Parallel Computing Systems and Applications
Introduction

Salvatore Filippone

salvatore.filippone@uniroma2.it

Parallelism: definitions, models, paradigms

Parallelism

PARALLELISM: The capability of having multiple operations executing (producing results!) at a given point in time

In the context of CSE the most important metrics is the number of floating-point operations (results).

Parallelism

We may have parallel capabilities at multiple levels

- A single instruction may specify multiple operations (ex: SSE);
- A single processor may execute (complete) multiple instruction in a single cycle;
- A single computer (node) may have multiple CPUs;
- A computer installation may be built connecting multiple compute nodes (cluster)

All of these options raise parallelism (and lower CPI)

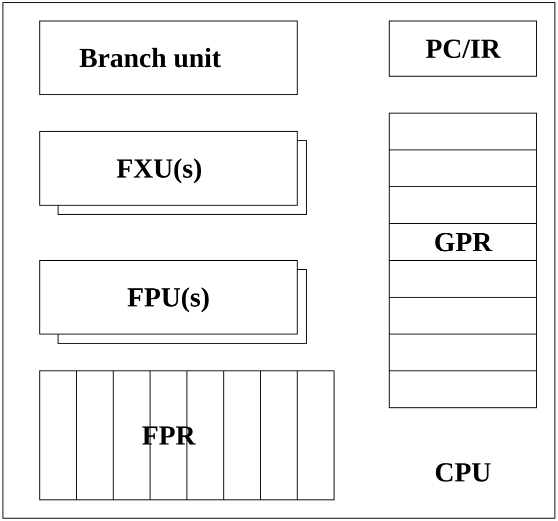
Multiple Units

A processor may have multiple execution units working independently:

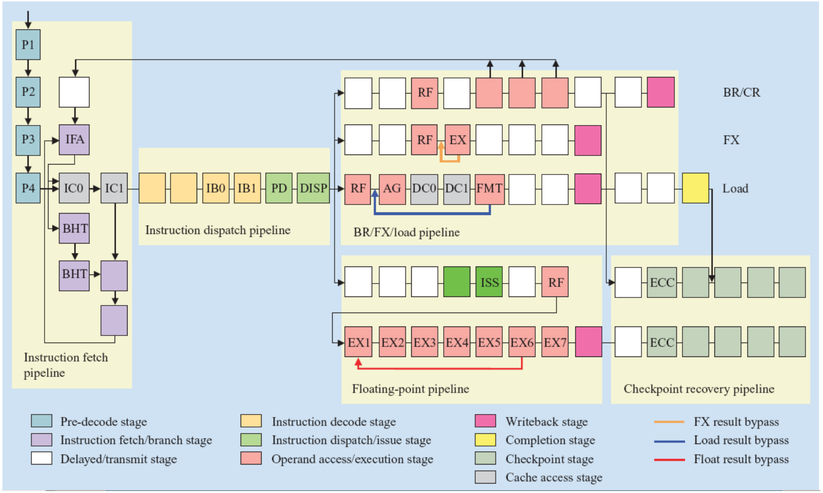
- POWER (IBM)
- ALPHA (DEC)
- Pentium (Intel, some limitations)

Practically all modern RISC processors.

multiple Units

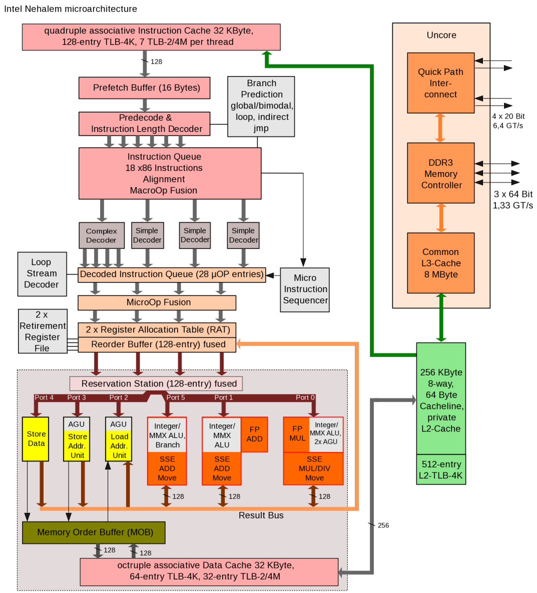


IBM POWER6



from IBM Journal of Research and Development.

Intel Nehalem

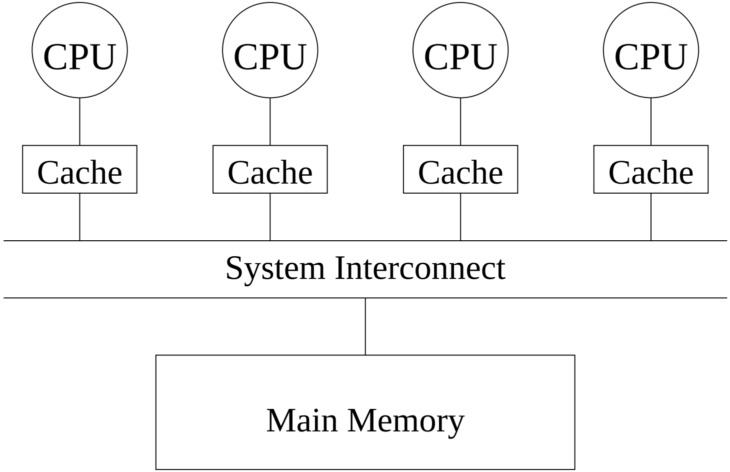


Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octruple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

Shared memory systems

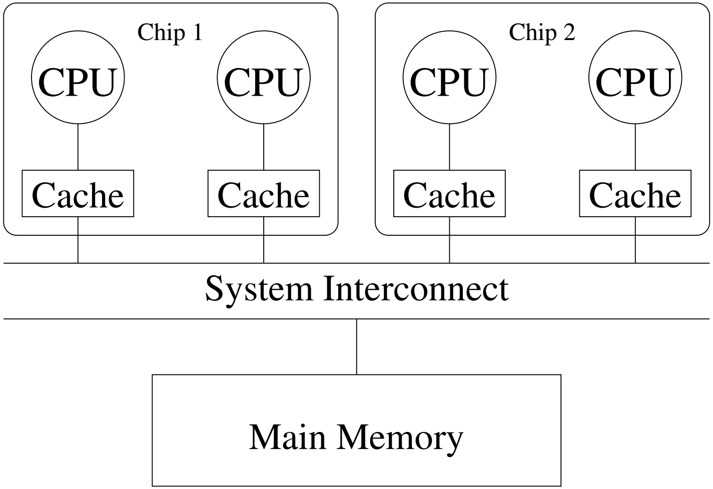
Two or more processor on the same board, coordinating memory accesses



May employ UMA (Uniform Memory Access) Model

Multi-core

Two or more processors on the same chip



From a logical and programming point of view, identical to shared memory multiprocessors; only implementation details are different.

Shared memory systems

Connection between memory and processors:

- common bus
- crossbar-switch
- multistage network

Critical system component: multiple tradeoffs between:

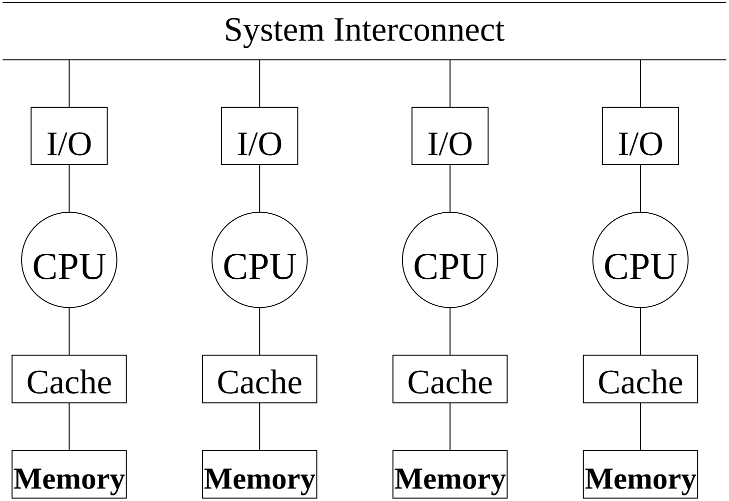
1. Latency
2. Bandwidth
3. Cost

Normally each processor has a private cache; operating systems and peripherals are shared.

Shared memory systems

Examples:	workstations (nodes) with dual/quad/octa processors/cores, vintage vector computers
Pros:	Shared memory programming paradigm
Cons:	Interconnection costs: limits on number of processors

Distributed memory systems



Distributed memory systems

- Extreme modularity;
- Often called “cluster” when the nodes are commodity;
- Critical component: interconnection network; ratio between computing time and communication time
- Typical paradigm: data access through *message passing*
- Programming is more complex.

Top500 supercomputing list `http://www.top500.org` is (mostly) clusters of shared-memory nodes (hybrid).

Parallel performance

Basic problem of parallel computing:

interaction among operation fluxes

- Conflicts among processor units;
- Memory access conflicts among processors;
- Sinchronization issues over a network;

Which has the most effective power, a horse or 10000 chickens?

Parallel performance

- How to measure parallel performance?
- How to compare measurements from different machines? (es: IBM BG, Cray XE6, generic Linux cluster)
- How to account for software development costs?
- Machine/environment prices?

Speed-up and efficiency

Define the size of a problem W: essentially equal to the number of operations necessary to arrive at the solution.

- $T_s = f(W)$: Serial time (between start and end on a single processor);
- $T_p = f(W, p, arch)$: Parallel time (between beginning of parallel execution and the time when the last processor completes) on p processors.
- Speedup:

$$S(W,p) = \frac{T_s(W)}{T_p(W,p)}$$

Should be always with "elapsed time"; given the same problem it may depend on the algorithm choice.

Speed-up and efficiency

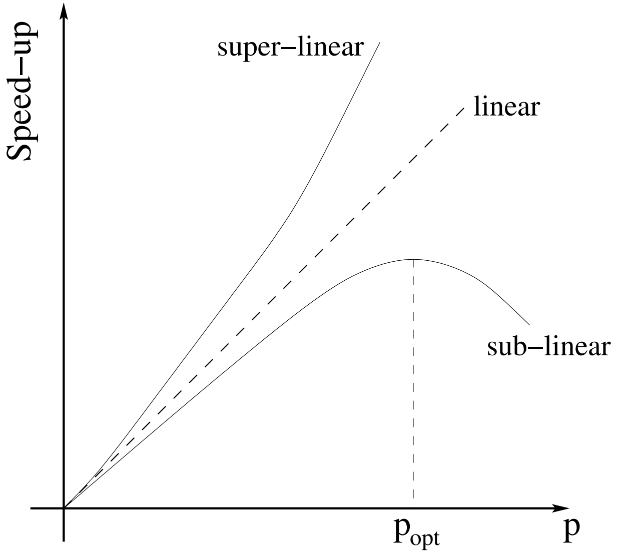
1. $S(W, p) = p$: *linear* speed-up
2. $S(W, p) < p$: *sub-linear* speed-up
3. $S(W, p) > p$: *super-linear* speed-up

Increasing p entails increase of

- startup time,
- *message-passing* fraction,
- synchronization overhead.

therefore we normally have $S(w, p) < p$
This is often referred to as *strong scalability* (i.e. constant workload).

Speed-up and efficiency with fixed problem size



Superlinear Speed-up anyone?

Actually, it *may* happen:

1. Better use of memory hierarchy;
2. Different code optimization;
3. Graph traversing order in search problems;

Speed-up and efficiency

Efficiency

$$E(W,p) = \frac{S(W,p)}{p} = \frac{T_s(W)}{T_p(W,p)\cdot p}$$

Amdhal's Law

Serial Fraction f_s: the ratio between the time spend in the strictly sequential part of a code and the total time $T_s(W)$;

Parallel Fraction f_p: the ratio between the time spent in the parallelizable part of the code and the total time $T_s(W)$.

$$T_p(W,p) = T_s(W) \cdot f_s + \frac{T_s(W)}{p} \cdot f_p \tag{1}$$

Therefore

$$\lim_{p\to+\infty} S(W,p) = \lim_{p\to+\infty} \frac{p}{1+(p-1)\cdot f_s} = \frac{1}{f_s} \tag{2}$$

Gustafson's Law

- Amdhal's Law is too pessimistic
- Parallelism: way to solve larger problems;
- Define α serial fraction, let the parallel part of the work grow;

$$T(N,P) = T_{\text{serial}}(N,P) + T_{\text{parallel}}(N,P) = \alpha T(1,1) + (1-\alpha)\frac{T(1,1)\cdot N}{P}$$

hence

$$S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{\alpha + (1-\alpha)N}{\alpha + (1-\alpha)\frac{N}{P}}$$

Setting $N = P$ you obtain

$$\begin{aligned} S(P,P) &= \alpha + (1-\alpha)P \\ E(P,P) &= \frac{\alpha}{P} + (1-\alpha) \end{aligned}$$

Note: a more precise model should take into account that α is not necessarily constant with a changing workload.
This is often referred to as *weak scalability*

Parallel Scalability

A Parallel System is said to be *scalable* if it keeps the same performance per processor as we increase at the same time the number of processors and the size of the problem to be solved.

Corresponds to the notion of *weak scalability*, because it is closer to typical usage.

The LINPACK benchmark

Example of performance measurement through a “kernel”.
Problem: solve $Ax = b$
Given the time to solution $T(N)$, compute MFLOPS R as

$$R = \frac{2}{3}\frac{N^3}{T}$$

Original LINPACK: $N = 100$ (very small), with a given Fortran code;

TPP: $N = 1000$ with any code;

HPC: Let N scale, then measure and publish:

R_{max}	: The best performance attained;
N_{max}	: N at which R_{max} was attained;
$N_{1/2}$	: N at which 50 % of R_{max} was attained;
R_{peak}	: The theoretical peak performance of the machine.

The LINPACK benchmark

Pros:

- Just a few, clear data values;
- The problem has (some) relation with real applications;
- Underlines the difference with R_{peak};

Cons:

- Just a few data values (modern machines are very complex);
- Difficult to project onto applications using very different algorithms.

Top500: a list of the 500 most powerful computing centers in the world, ranked by R_{max}.

Algorithms

As computer technology and performance progress, algorithms become more important!!!

- Computing system $\times 2$

Algorithms

As computer technology and performance progress, algorithms become more important!!!

- Computing system $\times 2$
- Problem size $N_2 = 2 \times N_1$

Algorithms

As computer technology and performance progress, algorithms become more important!!!

- Computing system $\times 2$
- Problem size $N_2 = 2 \times N_1$
- If $O(N^3)$ algorithm

Algorithms

As computer technology and performance progress, algorithms become more important!!!

- Computing system $\times 2$
- Problem size $N_2 = 2 \times N_1$
- If $O(N^3)$ algorithm
- Then computing time: $T_2 = 4 \times T_1$

Algorithms

As computer technology and performance progress, algorithms become more important!!!

- Computing system $\times 2$
- Problem size $N_2 = 2 \times N_1$
- If $O(N^3)$ algorithm
- Then computing time: $T_2 = 4 \times T_1$
- Therefore: always use the "best" algorithm available.

Parallelism

PARALLELISM

The capability of having multiple operations executing (producing results!) at a given point in time

In the context of CSE the most important metrics is the number of floating-point operations (results).

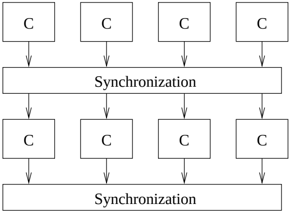
Definitions

Pattern:	Logical structure of a parallel algorithm;
Model:	Mechanism to express the desired logical structure in the source code;
Tool:	Software instrument (compiler, library, etc.) employed

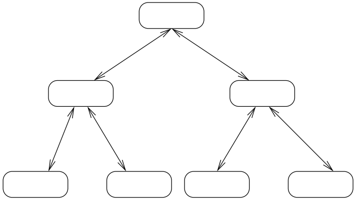
Algorithmic Patterns

Phase parallel:	A series of alternating computing phases (where the computations are independent) and synchronization phases;
Divide et impera:	A given task divides the problem into multiple subproblems; each of the subproblems is assigned to other tasks that recursively apply the same strategy; after the recursion, the solutions to the subproblems are combined to build the solution of the original problem;
Master-slave:	One of the tasks coordinates and distributes portions of the total work to the others, and collects the results;
Work pool:	There exists a shared structure hosting a queue of all subproblems to be solved to arrive at the overall problem solution

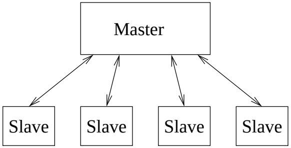
Algorithmic Patterns



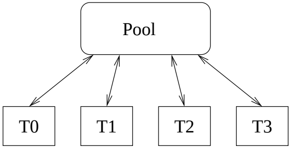
Phase parallel



Divide and conquer



Master−Slave



Work pool

Algorithmic Patterns

Domain Decomposition: Patterns in which the parallel opportunities are driven by the data:

- Geometric decomposition: The underlying geometry/topology of the problem guides the decomposition. Ex: PDEs discretized over a 2D/3D domain;
- Recursive Data Decomposition: Tree structures and the like (ex: the octa-tree for N-body problem)

Data Flow Decomposition: Patterns in which a *stream* of data must be processed through a set of parallel computing steps:

- Pipeline: There is a clear, 1D set of phases (Ex.: Unix pipes, processor pipelines)
- Irregular: Task-graphs, event-based coordination.

A (seemingly) perfect example

Consider the following C code

```
function foo(double x[], double y[], double z[])
{
  for (i=0; i<VERY_LARGE_SIZE; i++)
  z[i] = x[i] + y[i];
}
```

- What are the issues?
- How would you go about parallelizing this code?