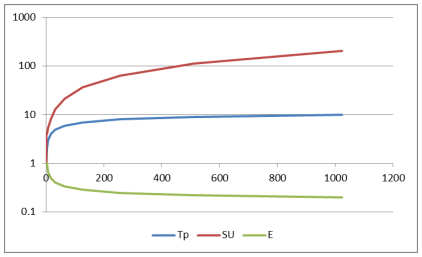


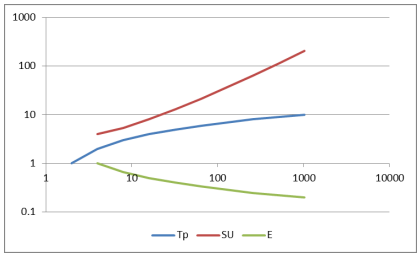
Figure 2: Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

Performance of sum

When the size of the array is equal to the number of processors ($N = P$), the **speedup and efficiency decrease**:

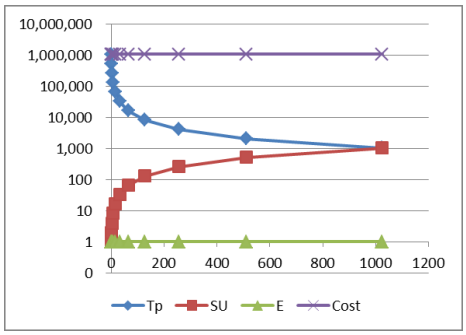
- $T^*(N) = T_1(N) = N$
- $T_{N=P}(N) = 2 + \log N$
- $\mathrm{SU}_P = \dfrac{N}{2 + \log N}$
- $T^*(N) = P \cdot (2 + \log N) \approx N \log N$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{N}{N \log N} = \dfrac{1}{\log N}$





If the size of the array is much larger than the number of processors ($N \gg P$), the **speedup and power are linear**, the **cost is fixed** and the **efficiency is maximum (equal to 1)**:

- $T^*(N) = T_1(N) = N$
- $T_p(N) = \frac{N}{p} + \log p$
- $\mathrm{SU}_P = \frac{N}{\frac{N}{p} + \log p} \approx P$
- $\mathrm{COST} = p\left(\frac{N}{p} + \log p\right) \approx N$
- $\mathrm{WORK} = N + P \approx N$
- $E_p = \frac{T_1}{pT_p} = \frac{N}{p\left(\frac{N}{p} + \log p\right)} \approx 1$



$n = 1'000'000$

Example 1

Refer to Figure 2 (page 12), the performance metrics are:

- $T_8 = 5$
- $C = 8 \cdot 5 = 40$ (could do 40 steps)
- $W = 2n = 16$ (16 on 40, wasted 24)
- $E_p = \frac{2}{\log n} = \frac{2}{3} = 0.67$
- $\frac{W}{C} = \frac{16}{40} = 0.4$

There is also the **Prefix Sum**, which takes **advantage of idle processors in the sum**. It computes all prefix sums:

$$S_i = \sum_{1}^{i} a_j \qquad a_1, \quad a_1 + a_2, \quad a_1 + a_2 + a_3$$

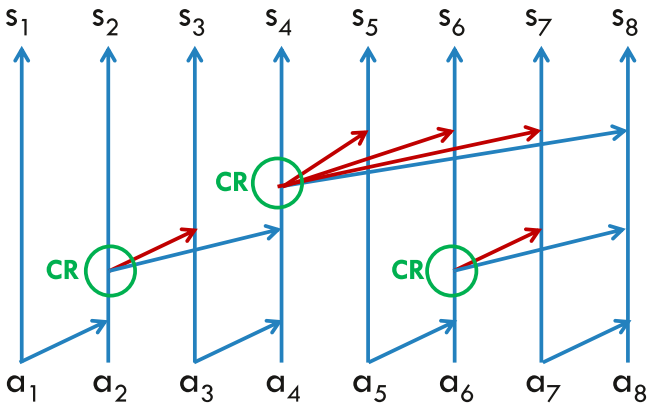


Figure 4: Prefix sum.

1.6 MM algorithm

The **Matrix Multiply (MM) algorithm** consists of three steps:

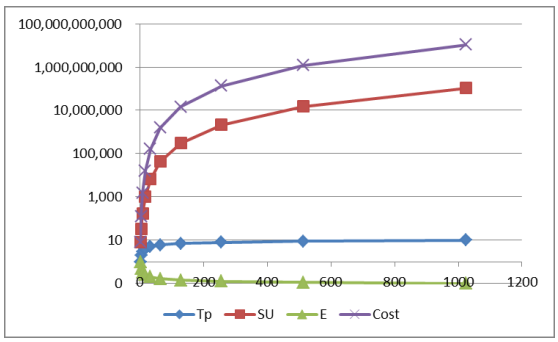
1. **Compute the two matrices** $A_{i,l}$ and $B_{l,j}$, so use the concurrent read.
2. Make the **sum**.
3. **Store** the result using exclusive write.

```
1 BEGIN
2     T_{i,j,l} = A_{i,l} B_{l,j}
3     FOR = H = 1 : K
4         IF l ≤ n ÷ 2^h THEN
5             T_{i,j,l} = T_{i,j,2l-1} + T_{i,j,2l}
6     IF l = 1 THEN
7         C_{i,j} = T_{i,j,1}
8 END
```

Algorithm 3: Matrix Multiply (MM)

Performance of MM

- $T_1 = n^3$
- $T_{p=n^3} = \log n$
- $\text{SU} = \dfrac{n^3}{\log n}$
- $\text{Cost} = n^3 \log n$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{1}{\log n}$



1.7 PRAM variants and Lemmas

The PRAM model presented here is one of the most commonly used. However, there are other important variants:

- PRAM model with a **limited number of shared memory cells** (small memory PRAM). If the input data set exceeds the capacity of the shared memory, the I/O values can be evenly distributed among the processors.
- PRAM model with **limited number of processors** (small PRAM). If the number of execution threads is higher, processors can interleave multiple threads.
- PRAM model with **limited size of one machine word**.
- PRAM model with **access conflicts handling**. These are restrictions on simultaneous access to shared memory cells.

Lemma 1. *Assume $P' < P$ and same size of shared memory. Any problem that can be solved for a P processor PRAM in T steps can be solved in a P' processor PRAM in:*

$$T' = O\left(\frac{TP}{P'}\right) \tag{8}$$

Proof. Partition P is simulated processors into P' groups of size $\frac{P}{P'}$ each. Associate each of the P' simulating processors with one of these groups. Each of the simulating processors simulates one step of its group of processors by:

- Executing all their read and local computation substeps first;
- Executing their write substeps then.

QED

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

Proof. Partition M simulated shared memory cells into M' continuous segments S, of size $\frac{M}{M'}$ each. Each simulating processor P'_I $(1 \leq I \leq P)$, will simulate processor P_I of the original PRAM. Each simulating processor P'_I $(1 \leq I \leq M')$, stores the initial contents of S_I into its local memory and will use $M'[I]$ as an auxiliary memory cell for simulation of accesses to cell of S_I.
Simulation of one original read operation:

```
1 EACH P'_I (I = 1, ..., max(P, M')) REPEATS FOR K = 1, ..., M/M'
2     WRITE THE VALUE OF THE K-TH CELL OF S_I INTO M'[I] (I = 1, ..., M')
3     READ THE VALUE WHICH THE SIMULATED PROCESSOR P_I (I = 1, ..., P)
      WOULD READ IN THIS SIMULATED SUBSTEP, IF IT APPEARED IN THE
      SHARED MEMORY
```

The local computation substep of P_I $(I = 1, \ldots, P)$ is simulated in one step by P'_I. SImulation of one original write operation is analogous to that of read.
QED

1.8 PRAM implementation

The PRAM is an ideal model for creating parallel algorithms. Now we look at "*is it really implementable?*" The short answer is yes.

The longest answer is the following. There are already some examples of PRAM being converted to real machine models, such as Explicit Multi-Threading (XMT), Rigel, Tilera, etc. If conversion is not easy or possible, the implementation can be "*direct*":

- The concurrent read is implemented as a detect-and-multicast technique.
- The concurrent write is implemented depending on the end result we want to achieve. Fetch-and-operate and prefix-sum are examples of serialized writing; otherwise, the CRCW technique is used:
  - Common CRCW: detect and merge
  - Priority CRCW: detect-and-priorities
  - Arbitrary CRCW: arbitrary

Example 2: Boolean DNF (sum of products) common CRCW

A logical formula is considered to be in DNF if it is a disjunction of one or more conjunctions of one or more literals.
Consider X as the sum of products of AND/OR operations:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM code, with X initialized to 0 and task index equal to \$, is:

```
if (a$b$) X = 1;
```

The common result is that not all processors write X and those that do write 1. The time complexity is $O(1)$. It works on common, priority and arbitrary CRCW.

Despite the previous example, exists also the PRAM SoP for the concurrent write. Let boolean X as:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM algorithm is:

```
if (aibi) X = 1;
```

Where all cores which write into X, **write the same value**.

✔ PRAM advantages

- Large body of algorithms.
- Easy to think about it.
- Sync version of shared memory. It eliminates sync and common issues, allows focus on algorithms, but allows adding these issues and allows conversion to async versions.
- Exists architectures for both synch (PRAM) and async (SM) model.
- PRAM algorithms can be mapped to other models.

1.9 Amdahl's and Gustafson's Laws

The **Amdahl's Law** is a formula which gives the **theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved**. The law can be stated as:

Definition 2: Amdahl's Law

The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

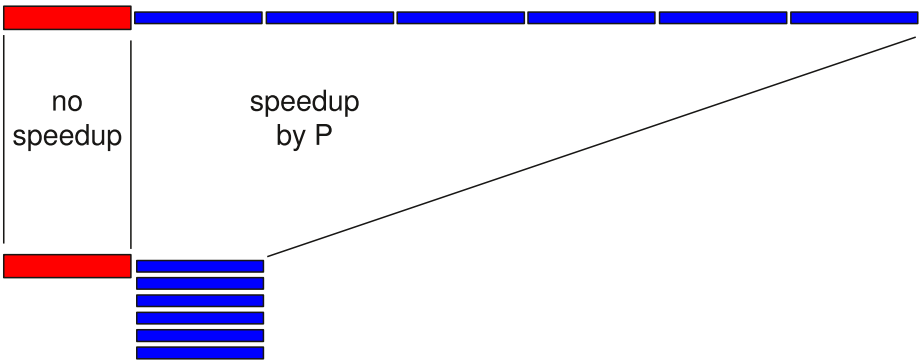
In practice, Amdahl's law says that the computation consists of interleaved segments of two types:

1. **Serial segments** (which cannot be parallelized);
2. **Parallelizable segments**.

Therefore, the metrics we can obtain are the time on P processors metric, that it is greater than the fraction of time on a processor divided by the processors P, and the speedup metric, that it is less than the number of processors P:

$$T_P > \frac{T_1}{P} \qquad SU < P$$

Graphically, we can see a fixed part of the line, which is the **serial segment** (no speedup), and a set of instructions that can be **parallelized** (the sum of these segments is equal to the unit time 1).



Furthermore, if we identify the parallelizable segment as f and the serial segment as $1 - f$, we obtain the following expressions:

$$\begin{aligned} SU\,(P, f) &= \frac{T_1}{T_P} = \frac{T_1}{T_1 \cdot (1-f) + \frac{T_1 \cdot f}{P}} = \frac{1}{(1-f) + \frac{f}{P}} \\ \lim_{P \to \infty} SU\,(P, f) &= \frac{1}{1-f} \end{aligned} \tag{9}$$

In the following figure we can see the speedup with parameter f. Note the pessimism: for a problem with inherent $f = 90\%$, there is no point in using more than 10 processors.

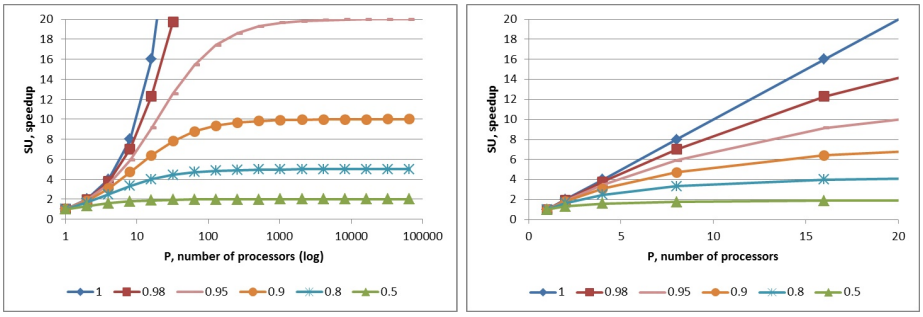


Figure 5: Amdahl's law, $SU(P)$, parameter f.

The original paper presenting Amdahl's Law [1] can be viewed by clicking on the link below or by scanning the QR code.

Amdahl's Law

Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, **Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance**. [8]

Gustafson's Law gives the speedup in the execution time of a task that theoretically gains from parallel computing, using a hypothetical run of the task on a single-core machine as the baseline. To put it another way, it is the **theoretical "slowdown" of an already parallelized task if running on a serial machine**.

Against Amdahl's law, Gustafson suggests the following ideas:

- Portion f is not fixed;
- The absolute serial time is fixed;
- Parallel problem size is increased to exploit more processors;
- Fixed serial time (s of total) and fixed parallel time ($1-s$ of total) are invariants;
- **Fixed time model** and not fixed size model (as Amdahl's law):

$$SU(P) = \frac{T_1}{T_P} = \frac{s + P \cdot (1-s)}{s + (1-s)} = s + P \cdot (1-s) \tag{10}$$

Gustafson's law suggests a **linear speedup** and is **empirically applicable to highly parallel algorithms**.

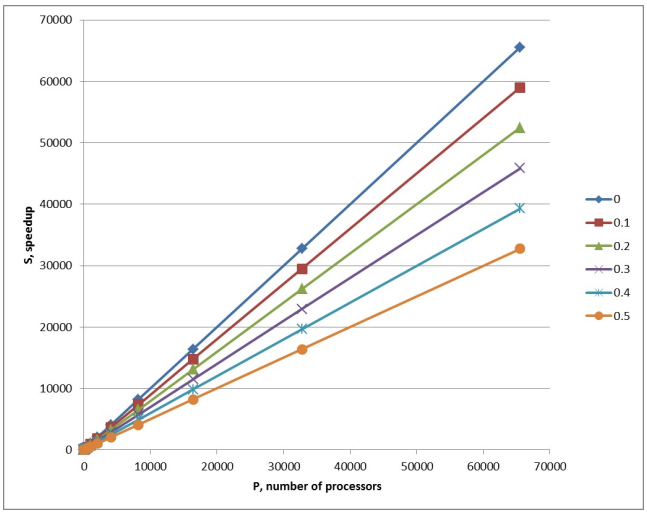


Figure 6: Gustafson's law.

Amdahl's Law states that as computing power increases, computational requirements remain the same. In other words, the **analysis of the same data will take less time with more computing power**.

Gustafson, on the other hand, argues that **more computing power leads to more careful and complete analysis of the data**. Where it would not have been possible or practical to simulate the impact of nuclear denotation on every building, car, and their contents (including furniture, structural strength, etc.) because such a calculation would have taken more time than was available to provide an answer, the increase in computing power will prompt researchers to add more data to more fully simulate more variables, giving a more accurate result.

The original paper presenting Gustafson's Law [4] can be viewed by clicking on the link below or by scanning the QR code.

Gustafson's Law

2 Fundamentals of architecture

The main purpose of this chapter is to introduce some basics of parallel computing theory. It will introduce the simplest and trivial processor and the more complex and efficient variants. The topics introduced are explained in a simple way and without any deepening, because it is only an introduction. For those who have studied computer science, this chapter might be a little boring and you might notice that some topics are explained in a simplistic way.

2.1 Introduction

2.1.1 Simplest processor

Inside a computer, a processor executes instructions.

- **Fetch/Decode**: Determine which instruction to run next;
- **ALU** (execution unit): Performs the operation described by an instruction, which may change values in the processor's registers or the computer's memory;
- **Registers**: maintain program state, store values of variables used as inputs and outputs to operations.

The simplest and most basic processor executes **one instruction per clock cycle**.

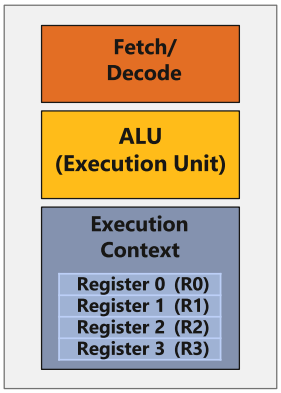


Figure 7: The simplest and most basic processor.

2.1.2 Superscalar processor

A more "complex" and realistic model is the **superscalar processor**. This **processor can decode and execute up to two instructions per clock**. The execution is slightly different from the simplest processor. The **processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units**.

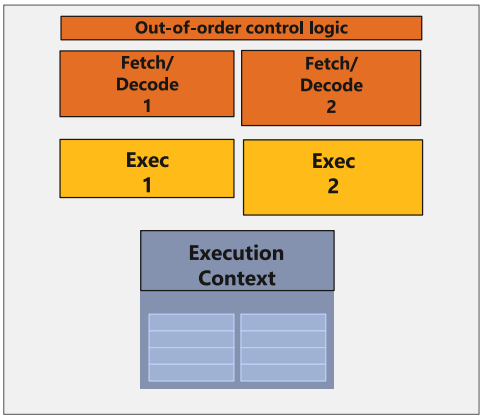


Figure 8: The superscalar processor.

The superscalar processor takes advantage of **Instruction-Level Parallelism (ILP)**[1] within an instruction stream.

- Processing **different instructions** from the same instruction stream **in parallel** (**within a core**).
- **Parallelism is automatically detected by the hardware during execution**.

[1] Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program. More specifically ILP refers to the average number of instructions run per step of this parallel execution.

2.1.3 Single Instruction, Multiple Data (SIMD) processor

Adding execution units (ALUs) to the simplest processor can increase compute capability. Amortize the cost/complexity of managing an instruction stream across many ALUs using **Single Instruction, Multiple Data (SIMD)** processing. Therefore, the **same instruction is sent to all ALUs**. This **operation is performed in parallel on all ALUs**.

✔ **Advantages**

- **Efficient for data-parallel workloads**: amortize control costs over many ALUs.
- Vectorization done by:
  - Compiler (**explicit SIMD**): parallelism is explicitly requested by the programmer through intrinsics, conveyed through parallel language semantics, and inferred through loop dependency analysis by the "auto-vectorizing" compiler. In other words, the **SIMD parallelization is done at compile time, and when we inspect the program binary, we can see the SIMD instructions**.
  - At runtime by hardware (**implicit SIMD**): the **compiler generates a binary with scalar instructions**, but n instances of the program are always executed together on the processor. The **hardware** (not the compiler) is **responsible for the simultaneous execution of the same instruction by multiple program instances on different data on SIMD ALUs**.

2.1.4 Multi-Core Processor

A **Multi-Core Processor (MCP)** is a **microprocessor** on a single integrated circuit (IC) with **two or more separate central processing units (CPUs)**, called *cores* to emphasize their multiplicity (e.g., *dual-core* or *quad-core*). Each core reads and executes program instructions, specifically ordinary CPU instructions (such as `add`, `move data`, and `branch`). However, the MCP can **execute instructions on separate cores simultaneously**, **increasing overall speed for programs that support multithreading or other parallel computing techniques**.

✔ **Advantages**

- Provides **thread-level parallelism**: execute a completely different instruction stream on each core simultaneously.
- **Software creates threads to expose parallelism to hardware** (e.g., via threading API)

2.2 Accessing Memory

2.2.1 What is a memory?

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

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Table 1: Example illustration of the program's memory address space of 32 bytes, range from `0x0` to `0x1F`.

From the processor's point of view, loading an instruction to access the contents present in memory is done with the `ld` assembly instruction. For example, `ld R0 ← mem[R2]` means "take the value from register `R2` and put that value into register `R0`".

Before we introduce new concepts, let us take a moment to explain some important **terminology**:

- **Memory Access Latency**, is the **time** it takes for the **memory system to deliver data to the processor**.
- **Processor Stall**. A **processor** stalls when it **cannot execute the next instruction in an instruction stream because of a dependency on a previous instruction that has not been completed**. Accessing memory is a major source of stalling, which is one of the main reasons why memory accesses should be limited.

For **example**, in the following three assembler instructions, the add has to wait for the loading of `R2` and `R3` values, making parallelization more complicated:

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

- **Memory Bandwidth**, is the **rate at which the memory system can provide data to a processor**.

  Bandwidth is the critical resource in modern computing.

  High-performance parallel programs will:

  1. **Organize computation to fetch data from memory less frequently**. For example, reuse data previously loaded by the same thread (temporal locality optimizations) or share data across threads (inter-thread cooperation);
  2. Prefer to **perform additional arithmetic to store/reload values**;
  3. **Programs need to access memory infrequently** to take advantage of modern processors.

2.2.2 How to reduce processor stalls

2.2.2.1 Cache

One of the most common solutions is caching.

A **cache** is a hardware or software **component that stores data so that future requests for that data can be served faster**; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

- A **Cache Hit** occurs when the **requested data is found** in a cache;
- A **Cache Miss** occurs when **it cannot**.

Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store, so the **more requests that can be served from the cache, the faster the system performs**. [11]

Many modern CPUs have logic that predicts what data will be accessed in the future and "pre-fetches" that data into caches. **Prefetching** reduces stalls because the data is resident in the cache when it is accessed. But beware, the other side of the coin is that if the **guess is wrong**, the **performance is worse than the system without prefetching**!

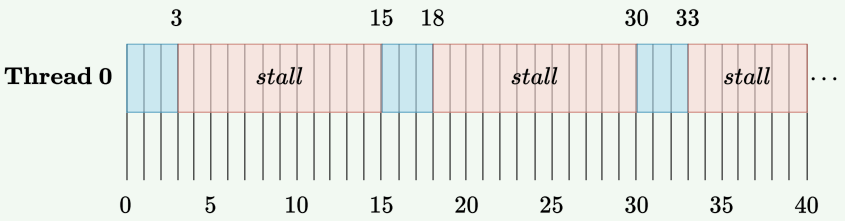
2.2.2.2 Multi-threading

A **Multithreaded Processor** is one that has the **ability to follow multiple instruction streams without software intervention**. In practice, then, this **includes any machine that stores multiple program counters (PCs) in hardware within the processor** (i.e., on chip, for microprocessor-era machines). [10]

The main idea in this architecture is to **interleave processing of multiple threads on the same core to hide stalls**. In other words, if we can't make progress on the current thread, we work on another one.

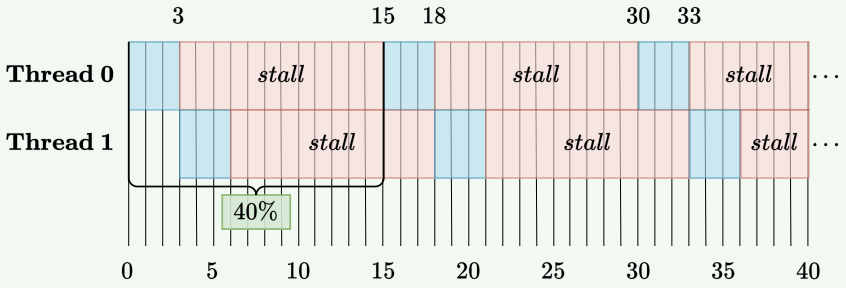
Example 1: core utilization

To better understand our explanation, suppose we are running a program where threads perform **three arithmetic instructions followed by a memory load** (with 12 cycle latency).

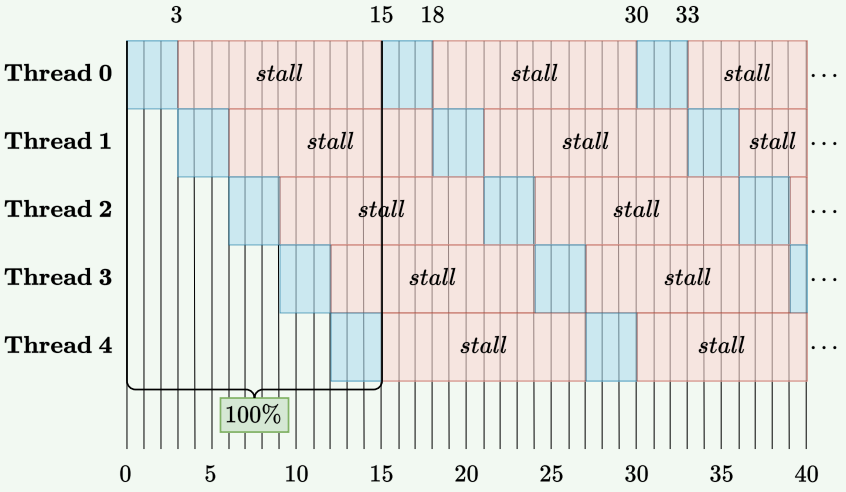


From the figure, it is clear that if we consider an arithmetic instruction and a memory stall, the core is not fully optimized to work at 100%. In

practice, we see that a single arithmetic instruction takes 3 clock cycles and the memory stall takes 12 clock cycles. This means that from this situation we are using the CPU at only 20% (3 work clock cycles on 15)! Without suggesting the final solution, try to see what happens when we add another thread.



We have gained three clock cycles, and now we are taking advantage of the 40% of the core.
Now, how many threads do we need to achieve 100% utilization? The answer is simple: the number of clock cycles of the operations to be done before the stall plus the clock cycles of the stall divided by the working operations (operations that are not stalls). In our case: $15 \div 3 = 5$.



Note that if we add more threads, there will be no benefit because the CPU is already at 100%.

✔ Multithreaded Processor benefits

- A processor with multiple hardware threads has the **ability to avoid stalls** by executing instructions from other threads when one thread must wait for a long latency operation to complete. The latency of the memory operation is not changed by multithreading, it just no longer causes reduced processor utilization.
- A **multithreaded processor hides memory latency by performing**

arithmetic from other threads. Program that feature more arithmetic per memory access need fewer threads to hide memory stalls.

Type of hardware-supported multithreading

- **Core manages execution contexts for multiple threads**. This type still has the same number of ALU resources: multi-threading only helps to use them more efficiently in the face of high latency operations such as memory access. The **processor decides which thread to run each clock cycle**.
- **Coarse-Grain Multithreading**, also called **Block Multithreading** or **Switch-On-Event Multithreading**, has multiple hardware contexts associated with each processor core. A hardware context is the program counter, register file, and other data required to enabled a software thread to execute on a core. However, only one hardware context has access to the pipeline at a time. [10]
- **Fine-Grain Multithreading (FGMT)**, also known as **Interleaved Multithreading** or **Temporal Multithreading**, is the type just described on the previous pages, as in the example on page 27.
- **Simultaneous Multithreading (SMT)** has multiple hardware contexts associated with each core. In a simultaneous multithreaded processor, instructions from multiple threads are available to be issued on any cycle. Therefore, all hardware contexts, and in particular all register files, must be accessible to the pipeline and its execution resources. [10]

  In other words, **each clock**, the **core selects instructions from multiple threads to execute on ALUs**.

3 Programming models

3.1 Implicit SPMD Program Compiler (ISPC)

Before introducing the ISPC compiler, we give the definition of SPMD.

Definition 1: Single Program, Multiple Data (SPMD)

Single Program, Multiple Data (SPMD) is a term that has been used to refer to computational models for exploiting parallelism, where **multiple processors work together to execute a program to achieve faster results**.
The **difference** between *SPMD* and *SIMD* (page 24) is that in SPMD parallel execution, **multiple autonomous processors simultaneously execute the same program at independent points**, rather than in SIMD it is vectorization at the instruction level so that **each CPU instruction processes multiple data elements**.
In other words:

- SPMD: is the **programming abstraction**, because the programmer has to think; the program is written in terms of this abstraction.
- SIMD: in general, the compilers (ISPC) issue special vector instructions that execute the logic performed by each parallel instance created (ISPC gang spawned). In addition, the compilers handle the mapping of conditional control flow to vector instructions.

The difference and the terminology used by ISPC will become clearer in the following pages. We suggest that finish this section and come back here in a moment.

Definition 2: Implicit SPMD Program Compiler (ISPC)

Implicit SPMD Program Compiler (ISPC) is a **compiler for a variant of the C programming language**, with extensions for *Single Program, Multiple Data (SPMD)* programming. Under the SPMD model, the programmer writes a program that generally appears to be a regular serial program, though the execution model is actually that a number of program instances execute in parallel on the hardware. In other words, the **ISPC gives the programmer some API to do parallelization on the code; it also generates high quality SIMD code to increase performance**.

The definition, implementation, and other details are explained in the official Intel GitHub repository.

How it works?

Let us take a general main program; when we call an `ispc` function, it causes a **spawn of gang of ISPC program instances upon return, all instances have completed**. These **instances execute the same ISPC code simultaneously**, and **each instance has its own copy of local variables**. Take the following ISPC code as an example:

```
1  export void ispc_sinx(
2      uniform int N,
3      uniform int terms,
4      uniform float* x,
5      uniform float* result
6  ){
7      // assume N % programCount = 0
8      for (uniform int i=0; i<N; i+=programCount) {
9          int idx = i + programIndex;
10         float value = x[idx];
11         float numer = x[idx] * x[idx] * x[idx];
12         uniform int denom = 6; // 3!
13         uniform int sign = -1;
14         for (uniform int j=1; j<=terms; j++) {
15             value += sign * numer / denom
16             numer *= x[idx] * x[idx];
17             denom *= (2*j+2) * (2*j+3);
18             sign *= -1;
19         }
20         result[idx] = value;
21     }
22 }
```

In the example, the `programCount` (row 8) and `programIndex` (row 9) variables, `uniform` (row 2, and so on) data type tell us:

- `programIndex` gives the index of the SIMD-lane being used for running each program instance (in other words, it's a varying **integer value that has value zero for the first program instance, and so forth**).
- `programCount` gives the **total number of instances in the *gang***.
- A variable that is declared with the `uniform` qualifier represents a **single value that is shared across the entire *gang***.

Together, these can be used to uniquely map executing program instances to input data (programIndex and programCount, uniform data type).

With the ISPC analogy, the **SPMD programming model** should be clear:

1. **Single thread of control** (typically a main program);
2. **Invoke** the **SPMD function** (in the previous example, the `ispc_sinx` function);
3. **SPMD execution**, then **multiple instances of the function run in parallel** (multiple logical threads of control);
4. **Returns** and **resumes a single thread** of control.

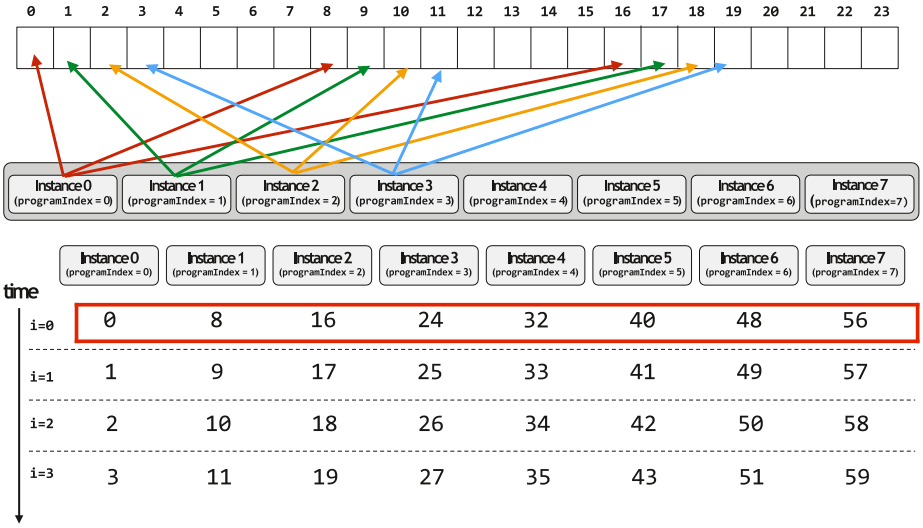


Figure 9: Example of execution with 8 instances (`programCount` equal to 8). For all program instances, there are eight non-contiguous values in memory. A special instruction called `gather` is needed to implement this, but unfortunately it is a more complex and expensive SIMD instruction rather than a contiguous implementation.

Figure 9 shows a possible execution of the ISPC function using 8 instances. The result is obtained and all is well. But there is one interesting observation. **Each ISPC instance writes each value in a non-contiguous way**. This can be done better:

```
export void ispc_sinx_v2(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result
){
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;
    for (uniform int i=0; i<count; i++) {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

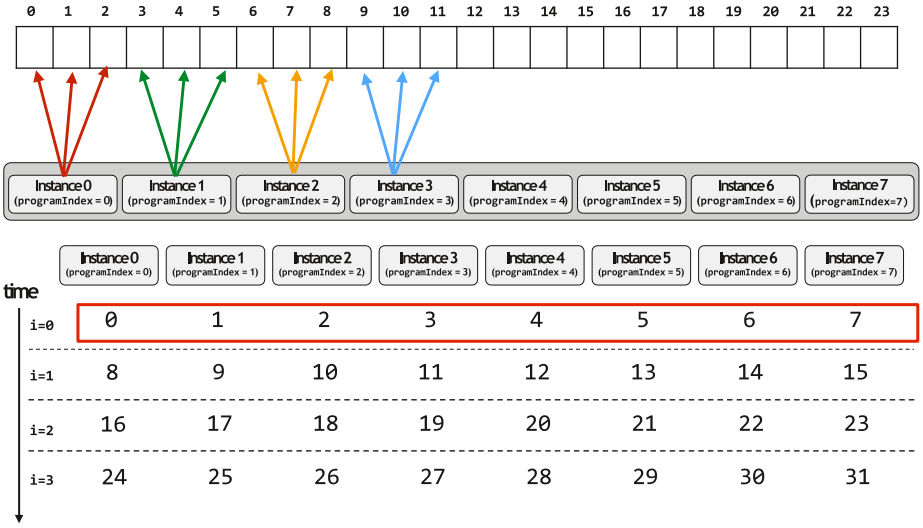


Figure 10: Example of execution with 8 instances (`programCount` equal to 8). A single "packed vector load" instruction efficiently implements this. For all program instances, since the eight values are contiguous in memory.

3.2 Shared Address Space Model

We give a general introduction to memory in the chapter 2.2. In parallel computing theory, **each thread communicates with other threads using read-/write operations**. These instructions operate on a special area called the **Shared Address Space** (also called **Shared Variables**).

Definition 3: Shared Address Space

The **Shared Address Space** view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. [6]

Now the first and trivial question should be: a powerful tool is the possibility to allow communication between threads, but *how can we guarantee that two or more threads accessing the same resource do not create well known problems, such as* race condition*?*

This property, commonly called **mutual exclusion** or **atomic operation**, can be guaranteed with some techniques:

- **Lock/Unlock mutex around a critical section**:

```
Lock lock_variable;

// some operations, such as spawn of threads

lock_variable.lock();
// critical section
lock_variable.unlock();
```

- Some languages have first-class **support for atomicity of code blocks**:

```
atomic {
    // critical section
}
```

- Intrinsics for **hardware-supported atomic read-modify-write operations**:

```
atomicAdd(x, 10);
```

The shared address space **requires hardware support to be efficiently implemented**. The main idea is that **each processor can directly reference the contents of any memory location**. Some interesting examples that can be explored in depth are: SUN Niagara 2, Knights landing (KNL): 2nd Generation Intel Xeon Phi processor.

3.3 Message Passing model of communication

In parallel computing, the **message passing model** **allows** threads (or processes) to **communicate by sending and receiving messages**. This model is used to **facilitate data exchange between threads running in their own private address spaces**.

Each thread operates within its private address space, meaning they **do not share memory directly**. When they need to communicate with a specific thread without using a *shared address space model*, they use the *message passing model* to send (or receive) the data.

- Usually, the **sender** specifies
  - the recipient;
  - the buffer to be transmitted;
  - an optional message identifier (a sort of tag).
- Meanwhile, the **receiver** specifies:
  - who the sender is;
  - the buffer where to store data;
  - an optional message identifier (again, a sort of tag).

The message passing model is the **only way to exchange data between threads** because it guarantees three main advantages:

- **Data Isolation**: Using message passing, **each thread maintains its address space**, reducing the risk of race conditions.
- **Scalability**: Message passing **scales well with distributed systems and multi-core architectures**, making it suitable for large-scale parallel computing.
- **Flexibility**: It **allows for explicit control over data exchange and synchronization, providing Flexibility in parallel program design**. Explicit control refers to the fact that in message passing, each communication action is explicitly defined by send and receive operations. This means we can precisely dictate which data is sent, when, and to whom it is sent. Furthermore, the synchronization is managed very well because message passing naturally incorporates synchronization. When a process sends a message, it can block until it is received, ensuring that the sender and receiver are synchronized.

Why message passing is preferable to the shared address space model

Unlike shared memory systems that require complex hardware mechanisms to implement system-wide load and store operations, message passing systems do not need this capability. They **only need to communicate messages between nodes**. It can be considered as a great advantage for message passing models, which for this reason is very much used in the supercomputers and clusters. Finally, this model has the ability to **connect commodity hardware systems together to form a large parallel machine**. This means we can use off-the-shelf components to build powerful computing clusters.

3.4 Data-Parallel model

In parallel computing, the **Data-Parallel model** is characterized by **applying the same operation simultaneously across multiple data points**. This model is particularly effective for tasks involving large datasets where the same computation needs to be performed on each data element.

This model organizes **computation as operations on sequences of elements** (e.g., perform the same function on all elements of a sequence). In programming languages, the basic data type used for this purpose is called `Sequence` (e.g., C++). Despite the name of the datatype used in the languages, the definition is that it **is an ordered collection of elements, where each element can be accessed and manipulated using various sequence operators**. The most common operators are:

- **Map**. The `map` function is a **higher-order function**[2] that **operates on sequences**. It applies a **side-effect-free unary function**[3] $f : a \to b$ **to all elements of an input sequence**, producing an **output sequence of the same length**.

  Example 1: Python Analogy

  For a better understanding, we provide a very simple Python code to see how a `map` function works. In Python there is a `map` function that does exactly what we say.

```python
# Define a trivial function that squares a number
def square(x):
    return x * x

# Create a sample list
numbers = [1, 2, 3, 4, 5]

# Use the map function to apply the 'square' function
# to each element in the 'numbers' list
squared_numbers = map(square, numbers)

# Convert the result to a list and print it
print(list(squared_numbers))
```

  And the result is:

```
[1, 4, 9, 16, 25]
```

  Now the main idea is: since the `map` is a function without side effects, we can **apply it to all elements of the sequence in any order without changing the output of the program**. This allows **reordering or parallel processing of sequence elements to optimize performance**.

[2] A **higher-order function** is a function that can do one or both of the following:
- Take other functions as arguments (parameters).
- Returns a function or value as its result.

[3] A function that takes only one argument and doesn't suffer from side effects.

- **Reduction**. The reduction born from the need to make parallel operations of iterations. For example, in a `for` loop, if we need a progressive sum of an element, we should implement some kind of mechanism to manage the synchronization. Using the reduction strategies, the compiler will do this job. We suggest reading the Reduction section (5.3.1.1, page 64) in the OpenMP chapter to understand what we mean.
- Others like **scan** and **shift**.

4 Parallel Programming Models and pthreads

4.1 How to create parallel algorithms and programs

Although *parallel algorithms* and *parallel programs* are in the same father set, the parallel computing topic, these two arguments are a little different.

Definition 1: Parallel Algorithms

A **parallel algorithm**, as opposed to a traditional serial algorithm, is an **algorithm which can do multiple operations in a given time**.

Definition 2: Parallel Programs

A **parallel program** is a **program that uses multiple CPU cores**, with **each core performing a task independently**.

However, designing *parallel algorithms* is not an easy task because there is no heuristic for designing *parallel algorithms*. There are some rules that help in the design. The same reasoning applies to *parallel programs*, because they depend on the chosen language and architecture.

Furthermore, there is no single correct solution, but several possible parallel solutions. A **good first approach is to start with machine-independent issues** (concurrency) and **delay target-specific issues as much as possible**.

Design a parallel algorithm	Design a parallel program
Understand the problem to be solved	Analyze the target architecture(s)
Analyze data dependencies	Choose the best parallel programming model and language
Partition the solution	Analyze the communications (cost, latency, bandwidth, visibility, synchronization, etc.)

Table 2: Design parallel algorithms and parallel programs.

The **PCAM (Partitioning, Communication, Agglomeration, Mapping)** methodology described by Argonne National Laboratory is intended to promote an exploratory **approach to design in which machine independent issues**, such as *concurrency*, are **considered early** and **machine specific aspects of design are deferred until late in the design process**. In other words, we immediately consider the machine-independent issues (e.g., concurrency) at the beginning of the design approach, and all machine-specific aspects are postponed to an advanced stage of the design process.

This methodology structures the design process into **four distinct stages**:

1. **Partitioning**. The **computation** that is to be performed and the **data** operated on by this computation are **decomposed into small tasks**. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

2. **Communication**. The communication required to **coordinate task execution** is determined, and appropriate **communication structures and algorithms are defined**.

3. **Agglomeration**. The task and **communication structures** defined in the first two stages of a design are **evaluated with respect to performance requirements and implementation costs**. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. **Mapping**. **Each task is assigned to a processor** in a manner that attempts to satisfy the competing goals of **maximizing processor utilization** and **minimizing communication costs**. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

In the **first two stages**, we focus on **concurrency** and **scalability** and seek to **discover algorithms with these qualities**. In the **third and fourth stages**, attention shifts to **locality** and **other performance-related issues**.

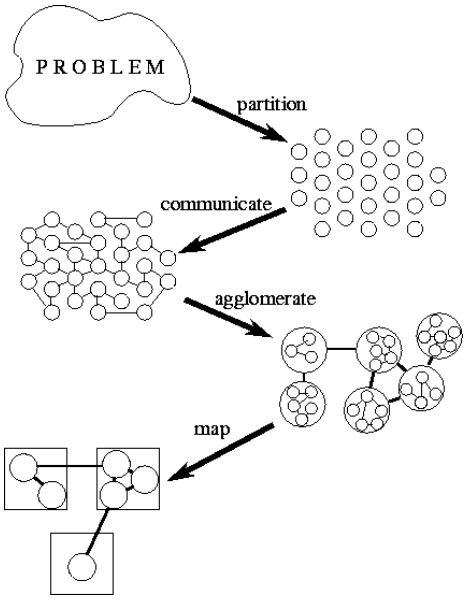


Figure 11: PCAM design methodology for parallel programs. Starting with a problem specification, develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.

4.2 Analyze parallel algorithms

Whether we want to analyze our parallel algorithm created with the PCAM model or evaluate a general parallel algorithm, we need some metrics.

The classical **metrics** needed to evaluate a parallel algorithm are:

- **Time complexity**: quantifies the amount of **time required to produce a solution**.
- **Resource complexity**: quantifies how many **resources are needed to produce the solution in that time**.

In general, to analyze a **parallel algorithm**, we can consider its **structure as a directed acyclic graph (DAG)**[4], where the nodes are the task and the edges are the data dependencies.

Parallel Algorithm Terminology and Metrics

- **Concurrent tasks**, each task is executed ***independently***.
- **Parallel tasks**, each task is executed at the ***same time*** (because multiple computing resources are available).
- **Work** W is the ***number of operations executed***. It may be higher than the sequential version of the algorithm due to communication overhead, etc.
- **Span** S is the ***longest chain of dependencies*** (i.e., the critical path) that determines the ***minimum time required to execute the algorithm***. This is a *lower bound* on the running time, regardless of the number of processors. The range indicates the ability of an algorithm to get better performance on more processors.

  #### How do we calculate the Span metric?

  1. As we just said, we **represent a parallel algorithm as a DAG** graph, where nodes represent tasks and edges represent dependencies between tasks;
  2. We **assign weights to each node** that represent the **time required to perform the corresponding task**;
  3. We try to **find the Critical Path**. In other words, we determine the path from the start node to the end node that has the *maximum cumulative weight*;
  4. Finally, the **sum** of the **weights of the nodes on the critical path** gives us the span value!

[4] A **Directed Acyclic Graph (DAG)** is a directed graph, i.e. with oriented edges, without cycles.

- **Parallelism** P is the ***measure of efficiency in the use of resources***. Trivially, it is the number of operations performed divided by the longest chain of dependencies:

$$P = \frac{W}{S} \tag{11}$$

It indicates **how many processors can be effectively used by the computation**. If the work is equal to the span, the parallelism is 1 and the computation is sequential. Ideally, but not necessarily, we win with polylogarithmic span, because if the work is $O(n \log n)$ and the span is $O(\log^2 n)$, then the parallelism is $O\left(\frac{n}{\log n}\right)$, which is actually quite high (and unlikely to be a bottleneck on most machines in the next 25 years). [2]

This measure is *one of the most important*. It indicates the **number of processors that are not idle**. It is obvious that a **good parallel algorithm is designed to have the lowest possible work** (less operation, then less resource usage, then less cost, and so on) **and the highest possible parallelism** (achievable by reducing the span, and this should be trivial, since the metric P is given by work divided by the span, so reducing the denominator, you can get a higher value).

As in all things, there is a **trade-off** between the lowest possible "work" and the highest possible "parallelism". Reducing the work too much could eliminate the possibility of parallelizing our algorithm, and on the other hand, reducing the span too aggressively could cause communication/synchronization overhead.

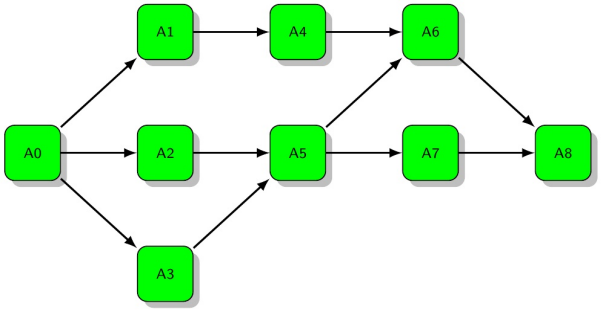


Figure 12: Example of DAG implementation with work equal to 9, span equal to 5, and parallelism equal to 1.8 ($9 \div 5$). The span calculus is not well known, has been calculated *a priori*.

Finally, we use a mathematical annotation and not only a graphical (DAG) annotation. The **Composition Rules** help determine how to combine smaller parallel tasks into a larger algorithm, while analyzing the Work and Span of the combined algorithm:

- **Single operation**. An operation takes 1 unit of work and 1 unit of span time.

$$W(op) = 1 \qquad S(op) = 1$$

- **Sequential Composition**.
  - The total work of executing e_1 and e_2 **sequentially** is the sum of their individual work.

$$W(e_1, e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 **sequentially** is the sum of their individual spans.

$$S(e_1, e_2) = S(e_1) + S(e_2)$$

- **Parallel Composition**.
  - The total work of executing e_1 and e_2 in **parallel** is still the sum of their individual works.

$$W(e_1||e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 in **parallel** is the ***maximum of their individual spans***, since they can be executed simultaneously.

$$S(e_1||e_2) = \max(W(e_1), W(e_2))$$

4.3 Technologies

Some famous architecture to work with parallel programming:

- **Verilog/VHDL** are *hardware description languages*. The target architectures are *ASIC and FPGA*. The **parallelism** and the **communication** are **explicit**.

  ✔ Pros

  - Complete control on computation and memory
  - No overhead introduced in the computation
  - Provides access to potentially large computational power

  ⚠ Cons

  - Requires specific hardware (e.g., ASIC or FPGA) to implement functionality
  - Difficult to learn: completely different programming language and programming paradigm
  - Depends on the chosen target architecture

- **MPI** is a *library*. The target architectures are *Multi CPUs*. The **parallelism** is **implicit** and the **communication** is **explicit**.

  ✔ Pros

  - Can be adopted on different types of architecture
  - Scalable solutions
  - Synchronization and data communication are explicitly managed

  ⚠ Cons

  - Communication can introduce significant overhead
  - Programming paradigm more difficult than shared memory-based ones
  - Standard does not reflect immediately advances in architecture characteristics

- **PThread** is a *library*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ✔ Pros

  - Can be adopted on different types of architecture
  - Explicit parallelism and full control over application

  ⚠ Cons

  - Task management overhead can be significant
  - Not easily scalable solutions
  - Low level API

- **OpenMP** is a *C/Fortran extensions*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ✔ Pros

  - Easy to learn
  - Scalable solution
  - Parallel applications can also be executed sequentially

  ⚠ Cons

  - Mainly focused on shared memory homogeneous systems
  - Requires small interaction between tasks

- **CUDA** is a *C extensions*. The target architectures are *CPU plus GPU(s)*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ✔ Pros

  - Provides access to the computational power of GPUs
  - Writing a CUDA kernel is quite easy
  - Already optimized libraries

  ⚠ Cons

  - Targets only NVIDIA GPUs
  - Difficult to extract massive parallelism from application
  - Difficult to optimize CUDA kernel

- **OpenCL** is a *C/C++ extensions and API*. The target architectures are *heterogeneous architecture*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ✔ Pros

  - Target-independent standard
  - Hides architecture details
  - Same programming infrastructure for every heterogeneous architecture: CPU + GPU (and FPGA)

  ⚠ Cons

  - Difficult programming paradigm for its heterogeneity
  - Hiding of architecture details makes difficult to obtain best performances
  - Gradually abandoned

- **Apache Spark** is an *API*. The target architectures are *multi CPUs*. The **parallelism** is **implicit** and the **communication** is **implicit**.

  ✔ Pros

  - API for different languages
  - Explicit parallelization and communication are not required
  - Preinstalled on cloud provide VMs

  ⚠ Cons

  - Suitable only for big data applications
  - Does not (yet) fully support GPUs

Regardless of these technologies, it is quite common to mix some of them:

- **OpenMP + CUDA**: allows to exploit multi-core CPU and GPU. CUDA is used to parallelize GPU code and OpenMP is used to parallelize CPU code.
- **MPI + OpenMP**: the most common scenario are:
  1. MPI used to express coarser parallelism (multi CPU) and OpenMP used to express finer parallelism (multi core).
  2. MPI used to implement communication and OpenMP used to parallelize computation.
- **OpenCL + Verilog or VHDL**: in principle, hardware kernels (implemented for example on FPGA) can be used as accelerators; OpenMP used to describe parallelism among different processing elements; Verilog/VHDL used to describe hardware kernel. An example of target: Intel Xeon Scalable.

4.4 Threads

4.4.1 Flynn's taxonomy

Flynn's taxonomy is a **classification of computer architectures**, proposed by Michael J. Flynn. The classification system has been used as a tool in the design of modern processors and their functionalities. Since the rise of multiprocessing central processing units (CPUs), a multiprogramming context has evolved as an extension of the classification system.

The four initial classifications defined by Flynn are based upon the number of concurrent instruction (or control) streams and data streams available in the architecture:

- **Single Instruction stream, Single Data stream (SISD)**
- **Single Instruction stream, Multiple Data streams (SIMD)**
- **Multiple Instruction stream, Single Data stream (MISD)**
- **Multiple Instruction stream, Multiple Data stream (MISD)**

It is important to quote it because it is the basis for the development of many advanced technologies.

4.4.2 Definition

A UNIX process can be created by the operating system and contains information about program resources and program execution status.

Definition 3: Thread

A **thread** is an **independent stream of instructions within a process**. Threads can be scheduled by the operating system, and each thread can run concurrently with other threads. A thread also has local resources and can access the shared process resources.

In other words, a thread can be thought of as any **procedure that runs independently of its main program**. We can create each thread dynamically during execution. A good point is that a multi-threaded program is lighter than a multi-process program.

When a thread exists within a process, it shares most of the process resources, for example:

- Changes made by one thread to 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.
- **Implicit communication** by reading and writing shared variables.
- **Reading and writing to the same memory locations requires explicit synchronization by the programmer**. If this rule is not followed, the code may suffer from a data race or race condition [5] problem.

The most common models for threaded programs are the manager / worker model[6] and pipeline.

This chapter introduces the POSIX threads model.

Definition 4: pthreads

POSIX Threads, commonly known as **pthreads**, is an **execution model** that exists independently from a programming language, as well as a parallel execution model. It **allows a program to control multiple different flows of work that overlap in time**. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

POSIX threads and OpenMP are two **implementations of a shared memory parallel programming model using threads**. The **programmer is responsible for handling parallelism and synchronization**, usually through a library of subroutines or a set of compiler directives. Typically, hardware vendors have implemented their own proprietary versions of threads, but in this course we will look at POSIX threads (pthreads) and OpenMP.

[5] In parallel computing, a **Data Race** or **Race Condition** is a software problem that occurs when two threads (or processes) access the same variables, and at least one does a write. They can finish in a different order than expected.

[6] The manager/worker pattern is described as follows. The idea is that the work that needs to be done can be divided by a "manager" into separate pieces and the pieces can be assigned to individual "worker" processes. Thus the manager executes a different algorithm from that of the workers, but all of the workers execute the same algorithm. Most implementations of MPI allow MPI processes to be running different programs (executable files), but it is often convenient (and in some cases required) to combine the manager and worker code into a single program.

4.4.3 pthreads API

In 1995, the IEEE POSIX 1003.1c standard specified the API for explicitly managing threads. An **API is a set of C language programming types and procedure calls**.

- **Header file** to include in the main file: `pthread.h`.
- To **compile** and use it, it is necessary to add the **flag** `-pthread` to the gcc (or g++) options.

The API are divided by what we want to do. In general, there are two sets: thread management and thread synchronization.

- Thread Management
  - Creation (page 48)
  - Termination (page 49)
  - Joining (page 50)
  - Detaching (page 51)
  - Joining through Barriers (page 52)
- Thread Synchronization
  - Mutexes (page 53)
  - Condition variables (page 53)

4.4.3.1 Creation

Once threads are created, they are peers, and **may create other threads**. There is **no implied hierarchy or dependency between threads**. The **maximum number of threads depends on the implementation**. Doc.

pthread API: `pthread_create`

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

- **Return value**: on success, `pthread_create()` returns 0; on error, it returns an error number, and the contents of `*thread` are undefined.
- **Arguments**:
  - `thread`: identifier for the new thread returned by the subroutine.
  - `attr`: used to set thread attributes, such as joinable, detached, scheduling and stack size.

- `start_routine`: the C routine that the thread will execute once it is created.
- `arg`: argument passed to `start_routine`. It must be passed by address as a pointer cast of type void.

4.4.3.2 Termination

The thread returns from its startup routine when its "life" ends. The thread makes a call to the `pthread_exit` subroutine. Doc.

pthread API: `pthread_exit`

```
1 void pthread_exit(void *retval)
```

- **Return value**: this function does not return to the caller.
- **Arguments**:
  - `retval`: function terminates the calling thread and returns a value via `retval`.

The thread is canceled by another thread via the `pthread_cancel` routine. Doc.

pthread API: `pthread_cancel`

```
1 int pthread_cancel(pthread_t thread)
```

- **Return value**: on success, `pthread_cancel()` returns 0; on error, it returns a nonzero error number.
- **Arguments**:
  - `thread`: the `pthread_cancel()` function sends a cancellation request to the thread `thread`.

4.4.3.3 Joining

The join function **blocks the calling thread until the specified thread exits**. Doc.

pthread API: `pthread_join`

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

- **Return value**: on success, `pthread_join()` returns 0; on error, it returns an error number.
- **Arguments**:
  - `thread`: the `pthread_join()` function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by `thread` must be joinable.

    If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is canceled, then the target thread will remain joinable (i.e., it will not be detached).
  - `retval`: if `retval` is not `NULL`, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit()`) into the location pointed to by `retval`. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in the location pointed to by `retval`.

4.4.3.4 Detaching

The detach function **marks a thread as detached**. When a thread is detached, its **resources are automatically released back to the system when the thread terminates**, without the need for another thread to join with it.

Why would I need to detach a thread and not join it?

Good question. The answer depends on what we have to do.

- **Fire and forget tasks**. When we start a thread to perform a task that doesn't require further interaction or result processing, releasing it ensures that the resources are automatically cleaned up when the task is complete.
- **Resource management**. Detaching avoids the need for another thread to call `pthread_join()`, which can save system resources and reduce the complexity of our code. It's especially useful in a highly concurrent application with many short-lived threads.
- **Avoid deadlocks**. When we have potential circular dependencies or complex synchronization between threads, detaching threads can help avoid deadlocks by eliminating the need for one thread to wait on another.
- **Long-running background tasks**. For tasks that should run independently in the background and not block the main program flow, detaching makes sense. We make sure they clean up after themselves without having to explicitly manage their lifecycle.

Doc.

pthread API: `pthread_detach`

```
1 int pthread_detach(pthread_t thread)
```

- **Return value**: on success, `pthread_detach()` returns 0; on error, it returns an error number.
- **Arguments**:
  - `thread`: the `pthread_detach()` function marks the thread identified by `thread` as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

    Attempting to detach an already detached thread results in unspecified behavior.

4.4.3.5 Joining through Barriers

The *barrier init* function **initializes a barrier object**, and the *barrier wait* function **blocks a thread until the specified number of threads have called it**. A **barrier object** is, in parallel computing, a synchronization tool that ensures that multiple threads reach a certain point of execution before any of them continue. It's like a **checkpoint that everyone must reach before continuing**, ensuring coordinated progress in a parallel algorithm.[7] Doc.

pthread API: `pthread_barrier_init`

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

pthread API: `pthread_barrier_wait`

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

- **Return value**: on success, function return 0; on error, they return an error number.
- **Arguments**: the main and most important argument is `count`, which specifies the **number of threads to wait** for.

[7] For example, imagine multiple threads working on different parts of a matrix. A barrier can ensure that all threads finish their part of the computation before moving on to the next phase, such as combining results or performing subsequent operations.

4.4.3.6 Mutexes

Mutex variables are the basic **method of protecting shared data when multiple writes occur**. Only **one thread can lock a mutex variable at a time**. If multiple threads attempt to lock a mutex, only one thread will succeed. **Threads that fail to acquire the mutex are blocked**. There is also the `trylock` function, which returns immediately if the mutex is currently locked (by any thread, including the current thread). Note that a **lock function has the potential to create a deadlock situation**.

There are also **three types of mutex** that can be set using the `settype` function:

- **Normal Mutex** (`PTHREAD_MUTEX_NORMAL`). A **normal mutex does not check for errors such as deadlock**. If a thread tries to lock a mutex it already owns, the thread will deadlock.
- **Error Check Mutex** (`PTHREAD_MUTEX_ERRORCHECK`). Provides **error checking**. If a thread tries to lock a mutex it already owns, `lock` function will return an error instead of deadlocking.
- **Recursive Mutex** (`PTHREAD_MUTEX_RECURSIVE`). Allows the same thread to lock the mutex multiple times without deadlocking. Each lock must have a corresponding unlock.

Doc.

pthread API: `pthread_mutex_lock`

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex)
```

pthread API: `pthread_mutex_trylock`

```
1 int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

pthread API: `pthread_mutex_unlock`

```
1 int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

4.4.3.7 Condition variables

Mutexes implement synchronization by serializing data accesses. Condition variables allow threads to synchronize explicitly by signaling the meeting of a condition. Without condition variables, the programmer would need to poll to check if the condition is met.

5 OpenMP v5.2

5.1 Introduction

OpenMP is a scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran.

- **Header file** to include in the main file: `omp.h`.
- To `compile` and use it, it is necessary to add the `flag -fopenmp` to the gcc (or c++) options.

The following link contains about 200 slides on OpenMP. It is an introduction, but it covers every argument required in this PoliMI course. The slides are made by a Senior Principal Engineer, Mattson Tim. He's a Senior Principal Engineer at Intel, where he's been since 1993. The slide link.

✔ Benefits

- Standard across a variety of shared memory architectures and platforms.
- Supports three famous languages: Fortran, C and C++.
- Scalable from embedded systems to the supercomputer.
- The directives are intuitive, and with a limited set of directives we can implement parallel algorithms.
- Incremental parallelization of serial program.
- Coarse-grained and fine-grained parallelism. See here here an interesting difference between coarse-grained and fine-grained architecture:

? How it works?

OpenMP is based on the **fork-join paradigm**. A "master" thread forks a specified number of "slave" threads. Tasks are divided among the "slaves", and each "slave" runs concurrently as the runtime allocates threads to different processors.

1. `Thread #0` **born**. OpenMP programs start with a single thread;
2. **Fork**. At the start of a parallel region, the *master* creates a team of parallel worker threads (*slaves*).
3. **OpenMP code block**. Statements in the parallel block are executed in parallel by each thread.
4. **Join**. At the end of the parallel region, all threads synchronize (implicit barrier, see definition on page 52) and join the master thread. [5]

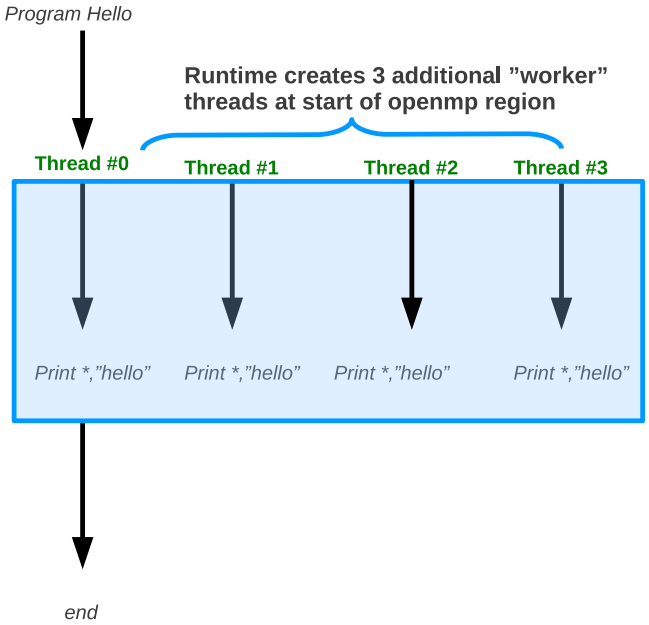


Figure 13: Example of an OpenMP program called `Hello`. At runtime, the *master* thread forks 3 additional *slave* threads to print `hello`. The example is very trivial, but here is a graphical representation of the workflow. An interesting thing to note is that **OpenMP creates** a sort of **region where each thread executes all the instructions in the OpenMP block**. This is important to understand. [5]

5.2 Basic syntax

We can manage OpenMP work flow using the directive syntax. We remember that the reference guide of OpenMP is available on their website:

Reference Guide 

A **directive** is a combination of the base-language mechanism and a *directive-specification* (the *directive-name* followed by *optional clauses*). A construct consists of a directive and, often, additional base language code. In C++ directives are formed from either pragmas or attributes.

OpenMP: `pragma omp`

```
#pragma omp directive-specification
```

The **number of OpenMP threads can be set** using:

- At compilation time: using the environment variable `OMP_NUM_THREADS`
- At runtime: using the function

  OpenMP: `omp_set_num_threads`

  ```
  void omp_set_num_threads(int num_threads)
  ```

Other useful function to get information about threads:

- The **number of threads in the current team**:

  OpenMP: `omp_get_num_threads`

  ```
  int omp_get_num_threads()
  ```

  The binding region for an `omp_get_num_threads` region is the innermost enclosing `parallel` region. If called from the sequential part of a program, this routine returns 1.

- The **upper bound on the number of threads** that could be used to form a new team if a `parallel` construct without a `num_threads` clause were encountered after execution returns from this routine.

OpenMP: `omp_get_max_threads`

```
int omp_get_max_threads()
```

- The **thread number of the calling thread**, within the current team.

OpenMP: `omp_get_thread_num`

```
int omp_get_thread_num()
```

- The **elapsed wall clock time in seconds**.

OpenMP: `omp_get_wtime`

```
double omp_get_wtime()
```

- The **precision of the timer (seconds between ticks)** used by `omp_get_wtick`.

OpenMP: `omp_get_wtick`

```
double omp_get_wtick()
```

OpenMP programs execute serially until they reach a `parallel` directive. As we have explained at page 55, the thread that was executing the code spawns a group of "slave" threads and becomes the "master" (thread ID 0). The code in the structured block is replicated, each thread executes a copy. At the end of the block there is an implied barrier, only the "master" thread continues.

OpenMP: `pragma omp parallel`

```
#pragma omp parallel optional-clauses
```

The parallel directive has **optional** clauses, the most commonly used are:

- Specify the **number of threads to spawn**:

```
#pragma omp parallel num_threads(int)
```

- Conditional parallelization with:

```
#pragma omp parallel if (condition)
```

Example 1: parallel if condition

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

void test(int val)
{
    #pragma omp parallel if (val != 0)
    if (omp_in_parallel()) {
        #pragma omp single
        printf_s(
            "val = %d, parallelized with %d threads\n",
            val, omp_get_num_threads()
        );
    } else {
        printf_s("val = %d, serialized\n", val);
    }
}

int main( )
{
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

The output will be:

```
val = 0, serialized
val = 2, parallelized with 2 threads
```

- Data scope clauses (explained in the following pages).

The **number of threads in a parallel region is determined by the following factors**, in order of priority (high to low):

1. Evaluation of the `if` clause;
2. Value of the `num_threads` clause;
3. Use of the `omp_set_num_threads()` library function;
4. Setting of the `OMP_NUM_THREADS` environment variable;
5. Implementation default, e.g., the number of CPUs on a node.

5.3 Work sharing

Work-sharing constructs divide the execution of a region of code among the team members who encounter it. A work-sharing construct must be enclosed in a parallel region for the directive to be executed in parallel. Note that the **constructs do not start new threads**. Also, there is **no implicit barrier at the *entry*** of a work-sharing construct, but **there is an implicit barrier at the *exit*** of a work-sharing construct.

5.3.1 For

The for directive **shares iterations of a loop across the team** (data parallelism).

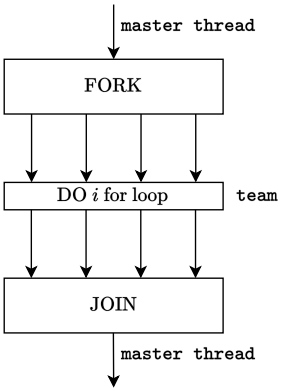


Figure 14: OpenMP for loop.

OpenMP: `pragma omp for`

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

The for directive parallelize execution of iterations. The number of iteration cannot be internally modified. Some common clauses are:

- *`schedule`* that describes **how iterations of the loop are distributed among the threads in the team**. The schedule type can be either `dynamic`, `guided`, `runtime`, or `static`.
  - *`static`*. Loop **iterations are divided into blocks of size** `chunk` and then statically allocated to threads. If `chunk` is **not specified**, the **iterations are divided evenly** (if possible) **among the threads**.

OpenMP: `static` schedule

```
#pragma omp for schedule(static, chunk-size)
```

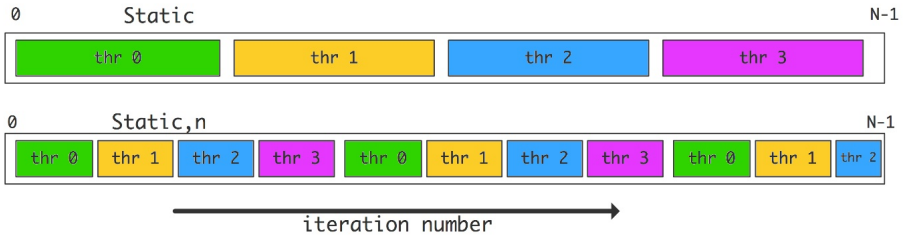


Figure 15: `static` schedule.

– *`dynamic`*. Loop **iterations are divided into blocks of size** `chunk` and **distributed among the threads *at runtime***; when a thread completes one chunk, it is **dynamically allocated another**. The default chunk size is 1. In fact, we can see in the image that the order is not always the same.

OpenMP: `dynamic` schedule

```
#pragma omp for schedule(dynamic, chunk-size)
```

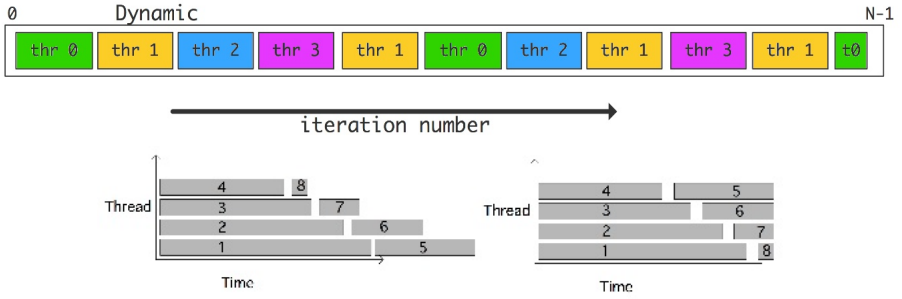


Figure 16: `dynamic` schedule.

– *`runtime`*. Depends on environment variable `OMP_SCHEDULE`.

– *`guided`*. Static, gradually decreases the chunk size (`chunk` specifies the smallest one).

OpenMP: `guided` schedule

```
#pragma omp for schedule(guided)
```

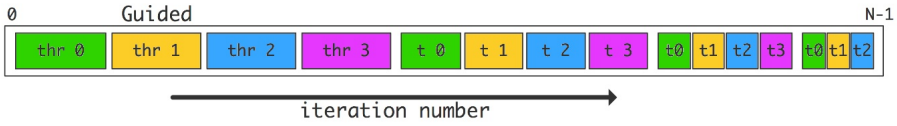


Figure 17: `guided` schedule.

Example 2: `schedule` types

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

#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3

int main( )
{
    int nStatic1[NUM_LOOPS],
        nStaticN[NUM_LOOPS];
    int nDynamic1[NUM_LOOPS],
        nDynamicN[NUM_LOOPS];
    int nGuided[NUM_LOOPS];

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStatic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(static, STATIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStaticN[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nDynamic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
```

```
50                  Sleep(0);
51              nDynamicN[i] = omp_get_thread_num( );
52          }
53
54          #pragma omp for schedule(guided)
55          for (int i = 0 ; i < NUM_LOOPS ; ++i)
56          {
57              if ((i % SLEEP_EVERY_N) == 0)
58                  Sleep(0);
59              nGuided[i] = omp_get_thread_num( );
60          }
61      }
62
63      printf_s("
        -------------------------------------------------\n")
        ;
64      printf_s("| static | static | dynamic | dynamic |
        guided |\n");
65      printf_s("|    1   |    %d   |    1    |    %d    |
              |\n",
66                  STATIC_CHUNK, DYNAMIC_CHUNK);
67      printf_s("
        -------------------------------------------------\n")
        ;
68
69      for (int i=0; i<NUM_LOOPS; ++i)
70      {
71          printf_s("|    %d   |    %d   |    %d    |    %d
            |"
72                      "    %d   |\n",
73                      nStatic1[i], nStaticN[i],
74                      nDynamic1[i], nDynamicN[i], nGuided[
        i]);
75      }
76
77      printf_s("
        -------------------------------------------------\n")
        ;
78  }
```

The result will be:

```
1  -------------------------------------------------
2  | static | static | dynamic | dynamic | guided |
3  |    1   |    5   |    1    |    5    |        |
4  -------------------------------------------------
5  |    0   |    0   |    0    |    2    |    1   |
6  |    1   |    0   |    3    |    2    |    1   |
7  |    2   |    0   |    3    |    2    |    1   |
8  |    3   |    0   |    3    |    2    |    1   |
9  |    0   |    0   |    2    |    2    |    1   |
10 |    1   |    1   |    2    |    3    |    3   |
11 |    2   |    1   |    2    |    3    |    3   |
12 |    3   |    1   |    0    |    3    |    3   |
13 |    0   |    1   |    0    |    3    |    3   |
14 |    1   |    1   |    0    |    3    |    2   |
15 |    2   |    2   |    1    |    0    |    2   |
16 |    3   |    2   |    1    |    0    |    2   |
17 |    0   |    2   |    1    |    0    |    3   |
18 |    1   |    2   |    2    |    0    |    3   |
19 |    2   |    2   |    2    |    0    |    0   |
```

```
|     3     |     3     |     2     |     1     |     0     |
|     0     |     3     |     3     |     1     |     1     |
|     1     |     3     |     3     |     1     |     1     |
|     2     |     3     |     3     |     1     |     1     |
|     3     |     3     |     0     |     1     |     3     |
-------------------------------------------------
```

- *nowait* to **avoid synchronization at the end of the parallel loop**. It overrides the barrier implicit in a directive.

```
#pragma omp for nowait
```

Example 3: nowait clause

```
#include <stdio.h>

#define SIZE 5

void test(int *a, int *b, int *c, int size)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp for nowait
        for (i = 0; i < size; i++)
            c[i] = a[i]/2;
    }
}

int main( )
{
    int a[SIZE], b[SIZE], c[SIZE];
    int i;

    for (i=0; i<SIZE; i++)
        a[i] = i;

    test(a,b,c, SIZE);

    for (i=0; i<SIZE; i++)
        printf_s("%d, %d, %d\n", a[i], b[i], c[i]);
}
```

The output will be:

```
0, 0, 0
1, 1, 0
2, 4, 1
3, 9, 1
4, 16, 2
```

5.3.1.1 Reduction

In parallel programming, sometimes there are some exceptional cases when we use the `for` statement where the variables inside the code block are not so easy to manage (memory viewpoint). For example, consider the following case:

```c
#include "stdio.h"
#include "omp.h"
#define MAX 10


int main(int argc, char const *argv[])
{
    double ave = 0.0;
    double A[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    #pragma omp parallel for
    for(i = 0; i < MAX; i++) {
        ave += A[i];
    }
    ave = ave / MAX;
    printf("Value %f\n", ave);
    return 0;
}
```

⚠ Too many threads modifying the same variable

In this case, we are combining values into a single accumulation variable (called `ave`). There is a true dependence between loop iterations that can't be trivially removed. A **continuous execution produces different results!**

```
$ ./example.out
Value 1.300000
$ ./example.out
Value 2.400000
$ ./example.out
Value 2.500000
$ ./example.out
Value 2.100000
$ ./example.out
Value 1.900000
```

This is a very common situation and a solution, which can also be **used as a synchronization technique**, is called a **reduction**.

A reduction variable in a loop **aggregates** (i.e., accumulates) a **value that depends on each iteration of the loop** and **doesn't depend on the iteration order**.

OpenMP: *reduction*

```c
#pragma omp parallel for reduction(operator: list)
```

A reduction clause:

- It **makes a local copy** of each *list* variable and initialized depending on the *operator*;

- It **updates occur on the local copy**;
- **Local copies are reduced into a single value and combined with the original global value**.

Therefore, the variables in *list* must be shared in the enclosing parallel region.

Many different associative operands (*operator* value) can be used with reduction:

- `+` with initial value 0
- `*` with initial value 1
- `-` with initial value 0
- `min` with initial value as largest positive number
- `max` with initial value as most negative number
- `&` with initial value ~ 0
- `|` with initial value 0
- `^` with initial value 0
- `&&` with initial value 1
- `||` with initial value 0

Using the `reduction` clause, the code written at the beginning of the paragraph can be fixed as follows:

```c
#include "stdio.h"
#include "omp.h"
#define MAX 10


int main(int argc, char const *argv[])
{
    double ave = 0.0;
    double A[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    // use reduction
    #pragma omp parallel for reduction(+: ave)
    for(i = 0; i < MAX; i++) {
        ave += A[i];
    }
    ave = ave / MAX;
    // now it prints the correct result 5.5
    printf("Value %f\n", ave);
    return 0;
}
```

✔ **Avoid race condition**

5.3.2 Sections

Section identifies code **sections to be divided among all threads**.

Sections allow to specify that the enclosed section(s) of code are to be executed in parallel. **Each section is executed once by a thread in the team.**

OpenMP: sections

```
#pragma omp [parallel] sections [clauses]
{
    #pragma omp section
    {
        code_block
    }
}
```

The sections directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.

Each section is preceded by a `section` directive, although the `section` directive is optional for the first section. The `section` directives must appear within the lexical extent of the `sections` directive. There's an **implicit barrier** at the end of a `sections` construct, unless a `nowait` is specified.

Restrictions to the sections directive are as follows:

- A `section` directive must not appear outside the lexical extent of the `sections` directive.
- Only a single `nowait` clause can appear on a `sections` directive.

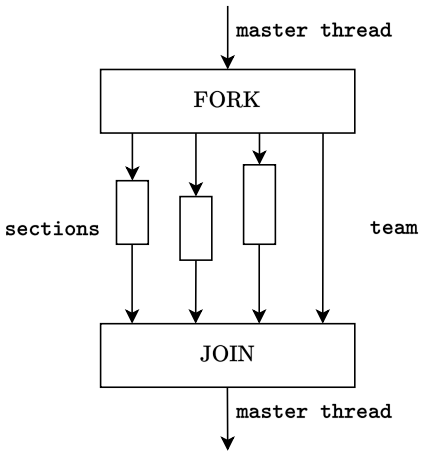


Figure 18: Breaks work into separate, discrete sections, each executed by a thread (functional parallelism).

5.3.3 Single/Master

A **section** (not the directive) **of code should be executed on a single thread, not necessarily the main (master) thread**. The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).

OpenMP: `single` and `master`

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

- `single` specifies that a section of a code is **executed only by a single thread**.
- `master` specifies that a section of a code is **executed only by the master**.

There's an **implicit barrier** after the `single` construct unless a `nowait` clause is specified.

5.3.4 Tasks

The following section has been enhanced with slides from Senior Principal Engineer Mattson Tim. He's a senior principal engineer at Intel, where he's been since 1993. His profile can be seen here and the slides are available online here. He has also made an interesting YouTube series on the introduction to OpenMP.

Tasks are **independent units of work**. They consist of: *code to execute*, *data environment*, and *internal control variables* (ICV). **Threads perform the work of each task**. The runtime **system decides when to execute tasks**; each task can be deferred or executed immediately.

Some useful terminology:

- ***Task construct***. It identifies the **task directive plus the structured block**.
- ***Task***. It is the **package of code and instructions** for **allocating data** created when a **thread encounters a task construct**.
- ***Task region***. It is the dynamic sequence of **instructions generated by the execution of a task by a thread**.

Tasks are guaranteed to complete at thread barriers (using the `barrier` directive) or at task barriers (using the `taskwait` directive):

```
#pragma omp parallel // omp directive to parallel the code
{
    #pragma omp task // multiple foo tasks created here,
                     // one for each thread
    foo();
    #pragma omp barrier // all foo tasks guaranteed
                        // to be completed here
    #pragma omp single  // only one thread can access to
                        // this piece of code
    {
        #pragma omp task // one bar task created here
        bar();
    }
    // foo task guaranteed to be completed here
}
```

Example 4: Fibonacci with tasks

Let us see a Fibonacci example of data scoping using the tasks. In the following code, we create the Fibonacci function and we create two tasks, but each task has a private variable and these variables are also used in the return statement:

```
int fib(int n) {
    int x, y;
    if(n < 2)
        return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
```

```
9       #pragma omp taskwait
10      return x + y;
11  }
```

A good solution is to "share" the `x` and `y` variables because we need both values to calculate the sum.

```
1   int fib(int n) {
2       int x, y;
3       if(n < 2)
4           return n;
5       #pragma omp task shared(x)
6       x = fib(n-1);
7       #pragma omp task shared(y)
8       y = fib(n-2);
9       #pragma omp taskwait
10      return x + y;
11  }
```

✔ Main advantage

Note the following code:

```
1   // create a team of threads
2   #pragma omp parallel
3   {
4       // one thread executes the single construct
5       // and other threads wait at the implied
6       // barrier at the end of the single construct
7       #pragma omp single
8       { // block 1
9           node *p = head;
10          while(p) { // block 2
11              // the single thread creates a task
12              // with its own value for the pointer p
13              #pragma omp task firstprivate(p)
14                  process(p);
15              p = p -> next; // block 3
16          }
17          // execution moves beyond the barrier
18          // once all the tasks are complete
19      }
20  }
```

The tasks have the **potential to parallelize irregular patterns and recursive function calls**. See Figure 19 (page 70) to understand how the runtime system can optimize execution using the tasks.

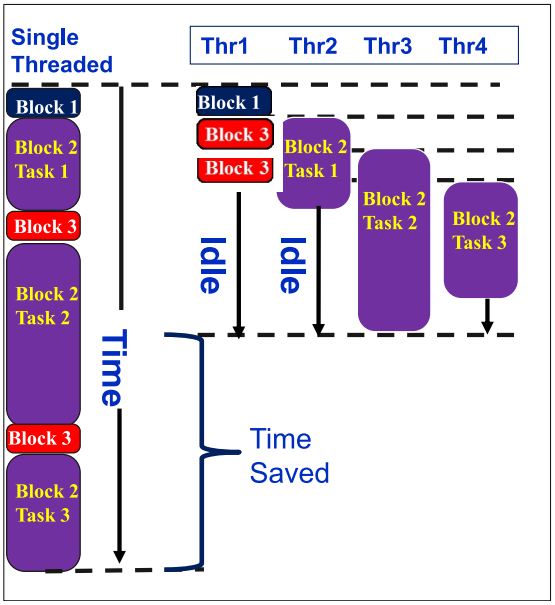


Figure 19: The main advantage of the tasks is to parallelize irregular patterns and recursive function calls.

5.4 Synchronization

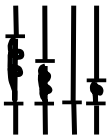
The following section has been enhanced with slides from Senior Principal Engineer Mattson Tim. He's a senior principal engineer at Intel, where he's been since 1993. His profile can be seen here and the slides are available online here. He has also made an interesting YouTube series on the introduction to OpenMP.

OpenMP is a multi-threaded, shared address model. This means that threads communicate by sharing variables. Unfortunately, this can cause some problems such as race conditions (page 47). The solution is to **use synchronization to protect against data conflicts**. The good news is that synchronization can avoid data race problems, but it is also an **expensive method**. So we can use synchronization, but we **need to change the way data is accessed to minimize the need for synchronization**.

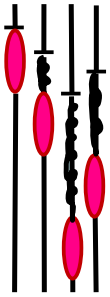
Synchronization brings **one or more threads to a well-defined and known point in their execution**. The two most common forms of synchronization are:

- ***Barrier***: each thread wait at the barrier **until all threads arrive**.
- ***Mutual exclusion***: define a block of code that **only one thread at a time can execute**.

The OpenMP directives are: `critical`, `atomic`, and `barrier`. These are high-level synchronization directives, but there are also low-level synchronization directives, such as `flush` and `locks`, but they are too complex at the moment, we will see them later.



(a) Barrier. Each thread wait at the barrier until all threads arrive.



(b) Mutual exclusion. Define a block of code that only one thread at a time can execute.

Barrier. Each thread waits until all threads arrive.

OpenMP: `barrier`

```
#pragma omp barrier name
```

An optional `name` may be used to identify the `critical` region. A **thread waits at the beginning of a critical region** until no other thread is executing a critical region (anywhere in the program) **with the same name**. **All unnamed `critical` directives map to the same unspecified name**.

Example 5: synchronization with `barrier` directive

The following example includes several critical directives. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against many threads dequeuing the same task, the dequeuing operation must be in a `critical` section. Because the two queues in this example are independent, they're protected by `critical` directives with different names, *xaxis* and *yaxis*. [9]

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
    #pragma omp critical ( xaxis )
        x_next = dequeue(x);
    work(x_next);
    #pragma omp critical ( yaxis )
        y_next = dequeue(y);
    work(y_next);
}

```

Mutual exclusion. Only one thread at a time can enter a *critical* region.

OpenMP: `critical`

```
#pragma omp critical
```

	`omp critical`	`omp single`
Meaning	Run code segment one by one by all threads	Run code segment once by any thread
Number of times code is executed	Number of threads	Only one
Use case	Avoid race condition	Manage control variables or signals

Table 3: ⚠ `omp critical` vs `omp single`

Example 6: synchronization with `critical` directive

Note the following code:

```
float res;

#pragma omp parallel
{
    float B;
    int i, id, nthrds, niters = big_number;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i = id; i < niters; i += nthrds) {
        B = big_job(i);
        #pragma omp critical // threads wait their turn;
        res += consume(B);   // only one at a time
    }                        // calls consume
}

```

Atomic. Provides mutual exclusion, but only when updating a memory location. It **ensures that a particular memory location is accessed atomically**. It is **valid only for the following statement** and not for a structured block.

The statement inside the atomic must be one of the following forms:

- `x binop = expr`
- `x++` or `++x`
- `x--` or `--x`

Where `x` is an lvalue of scalar type and `binop` is a non-overloaded builtin operator.

OpenMP: `atomic`

```
#pragma omp atomic
```

Example 7: synchronization with `atomic` directive

```
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}

```

5.5 Data environment

OpenMP is based on the shared memory programming model, so most **variables are shared by default**. **Global variables are also shared between threads**. But not everything is shared; for example, *stack variables* in functions called from parallel regions are **private**, as are *automatic variables* within a statement block.

Example 8: data sharing

In the following code, the variable `temp` is private (local to each thread) because it is in the stack of the function `work`; meanwhile, the variables `A`, `index`, and `count` are shared by all threads.

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index);
    printf("%d\n", index[0]);
}

void work(int *index) {
    double temp[10];
    static int count;
    /* other code */
}
```

We can refer to these arguments as **Data Scope Attribute Clauses** because the issue is really about the visibility and value of each data in each scope. Although OpenMP shares variables by default, there is (and it is a very common and *best practice*) the option to:

- Selectively **change storage attributes for construct** using the following clauses: `shared`, `private` and `firstprivate`.
- The **final value** of a private inside a parallel loop can be **transmitted to the shared variable outside the loop** with: `lastprivate`.
- The **default attributes can be overridden** with:
  `default(private | shared | none)`

⚠ Note that when we say "copy" we mean the *shallow copy*, not the *deep copy*. So when the following clauses create a local copy, they create a *shallow copy*.

`private` **clause**. The statement `private(var)` creates a **new local copy** of `var` **for each thread**. The value of the private copies is ***uninitialized*** and also the **original variable value remains unchanged after the region**.

OpenMP: `private(...)`

```
#pragma omp parallel directive private(var1-name, var2-name, ...)
```

Example 9: `private` clause and *dirty memory location*

In the following code we try to use the private clause inside a parallel for. The code is very trivial, we set the number of threads to two for better understanding, so we start the parallel code with the for directive and the private clause. It has `private_test` as a private variable. So each thread will copy the variable and the initial value will be undefined.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[])
{
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for private(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        private_test = i == 0 ? 0 : ++private_test;
        printf(
            "Thread #%i, iter_i: %d, private_test: %d\n",
            omp_get_thread_num(), i, private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Unfortunately, when we examine the output, we see a problem. Thread zero (*master*) executes the `for` statement for the first 3 iterations, while thread one (*slave*) executes the `for` statement for the last 3 iterations. The zero thread behaves as expected because we initialize the `private_test` variable to zero on the first `for` iteration (when `i` is 0). The thread one, has made a copy of the variable in another memory location and the value is unknown; so it continues to add a single value to each iteration in a *dirty memory location*. This is a trivial example that highlights the unknown values that we can find inside the private variables if we don't do any initialization.

```
$ g++ -fopenmp example.cpp -o example
$ ./example
Memory location of private_test: 0x7ffecdfa3aa4
Master will execute the for statement in parallel!
Thd #0, i:0, private_test:0,         mem: 0x7ffecdfa3a40
Thd #0, i:1, private_test:1,         mem: 0x7ffecdfa3a40
Thd #0, i:2, private_test:2,         mem: 0x7ffecdfa3a40
Thd #1, i:3, private_test:687869953, mem: 0x76a0297ffdd0
Thd #1, i:4, private_test:687869954, mem: 0x76a0297ffdd0
Thd #1, i:5, private_test:687869955, mem: 0x76a0297ffdd0
private_test outside the parallel region: 10
```

`firstprivate` **clause**. The **variables are initialized from the shared variable**, but as in the `private` clause, the updated value doesn't leave the parallel region.

OpenMP: `firstprivate(...)`

```
#pragma omp parallel directive firstprivate(var1-name, ...)
```

Example 10: `firstprivate` **clause**

A very trivial example to see how the firstprivate clause works. The variable `private_test` is copied to local and initialized with the value outside the parallel region.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[])
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for firstprivate(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        ++private_test;
        printf(
            "Thd #%i, i:%d, private_test:%d, mem: %p\n",
            omp_get_thread_num(), i,
            private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Note that the value is not propagated outside the parallel region.

```
$ g++ -fopenmp example.cpp -o example
$ ./example
Memory location of private_test: 0x7ffc5cd153b0
Master will execute for in parallel!
Thd #0, i:0, private_test:11, mem: 0x7ffc5cd15350
Thd #0, i:1, private_test:12, mem: 0x7ffc5cd15350
Thd #0, i:2, private_test:13, mem: 0x7ffc5cd15350
Thd #1, i:3, private_test:11, mem: 0x77001d9ffdd0
Thd #1, i:4, private_test:12, mem: 0x77001d9ffdd0
Thd #1, i:5, private_test:13, mem: 0x77001d9ffdd0
private_test outside the parallel region: 10
```

Example 11: be careful with pointers using the `firstprivate` clause

The following code is very similar to the previous one. The difference here is that the parallel region also gets a pointer. Note that the pointer is in C style, because using the unique pointer technique (suggested in C++) will create an exception, because C++ doesn't allow the copy of a unique pointer. However, each thread creates a shallow copy of the pointer, but not a copy of the value pointed to! In this test, we use the pointer to the value of `private_test` to modify the value of `private_test`.

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
    bool print_flag = true;
    int i, private_test = 10;
    // C pointer, not a good practice in C++...
    // used only for the example
    int *ptr_private_test = &private_test;
    printf("Memory location of private_test    : %p\n",
           &private_test);
    printf("Memory location of ptr_private_test: %p\n",
           &ptr_private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n\n");
    #pragma omp parallel for firstprivate(private_test,
    ptr_private_test, print_flag)
    for(i = 0; i < MAX; ++i) {
        if (print_flag) {
            printf("Memory location ptr %p\n",
                   &ptr_private_test);
            print_flag = false;
        }
        // increase value pointed to by ptr
        ++*ptr_private_test;
        // increase simple variable
        ++private_test;
        printf(
            "Thread #%i, i:%d\n- private_test:%d,
    ptr_private_test:%d, mem: %p\n\n",
            omp_get_thread_num(), i, private_test,
            *ptr_private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

Note an interesting observation. The variable `private_test` is incremented at each iteration; in the same way, the value pointed to by the pointer `ptr_private_test` is also incremented. Finally, the variable

`private_test` is modified because the pointer was copied in each thread and each slave, including the master, increased the value. This is a very bad practice and we want to suggest to use unique pointers of C++ or to avoid shallow copies.

```
Memory location of private_test    : 0x7fff3849c224
Memory location of ptr_private_test: 0x7fff3849c228
Master will execute for in parallel!

Memory location ptr 0x7fff3849c1a0
Thread #0, i:0
- private_test:11, ptr_private_test:11, mem: 0x7fff3849c198

Thread #0, i:1
- private_test:12, ptr_private_test:12, mem: 0x7fff3849c198

Thread #0, i:2
- private_test:13, ptr_private_test:13, mem: 0x7fff3849c198

Memory location ptr 0x7b710cfffdb0
Thread #1, i:3
- private_test:11, ptr_private_test:14, mem: 0x7b710cfffda8

Thread #1, i:4
- private_test:12, ptr_private_test:15, mem: 0x7b710cfffda8

Thread #1, i:5
- private_test:13, ptr_private_test:16, mem: 0x7b710cfffda8

private_test outside the parallel region: 16
```

The expected value for `private_test` should remain 10 because the firstprivate doesn't affect the values of the variables after the parallel region, but in this case we are using a pointer and a bad practice.

`lastprivate` **clause**. The variables **update shared variables with the value from the last iteration**, so the order of execution of the threads is important here.

OpenMP: `lastprivate(...)`

```
#pragma omp parallel directive lastprivate(var1-name, ...)
```

Example 12: `lastprivate` **clause**

In the following example, the value of the last iteration is passed (and overwritten) to the original value:

```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
```

```
    int i, private_test = 10;
    printf("Memory location of private_test: %p\n",
           &private_test);
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    printf("Master will execute for in parallel!\n");
    #pragma omp parallel for lastprivate(private_test)
    for(i = 0; i < MAX; ++i) {
        // initialize private_test
        private_test = i;
        printf(
            "Thd #%i, i:%d, private_test:%d, mem: %p\n",
            omp_get_thread_num(), i,
            private_test, &private_test
        );
    }
    printf(
        "private_test outside the parallel region: %d\n",
        private_test
    );
    return 0;
}
```

The `private_test` variable initially has a value equal to 10, but with `lastprivate` we have overwritten it.

```
Memory location of private_test: 0x7ffc4c82b8c0
Master will execute for in parallel!
Thd #0, i:0, private_test:0, mem: 0x7ffc4c82b860
Thd #0, i:1, private_test:1, mem: 0x7ffc4c82b860
Thd #0, i:2, private_test:2, mem: 0x7ffc4c82b860
Thd #1, i:3, private_test:3, mem: 0x72c7dddffdd0
Thd #1, i:4, private_test:4, mem: 0x72c7dddffdd0
Thd #1, i:5, private_test:5, mem: 0x72c7dddffdd0
private_test outside the parallel region: 5
```

`default` clause. The default storage attribute is `default(shared)`. To change the default we can write simply the value shared or none inside the brackets:

- `default(share)` is the default choice for OpenMP, so there is no need to use it except for the clause `pragma omp task`.

OpenMP: `default(share)`

```
#pragma omp parallel directive default(share)
```

Against the `private` clauses, if we want to share some variables, we can use `shared` clause.

OpenMP: `shared`

```
#pragma omp parallel directive shared(var1-name, ...)
```

- `default(private)`, each variable in the construct is made private as if specified in private clause.

OpenMP: `default(private)`

```
#pragma omp parallel directive default(private)
```

- `default(none)`, no default for variables in static extent. Must list storage attribute for each variable in static extent. It is a good programming practice.

OpenMP: `default(none)`

```
#pragma omp parallel directive default(none)
```

Example 13: `default(none)`

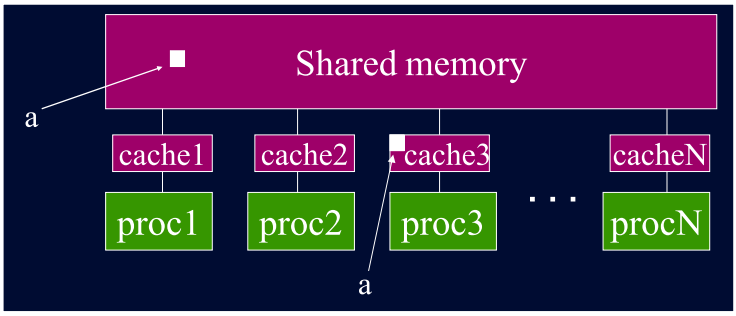
```
#include <iostream>
#include "omp.h"
#define MAX 6

int main(int argc, char const *argv[]) {
    double private_test = 10.0;
    int i;
    // set limit to 2 threads for better understanding
    omp_set_num_threads(2);
    #pragma omp parallel for default(none) private(i,
    private_test)
    for(i = 0; i < MAX; i++) {
        private_test = i;
        printf(
            "Thread #%i, value: %f\n",
            omp_get_thread_num(), private_test
        );
    }
    printf(
        "private_test outside the parallel region: %f\n"
    ,
        private_test
    );
    return 0;
}
```

```
Thread #0, value: 0.000000
Thread #0, value: 1.000000
Thread #0, value: 2.000000
Thread #1, value: 3.000000
Thread #1, value: 4.000000
Thread #1, value: 5.000000
private_test outside the parallel region: 10.000000
```

5.6 Memory model

OpenMP supports a **shared memory model**. It is a model where **all threads share an address space**, but it can get complicated, for example in the following example (picture) we can see a shared space that is also in the cache of process 3; so how can we manage this situation? What are the methods to adapt? Furthermore, what happens when a thread modifies a shared address space? How does the system behave?



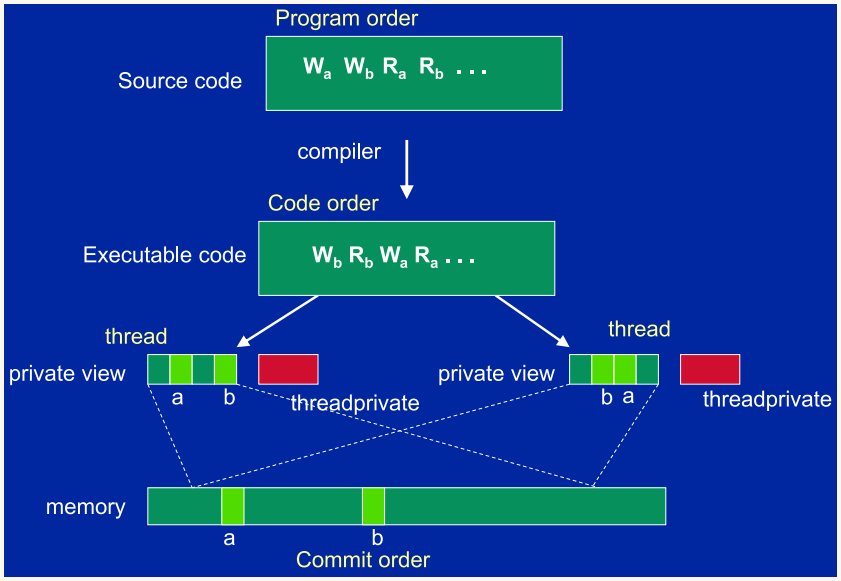
In general, a memory model is defined in terms of:

- **Coherence**: behavior of the memory system when a **single address is accessed by multiple threads**.
- **Consistency**: read, write, or synchronization (RWS) orders with **different addresses and through multiple threads**.

Remark: what the compiler does at the low level

When we write a program and ask the compiler to compile it, it does a lot of "magic" under the hood.

1. Our source code is decoded into very low level operations. These operations respect the order of the high-level operations of our code.
2. The compiler, smarter then us, tries to optimize the code while creating the executable code; so it reorders the low-level operations that are **semantically equivalent** to our program, but often allow to gain performance.
3. During the execution of the code, if it is parallel, some threads are created and each of them has a private memory address.
4. Finally, each thread writes/reads from memory, also called the commit order, using some rules.



Note that the re-ordering are made by:

- **Compile** re-orders *program order* → *code order*.
- **Machine** re-orders *code order* → *memory commit order*.

At any given time, the private view seen by a thread may differ from the view in shared memory. For this reason, there are **consistency models** that **define constraints on the order of RWS** (Reads, Writes, Synchronizations) **operations**.

In general, a multiprocessor adopts the **Sequential Consistency Model**. It says that given n operations (RWS), they are sequentially consistent if:

- They **remain in *program order* for each processor**.
- They are **seen to be in the same overall order by each of the other processors**.

Also, in a sequential consistency model, program order is the same as code order and commit order.

OpenMP uses a **Relaxed Consistency Model** where the **compiler cannot reorder synchronization operations with read or write operations on the same thread**. The consequences are:

- ✓ All threads have the same view of memory at specific points in the code, called ***Consistency Points***.
- ✓ Between two consistency points, each thread has its own temporary view of memory, which may be different from the other temporary views of other threads.

✓ Data are read-only, this guarantee to avoid consistency issues.

✗ Shared data that need to be modified can create possible race conditions.

`flush` **directive**. Defines a *sequence point* at which a **thread is guaranteed to see a consistent view of memory** with respect to the `flush-set`. The `flush-set` means ***all thread visible variables*** for a `flush` construct without an argument list, and it also means a list of variables when the `flush(list)` construct is used. The action of `flush` is to **guarantee** that:

- All read/write **operations that overlap** the `flush-set` and **occur *before*** the `flush` will be **completed before the `flush` is executed**.
- Any read/write **operations that overlap** the `flush-set` and **occur *after*** the `flush` will **not be performed until after the `flush`**.
- Flushes with overlapping `flush-set`s can not be reordered.

In other words, the `flush` **forces data to be updated in memory so that other threads see the most recent value**.

OpenMP: `flush`

```
1 #pragma omp flush flush-set
```

A flush operation is **implied** by default by OpenMP synchronizations:

- At entry/exit of `parallel`/`critical` regions;
- At implicit and explicit `barrier`s;
- At exit from work-sharing constructs, unless `nowait` is specified.

And it is **not implied**:

- At entry of work-sharing constructs;
- At entry and exit of `master`.

5.7 Nested Parallelism

How does OpenMP create multiple threads?

OpenMP uses a **fork-join model** of parallel execution. When a thread encounters a `parallel` construct, the **thread creates a team composed of itself and some additional** (possibly zero) **number of threads**. The encountering **thread becomes the *master* of the new team**. The **other threads of the team are called *slave*** threads of the team.

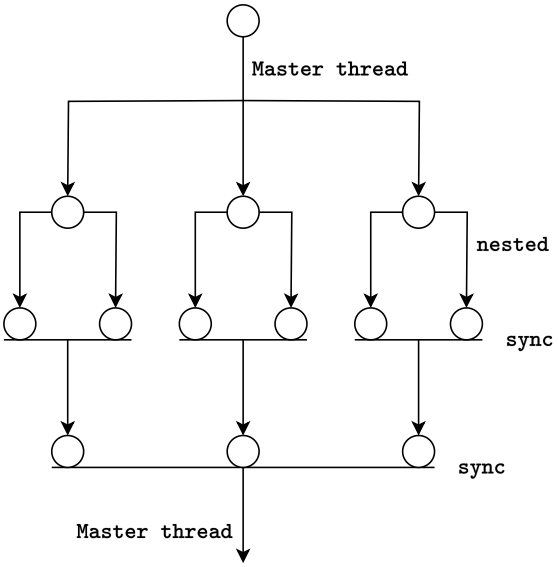


Figure 21: OpenMP fork-join model.

What is the life history of each thread (*slave*) created?

All team members execute the code inside the `parallel` construct. When a thread finishes its work within the parallel construct, it waits at the implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, the threads can leave the barrier. The *master* thread continues execution of user code beyond the end of the `parallel` construct, while the *slave* threads wait to be summoned to join other teams.

OpenMP **parallel regions can be nested inside each other**. If nested parallelism is:

- **Disabled**, then the **new team** created by a thread encountering a parallel construct inside a parallel region **consists only of the encountering thread**.
- **Enabled**, then the **new team may consist of more than one thread**.

How does OpenMP manage the available threads? Thread Pool

The OpenMP runtime library maintains a pool of threads that can be used as *slave* threads in parallel regions. When a thread encounters a parallel construct and needs to create a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them *slave* threads of the team. The *master* thread might get fewer *slave* threads than it needs if there is not a sufficient number of idle threads in the pool. When the team finishes executing the parallel region, the *slave* threads return to the pool.

Summary

1. **Parallel construct**. Our main thread starts to execute our code. It encounters a `parallel` construct.
2. **Pool verification**. The OpenMP library checks its pool of threads. If within its pool of available threads have something of disposable, it allocates the *slave* (thread requested) to the *master* (thread that want to create the team). We refer to a single thread, but obviously this can be extended to multiple thread request (e.g. *master* thread requests 3 threads to OpenMP). Finally, the number of requested *slave*s cannot always be satisfied; OpenMP guarantees the best, so it continues to give the requested threads to the applicants. If it cannot satisfy the request, it returns the maximum number of *slave*s it can satisfy (e.g. *master* requests 4 threads, but OpenMP has a pool of only 2 threads available; therefore it returns 2 *slave*s).
3. **Assign and start execution**. Ideally, OpenMP returns the number of threads requested by the *master*. Otherwise, it returns the maximum number. The team now consists of the main thread, called the *master*, and its *slave*s. Each member of the team executes the code specified by the programmer within the parallel construct.
4. **End of execution of a thread**. A thread of the team finishes its work. It can finally rest, and its state changes from "running" to "waiting". It waits for its other thread friends. Each thread has an implicit barrier at the end of the parallel construct.
5. **Any thread finish**. Finally, each thread finishes its work. The *master* releases each member of the team to the OpenMP library. OpenMP updates its thread pool with the returning threads, and the *master* thread continues its life independently from the released threads.

The previous flow works if and only if nested parallelism is enabled. Otherwise, if a *master* asks to create a new team, it will ignore the request and tell the requester to continue alone.

Implementation

Nested parallelism can be enabled or disabled by passing true or false as arguments to the runtime function:

OpenMP: `omp_set_nested`

```
void omp_set_nested(int nested)
```

We can set a **default number of threads at different levels of nested parallelism** with:

```
OMP_NUM_THREADS = [list, of, integers]
```

If the nesting level is deeper than the number of entries in the list, the last value is used for all subsequent nested parallel region.

- `OMP_MAX_ACTIVATE_LEVELS` defines the upper **limit on the number of active parallel regions that may be nested**.
- `OMP_THREAD_LIMIT` **avoids** that recursive applications **create too many threads**.

Finally, since we are in nested parallelism, the thread number returns the thread number partially and not globally, we need other useful functions:

- Returns the maximum number of OpenMP threads available in contention group:

  OpenMP: `omp_get_thread_limit`

  ```
  int omp_get_thread_limit()
  ```

- Returns the maximum number of nested active parallel regions when the innermost parallel region is generated by the current task.

  OpenMP: `omp_get_max_active_levels`

  ```
  int omp_get_max_active_levels()
  ```

- Limits the number of nested active parallel regions when a new nested parallel region is generated by the current task.

  OpenMP: `omp_set_max_active_levels`

  ```
  void omp_set_max_active_levels(int max_levels)
  ```

- Returns the number of nested parallel regions on the device that enclose tha task containing the call.

OpenMP: `omp_get_level`

```
int omp_get_level()
```

- Returns the number of active, nested parallel regions on the device enclosing the task containing the call.

OpenMP: `omp_get_active_level`

```
int omp_get_active_level()
```

- Returns, for given nested level of the current thread, the thread number of the ancestor of the current thread.

OpenMP: `omp_get_ancestor_thread_num`

```
int omp_get_ancestor_thread_num(int level)
```

- Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor of the current thread belongs.

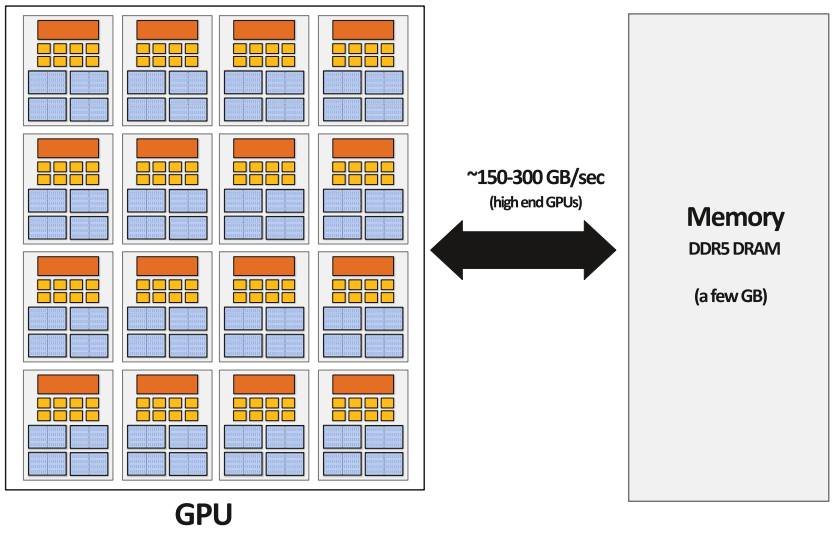
OpenMP: `omp_get_team_size`

```
int omp_get_team_size(int level)
```

6 GPU Architecture

6.1 Introduction

In the image below, we can see a very basic GPU architecture:



- **GPU Structure** (left). The GPU is made up of a **grid of** smaller blocks, each representing a **core**. All the blocks form a **multi-core structure** that allows the GPU to handle many tasks simultaneously.

  Within a **single core**, the GPU uses **SIMD** (Single Instruction, Multiple Data) **execution**. This means that many execution units within a core can simultaneously execute the same instruction on different pieces of data. This results in highly efficient execution of parallel tasks such as rendering graphics or running simulations.

  In addition, **each core supports multi-threaded execution**, which allows multiple threads to be processed simultaneously. This further enhances the GPU's ability to perform multiple tasks simultaneously.

- **Memory Connection** (right). The GPU is connected to DDR5 DRAM, a type of dynamic random access memory. Fast memory is essential to handle the large amounts of data that GPUs process. The more powerful the DRAM, the faster the data transfer.

Initially, GPUs were designed with a specific purpose: to render graphics quickly and efficiently. However, their role has expanded significantly over the years.

General-Purpose computing on Graphics Processing Units (GPGPU) was originally designed to **render graphics**, but GPUs have evolved to perform a wide range of computations beyond traditional graphics tasks. GPGPU takes advantage of the parallel processing capabilities of GPUs to perform computations typically handled by the CPU.

6.2 GPU compute mode

GPU compute mode refers to **GPU hardware that is optimized for general-purpose computing rather than graphics rendering**. This mode allows users to run non-graphics programs on the GPU's programmable cores, taking advantage of the GPU's parallel processing capabilities for tasks such as scientific simulations, data analysis, and machine learning.

Example 1: how to run code on a GPU (prior to 2007)

Now let us see how to run a simple code on a GPU. Suppose a user wants to draw a picture on a GPU:

- **GPU Shader Program Binaries**. The application, via the graphics driver, supplies the GPU with shader program binaries. These are compiled programs that the GPU will execute to perform rendering tasks.
- **Graphics Pipeline Parameters**. The application sets various parameters for the graphics pipeline, such as the output image size, to control how the rendering should be processed.
- **Vertex Buffer**. The application provides the GPU with a buffer of vertices. Vertices are data points that define the shape of the objects to be rendered.
- **Draw Command**. The application sends a draw command to the GPU using the function call `drawPrimitives(vertex_buffer)`. This command instructs the GPU to start rendering using the provided vertex data.

The stages of the graphics pipeline:

1. **Input Vertex Buffer**: The initial stage where the vertex data is input to the pipeline.
2. **Vertex Generation**: Vertices are generated or fetched from the vertex buffer.
3. **Vertex Processing**: The vertices undergo various transformations and shading calculations.
4. **Primitive Generation**: The processed vertices are used to generate geometric primitives (such as triangles).
5. **Fragment Generation (Rasterization)**: The primitives are converted into fragments (potential pixels).
6. **Fragment Processing**: Fragments undergo shading and texturing calculations to determine their final color and properties.
7. **Pixel Operations**: Final operations are performed on the fragments, such as depth testing and blending.
8. **Output Image Buffer**: The processed fragments are written to the output image buffer, resulting in the final rendered image.

Some history

Before 2007 the **only way to interface with GPU hardware was through the graphics pipeline**. This meant that GPUs were ***designed and used specifically for tasks related to graphics rendering***. The pipeline stages (such as vertex processing, fragment generation, and pixel operations) were all designed to transform vertex data into pixels displayed on the screen.

Because they were optimized to handle parallel tasks associated with rendering images, their architecture and interfaces were tightly coupled with graphics APIs.

By 2007, the concept of using GPUs for **General-Purpose computing on Graphics Processing Units (GPGPU)** was emerging. Thanks mainly to the introduction of a new architecture signed NVIDIA called Tesla and CUDA, a parallel computing platform and programming model (also OpenCL was emerging).

The **NVIDIA Tesla architecture**, introduced with the GeForce 8800 GPU in 2006, marked a significant shift in GPU design by unifying graphics and computing capabilities. This architecture featured a scalable parallel array of processors that could be programmed in `C` or via graphics `APIs2`. The Tesla architecture enabled **flexible**, **programmable graphics and high-performance computing**, making it possible to use GPUs for a wide range of applications beyond traditional graphics rendering. This unification enabled massive multithreading and parallel processing, dramatically improving performance for computationally intensive tasks. [7]

From this point on, the programmable cores of the GPU are used:

- Application could allocate buffers in GPU memory and copy data to/from buffers;
- Application (via the graphics driver) provides a single kernel program binary to the GPU;
- Application tells GPU to run kernel in SPMD fashion.

6.3 CUDA

6.3.1 Basics of CUDA

What is CUDA?

Compute Unified Device Architecture (CUDA), is a **parallel computing platform and Application Programming Interface** (API) **model** created by NVIDIA. It allows developers to utilize NVIDIA GPUs for general-purpose processing, enabling them to perform a wide range of computations more efficiently than with traditional CPU processing alone.

CUDA was introduced with the NVIDIA Tesla architecture, which marked a significant shift in GPU capabilities, enabling general purpose computing on GPUs. **CUDA is designed to be similar to the `C` programming language**, making it familiar to many developers. It allows programmers to write code that runs on GPUs using the compute-mode hardware interface.

CUDA's abstractions are relatively low-level and closely match the performance characteristics and capabilities of modern GPUs. This design goal helps maintain a low abstraction layer, ensuring that developers can take full advantage of the hardware's potential.

Note that **Open Computing Language (OpenCL)** is an open standards version of CUDA that runs on both CPUs and GPUs from multiple vendors. **While CUDA runs only on NVIDIA GPUs, OpenCL is designed to be more versatile and work on hardware from multiple vendors**.

CUDA Thread Hierarchy

CUDA organizes threads into a hierarchical structure to efficiently manage parallel computations on GPUs. To understand how it works, let's look at it from the deepest level up:

1. **Threads**. Threads (or **CUDA Threads**) are the **smallest unit of execution** in CUDA. Each thread runs a single instance of a kernel function[8].

   Threads are **identified by their unique thread IDs**, which are used to calculate memory addresses and control decisions.

2. **Thread Blocks**. Threads are grouped into blocks (called also **CUDA Blocks**). Each block can contain multiple threads that execute concurrently.

   Threads within the same block can communicate and share data through shared memory and synchronization primitives like barriers and atomic operations.

   The *maximum number* of threads per block is limited by the GPU architecture, typically up to 1024 threads per block.

[8] A kernel function in CUDA is a function that runs on the GPU (device) but is called from the CPU (host). These functions are executed by many parallel threads on the GPU

3. **Grids**. Blocks are organized into a grid (called also **CUDA Grids**). A grid is a **collection of blocks that execute the same kernel function**.

   All **threads in a grid share the same global memory space**.

   The grid can be multi-dimensional, allowing for flexible organization of blocks to match the problem's dimensions.

Note that CUDA uses the `x`, `y`, and `z` dimensions for threads, blocks, and grids because this **multi-dimensional structure** aligns well with many common computational problems. Many computational problems naturally fit into a multi-dimensional space. For example, image processing involves 2D data, and volumetric simulations involve 3D data.

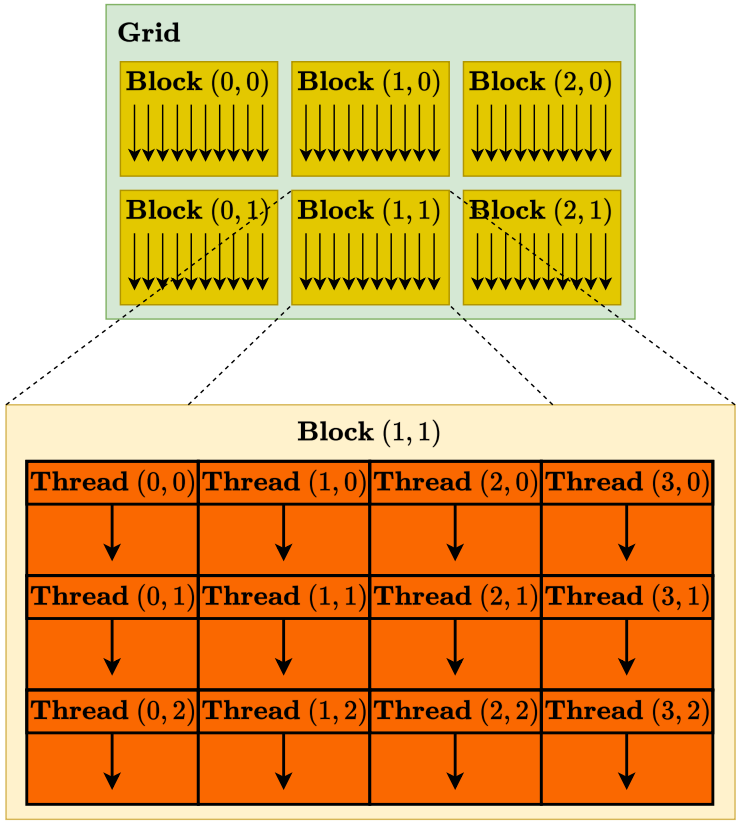


Figure 22: The figure shows an example of a CUDA grid consisting of 6 blocks, each block consisting of 12 threads. The number of threads and the number of blocks per column/row are customizable, this is just an example. In total, there are 72 CUDA threads (12 threads per block times 6 blocks in the grid).

CUDA Kernels

A **CUDA kernel** is a **function that gets executed on the GPU**. It contains the parallel portion of the application, which is executed by multiple threads in parallel. Unlike regular C/C++ functions that run on a single thread, CUDA kernels can run thousands of threads simultaneously.

⚖ Device vs Host

In CUDA, there is a term to identify the code that runs on the CPU and GPU.

Definition 1: CUDA Host (CPU)

The **CUDA Host refers to the CPU** and its associated memory.
It is **responsible for managing the overall program execution**. This includes allocating memory, launching CUDA kernels, and transferring data between the CPU and the GPU.
It is typically written in the standard programming language used (e.g., C, C++, Java, Python, etc.) and contains the logic for setting up and controlling the execution of CUDA kernels on the device.

Definition 2: CUDA Device (GPU)

The **CUDA Device refers to the GPU** and its associated memory.
It is **responsible for running parallel code** (CUDA kernels) and **performs the bulk of the computation**. This takes advantage of the parallel processing power of the GPU.
The device code (kernels) is written in the CUDA programming language and is designed to run on the multiple parallel cores of the GPU.

</> Basic CUDA syntax

Here is a sample code written in CUDA:

```
#include <cuda_runtime.h>
#include <iostream>
#define Nx 12
#define Ny 6

// Kernel definition
__global__ void MatAdd(
    float A[Ny][Nx],
    float B[Ny][Nx],
    float C[Ny][Nx]
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    // guard against out of bounds array access
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    // Define the hierarchy
    dim3 threadsPerBlock(4, 3);
    dim3 numBlocks(
        Nx / threadsPerBlock.x,
        Ny / threadsPerBlock.y
    );
    // Kernel invocation
    // Assume matrices A, B, C are already allocated (dim Nx x Ny)
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Rows 3-4 Variables `Nx` and `Ny` define the dimensions of the matrices. In this case, they are 12×6.

Row 21 The `threadsPerBlock` function specifies that each block will have 4 threads in the x dimension and 3 threads in the y dimension. In other words, it defines how a block should be composed (see the figure 22).

Row 22 The `numBlocks` function defines the number of blocks required to cover the entire matrix. The number of blocks is calculated by dividing the matrix dimensions by the number of threads per block in each dimension. In other words, it defines how a grid should be composed (see the figure 22).

Row 28 The execution configuration syntax `<<<...>>>` is required to specify the number of threads per block and the number of blocks per grid (both numbers can be of type `int` or `dim3`).

The function invocation starts the `MatAdd` kernel with the given grid and block dimensions. This kernel runs on the GPU and performs the matrix addition.

Row 7 Defines a kernel function called `MatAdd` to be executed on the GPU. The `__global__` keyword indicates that this is a kernel function.

Rows 12-13 Computes the global column index i for each thread. This combines the block index and the thread index within the block to get the overall position.

It also does the same with the j index.

Row 15 The `if` condition ensures that the thread does not access out-of-bounds elements in the matrices.

Row 16 Finally, it performs the element-wise addition of matrices `A` and `B` and stores the result in matrix `C`.

As we said in Figure 22 (page 93), the example spawns 72 CUDA threads.

6.3.2 Memory model

The CUDA memory model consists of several types of memory, each with different characteristics and uses. In general, the **Host (CPU) and the Device (GPU) have different address spaces**, each one has its private memory address.

For example, the `cudaMemcpy` function in CUDA is used to **copy data** between different memory spaces, specifically **between host (CPU)** memory **and device (GPU) memory**.

```
// allocate buffer in host mem
float* A = new float[N];

// populate host address space pointer A
for (int i = 0; i < N; ++i) {
    A[i] = (float)i;
}

// allocate buffer in device (GPU) address space
int bytes = sizeof(float) * N;
float* deviceA;
cudaMalloc(&deviceA, bytes);

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
// deviceA:
//      Destination memory address (either on the host or device).
//
// A:
//      Source memory address (either on the host or device).
//
// bytes:
//      Number of bytes to copy.
//
// cudaMemcpyHostToDevice:
//      Type of memory copy operation.
//      Copies data from host memory to device memory.
```

Note that directly accessing `deviceA[i]` is an invalid operation, because we cannot manipulate the contents of `deviceA` directly from host, since `deviceA` is not a pointer to the host's address space.

Types of CUDA device memory models visible to kernels

1. **Per-thread Private Memory**. This memory is **private to each thread**. Each thread has its own memory space that other threads cannot access.

   Usage. It is **ideal for storing variables that are only relevant to individual threads** and do not need to be shared with other threads.

   Access. It has **fast access**, limited by the number of registers available.

2. **Per-block Shared Memory**. This memory is **shared by all threads within a block**. It allows threads within the same block to cooperate by sharing data.

   Usage. It is useful for tasks where **threads within a block need to communicate or share intermediate results**. It is often used to optimize memory access patterns and reduce global memory accesses.

   Access. It is **much faster than global memory, but limited in size**. Access is almost as fast as registers when used properly.

3. **Device Global Memory**. This memory is **accessible to all threads across all blocks**. It provides a large amount of memory, but has higher latency and lower bandwidth than shared memory.

   Usage. It is **suitable** for storing large amounts of **data that must be accessed by threads in different blocks**.

   Access. It has the **slowest access** of the three types, but is necessary for large data storage and inter-block communication.

6.3.3 NVIDIA V100 Streaming Multiprocessor (SM)

The NVIDIA V100 is a powerful GPU designed for data centers, primarily used for Artificial Intelligence (AI), High Performance Computing (HPC), and data science. Meanwhile, the **NVIDIA V100 Streaming Multiprocessor (SM)** is a key component of the V100 GPU architecture.

How is architecture composed?

- **Warp Selector and Fetch/Decode**:

  The *Warp Selector* and *Fetch/Decode* units are responsible for **managing the execution of warps** (groups of threads) and **decoding instructions**.

- **Functional Units**:

  - **SIMD fp32 functional unit** (Yellow).

    It **handles single-precision floating-point operations**. Control is shared across 16 units, allowing for 16 multiply-add (`MUL-ADD`) operations per clock cycle. This translates to one 32-wide SIMD operation every two clocks.

  - **SIMD int functional unit** (Orange).

    It **manages integer operations**, also shared across 16 units, with the same performance characteristics as the fp32 unit (16 × `MUL-ADD` per clock).

  - **SIMD fp64 functional unit** (Brown).

    It is **responsible for double-precision floating-point operations**. Control is shared across 8 units, allowing for 8 `MUL-ADD` operations per clock cycle, equating to one 32-wide SIMD operation every four clocks.

  - **Tensor core unit** (Red).

    It is **specialized for tensor operations**, which are crucial for deep learning and AI workloads.

  - **Load/store unit** (Green).

    It **handles memory operations**, such as loading data from and storing data to memory.

- **Warp Scheduler**:

  The diagram below shows the scheduling of warps (Warp 0, Warp 4, Warp 60, etc.) across the functional units, indicating how different warps are processed in parallel.

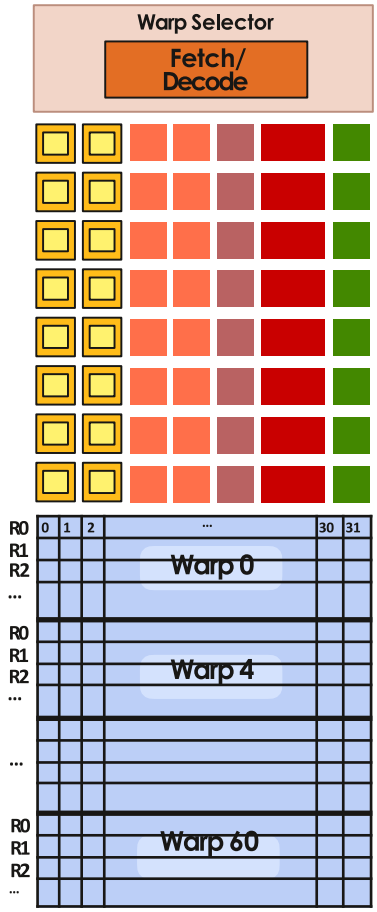


Figure 23: A "sub-core" of the NVIDIA V100 Streaming Multiprocessor (SM) architecture.

What is a Warp?

A **Warp** is a **group of 32 threads that execute the same instruction at the same time**. **Threads within a block are divided into warps**. For example, a block of 256 threads would have 8 warps (256 threads / 32 threads per warp). **Each SM** in the V100 **can schedule and interleave the execution of up to 16 warps**. This means that multiple warps can be executed simultaneously, improving the overall throughput of the GPU. Finally, each warp has some registers to store the data needed by the threads.

How is a Warp executed?

Threads within a warp execute the same instruction simultaneously, taking advantage of **SIMD execution** to improve performance.

When threads within a warp do not share the same instruction, it results in divergent execution, which can degrade performance due to the need to serialize instructions. However, the **check** of the same instruction by the 32 threads is done **dynamically by the GPU hardware**. Finally, although not part of CUDA, understanding warps is critical to optimizing CUDA programs on modern NVIDIA GPUs.

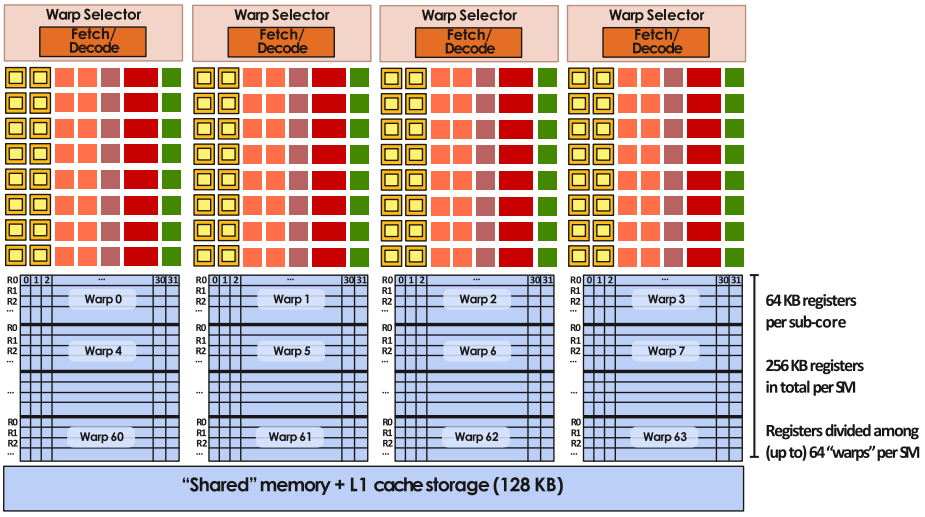


Figure 24: A NVIDIA V100 Streaming Multiprocessor (SM) architecture.

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[2] Guy E. Blelloch, Laxman Dhulipala and Yihan Sun. Introduction to parallel algorithms. https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf, 2024. [Accessed 22-10-2024].

[3] Ferrandi Fabrizio. Parallel computing. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[4] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[5] Johnston Hans. OpenMP by Example. https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf, 2024. [Accessed 23-10-2024].

[6] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[7] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[8] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. ITPro collection. Elsevier Science, 2012.

[9] Microsoft. The critical directive. https://learn.microsoft.com/en-us/cpp/parallel/openmp/a-examples?view=msvc-170#a5-the-critical-directive, 2024. [Accessed 29-10-2024].

[10] M. Nemirovsky and D. Tullsen. *Multithreading Architecture*. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2022.

[11] Wikipedia. Cache (computing) - Wikipedia. https://en.wikipedia.org/wiki/Cache_(computing). [Accessed 20-10-2024].

Index

S

T

W