Fundamental Properties of Program Behaviour

Exposing Parallelism

Tópico 3

Parallel Systems

Basic problems to be resolved:

1. How to split up or partition algorithms/programs and map them to some “abstractly partitioned” hardware. Will also need to evaluate the performance of the various options.

2. How to interconnect the hardware components to produce the required “abstractly partitioned” architecture.

Exploiting Parallelism

The objective is to exploit *Hardware* and *Software Parallelism* and the analysis comes down to the degree of (mis)match between them.

There are various levels of parallelism within these two forms: computational granularity; communication; scheduling; load balancing etc.

Highlights the need for interaction between hardware designers and software programmers in order to exploit parallelism for “revolutionary” performance improvements.

Hardware Parallelism

The available hardware parallelism is defined by the system architecture.

- The number and capacity of components and the capacity of the interconnection network.
- Example: the “5 characteristics” description of SIMD systems.
- Cost versus Performance (of basic operations, not programs).

An Analogy: Instruction Level Parallelism

Uniprocessors are either (super)pipelined to exploit temporal parallelism or superscalar to exploit spatial parallelism between instructions.

- Limits
- Tradeoffs

Software Parallelism

The two important types of software parallelism in parallel programming are:

- *Control Parallelism* - two or more operations are performed simultaneously; and
- *Data Parallelism* - the same operation over many data elements. Some advantages:
  - highest potential for concurrency;
  - parallelism is in proportion to the quantity of data; and
  - easier to write and debug.

Limits on Software Parallelism

The software parallelism available will depend on the algorithm, the programming style (perhaps) and compiler optimisations.

- Parallelism is limited by **dependencies**
  - between *data* in a program,
  - between *instructions* in a program, and
  - between program and operating system/microarchitecture *resources*.
- Programs can be translated to *dependence graphs* – directed graphs (e.g. flowcharts) which make dependencies explicit.

Now we just need to map or schedule graphs onto the architecture?

Data Dependence

- *Flow/True dependence* : the "output" of one instruction is required by the "input" of another.
- *Anti-dependence* : one instruction needs to do an "input" (from some variable) before another one can do an "output" to that variable (overwriting the original value).
- *Output dependence* : Two instructions "output" to the same variable (obviously, program order must be maintained for correctness).

What's the point?

The dependence relationships cannot be violated during program execution otherwise erroneous behaviour may result.

Usually used in order to come up with automatic means of analysing "abstract programs" (graphs) so that a compiler can schedule independent instructions to execute simultaneously.

Control Dependence

This refers to the situation where the order of execution of statements cannot be determined before run-time.

- E.g. loop iterations which depend (or do not) on values computed in other iterations.

- The same point as before: Compiler techniques are required to avoid these types of dependencies in order to exploit more parallelism.

The Aim

To create *processes* (sets of instructions) which can execute in parallel i.e. to have minimal dependencies between processes. (Usually in the context of writing *parallelising compilers.*)

"Bernstein's Conditions" define processes to be independent (and can run in parallel) if they don't write to common input/output data.

What this all means is that certain program properties (here defined in terms of conventional sequential-like programs) can be used to **create** software parallelism which can be matched to the kind of hardware parallelism available.

The Approach

Will need to use parallelising compilers **and** the appropriate hardware architecture to solve the mismatch problem.

Remember, a tradeoff exists in terms of cost, complexity and performance etc.

An Analogy:

- Optimising compilers and RISC architectures.
- Tradeoffs – e.g. complexity is moved to the compiler. Another, due to parallelism, is register pressure (i.e. larger register files versus code spills).

Compiling for parallel processors is much more involved.

More than one processor – So what?

Granularity is a measure of the amount of computation involved in a “software process”.

Grain size is the size of the basic program segment chosen for parallel processing, and is often described as *fine*, *medium* or *coarse* depending on the processing level.

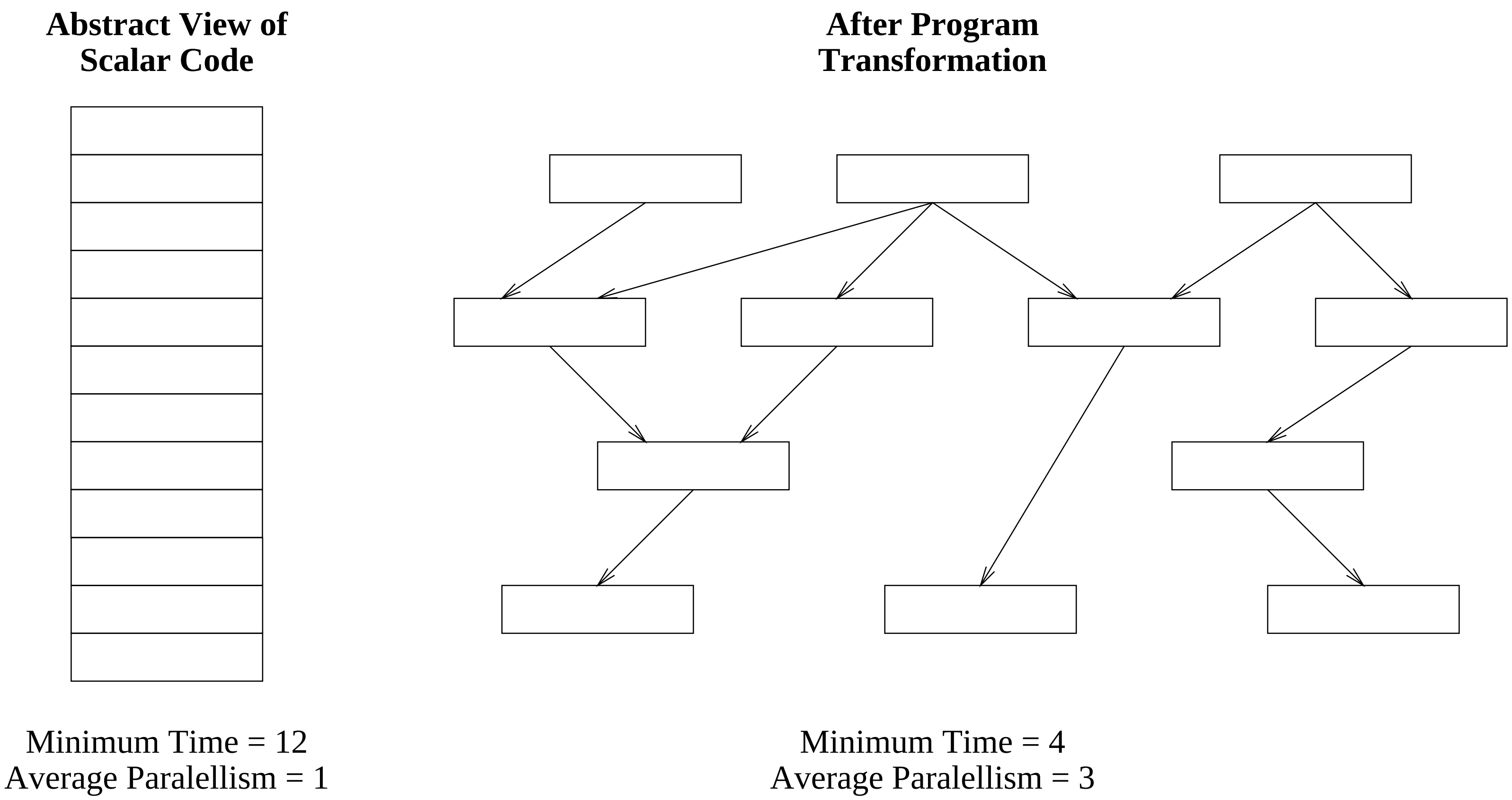
Latency is a measure of the “communication” overhead (e.g. memory, synchronisation and communication latencies).

More than one processor – So what?$_{(cont)}$

A close relationship exists between computational granularity and communication latency: better performance can be achieved by balancing them both.
Analogy : pipeline stages

Latency imposes a limiting factor on the scalability of a machine. System designers try to minimise inter-processor communication (IPC), but IPC is also affected by communication patterns which are determined by algorithms.

Program Transformations



Loop Structured Parallelism

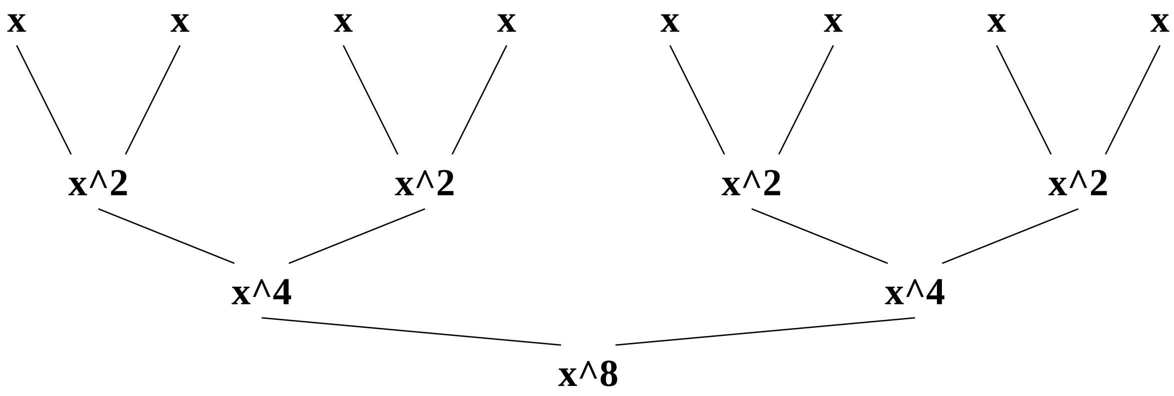
Example : raising to a power $y = x^n$.

$y := x$

for $i = 1$ to $n - 1$ do

$\quad y := y * x;$

Looks pretty sequential, but remember "program transformations"



Loop Structured Parallelism$_{(cont)}$

However this loop cannot be transformed:

for $i = 1$ to $n - 1$ do

$a[i] := a[i-1] * b[i-1];$

Because of

- data dependencies, and
- a linear recurrence relation.

Data-Level Parallelism

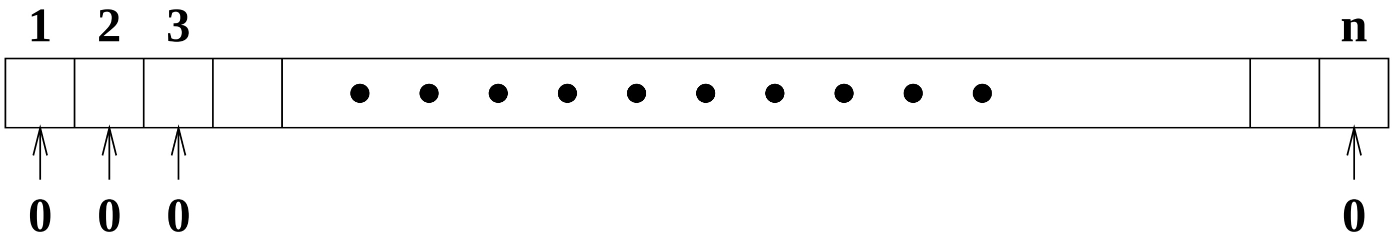
Simultaneous application of the *same* operation to multiple data objects. Data objects can be organised into data structures and thus a parallel operation is applied to the structure.

A simple (and typical) data structure is the **vector** (1D array, fixed size, independent elements). Easy to see how data parallelism applies to other structures: arrays (vectors of vectors); graphs (represented by matrices, i.e. 2D arrays).

Broadcasting (or "ascryption")

[scalar $\rightarrow$ vector]

for $i = 1$ to n do
    $x[i] := 0;$



All assignments could be done simultaneously given enough processors, or "quasi-simultaneously" given limited processors.

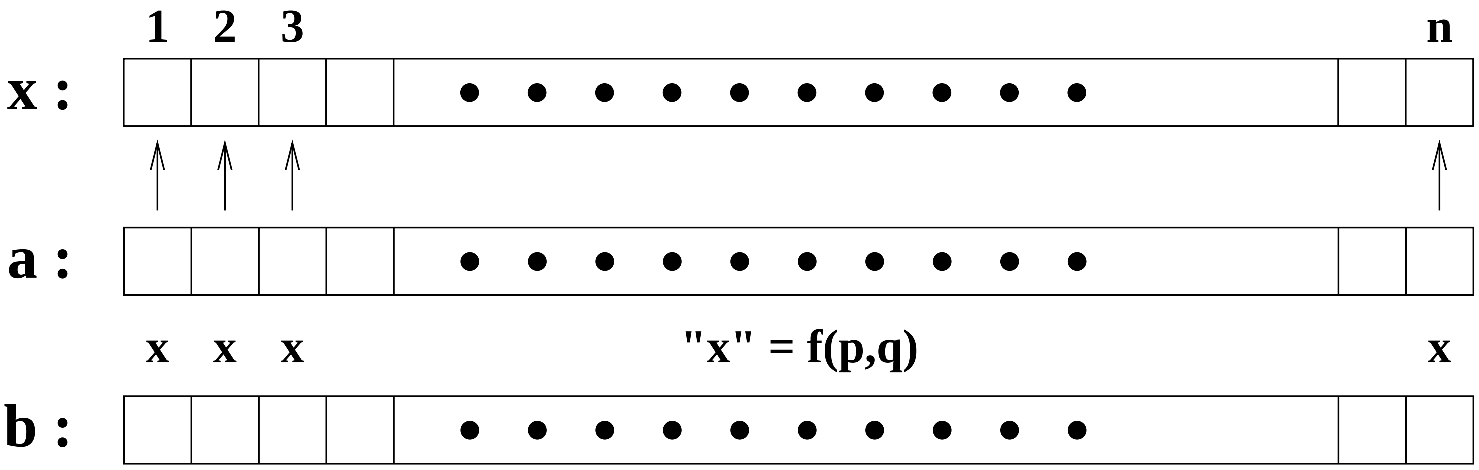
Notice the link with the architecture and performance.

Dyadic Functions

[vector $\times$ vector $\rightarrow$ vector]

for $i = 1$ to n do

$x[i] := f(a[i], b[i]);$



These can be generalised (e.g. triadic) in the obvious way.

Cumulative Reduction

[vector → scalar]
[(vector × vector) × scalar → scalar]

e.g. $\sum_{i=1}^{n} X_i$ $\prod_{i=1}^{n} X_i$ $\sum_{i=1}^{n} a_i * b_i$

Dot product is the basic constituent of matrix multiplication and many other numerical algorithms.

Usually done in $O(\log \mathrm{n})$ time (given $\geq \frac{n}{2}$ processors), giving a (rough form) speedup of $O(\frac{n}{log n})$. (Using the "binary tree" approach of earlier examples.) This compares with $O(1)$ time for the two other operations on the previous slides.

[It is important to get the "final result" of a parallel computation.]

Non-vector Data Structures

- Obviously, arrays
- Graphs

Graph edges *"encode"* communication required between nodes (or the data they represent). "Simultaneous" communication since the operation applied to all data.

"encode" is in quotes because there is not necessarily direct communication, just a data dependency that an algorithm has to deal with.

See M. Quinn, "Designing Efficient Algorithms for Parallel Computers" Chapter 8.

Process-Level Parallelism

Previous examples involved a number of parallel basic (atomic) operations.

Low complexity; overall parallelism determined by the number of atomic parallel operations (and the number of processors available!)

Alternative is to divide the problem into *independent* processes (which can communicate with each other).

Hardware for exploiting such parallelism is much more *loosely coupled* than that for data parallelism. Thus much more (conceptually) complex!

Parallelism

Software Parallelism		Hardware Parallelism
	$\xleftrightarrow{\textit{explicitly}}$	
data parallelism		SIMD, MIMD
process parallelism		MIMD

Hence the connection to UMA, NUMA shared and distributed memory etc.

Software Parallelism		Hardware Parallelism
	$\xleftrightarrow{\textit{implicitly}}$	
Instruction-level parallelism Loop-structured parallelism	i.e. via parallelising compilers	pipeline/vector/SIMD/MIMD depending on the compiler technology.

Last point

One other issue, before proceeding deeper into hardware parallelism (i.e. the architecture), is the *"flow"* of parallelism.

We concentrate on *control-flow* parallelism (program counter driven rather than data driven) because data-flow is still very much an open question.
[Big potential perhaps??]