BCS-29
Advanced Computer Architecture

Parallel Computing

Programming Environments

Issues in Parallel Computing

- Design of parallel computers
- Design of efficient parallel algorithms
- Parallel programming models
- Parallel computer language
- Methods for evaluating parallel algorithms
- Parallel programming tools
- Portable parallel programs

Programming Environments

- Programmability depends on the programming environment provided to the users.
- Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer.
- Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.
- **Implicit Parallelism:**
- **Explicit Parallelism**

Programming Environments

- **Implicit Parallelism:**
  - Use a conventional language (like C, Fortran, Lisp, or Pascal) to write the program.
  - Use a parallelizing compiler to translate the source code into parallel code.
  - The compiler must detect parallelism and assign target machine resources.
  - Success relies heavily on the quality of the compiler.
- **Explicit Parallelism**
  - Programmer write explicit parallel code using parallel dialects of common languages.
  - Compiler has reduced need to detect parallelism, but must still preserve existing parallelism and assign target machine resources.

Important Issues in Parallel Programming

Important Issues:

- Partitioning of data
- Mapping of data onto the processors
- Reproducibility of results
- Synchronization
- Scalability and Predictability of performance
- Success depends on the combination of
  - Architecture, Compiler, Choice of Right Algorithm, Programming Language
  - Design of software, Principles of Design of algorithm, Portability, Maintainability, Performance analysis measures, and Efficient implementation

Exploitation of PARALLELISM

Attributes of parallelism

- Computational granularity,
- Time and space complexities,
- Communication latencies,
- Scheduling policies,
- Load balancing, etc.

Types of Parallelism

- Data parallelism
- Task parallelism
- Combination of Data and Task parallelism
- Stream parallelism

Data Parallelism

- Identical operations being applied concurrently on different data items is called data parallelism.
- It applies the SAME OPERATION in parallel on different elements of a data set.
- It uses a simpler model and reduce the programmer’s work.
- Responsibility of programmer is to specify the distribution of data for various processing elements.

Task Parallelism

- Many tasks are executed concurrently is called task parallelism.
- This can be done (visualized) by a task graph. In this graph, the node represent a task to be executed. Edges represent the dependencies between the tasks.
- Sometimes, a task in the task graph can be executed as long as all preceding tasks have been completed.
- Let the programmer define different types of processes. These processes communicate and synchronize with each other through MPI or other mechanisms.
- Programmer’s responsibility is to deal explicitly with process creation, communication and synchronization.

Data and Task Parallelism

Integration of Task and Data Parallelism

- Two Approaches
  - Add task parallel constructs to data parallel constructs.
  - Add data parallel constructs to task parallel construct
- Approach to Integration
  - Language based approaches.
  - Library based approaches.

Stream Parallelism

- Stream parallelism refers to the simultaneous execution of different programs on a data stream. It is also referred to as ***pipelining***.
- The computation is parallelized by executing a different program at each processor and sending intermediate results to the next processor.
- The result is a pipeline of data flow between processors.
- Many problems exhibit a combination of data, task and stream parallelism.
- The amount of stream parallelism available in a problem is usually independent of the size of the problem.
- The amount of data and task parallelism in a problem usually increases with the size of the problem.

Conditions of Parallelism

- The exploitation of parallelism in computing requires understanding the basic theory associated with it. Progress is needed in several areas:
  - computation models for parallel computing
  - Inter-processor communication in parallel architectures
  - integration of parallel systems into general environments

Data and Resource Dependencies

- Program segments cannot be executed in parallel unless they are independent.
- Independence comes in several forms:
  - **Data dependence:** data modified by one segement must not be modified by another parallel segment.
  - **Control dependence:** if the control flow of segments cannot be identified before run time, then the data dependence between the segments is variable.
  - **Resource dependence:** even if several segments are independent in other ways, they cannot be executed in parallel if there aren’t sufficient processing resources (e.g. functional units).

Data Dependence

- **Flow dependence:** S1 precedes S2, and at least one output of S1 is input to S2.
- **Anti-dependence:** S1 precedes S2, and the output of S2 overlaps the input to S1.
- **Output dependence**: S1 and S2 write to the same output variable.
- **I/O dependence**: two I/O statements (read/write) reference the same variable, and/or the same file.
- **Unknown dependence:** Dependence relationships cannot be determined in the following situations:
  - Indirect addressing
  - The subscript of a variable is itself subscripted.
  - The subscript does not contain the loop index variable.
  - A variable appears more than once with subscripts having different coefficients of the loop variable (that is, different functions of the loop variable).
  - The subscript is nonlinear in the loop index variable.
- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

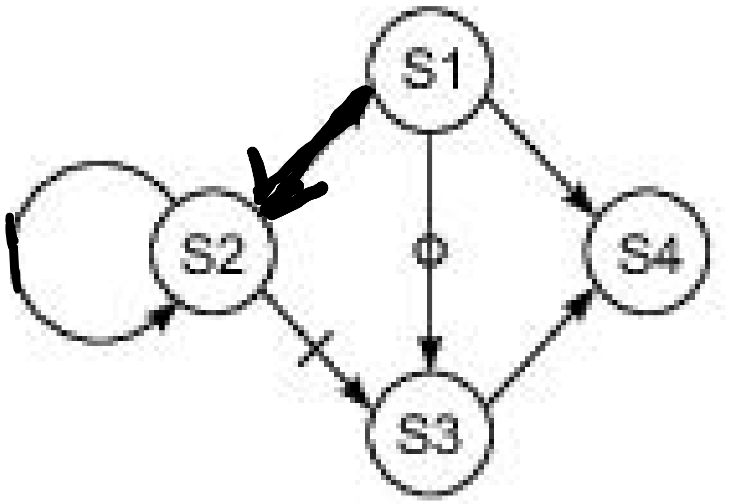
Example

S1:	Load R1, A	/R1 ← Memory(A)/
S2:	Add R2, R1	/R2 ← (R1) + (R2)/
S3:	Move R1,R3	/R1 ← (R3)/
S4:	Store B, R1	/Memory(B) ← (R1)/

S2 is flow dependent on S1 because the variable R1

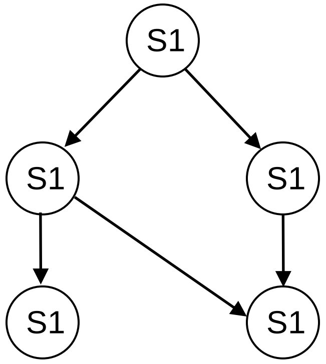
S3 is anti-dependent on S1 because of register R1.

S3 is output-dependent on S1 because of register R1and more …..



Program Transformation and Code scheduling

```
S1:   A = 1
S2: B = A + 1
S3: C = B + 1
S4: D = A + 1
S5: E = D + B
```


$$\begin{tikzcd}
 & S1 \arrow[dl] \arrow[dr] & \\
S1 \arrow[d] \arrow[drr] & & S1 \arrow[d] \\
S1 & & S1
\end{tikzcd}$$

```
S1:  A = 1
     cobegin
S2:          B = A + 1
             post (e)
S3:          C = B + 1
                   ||
S4:          D = A + 1
              wait (e)
S5:          E = D + B
     coend
```

Control Dependence

- It is the situation, when the order of the execution cannot be determined before run time.
  - Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions.
  - Dependence may also exist between operations performed in successive iterations of a looping procedure.

- Control-independent example:

```
for (i=0;i<n;i++) {
a[i] = c[i];
      if (a[i] < 0) a[i] = 1;
}
```

- Control-dependent example:

```
for (i=1;i<n;i++) {
      if (a[i-1] < 0) a[i] = 1;
}
```

- Compiler techniques are needed to get around control dependence limitations.

Control Dependences

```
S :   if $A \neq 0$ then
T :        C=C+1
U :        D = C/A
      else
V :        D = C
      end if
W :        X = C + D
```

```
S:    b =  [$A \neq 0$]
T:    C = C+ 1 when b
U:    D = C/A when b
V:    D = C when not b
W:    X = C + D
```

Resource Dependence

- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc.
- ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions

- Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.
- Notation
  - I_i is the set of all input variables for a process P_i .
  - O_i is the set of all output variables for a process P_i .
- If P_1 and P_2 can execute in parallel (which is written as $P_1 \,||\, P_2$), then:

$$I_1 \cap O_2 = \varnothing$$

$$I_2 \cap O_1 = \varnothing$$

$$O_1 \cap O_2 = \varnothing$$

Bernstein's Conditions

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative (P_i || P_j implies P_j || P_i), but not transitive (P_i || P_j and P_j || P_k does not imply P_i || P_k) . Therefore, || is not an equivalence relation.
- Intersection of the input sets is allowed.

Detection of Parallelism

- Example

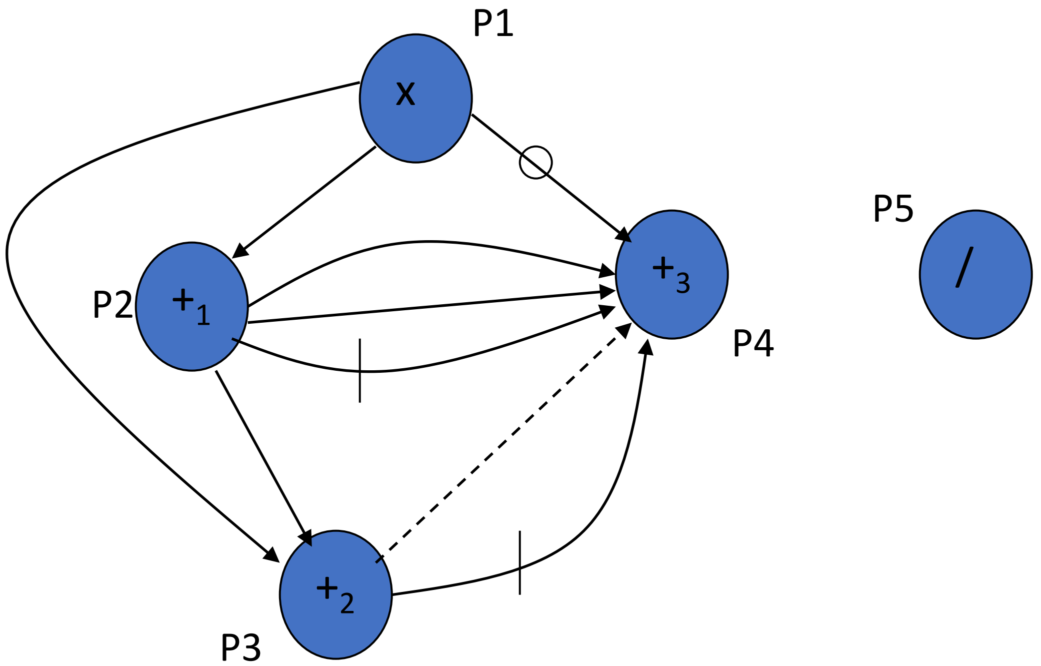
  P_1: C = D x E

  P_2: M = G + C

  P_3: A = B + C

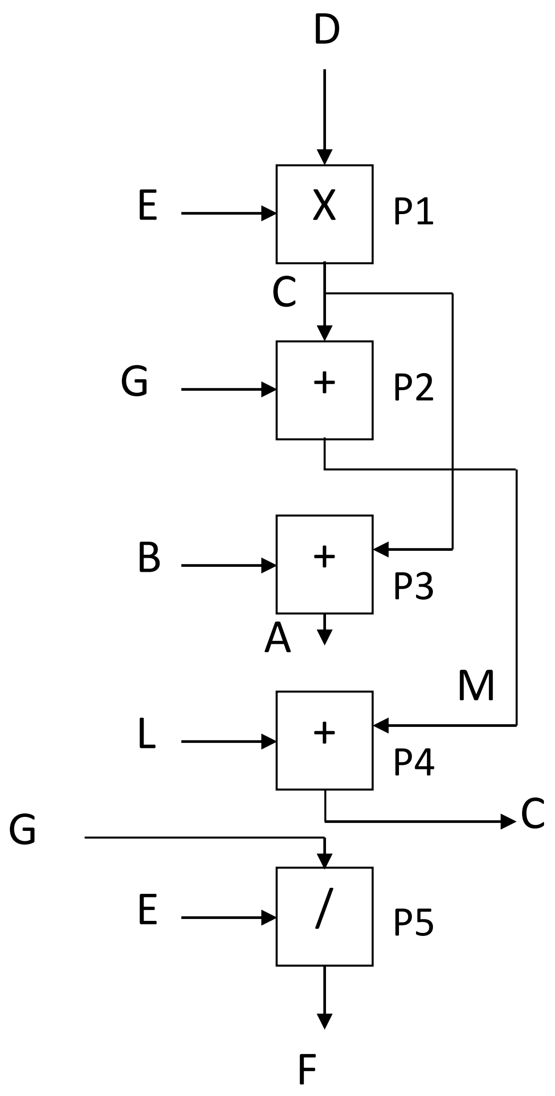
  P_4: C = L + M

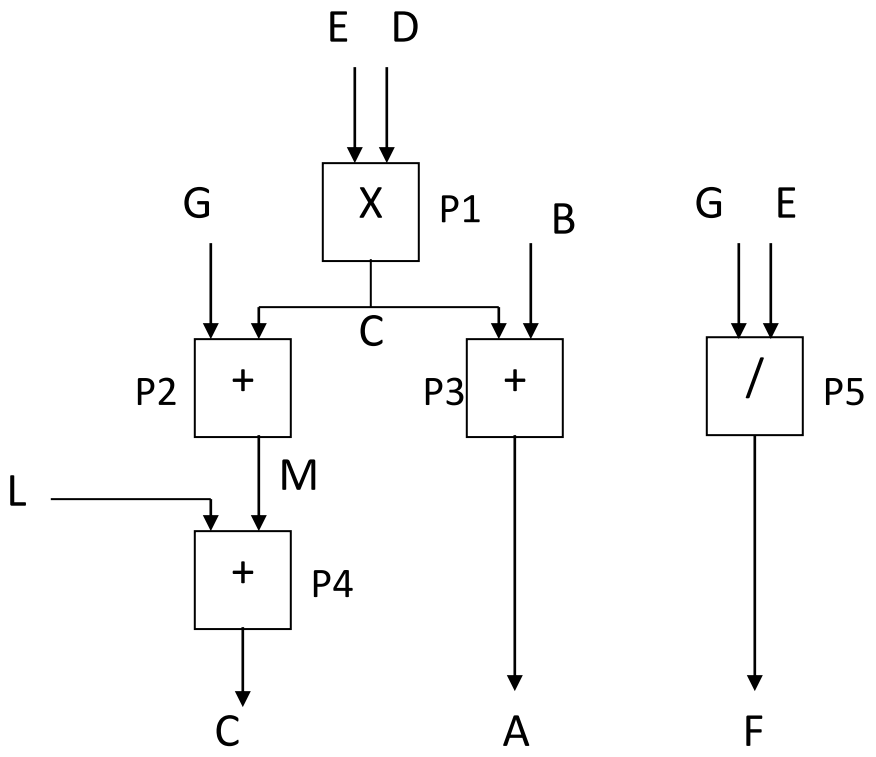
  P_5: F = G / E



Dependence Graph

Execution (Data-flow)





Hardware Parallelism & Software Parallelism

Hardware parallelism

- Hardware parallelism is defined by machine architecture and hardware multiplicity.
- It can be characterized by the number of instructions that can be issued per machine cycle. If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor. Conventional processors are *one-issue* machines.
- Examples. Intel i960CA is a three-issue processor (arithmetic, memory access, branch). IBM RS-6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).
- A machine with *n* *k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

Software Parallelism

- Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph.
- It is a function of algorithm, programming style, and compiler optimization.

Mismatch between software and hardware parallelism

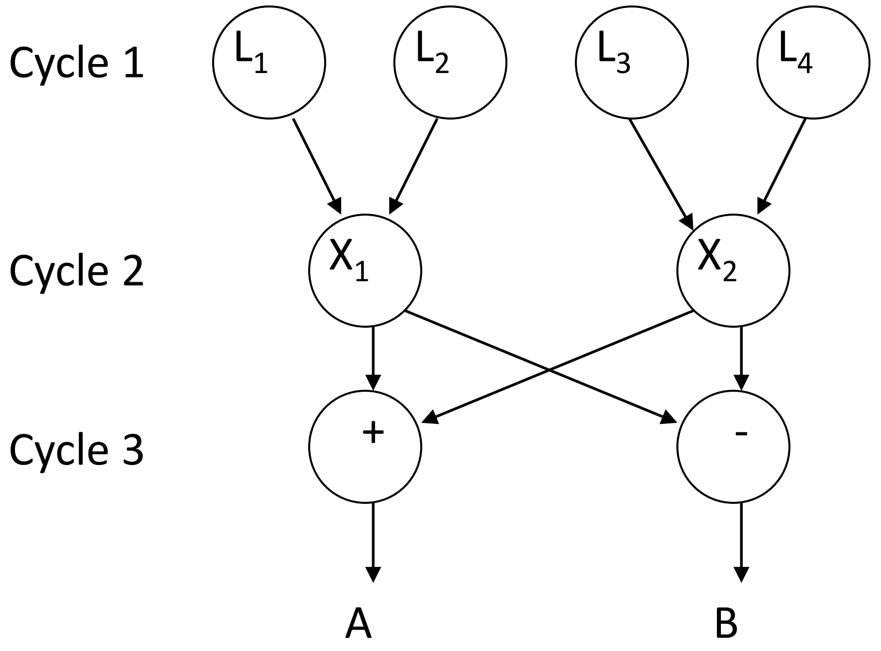
Example:

$$A = (P \times Q) + (R \times S)$$

$$B = (P \times Q) - (R \times S)$$

Code Sequence

L_1	Load P
L_2	Load Q
L_3	Load R
L_4	Load S
X_1	Mul P, Q
X_2	Mul R, S
+	Add X_1, X_2
-	Sub X_1, X_2



Maximum software parallelism: No limitation of functional units (L=load, X/+/- = arithmetic).

Mismatch between software and hardware parallelism

Example:

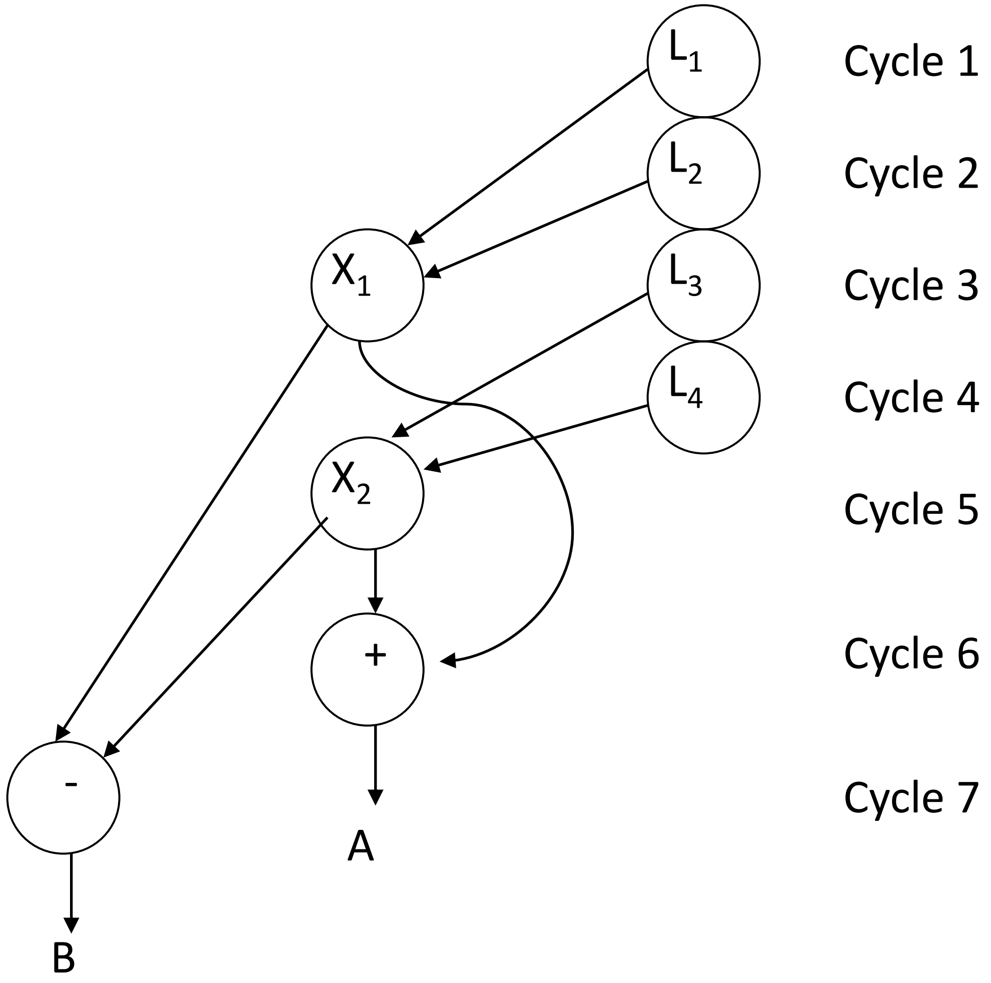
A = (P X Q) + (R X S)

B = (P X Q) - (R X S)

Code Sequence

L_1	Load P
L_2	Load Q
L_3	Load R
L_4	Load S
X_1	Mul P, Q
X_2	Mul R, S
+	Add X_1, X_2
-	Sub X_1, X_2

Execution Using Single Functional Unit for Load, Mul and Add/Sub



Mismatch between software and hardware parallelism

Example:

$A = (P \times Q) + (R \times S)$

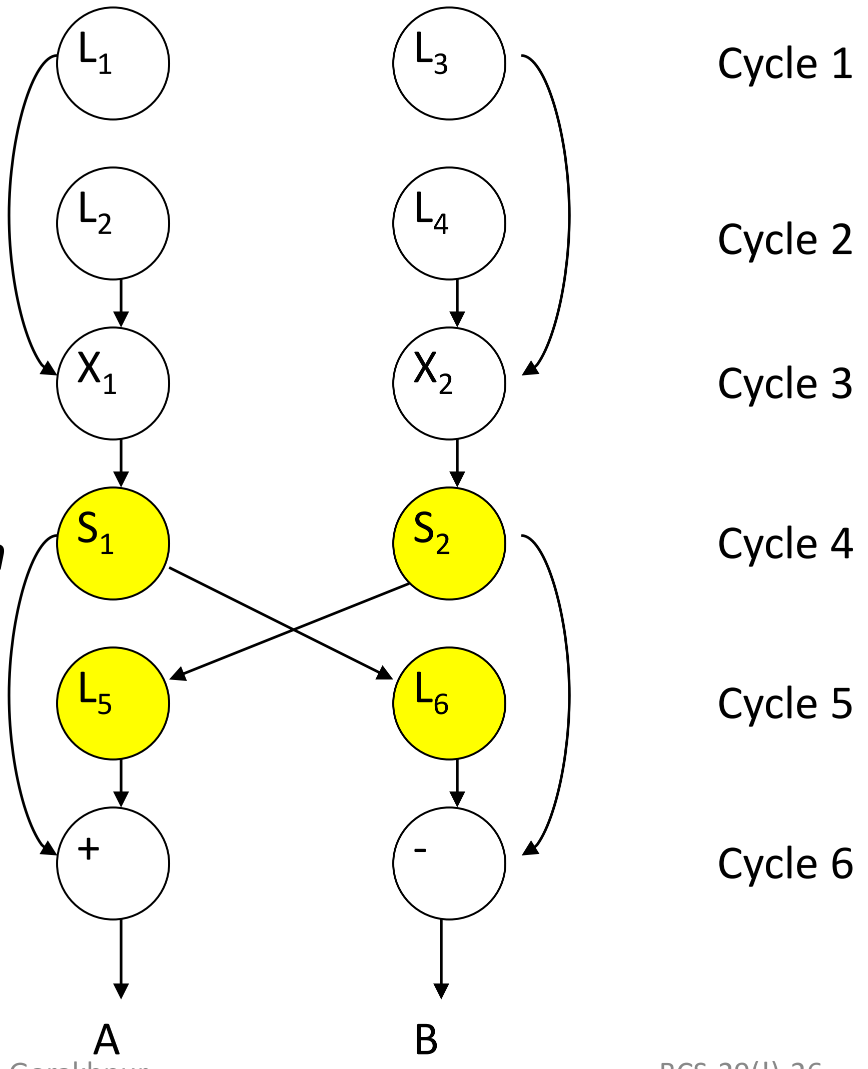
$B = (P \times Q) - (R \times S)$

Code Sequence

L_1	Load P
L_2	Load Q
L_3	Load R
L_4	Load S
X_1	Mul P, Q
X_2	Mul R, S
+	Add X_1, X_2
-	Sub X_1, X_2

Execution Using Two Functional Units for each of Load, Mul and Add/Sub operations

● *= inserted for synchronization*



Program Partitioning & Scheduling

- Program Partitioning
  - The transformation of sequentially coded program into a parallel executable form can be done manually by the programmer using explicit parallelism or by a compiler detecting implicit parallelism automatically.
  - Program partitioning determines whether the given program can be partitioned or split into pieces that can be executed in parallel or follow a certain pre-specified order of execution.

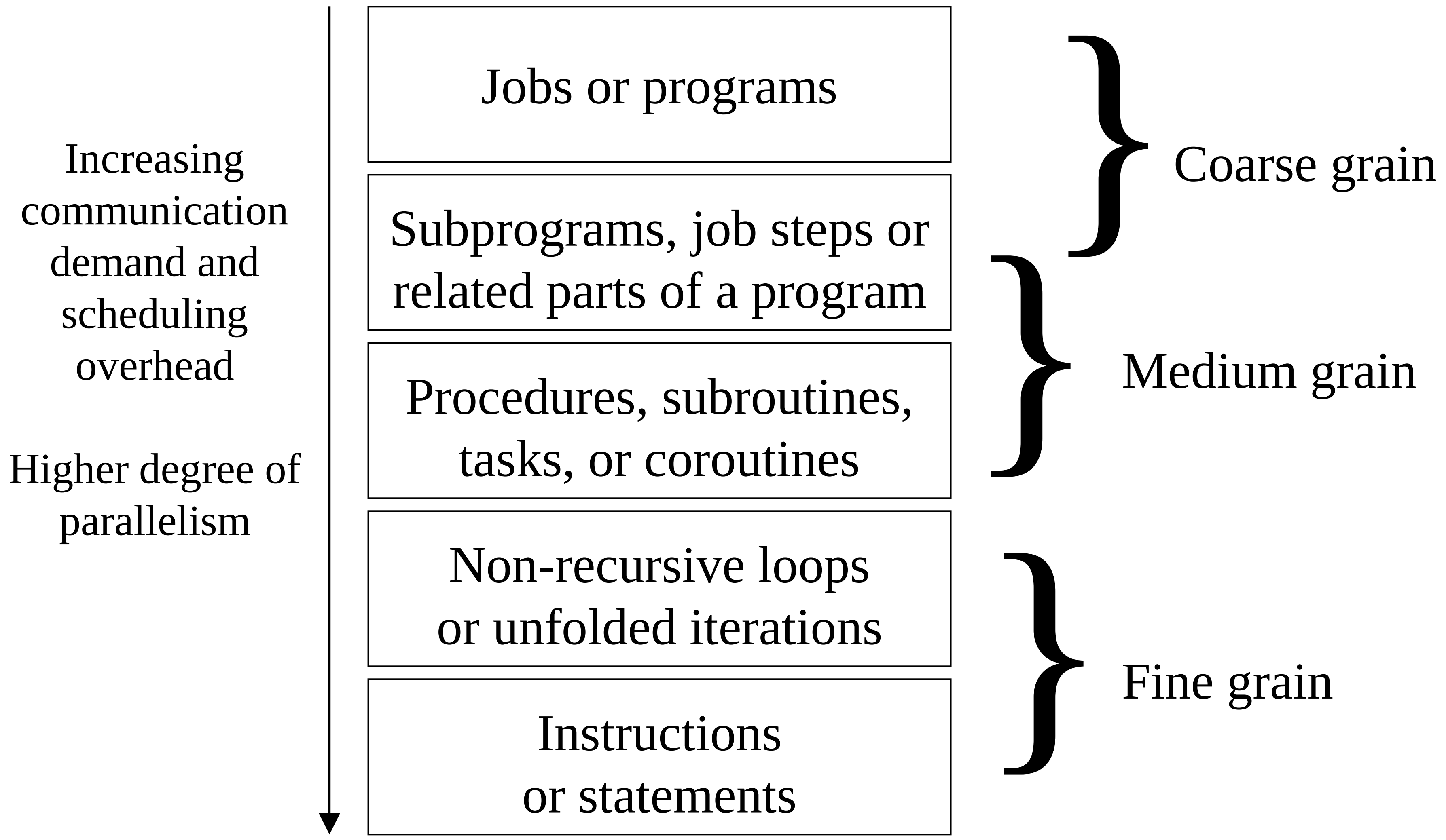
Program Partitioning & Scheduling

- **Grain size or Granularity**
    - It is the size of the parts or pieces of a program that can be considered for parallel execution.
    - Grain size is the simplest measure to count the number of instructions in a program segment chosen for parallel Execution.
    - Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved
- **Latency**

Latency is the time required for communication between different subsystems in a computer.

- Memory latency, for example, is the time required by a processor to access memory.
- Synchronization latency is the time required for two processes to synchronize their execution.
- Computational granularity and communication latency are closely related.

Levels of Parallelism



Increasing communication demand and scheduling overhead
Higher degree of parallelism
Jobs or programs
Subprograms, job steps or related parts of a program
Procedures, subroutines, tasks, or coroutines
Non-recursive loops or unfolded iterations
Instructions or statements
Coarse grain
Medium grain
Fine grain