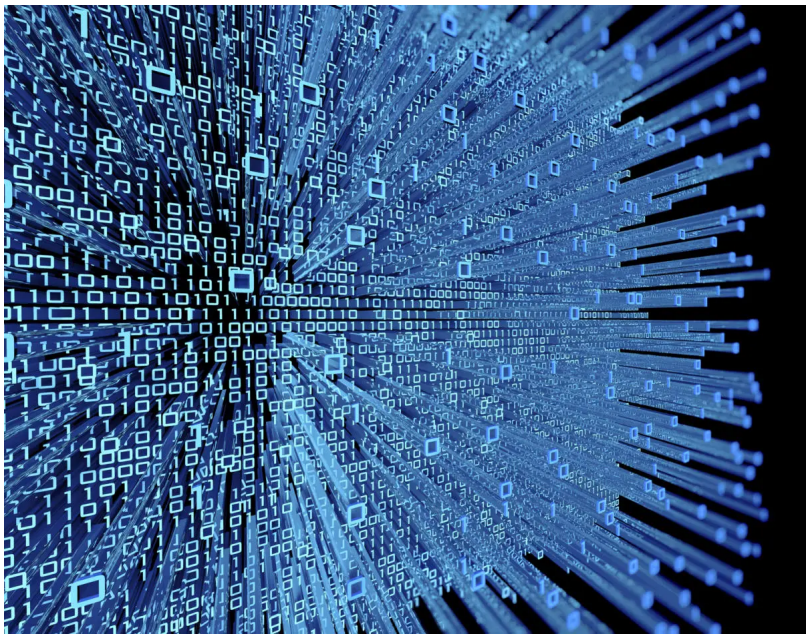


LINMA2710 Scientific Computing

Issambre L'Hermite Dumont
Simon Desmidt

Academic year 2024-2025 - Q2

UCLouvain

Contents

Shared-Memory Multiprocessing

1.1 How memory works

1.1.1 Memory hierarchy

To store the data that it will use, the CPU uses memory. Memory is hierarchical like a pyramid. The higher it is, the faster it goes, but the less space there is.

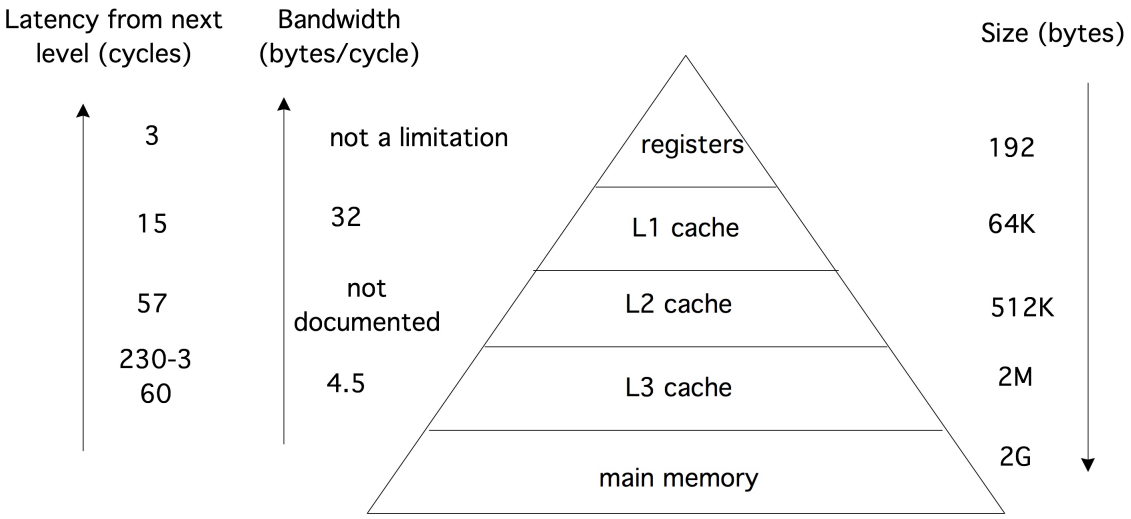


Figure 1.1: Memory hierarchy

First we need to define a cycle. It's an unit of time that is defined like this: $cycle = \frac{1}{\text{CPU freq}}$. For example, for the CPU *AMD Ryzen 5 5600X*, the maximum frequency is $4.6GHz$, so the cycle is $cycle = \frac{1}{4.6GHz} \approx 0.217ns$. The **bandwidth** is the number of bytes that can be transferred in one cycle. And the **latency** is the number of cycles required to access a level of memory. There's also the **latency** but for a number of bytes, its formula is $\alpha + \beta n$ where α is the level latency, β is the inverse of the bandwidth and n is the number of bytes.

Level	Level Latency $[cycle]$	Bandwidth $[bytes/cycle]$	Size	What is stored	Example
Register	3	No limit	$\pm 192B$	"Immediate" data for the CPU	Results of addition, memory address
Cache L1	15	$32 - 64$	$\pm 64KB$	Instructions and "Immediate" data	Local variable
Cache L2	57	$16 - 32$	$\pm 512KB$	Data used recently	Data struct, code part
Cache L3	$230 - 360$	$4 - 10$	$\pm 64MB$	Data shared between core	Global variable
RAM	$300 - 500$	$9 - 50GB/s$	$\geq 4GB$	Running programs	Running software, open document
Disks	$\geq 10^5$	Usually $\leq 3GB/s$	$\geq 128GB$	Persistent data	Document, OS, etc

1.1.2 Caches lines and prefetching

A **cache line** is a small fixed-size contiguous block of memory, usually 64 or 128 bytes. It's not necessarily stored in the cache. We use them to organize the memory, because it is easier to deal with fixed size block. When the CPU need to access a memory location, it loads the entire cache line into the cache of the CPU. If the wanted data is not in the cache, there will be a **cache miss**. After that the CPU will load the entire cache line into the cache.
The **prefetching** is the fact that the CPU will load the cache line that is next to the one that is needed. It's because of the spacial locality. Spacial locality is the reason why we use cache lines. For example, if we store an array of data, it may use some space greater than one cache line so for precaution, the CPU will load the next cache line too. And so we save time, by anticipating.
The data locality is important for at least the two following reasons:

- **Temporal locality**: If a data is used frequently, we will keep it in the cache.
- **Spacial locality**: If a data is used, the data next to it could be useful too.

1.1.3 Arithmetic intensity

The **arithmetic intensity** is a concept in performance analysis for memory-bound and compute-bound programs. Let's consider a program that does o arithmetic operations and m memory operations, we define:

- **Arithmetic intensity**:
$$a = \frac{o}{m} \tag{1.1}$$
It helps find if the program is limited by a compute-bound or memory-bound.

- **Arithmetic time**:
$$t_{arith} = \frac{o}{\text{CPU freq}} \tag{1.2}$$
It's the time needed to perform o operations.

- **Memory transfer time**:
$$t_{mem} = \frac{m}{\text{bandwidth}} = \frac{o}{a \times \text{bandwidth}} \tag{1.3}$$
It's the time needed to do m memory operations.

The overall performance of a program is thus defined by the worst component of the PC, and so we get the **time per iteration**:

$$\min\left(\frac{t_{arith}}{o}, \frac{t_{mem}}{o}\right) \tag{1.4}$$

With some algebra, we can find the **number of operations per second**:

$$\max\left(\text{CPU freq}, a \times \text{bandwidth}\right) \tag{1.5}$$

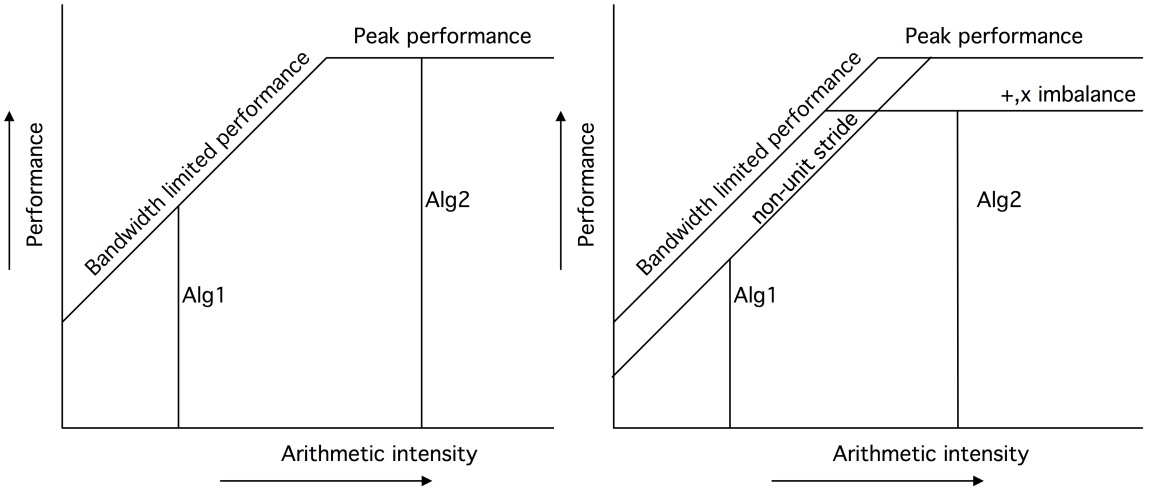
To resume:

- If a is low then the performance is memory-bounded $\Rightarrow$ **Need cache optimization**.
- If a is high then the performance is compute-bounded $\Rightarrow$ **Need more CPU cores or vectorization**.

The arithmetic intensity can be represented with the **roofline model** which we'll talk now.

1.1.4 The roofline model

The two following graphs link the performances of a program with the arithmetic intensity. The "performances" means the number of operations per second and the "arithmetic intensity" is the number of operations per byte of memory transferred. The left graph is the case where we consider a perfect usage of the PC, and the right graph is the case where we consider a bad usage of the PC.



(a) Roofline model (b) Model with worse performances

As we can clearly see, we have two limitations:

- Bandwidth limit (slopped line)
- Computing limit (flat line)

These two limits lines are upper bound for performances and are limited from above by the hardware (we can't do better than what our CPU can do), but they can be lower if the program is not optimized. For example the slopped line can be lower if we don't use the cache correctly. The flat line can be lower if we don't use the CPU correctly (e.g. not using SIMD).

1.1.5 Cache hierarchy for a multi-core CPU

In the case where we have a multi-core CPU, the organization of the memory isn't always the same, it depends of the CPU architecture. For example, we can have a hierarchy like this:

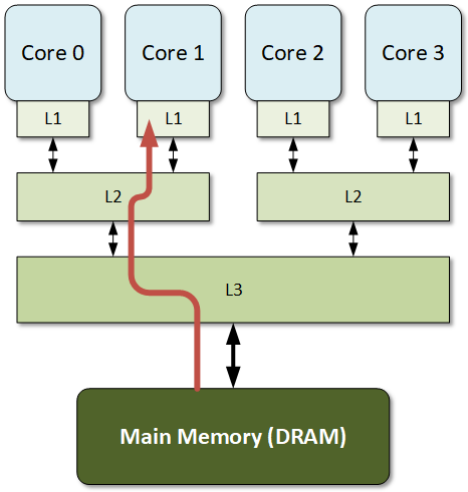


Figure 1.3: Cache hierarchy for a multi-core CPU

Here the L1 cache is private to each core, the L2 cache is shared between two cores and the L3 cache is shared between all cores, as well as the RAM.

1.2 Parallelism

In this course, we use *OpenMP* to parallelize our code instead of *lpthread*. The advantages of *OpenMP* are detailed in the chapter **useful tools** at the section 5.1.
Parallelism is a very useful tool to optimize the performances of a program. It allows to use multiple cores of the CPU to do multiple tasks at the same time. But sometimes it may arise some problems of performances or computational error.

1.2.1 computational error

When using multi-threading, we may modify and use some variables at the same time which could cause computational error. So to avoid that we can either protect the variable (with some adapted tools) but this option is not very efficient because it takes some time to *lock* and *unlock* and to wait for the variable to be available. Or we can make each thread use its own version of the variable, this introduces the next problems.

1.2.2 Race condition

The race condition is a problem that occurs when two or more threads try to modify the same variable at the same time, here there are no computational error possible. For example if all the threads want to modify a variable a the same time, they will need to wait for the variable to be available, and it is a waste of time. The solution for that, as we said before is to make each thread use its own version of the variable. Let's say they need to compute the sum of an array. The solution is to make each thread compute the

sum of a part of the array and then by "reduction" sum all the sums. We will see this in more detail in the next chapter.

1.2.3 False sharing

As we say before, in the section about cache lines 1.1.2, the CPU loads the entire cache line into the cache when it need to use some data that is on the cache line. So here is the problem, let's say that all of the partial sum of the array are stored in the same cache line, when a thread modify its partial sum, the entire cache line is loaded into the cache of the CPU, and so the other threads will have to wait for the cache line to be available. The solution is to make each thread use its own cache line. We can solve this by adding some padding (shift the data) to the data structure.

1.3 Amdahl's law

First let's define the **speedup** of a program, it represents the improvement of running time while using p threads. It is defined by:

$$S_p = \frac{T_1}{T_p} \tag{1.6}$$

Where T_1 is the time needed to run the program with one thread and T_p is the time needed to run the program with p threads.
Then let's define the **efficiency** of a program, is mesures how well the parallelization is done. It is defined by:

$$E_p = \frac{S_p}{p} \tag{1.7}$$

Where S_p is the speedup of the program with p threads. We get 3 cases:

- If $E_p = 1 \Rightarrow$ **Ideal case**, the parallelization is perfect.
- If $E_p < 1 \Rightarrow$ **Realistic case**, there's some inefficiency.
- If $E_p > 1 \Rightarrow$ **Unrealistic case**, would imply a super-linear speedup.

We can now define **Amdahl's law**, which explain the limitation of parallelization due to the presence of a sequential portion in a program. It is defined like this:

$$T_p = F_s T_1 + \frac{(1 - F_s)T_1}{p} \tag{1.8}$$

F_s is the percentage of the program that is sequential (impossible to parallelize). With that we can redefine the **speedup** and the **efficiency** of a program:

$$S_p = \frac{T_1}{F_s T_1 + \frac{(1-F_s)T_1}{p}} = \frac{1}{F_s + \frac{1-F_s}{p}} \Rightarrow \lim_{p\to\infty} S_p = \frac{1}{F_s} \tag{1.9}$$

And

$$E_p = \frac{S_p}{p} = \frac{1}{pF_s + 1 - F_s} = \frac{1}{F_s(p-1) + 1} \tag{1.10}$$

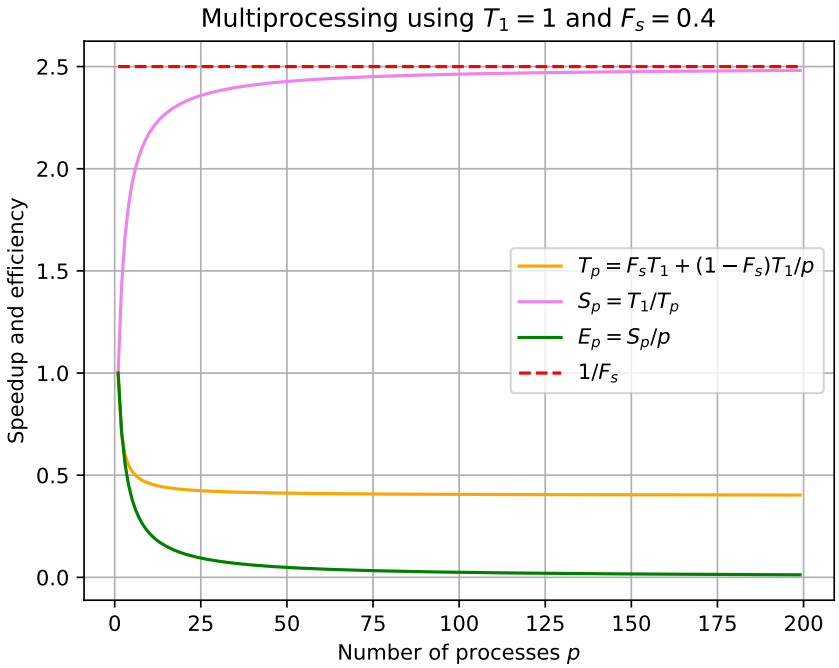


Figure 1.4: Amdahl's law

1.3.1 Application of Amdahl's law to parallel sum

Let's consider the sum of an array, we want to parallelize it. The time without parallelization is n and with parallelization is $\frac{n}{p} + \log_2(p)$, the log term is the time needed to sum all the partial sums. So the speedup is:

$$S_p = \frac{n}{\frac{n}{p} + \log_2(p)} = \frac{1}{\frac{1}{p} + \frac{\log_2(p)}{n}} \tag{1.11}$$

And the efficiency is:

$$E_p = \frac{S_p}{p} = \frac{1}{1 + \frac{p}{n}\log_2(p)} \tag{1.12}$$

We clearly see that if we use more than n processes the efficiency will decrease. Because if $p \geq n$ then we have $T_p = \log_2(n)$ and $\lim_{s\to\infty} S_p = \frac{1}{\log_2(n)}$ and $F_s = \log_2(n)$

Single Instruction Multiple Data (SIMD)

Maybe to improve

SIMD (Single Instruction Multiple Data) is a parallel computing architecture used to process multiple data points simultaneously with a single instruction.
The idea of SIMD is that instead of executing the same instruction separately for each data element, SIMD processes multiple elements in parallel within a single CPU cycle.
To simplify SIMD takes advantages of **vectorization**, to optimize the performances of the program. It uses register to store the vector on which it does the computing.
Let's do a little example, consider the following code:

```
int a[] = {...};
int b[] = {...};
int c[size];
for(int i = 0; i < size; i++){
  c[i] = a[i] + b[i];
}
```

Here we use `int` which is 32 bytes so if we use SIMD with AVX2 registers, we can store $(512/32 =)16$ integers in a register. And so SIMD with that type of registers will automatically translate the previous code to:

```
int a[] = {...};
int b[] = {...};
int c[size];
for(int i = 0; i < size; i+=16){
  __m512_add_epi16(c+i, a+i, b+i);
}
```

We can write the code with the intrinsics (`__m512_add_epi16`) but it's not necessarily. The compiler will do it for you.
To use SIMD, you need to give to your compiler some flags, for example:

- `-O2` or `-O3` for optimization
- `-mavx` for AVX
- `-mavx2` for AVX2
- `-mavx512` for AVX512

TODO: explain AVX

Distributed Computing with MPI

3.1 Single Program Multiple Data (SPMD)

Single Program Multiple Data means that we make multiple processes execute the same program but operate on different data. It's a kind of parallel computing that we call distributed computing.

3.1.1 Message Passing Interface (MPI)

MPI is a standardized and portable message-passing system used to enable parallel computing across multiple nodes. Here are some key steps with MPI:

- Initialization: `MPI_Init(&argc, &argv);`, initialize MPI and the processes.
- Getting the number of processes: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs);`, determines how many processes are running (`nprocs` is the same for all processes.)
- Getting the rank of the process: `MPI_Comm_rank(MPI_COMM_WORLD, &procid);` Give a unique ID (rank) to each process, which allows them to perform different tasks despite running the same program.
- Finalizing MPI: `MPI_Finalize();`, finalize MPI and free the ressources.

3.2 Collectives

Lets consider a system that can support 4 processes for the explcit examples (tables). And for the bounds consider as usual p proccess and the variables defined for the latency 1.1.1 (α, β, n) and an arithmetic intensity parameter (γ). Here is some illustration of how collectives work:

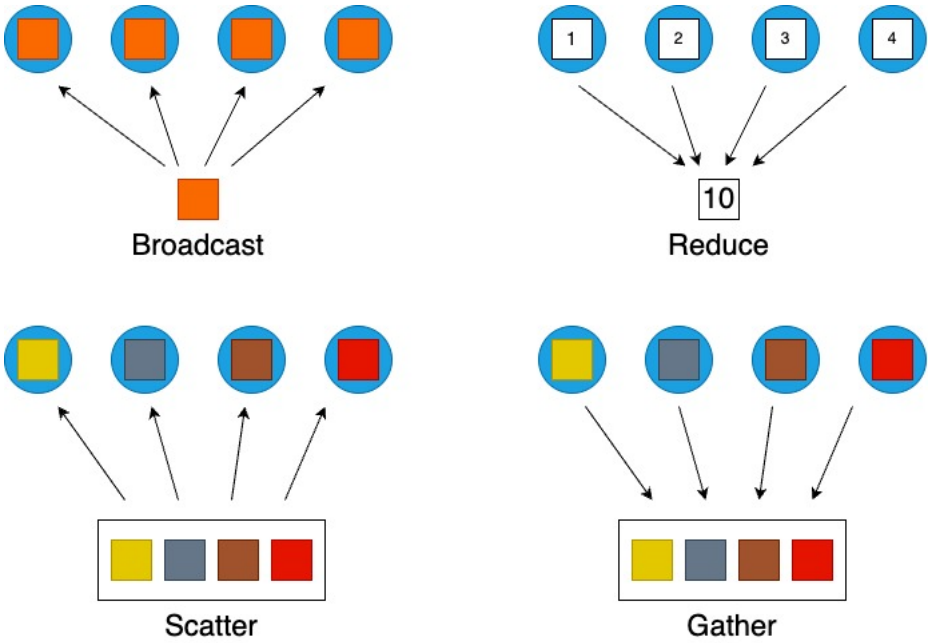


Figure 3.1: Collectives

TODO: give the function for each and the args

3.2.1 Broadcast

Broadcast makes a process gives a variable to the other processes.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x						x	x	x	x

Using *spanning tree algorithm*, we can **broadcast** in $\log_2(p)(\alpha + \beta n)$.

3.2.2 Gather

Gather makes a process gather the variables of the other processes.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0						x_0			
		x_1					x_1			
			x_2				x_2			
				x_3			x_3			

Using *spanning tree algorithm*, we **gather** in $\log_2(p)\alpha + \beta n$.

3.2.3 Reduce

Reduce sums up all the values of the variables on the differents processes and store the sum on one process.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0	x_1	x_2	x_3			$\sum_{i=0}^{3} x_i$			

Using *spanning tree algorithm*, we can **reduce** in $\log_2(p)(\alpha + \beta n) + \log_2(p)\gamma n$. To simplify we have: $\log_2(p)(\alpha + (\beta + \gamma)n)$.

3.2.4 All gather

All gather is a sort of combination of **gather** and **broadcast**. It gathers all the variables of the processes and broadcast them to all the processes but more efficiently.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0						x_0	x_0	x_0	x_0
		x_1					x_1	x_1	x_1	x_1
			x_2				x_2	x_2	x_2	x_2
				x_3			x_3	x_3	x_3	x_3

Using *spanning tree algorithm*, we can **all gather** in $\log_2(p)\alpha + \beta n$.

3.2.5 Reduce scatter

Reduce scatter is a sort of combination of **reduce** and **scatter**. It reduces the variables of the processes and scatter them to all the processes, but more efficiently.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$			$\sum_{j=0}^{3} x_{0,j}$			
	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$				$\sum_{j=0}^{3} x_{1,j}$		
	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$					$\sum_{j=0}^{3} x_{2,j}$	
	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$						$\sum_{j=0}^{3} x_{3,j}$

Using *spanning tree algorithm*, we can **reduce scatter** in $\log_2(p)\alpha + (\beta + \gamma)n$

3.2.6 All reduce

All reduce is a sort of combination of **reduce** and **broadcast**.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0	x_1	x_2	x_3			$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$

We explained above that we can do a combination of **reduce** and **broadcast**, but we can **all reduce** more efficiently doing **reduce scatter** then **all gather**, and we can do it in $\log_2(p)\alpha + \beta n$.

3.3 Point-to-point communication

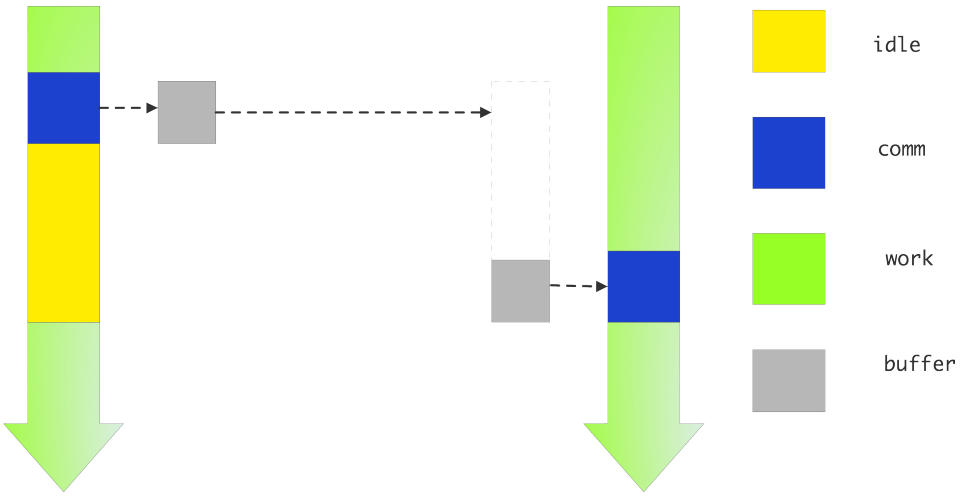


Figure 3.2: Point-to-point blocking communication

This figure shows the problem of point-to-point blocking communication in MPI. As we can see, the left process is waiting for the right process to send the data, and thus is idling and wasting time. It does that because the sender cannot continue until the message is completely transferred. And in the case of a too big message that can't be stored in the buffer then the sender must wait for the receiver.
There is two ways to solve this problem:

- **Rendezvous protocol**: It works like an handshake process, so we have:
  - The sender sends a header to notify the receiver that he will send data.
  - The receiver replies with a "ready-to-send" signal.
  - The sender sends the actual data.
- **Eager protocol**: The message is quickly buffered in the sender's buffer to allow `MPI_Send()` to finish "eagerly" and when the receiver is ready, it can receive the message. The sender doesn't wait for the receiver to be ready. It's useful when the message is small and thus fit in the buffer.

The eager protocol can be represented like this:

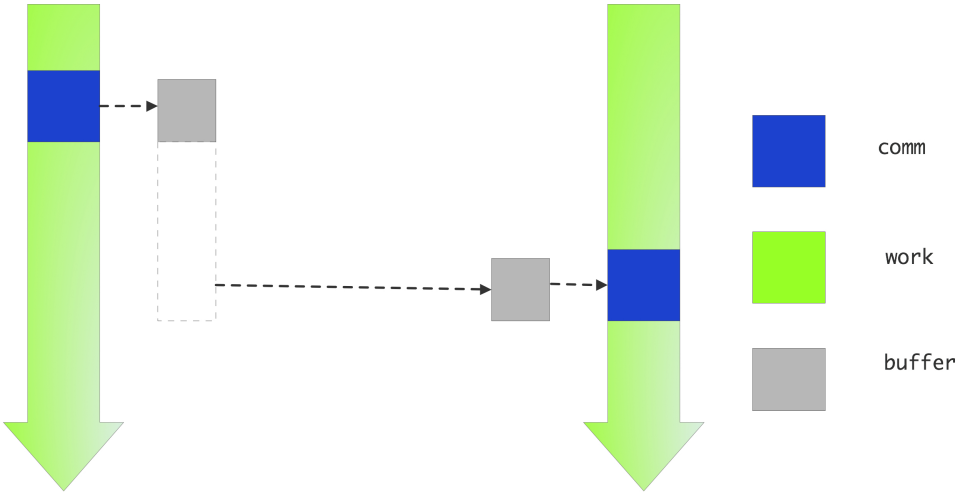


Figure 3.3: Eager protocol

Scientific computing on GPU

4.1 What is a GPU?

4.1.1 GPU vs CPU

A **GPU** (Graphics Processing Unit) is simply a CPU but specialized for parallel computing and is commonly used for matrix computations. In order to understand why let's do a little comparison between CPU and GPU:

	CPU	GPU
Purpose	brain of the PC	graphic rendering
Number of Cores	a few $2-32$	thousands
Processing	serial work	parallel work
Memory (global)	RAM $(8-64+GB)$	VRAM $(4-24+GB)$
Memory (local)	cache L1, L2, L3 (cf 1.1)	$32-128KB$ per block
Frequency	high $(3-5GHz)$	low $(1-2GHz)$
Flexibility	versatile	specialized

So to summarize, the CPU is a group of a few powerful worker that do different work each while the GPU have an army of workers that do the same simple task in parallel to contribute to a greater goal.
GPU are used commonly for graphics rendering, but it's not the only thing that they can do, we can use them for any kind of parallel computing. For example: machine learning, deep learning, simulations, etc. To do that we can run code using different "translations platforms" like **CUDA** (NVIDIA only), **ROCm** (AMD only) or open source alternative like **OpenCL**.

4.1.2 Hierarchy

The hierarchy is pretty simple, we have the host that send the jobs to the rest of the CPU and GPU. Some computing devices, all of the cores of the CPU are managed as one, and each GPU is an independant computing device. After that we have the computing units. Typically, for a CPU each core is a computing unit, and for a GPU each core is a block of processors that can run multiples threads at the same time. And at the lowest level is the processing elements, it is ab ALU (Arithmetic Logic Unit), to simplify it's a single thread that does basic operations.

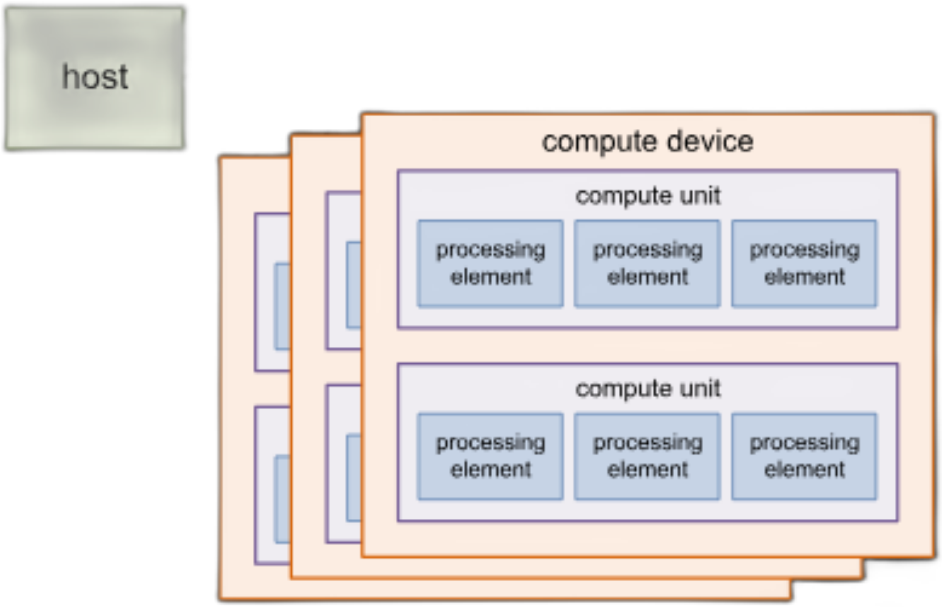


Figure 4.1: Hierarchy

And we can get informations from all of those levels by the following functions:

compute device	compute unit	processing element
get_global_id	get_group_id	get_local_id
get_global_size	get_num_groups	get_local_size

4.1.3 Memory hierarchy

The memory hierarchy follows the same principle as the hierarchy. The host has its own memory (RAM). The global memory is the memory of the GPU, it is shared between all the threads of the GPU but is "slow" (VRAM). The local memory is shared between all the threads of a computing unit, and is faster than the global memory. And finaly the private memory is the memory of a thread, it is the fastest memory. The size of the memory is decreasing as its locality increases.

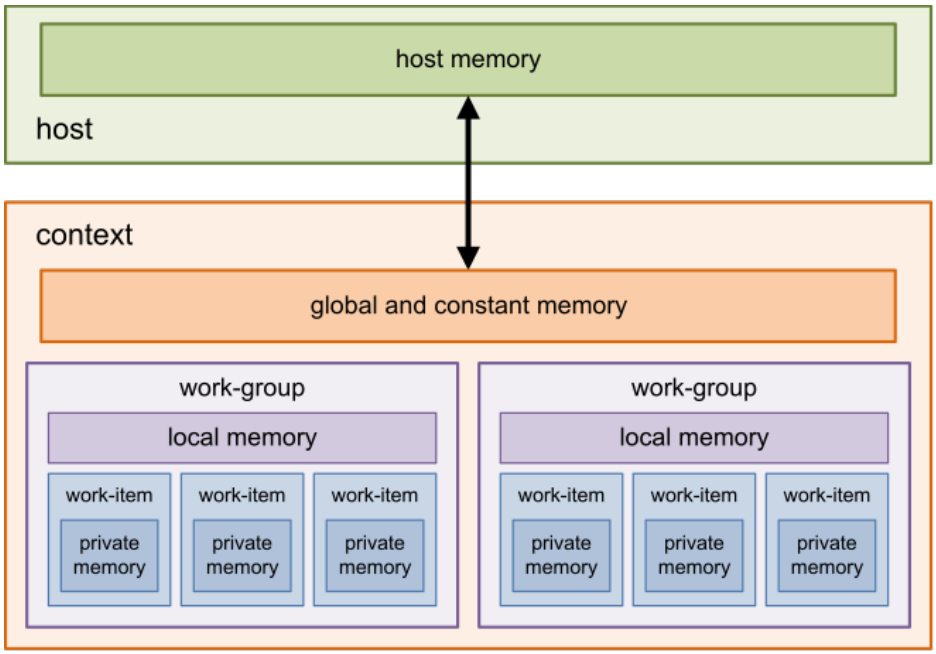


Figure 4.2: Memory hierarchy

Useful tools

5.1 OpenMP

TODO: Improve
OpenMP is a library that allows to parallelize code, it is better than *lpthread* for multiple reasons:

- Easy to add to an existing code, because of the compilation directives (`#pragma omp`). It's an implicit gestion of the threads unlike *lpthreads* that require an explicit gestion. It allows to parallelize a code without changing the whole structure of the code.
- Automatic gestion of the threads, for example if you want to parallelize a loop, you don't have to create the threads, the library will do it for you and optimize the number of threads.
- Simplified synchronization with tools. *OpenMP* provides tools to synchronize the threads like `critical`, `atomic`, `barrier`, etc. They help to protect the shared data between the threads.
- Optimized use of the cache
- Compatible with *SIMD*
- Portability: it is compatible with *Windows*, *Linux*, *MacOS*. With multiple CPU architectures like *Intel*, *AMD*, *ARM*, etc.

5.2 Slurm

5.2.1 What is Slurm?

Slurm stands for *Simple Linux Utility for Resource Management*. It is an open-source job scheduler used in high-performance computing environments, such as university clusters or research supercomputers.

5.2.2 How to use Slurm?

5.2.3 Slurm Topology

5.3 OpenCL

PAA - PDEs

6.1 Introduction

Throughout this chapter, we will use the simple example of the diffusion equation. It is expressed as

$$\partial_t u(x,y,z,t) = \kappa \nabla^2 u(x,y,z,t) + f(t,x,y,z) \tag{6.1}$$

To that, we add an initial condition and boundary conditions:

- Initial condition: $u(0,x,y,z) = u_0(x,y,z)$;
- General boundary condition: $n \cdot \nabla u = 0$ on Ω;
  - Dirichlet boundary condition: $u(t,0)$ given for all $t \geq 0$;
  - Neumann boundary condition: $\partial_x(t,0)$ given for all $t \geq 0$;

This equation can represent the diffusion of heat, particles, pollutants, etc. It is derived from two basic laws of physics:

- Fourier law for heat: $\vec{q} = -\kappa \nabla u$;
- Conservation law: $\partial_t u = -\partial_x \vec{q}$;

6.2 Finite differences methods

We will now consider more specifically the 1D case:

$$u_t(t,x) = u_{xx}(t,x) \qquad x \in [0,1] \qquad t \geq 0 \tag{6.2}$$

with conditions

$$\begin{cases} u(t,0) = u(t,1) = 0 \quad \forall t \\ u(0,x) = u^0(x) \end{cases} \tag{6.3}$$

There is usually no analytical solution to the equation, and this is why we can use numerical method to find the solution.

6.2.1 The θ-method

In finite differences, we discretize the domain in a mesh, like displayed on 6.1

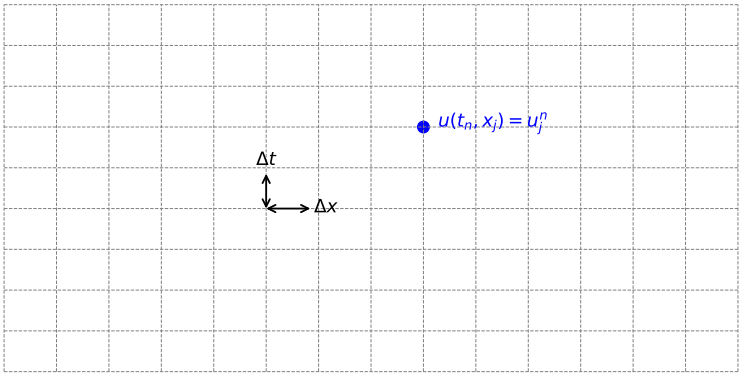


Figure 6.1: Discretization of the domain

The θ-method consists in taking a linear combination of two approximations of the derivatives through finite differences:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = (1-\theta)\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} + \theta\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{\Delta x^2} + T_j^n \tag{6.4}$$

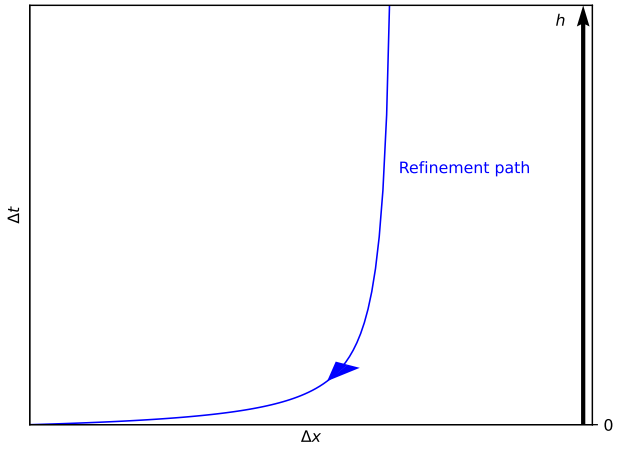
Where u_j^n denotes the exact value of the function $u(\cdot)$ at the given point, and T_j^n is called the truncation error.
Using a truncation, i.e. using approximate values and deleting the truncation error term, the formula is

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = (1-\theta)\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{\Delta x^2} + \theta\frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{\Delta x^2} \tag{6.5}$$

6.2.2 Analysis principles of FD methods

To be valid, a finite differences method needs to verify some basic principles:

- Consistency along a refinement path: $T_j^n \xrightarrow{h \to 0} 0$;



- Accuracy: $|T_j^n| \leq c(\Delta t^p + \Delta x^q)$, for all $\Delta t, \Delta x$ small, c a constant and p, q some accuracy parameters;
- Stability along a refinement path: For all solutions v, w of the method and for all $t_F > 0$, $\exists K_{t_F}$ such that $\|V^n - W^n\| \leq K_{t_F}\|V^0 - W^0\|$ for all n such that $n\Delta t \leq t_F$, for t_F the biggest time value;
- Lyapunov stability;
- Convergence along a refinement path: $U_j^n \xrightarrow{h\to 0} u(t,x)$ when $j\Delta x \to x$ and $n\Delta t \to t$.

6.2.3 Value of θ

$\theta = 0$**: FTCS (forward time, centered space)**

The formula becomes, when defining $\mu := \kappa\Delta t/\Delta x^2$,

$$U_j^{n+1} = \mu U_{j+1}^n + (1-2\mu)U_j^n + \mu U_{j-1}^n \tag{6.6}$$

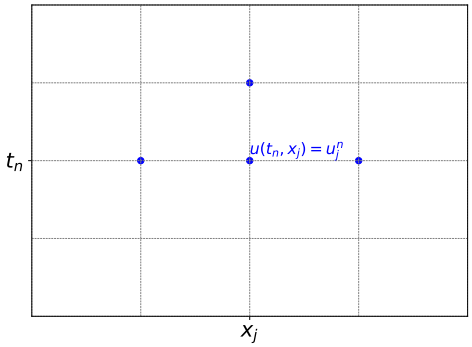


Figure 6.2: Points used in the FTCS scheme.

$\theta = 1$**: BTCS (backward time, centered space)**

The formula becomes

$$U_j^n = -\mu U_{j+1}^{n+1} + (1+2\mu)U_j^{n+1} - \mu U_{j-1}^{n+1} \tag{6.7}$$

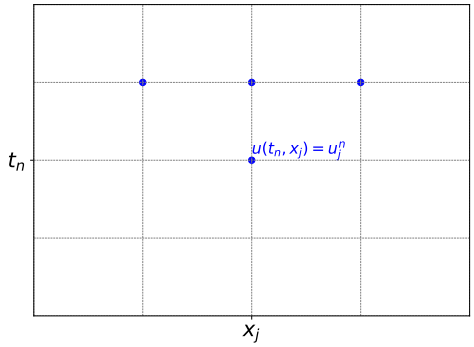


Figure 6.3: Points used in the BTCS scheme.

$\theta = 1/2$**: Crank-Nicholson**

Crank Nicholson uses both the backward and forward time specifications. The formula is determined by replacing the value of θ but is tedious to write. The points it uses are displayed on figure 6.4

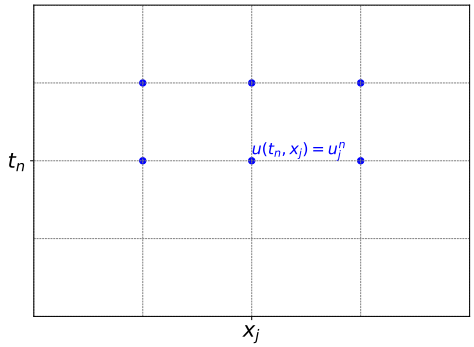


Figure 6.4: Points used in the Crank-Nicholson scheme.

6.2.4 Truncation error

By definition, the truncation error term is given by

$$T = \frac{\delta_t u_j^{n+\frac{1}{2}}}{\Delta t} - \frac{\theta\delta_x^2 u_j^{n+1} + (1-\theta)\delta_x^2 u_j^n}{\Delta x^2} \tag{6.8}$$

where we define $\delta_t u_j^{n+\frac{1}{2}} = u_j^{n+1} - u_j^n$ and $\delta_x^2 u_j^n = u_{j+1}^n - 2u_j^n + u_{j-1}^n$. Using a Taylor development series for each of the u_j^n, with reference point $(t_{n+\frac{1}{2}}, x_j)$, we can show that

$$T = \left(\left(\frac{1}{2} - \theta\right)\Delta t - \frac{1}{2}\Delta x^2\right) u_{xxxx} - \frac{1}{12}\Delta t^2 u_{ttt} + \cdots = \mathcal{O}(\Delta t, \Delta x^2) \tag{6.9}$$

And for Crank-Nicholson, this becomes $\mathcal{O}(\Delta t^2, \Delta x^2)$.

6.2.5 Fourier stability analysis

From equation (6.5), we see that U^{n+1} depends linearly in U^n and we can make the guess that the solution will thus have the form

$$U_j^n = (\lambda(k))^n e^{ikj\Delta x} \qquad k \in \mathbb{R} \tag{6.10}$$

Replacing in equation (6.5) and solving for $\lambda(k)$, we get

$$\lambda(k) = \frac{1 - 4\mu(1-\theta)\sin^2\frac{k\Delta x}{2}}{1 + 4\mu\theta\sin^2\frac{k\Delta x}{2}} \tag{6.11}$$

By the discrete Fourier transform, every solution can be written

$$U_j^n = \sum_k c_k(\lambda(k))^n e^{ikj\Delta x} \tag{6.12}$$

We know that, if $|\lambda(k)| \leq 1$ for all k, then the θ-scheme is Lyapunov stable. This condition is equivalent to

$$\mu(1-2\theta) \leq \frac{1}{2} \tag{6.13}$$

$\rightarrow$ Note: this property is sufficient but not necessary.

Let us now define the matrix

$$C(\beta) = \begin{bmatrix} 1-2\beta & \beta & \ddots & & \\ \beta & 1-2\beta & \beta & \ddots & \\ 0 & \beta & 1-2\beta & \beta & \\ & & \ddots & \ddots & \ddots \\ & & \beta & 1-2\beta & \beta \\ & & & \beta & 1-2\beta \end{bmatrix} \in \mathbb{R}^{J-1\times J-1} \tag{6.14}$$

The θ-scheme can be written in matrix form:

$$\underbrace{C(-\mu\theta)}_{=:B_1} U^{n+1} = \underbrace{C(\mu(1-\theta))}_{=:B_0} U^n \iff U^{n+1} = \underbrace{(C(-\mu\theta))^{-1}C(\mu(1-\theta))}_{=B_1^{-1}B_0=:A} U^n \tag{6.15}$$

We can show that the eigenvalues of $C(\beta)$ and their corresponding eigenvectors are

$$v_\ell = 1 - 4\beta\sin^2\frac{\pi\ell}{2J} \qquad x_\ell = \begin{bmatrix} \sin\frac{\pi\ell}{J} \\ \vdots \\ \sin\frac{\pi\ell(J-1)}{J} \end{bmatrix} \qquad \ell = 1,\dots,J-1 \tag{6.16}$$

Hence the eigenvalues of the matrix A are

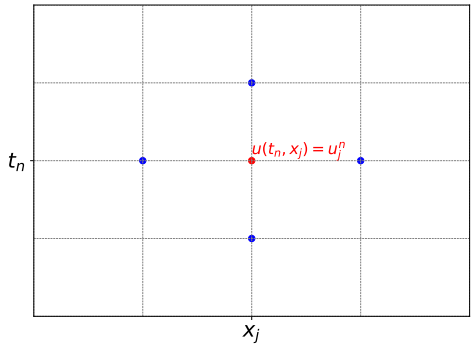
$$v_e ll = \frac{1 - 4\mu(1-\theta)\sin^2\frac{\pi\ell}{2J}}{1 + 4\mu\theta\sin^2\frac{\pi\ell}{2J}} \qquad \ell = 1,\dots,J-1 \tag{6.17}$$

Their module must be smaller than 1 for the system to be Lyapunov stable (for all $\ell = 1,\dots,J-1$). This is equivalent to the condition

$$\frac{1 - 4\mu(1-\theta)\sin^2\frac{\pi(J-1)}{2J}}{1 + 4\mu\theta\sin^2\frac{\pi(J-1)}{2J}} \geq -1 \tag{6.18}$$

6.2.6 Dufort-Frankel scheme

$$U_j^{n+1} = U_j^{n-1} + 2\mu \left(U_{j-1}^n + U_{j+1}^n - U_j^{n+1} - U_j^{n-1} \right) \tag{6.19}$$



Let us analyse the stability, given a solution of the form $U_j^n = \lambda^n e^{ikj\Delta x}$, which gives us the values

$$\lambda_\pm = \frac{2\mu \cos(k\Delta x) \pm \sqrt{1 - 4\mu^2 \sin^2(k\Delta x)}}{1 + 2\mu} \tag{6.20}$$

We can show through the analysis of the λ that this method is unconditionnaly stable, but not unconditionnaly consistent.

→ Note: a scheme taking Dufort-Frankel with the additional point (t_n, x_j) is unconditionnally unstable.

6.3 Finite volume methods

Let us study the Poisson equation, written in the following form

$$\nabla \cdot (\kappa(x) \nabla u) = f \quad \text{in} \quad \Omega \subseteq \mathbb{R}^n \tag{6.21}$$

6.3.1 Voronoi cells

The idea of finite volumes consists in creating cells given some vertices $\{x_i\}_i \subset \Omega$. The Voronoi cells are defined as

$$V_i = \{x \in \Omega : \ \|x - x_i\| \leq \|x - x_j\|, \ \forall j \neq i\} \tag{6.22}$$

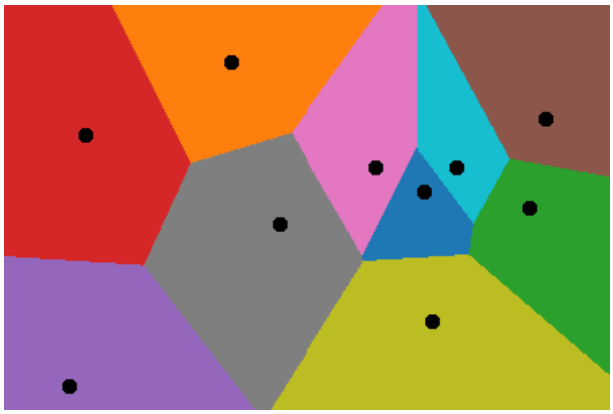


Figure 6.5: Voronoi cells.

The method is the following: by integrating over each cell, we can then use the divergence theorem to approximate using finite differences.

$$\begin{aligned} \int_{V_i} \nabla \cdot (\kappa(x)\nabla u)dx &= \int_{V_i} f dx \\ \int_{\partial V_i} \kappa(x)\nabla u \cdot n ds &\approx f_i Vol(V_i) \\ \sum_{j\sim i} K_{ij} \underbrace{\frac{u_j - u_i}{\|x_j - x_i\|}}_{=:h_{ij}} \ell_{ij} &\approx f_i Vol(V_i) \end{aligned} \tag{6.23}$$

where the notation $j \sim i$ denotes the neighbours j of point i, $K_{ij} = \kappa\left(\frac{x_i+x_j}{2}\right)$, ℓ_{ij} is the length of the interface between cells i and j. This gives the finite volume scheme:

$$\sum_{j\sim i} K_{ij} \frac{U_j - U_i}{h_{ij}} \ell_{ij} = Vol(V_i) f_i \qquad \forall i \tag{6.24}$$

6.3.2 Boundary conditions

- Dirichlet: if the condition is $u(x) = g(x)\ \forall x \in \partial\Omega$, we write for points x_k on the boundary $U_k = g(x_k)$;
- Neumann: the condition is $\nabla u \cdot n = g$ on $\partial\Omega$ and we use it in the form

  $$\int_{\partial V_i} \kappa(x)\nabla u \cdot n ds \approx \sum_j K_{ij} \frac{u_j - u_i}{h_{ij}} \ell_{ij} + \int_p \kappa(x)\nabla u \cdot n ds,$$

  where p is the boundary of cell V_i, and the integral can be approximated by quadrature.

6.3.3 Analysis of principles

The analysis of principles is complicated using finite volumes. However, here are the results:

- Consistency: finite volume schemes are not consistent in general;
- Convergence: yes;

6.4 Summary

	+	-
Finite differences	Easy to understand and program	Cumbersome for non-rectangular domains or non-Dirichlet boundary conditions
Finite volumes	Works for arbitrary geometries and meshes	Higher order methods are difficult to obtain
Finite elements	Works for arbitrary geometries and meshes; Higher order methods exist	More complicated to implement

Table 6.1: Comparison of numerical methods.

6.5 Hyperbolic equations in 1 space dimension

6.5.1 Linear advection equation

$$u_t + a u_x = 0 \qquad u(x,t) = u^0(x) \tag{6.25}$$

6.5.2 CFL condition

Consider the finite differences scheme:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + a \frac{U_j^n - U_{j-1}^n}{\Delta x} = 0 \tag{6.26}$$

which can also be written

$$U_j^{n+1} = (1-\nu) U_j^n + \nu_{j-1}^n \qquad \nu := \frac{a \Delta t}{\Delta x} \tag{6.27}$$

The CFL condition states that ν must be smaller than 1. If the scheme is convergent, then the condition holds. The meaning of the CFL condition is that the domain of dependence of the PDE, i.e. all the points that depend on U_j^n, is a subset of the domain of dependence of the numerical scheme.