Arquiteturas Paralelas I

Prof. Vinod Rebello (DCC-UFF)
Sala 302, 3º andar do Bloco E
vinod@ic.uff.br

O Que, Por Que e Onde do Paralelismo

One Definition of a Lecture

"A lecture should not be where the notes of the professor become the notes of the student without passing through the mind of either one."

Outline for Tópico 1

- The Objectives and Syllabus of the Course.

- What is Parallel Computing all about?
  Why Study it?

- An Introduction to **Parallelism**
  and the Big Picture.

The Aims of the Courses

- To explain the need for *Parallel Machines*;
- To outline the principles of machine design (*Parallel Architectures*), and the relationships to software issues;
- To explore the design space of architectural classes and concepts, and design aspects and choices;
- To provide you with an ability to objectively evaluate current and future parallel architectures; and finally,
- To provide a basic background (and hopefully generate interest :-) for further study and research in *Parallel Systems*.

Objetivos dos Cursos

- Expor a necessidade de *Máquinas Paralelas*;
- Apresentar os princípios do projeto da arquitetura (*Arquiteturas Paralelas*) e sua relação com aspectos relevantes em software;
- Explorar as alternativas de projeto de classes de arquiteturas, seus conceitos, e aspectos dos projetos;
- Formar o estudante tal que ele seja capaz de analisar com objetividade arquiteturas paralelas atuais e futuras; e finalmente,
- Prover uma fundação básica (e um provável interesse) para estudos e pesquisa futura em *Sistemas Paralelos*.

The Syllabus is based on

Sima, Fountain and Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, Addison-Wesley, 1997.

- Principles of parallel processing (Chapter 3).
- Parallel processors systems and structures (Chapters 4, and 6 to 18).

Kai Hwang, *Advanced Computer Architectures: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

- Principles of parallel processing (Chapters 1, 2 and parts of 3).
- Parallel processors systems and structures (Chapters 7, 8 and 9).

Hennessey and Patterson, *Computer Architecture: A Quantitive Approach*, 2nd edition, Morgan Kauffman, 1996.

- Advanced pipelining techniques (Chapter 4).
- Parallel processors systems (Chapters 7 and 8).

What is Parallel Computing about?

Speed:

- Solving problems (e.g. numerical computing, transaction processing, logical reasoning) faster;

Scalability:

- Solving a larger version of a problem in the same amount of time; or
- Solving (perhaps small versions of) problems not previously attemptable.

What is Parallel Computing about?$_{(cont)}$

Money:

Rather bluntly, a company will only stay in business if

- its machine is "revolutionary", or (and)
- the machine is *cost-effective*.

The bottom line: Price versus Performance.

So why is studying Parallel (and Distributed) Systems important?

- There **is** a hard limit to the performance of sequential machines.
- The use of these systems in both academia and industry is on the increase.
- Replicating simple hardware (to make a parallel machine) is ultimately more cost-effective than constructing exotic ASICs, so parallelism is still likely to become successful.

  [Comment: However, don't expect a dramatic switch away from "sequentialism".]

Why Parallelism? – The Argument Against

Fact – Sequential machines have gotten faster and cheaper at a tremendous rate.

Fact – Not all problems can be solved ("efficiently") in parallel, there exist inherently sequential problems.

Question: Why spend a lot of time and effort to make parallelism work?

A quick answer: We are seeking *revolutionary* rather than evolutionary increases in performance.

Also....

Sequentialism – The Argument Against

Fact – *The von Neumann Bottleneck.*

A "conventional" processor, fetching and executing instructions and data from a single memory, will have its execution rate bounded by the memory bandwidth.

A communication or data limit.

Sequentialism – The Argument Against$_{(cont)}$

Fact – *The Flynn (or speed of light) limit.*

In every "conventional" scalar processor there is a path through which instructions will flow at a rate of ≤ 1 instruction per clock cycle.

Since the maximum clock cycle time is bounded by speed of light constraints, this establishes an upper bound on processor performance.

A computation or control limit.

Why Parallelism? – The Argument For

In theory, parallelism bypasses both of these limits (through multiple memories and multiple processors), but .....

Question: How well does it work?

Drawing the Big Picture

The big picture must capture atleast these three basic issues:

1. How is parallelism expressed (by programmers in algorithms) for the target machine?
   [*Software (SW)*]
2. How can machine components be configured (via the architecture) to exploit the parallelism available, both, among these components and the software?
   [*Hardware (HW)*]
3. How do the resultant systems behave and perform?
   [*Interaction of SW and HW*]

Parallelism - What is it?

The notion of *Parallelism* is used in two different contexts:

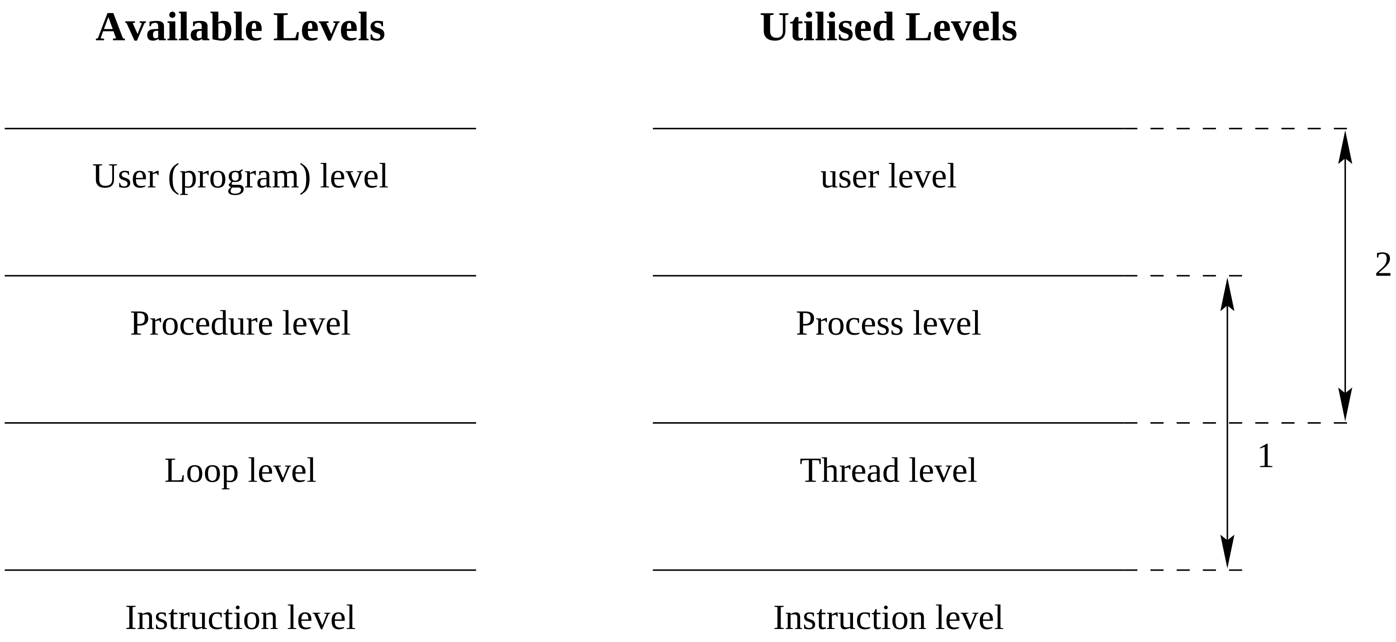
1. the **available parallelism** in programs:
   - Functional parallelism arises from the logic of the problem solution - this form is generally *irregular* and *small* (loop parallelism excepted); and
   - Data parallelism comes from using data structures that allow parallel operations on their elements - this form is *regular* and often *massive*.
2. the **utilised parallelism** occurring during execution.

Parallelism - Where is it?

Programs embody functional parallelism at different levels or sizes of granularity:

- Parallelism at the *instruction-level* (fine-grained parallelism);
- Parallelism at the *loop-level* (fine/medium-grained parallelism);
- Parallelism at the *procedure-level* (medium-grained parallelism); and
- Parallelism at the *program-level* (course-grained parallelism).

Utilising Functional Parallelism



1: Exploited by architectures.

2: Exploited by means of the operating system.

Utilising Data Parallelism

1. Directly by dedicated architectures that permit parallel or pipelined operations on data elements, called data-parallel architectures (DP-architectures).

2. Convert data parallelism into functional parallelism by expressing parallel executable operations on data elements in a sequential manner through the use of loop constructs.

Parallelism in Algorithms (or Programs)

The expression of an algorithm in a programming language defines how actions can be carried out in parallel. (This depends on the control and data structures available.)

Ideally, of course, we would like to be able to express the algorithm in any correct fashion and have the parallelism exposed and exploited automatically.

Data-Level Parallelism

This is the execution of the same function on a number of data items simultaneously.

The *degree of parallelism* is determined by the quantity of data rather than the algorithm. Although language constructs are needed to specify parallel operations on "parallel data structures",

- e.g. vectors, arrays and graphs.

[Comment: Obviously, when mapping an algorithm onto an architecture, the number of processors in the machine will limit the degree of parallelism.]

Process-level Parallelism

This is the execution of a number of independent instruction streams (processes) in parallel.

The process may or may not be part of the same overall program (actually, not quite so simple: different groups of different processes may work on different tasks at different times), e.g. in distributed systems or operating systems.

[Comment: A program may contain parallelism of various forms, but the architectural techniques for exploiting them are very different!]

Processes and Threads

In operating system terminology, the notion of a *process* is used in connection with the execution of a program. Processes are the tasks required to be carried out by a program or application. *Threads* are smaller chunks of code (and are also called *lightweight processes*). Multiple threads may be generated for each process.

[Comment: There are a number of concurrent and parallel languages that enable parallelism to be expressed at the language level by providing constructs to specify the creation of processes and threads.]

Concurrent and Parallel Execution

Concurrent Execution is the temporal behaviour of the *N-client 1-server model* where one client is served at any given moment. This model has a dual nature – it is *sequential* in a small time scale, but *simultaneous* in a large scale.

A scheduling policy determines how clients should be chosen for service and covers two aspects – whether a client can be interrupted or not (*the pre-emption rule*) and how one of the competing client is chosen (*the selection rule*).

Concurrent and Parallel Execution$_{(cont)}$

Parallel Execution is associated with the *N-client N-server model* where having more than one server allows the servicing of more than one client at the same time. Two schemes of execution exist – *lock-step* or *synchronous* (where each server starts service at the same time) and *asynchronous* (where the servers do not work in concert).

Concurrent and Parallel Programming Languages

Classifying languages according to constructs:

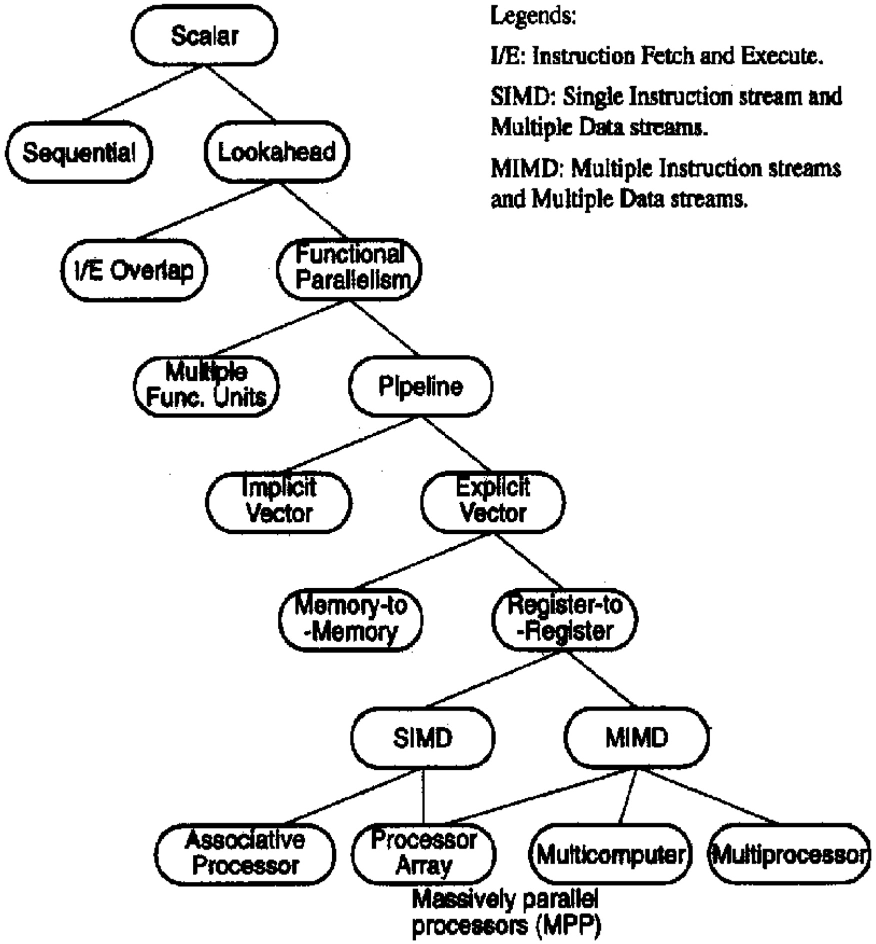
- *Sequential (traditional) languages* do not contain any constructs to support the N-client model (e.g. C, Pascal, Fortran).
- *Concurrent languages* employ constructs to implement the N-client 1-server model by specifying concurrent threads and processes but lack constructs to describe the N-server model (e.g. Ada, Concurrent Pascal, Modula-2).

Concurrent and Parallel Languages$_{(cont)}$

- *Data-parallel languages* introduce special data structures that are processed in parallel, element by element. Also, special mapping directives help the compiler in the optimal distribution of the structures among processors (e.g. High Performance Fortran, DAP Prolog).
- *Parallel languages* extend the specification of the N-client model of concurrent languages with processor allocation constructs (e.g. Occam-2, Parallel C, Strand-88).

[Comment: Although parallel languages have more constructs, it does not mean they are superior!]

Computer Architecture Evolution



Legends:

I/E: Instruction Fetch and Execute.

SIMD: Single Instruction stream and Multiple Data streams.

MIMD: Multiple Instruction streams and Multiple Data streams.

Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers.

A Classification of Parallel Architectures

Flynn's Taxonomy (1972), though not perfect, has become the standard and is based on instruction and data streams:

SISD – Single Instruction stream Single Data stream (Sequential machines)

SIMD – Single Instruction Multiple Data (Vector machines, DAP, MasPar)

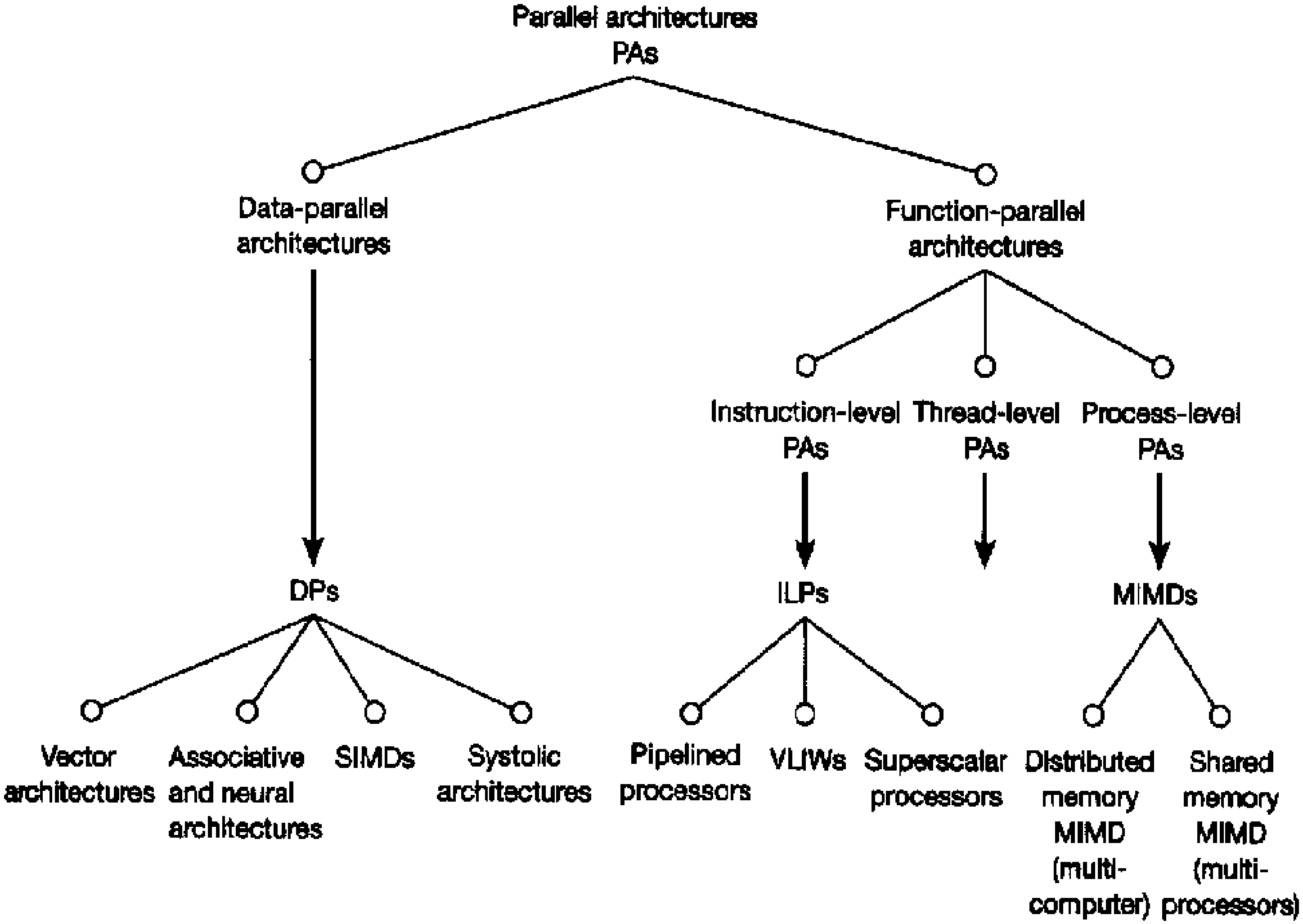
Flynn's Classification of Parallel Architectures$_{(cont)}$

MISD – Multiple Instruction streams Single Data stream (Systolic arrays)

MIMD – Multiple Instruction streams Multiple Data streams (NCube, Intel Paragon, Meiko Computing Surface)

There are some newer designs that are hard to classify e.g. mixed SIMD/MIMD (CM-5)

A Parallelism-based Classification



Basic Structures of Parallel Architectures

There are just 2 fundamental structures (*replication* and *pipelining*) derived as a result of partitioning the tasks either *spatially* or *temporally*.

Spatial Parallelism —> space —> array-like structures

- "Arrays" partition the "data" and/or "control" – SIMD, MIMD.

Temporal Parallelism —> time —> pipelined structures

- Pipelines partition the "function" – Vector processors, systolic arrays, MISD, possibly MIMD.

Basic Structures$_{(cont)}$

Examples of replication:

- Functional units in processors;
- Datapaths in superscalar and VLIW processors;
- Processors in SIMD and MIMD machines;
- Memory banks in interleaved memory systems.

Examples of pipelining:

- Processors pipeline instruction execution;
- Vector processors execute the same operation on pairs or tuple of elements;
- Systolic and wavefront arrays;
- Wormhole router of message passing computers.

Exposing Parallelism

1) Compiler generated or "implicit" parallelism:

- sequential code converted to parallel object code by *parallelising compiler*;
- loop parallelisation;
- vector operations on multiple processors;
- other "small" (limited utility) extractions of independent operations from code.
- difficult (using sequential languages even with parallel extensions) and benefit is limited.

Exposing Parallelism$_{(cont)}$

2) Hand generated or "explicit" parallelism:

- needs lots of work by the programmer;
- reduces burden on the compiler to detect parallelism (instead it has to preserve parallelism);
- high software development and maintenance costs.
- Research area (since a compiler can only do so much).

Programming Languages and Parallel Architectures

Programming languages and parallel architectures are NOT independent layers of a computer system!

Architectures have an influence on the language constructs applied to exploit parallelism.

- ILP and dataflow architectures:
  - are not supported by special constructs;
  - rely on intelligent optimising compilers;
- Vector processors:
  - do not often impose special language constructs;
  - rely on compiler support to exploit loop parallelism;

Languages and Architectures$_{(cont)}$

- SIMD machines:
  - the only data-parallel architecture sensitive to language;
  - parallel data structures handled as single objects;
  - SIMD machines have only one control unit and therefore require a construct to mask operations on processors;
  - constructs to specify the allocation of the parallel data structure elements to processors.
  - One of the reasons why SIMD isn't that popular.

Languages and Architectures$_{(cont)}$

- Single Procedure Multiple Data execution model:
  - A generalisation of SIMD execution model;
  - A thread can split into N threads that work on different invocations of the same loop;
  - All processors execute the same instructions in SIMD model, in the SPMD the same code can be executed at different speeds;
  - SIMD model applies instruction-by instruction synchronisation, whereas in SPMD it is sufficient to synchronise at the end of the loop. This is called *barrier synchronisation* and requires hardware support for efficent implementation.

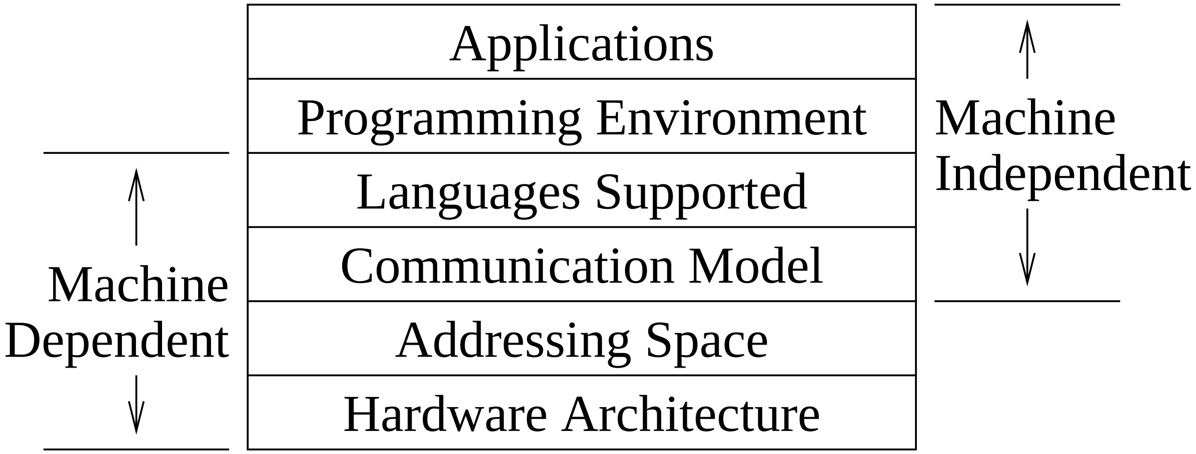
Languages and Architectures$_{(cont)}$

- MIMD (and multithreaded) architectures:
  - Beyond creation and termination of processes, languages provide "tools" for communication and synchronisation;
  - *mutual exclusion* is the main synchronisation problem in shared memory MIMD machines;
  - test_and_set operations (implemented in hardware) are used to support *semaphores*;
  - Distributed memory machines use message-passing operations;
  - PVM and MPI are recent parallel interfaces that realise communication constructs to simplify programming.

Exposing Parallelism - A Summary

- Parallelism exists in sequential programs.
- Difficult to detect and exploit it automatically.
- In terms of parallelism, sequential programming language syntax and semantics often impose unnecessary *causality* constraints between statements resulting in highly restrictive behaviour.
- Conclusion: Conventional languages are NOT good vehicles for exploiting parallelism.
- However, a significant investment exists in SW written in "old" languages (e.g. Fortran) and much effort (commercially) is placed in this area.

The Parallel System Big Picture



A Summary

- The *why*, *what* and *where* of parallelism.
- Various types (or levels) of parallelism: the program or algorithm view and the architecture view.
- Architectural techniques for exploitation depend on the type of parallelism.
- Parallelism has to be found implicitly or explicitly in programs.
- Have to match the available parallelism to the utilisable parallelism.
- The big picture and the interactions between the system levels.