# High Performance Computing - Notes

Author: Kevin Cattaneo

The following is a summary of the slides of HPC Master course of Computer Science (6 CFU). This summary involves a flow of thoughts that has been translated into a schema.

# Introduction

Before: we have big energy consumption computers
**Parallel computers**: used for solving problems and using wider resources over a network.
**Embarrassing parallel problems**: simple case to parallelize, like graphical rendering, event simulation, brute force search
Top 500 best architectures

**Speedup**: S(n,p) = T(n,1)/T(n,p) and 0<S<p
**Efficiency**: E(n,p) = S(n,p)/p and 0<E<1
**Amdhal law**:

- 1/(1-p) max speedup with p portion of code parallelizable
- 1/((p/N)+S) max speedup with p number of processors and S serial portion of code

Amdhal was optimistic: we don't achieve so beautiful results, we have parallel process overhead
Amdhal was pessimistic: superlinear speedup is not so rare if we exploit RAM

# Single Processing Computing

Before: Von Neumann architecture with CPU and Memory
We measure performance in terms of CPU **clock cycles**
The more transistors we fit on a chip, the more the performance we obtain.
Limits of **Moore law**: physical limits

- the minimum dimension of a transistor is the atom dimension
- we have temperature limits that materials can stand

**Hyperthreading**: running more threads within the same CPU
**Multicore**: running more CPUs on same chip
**Cache**: L1 (smallest, fastest), L2, L3 (wider, slowest)
**Cache line** is the minimum transfer unit
**Instruction Level Parallelism (ILP):** we can achieve optimization with **pipelining:** starting the next instruction while the corresponding CPU part is idle.
Possible hazard:

- **structural**: resource conflicts – we can solve with **superscalar execution** (executing 2 or more instructions phase in a clock cycle)
- **control flow**: changes in program behavior – can solve with **branch prediction** and **speculative execution**
- **data dependency**: can solve with **out of orders instructions** (scheduling in arbitrary order)

Before: CPU follows the control flow
Now: CPU follows data flow, adapting with optimization + solutions to hazard seen before
**Simultaneous Multi Threading (SMP)**: having a long pipeline is a problem, so here comes parallelization on multiple cores, exploiting different threads where one leaves parts of cache non-used and so usable by other threads.
**Latency** (alpha): delay between request of processor for data and the actual data arriving
**Bandwidth** (beta): rate with which data arrives after the initial delay
Time(n data) = alpha + beta*n
To achieve performance for cache I need to look for:

- **temporal locality**: access same data frequently
- **spatial locality**: access data near (e.g. matrix swap)

**Cache miss**: is the missing of data from cache (at least 1, the initial missing of data from cache)
**Conflict miss**: when one requested data is mapped in some location that have been evicted since there were no other candidates to evict as cache lines

# Level of parallelism

**Data parallelism**: same operation/instruction to many data elements
**Task parallelism**: same program in parallel to many data
Architectures: **SISD**, **SIMD** (ILP), **MIMD**:

- usually SIMD → **SPMD** (single program executed by several CPUs on multiple data), different control flow
- MIMD → usually used in parallelization of nodes

Double operands register widths:

- **SSE** = 2 operands
- **AVX** = 4 operands
- **AVX512** = 8 operands

**FLOPS** = instruction/cycle * operation/instruction * FLOP/operation
MIMD architecture: memory+socket < node < blade < chassis
Shared memory between different CPU:

- **Unified Memory Access (UMA)**: if CPU accesses local memory
- **Non-Unified Memory Access (NUMA)**: if a CPU accesses memory of another CPU, to be reduced!

Distributed memory over a network is a task of the programmer that needs to handle also synchronization
The distribution can be handled with different structures: **tuple spaces** or **different partitions**
To communicate and achieve the completion of a single task with multiple CPUs I need to have a structures between the nodes:

- star
- complete graph
- 2D grid
- ring
- torus – fast for neighbor to neighbor communication
- fat trees – CPUs working are the leafs
- hypercubes – useful for broadcast, only inner nodes have same number of neighbors

**Parallel file system**: can be used to allow concurrent access via the application of an offset on files accessed
**Cluster**: collection of independent nodes
**Massive Parallel Processors**: working entities that cannot run alone and need a network to communicate

# Main Parallel Concepts

**Critical paths**: chain of dependencies that force seriality
Scaling:

- **strong**: increase processors, total problem size fixed
- **weak**: increase processors, total work per processor fixed

**Arithmetic intensity**: f(n)/n – given f(n) the operations on n data
**Roofline model:** gives an idea of result achievable with best code, best architecture and compiling options (may not be reachable)
Approaches in parallelization:

- **MPI**: parallel process/node sends messages
- **OPENMP**: parallel process sends messages via memory
- **CUDA**: extension for GPU programming

Design of parallel algorithms:

- identify hotspots
- identify bottlenecks (usually I/O)
- identify inhibitors of parallelism
- investigate other better algorithms
- design solution

Scope of communication: **point-to-point**, **collective** (results share among a group)
**Granularity**: how much work per processor

- **fine grain**: less computation, more communication
- **coarse grain**: more computation, less communication

**Memory affinity**: is the act to allocate memory as close as possible to core; so NUMA effects must be reduced

# Vectorization

Important flags for compiler:

- O3: generate best optimized instructions
- xHost: generate instruction that exploit the best the current architecture
- fast-math: to increase speed but reduce precision in calculations
- ipo: interprocedural optimization between function calls

Vectorization examples: **loop unrolling**, needs **alignment** to vectorize
Limits of vectorization:

- number of loops must be known
- no breaks
- np switch
- if statement depends

- compiler may swap loops
- no function calls apart simple math ones

With the **vectorization report** I can know what has not been vectorized and why.
**Aliasing**: having more pointers to same data location
**Loop peeling**: to align the loops over data, usually the first and last iteration are isolated
**Loop Carried Dependence (LCD)**: data dependence on different iteration (a[i-1])
**Loop Independent Dependence (LID)**: data dependence on same iteration
Data dependence: **RAW** (read after write, non-vectorizable), **WAR** (vectorizable), **WAW** (non vectorizable, e.g. with a condition on writing)
**Reduction**: when same operation is applied on specific variable

# OpenMP

Guide parallelization via pragma instructions:

- parallel
- parallel for
- barrier
- no-wait
- collapse
- shared(default)
- reduction
- task

Avoid useless barriers and locks
Avoid NUMA and **false sharing** (when multiple threads acts on same cache line, making it going back and forth upon modifications (cache reloads the line each time))

# CUDA

**Kernel**: is a function that runs on GPU device
To call it pass **<<<Nblocks,NthreadsPerBlocks>>>**
**__global__**: function that runs on device, callable by host
**__device__**: function that runs on device, callable by device only
If I have unified memory between CPU and GPU, I don't need to specifically call **cudaMalloc**, but malloc is sufficient.
GPU threads are executed in groups called **warps** that serially executes each branch
**Coalescing**: when data for multiple threads are loaded in memory in a single transaction
Memories:

- **global memory**: larger, high bandwidth, high latency, like a RAM for GPU
- **shared memory**: one per block, shared between inner threads, very fast
- **constant memory**: for constant data, read-only, shared to blocks
- **texture memory**: read-only, set by host, follows dedicated pattern for graphics
- **registers**: accessible for each thread to store little data, fastest access
- **local memory**: not real, often mapped to global memory (in case of registry spilling)

Alternative approaches: **OpenACC** (with pragma like OpenMP) and **OpenCL** (framework and standard)