Parallel Processing

The parallel processing were classified by M.J. Flynn, accordingly this classification is called *Flynn's* classification. This classification is based the fact that how many *instructions* and *data* it can process simultaneously. The sequence of instructions read from the memory constitute an *instruction stream*. The operations performed on data in the processor constitutes a *data stream*. The Parallel processing may occur in *instruction stream* or in *data stream*, or both. Following are the alternatives:

1. *Single-instruction single-data streams (SISD)*: All the instructions are executed sequentially.

2. *Single-instruction multiple-data streams (SIMD)*: All processors receive the same instruction from control unit, but operate in different sets of data. For example, in vector processors, all the elements of a vector are processed in a similar way.

3. *Multiple-instruction single-data streams (MISD)*: It is of theoretical interest only as no practical organization can be constructed using this organization.

4. *Multiple-instruction multiple-data streams (MIMD)*: Several programs can execute at the same time, operating on multiple data values. Most multiprocessors come in this category.

The Flynn's taxonomy of computer architecture is shown in figure 1 for SISD and SIMD architectures. The figure 2 shows the MIMD architecture.

One of the parallel processing class that does not fit into this classification is *pipeline processing*, in which, the same instruction is split into may stages, and all stages work in parallel processing the different stage of an instruction, giving view of a pipeline.

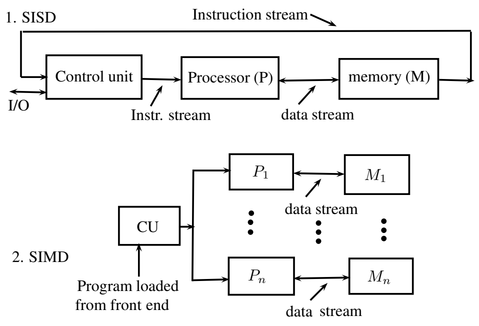

Figure 1: SISD and SIMD architecture.

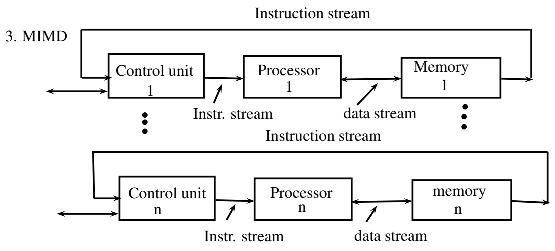

Figure 2: MIMD Architecture.

0.1 Types of Parallel Processing

Parallel processing means, increasing speed by doing many things in parallel. Let P is a sequential processor processing the task T in sequential manner. If T is partitioned into n subtasks $T_1, T_2, \ldots, T_n$ of approximately same size, then a processor P' (say) having n processors of P, can be programmed so that all the subtasks of T can executed in parallel. Then P' executes n times faster than P. A failure of CPU is fatal to a sequential processor, but not in the case of parallel processor. The parallel processors are thus able to execute the programs much faster. Some of the applications of parallel computer (processors) are:

1. Expert system for AI
2. Fluid flow analysis,
3. Seismic data analysis
4. Long range weather forecasting,

5. Computer Assisted tomography

6. Nuclear reactor modeling,

7. Visual image processing

8. VLSI design

The typical characteristic of parallel computing are vast amount of computation, floating point arithmetic, vast number of operands.

We can derive different *computation models* based on the following logical conclusions. We assume that a given computation can be divided into concurrent tasks for execution on a multiprocessor. As per the *equal duration model* a given task can be divided into n equal subtasks, each of which can be executed by one processor. If t_s is the execution time of the whole task using a single processor, then the time taken by each processor to execute its subtask is $t_m = t_s/n$. Since, according to this model, all processors are executing their subtasks in parallel, then the time taken to execute the whole task is $t_m = t_s/n$.

The speedup factor of a parallel system can be defined as the *ratio between the time taken by a single processor to solve a given problem instance to the time taken by a parallel system consisting of* n *processors* to solve the same problem instance. We can define an *speedup factor* $S(n)$ as,

$$\begin{aligned} S(n) &= \frac{t_s}{t_m} \\ &= \frac{t_s}{t_s/n} = n \end{aligned} \tag{1}$$

0.2 Pipelining

A typical example of parallel processing is a one-dimensional array of processors, where there are n identical processors $P_1 \ldots P_n$ and each having its local memory. These processors communicate by message passing (send - receive). The figure 3 shows the pipelining model with m segments.

Consider that there are total n operations going on in parallel. A pipe line constitutes a sequence of processing circuits, called segments or stages. The m stage pipeline has same throughput as m separate units. Figure 4 shows a pipeline model with segments.

The pipeline processors are divided into tow major categories, as follows.

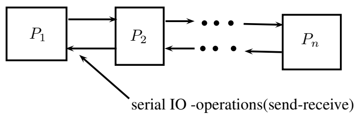

Figure 3: Pipeline processing.

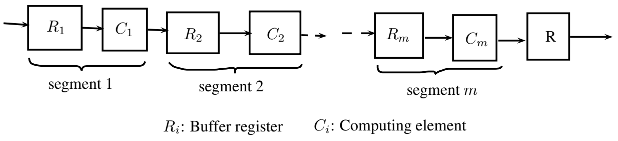

Figure 4: Pipeline segments.

Instruction pipeline: Transfer of instructions takes place through various stages of CPU executions, during an instruction cycle, for example, *fetch*, *decode*, *execute*. Thus, there can be three different instructions in different stages of execution: one getting fetched, previous of that is getting decoded, and previous to that is getting executed.

Arithmetic pipeline: The data is computed through different stages, like, the instructions process in part.

The figure 5 shows the *instruction* and *data pipeline* processor.

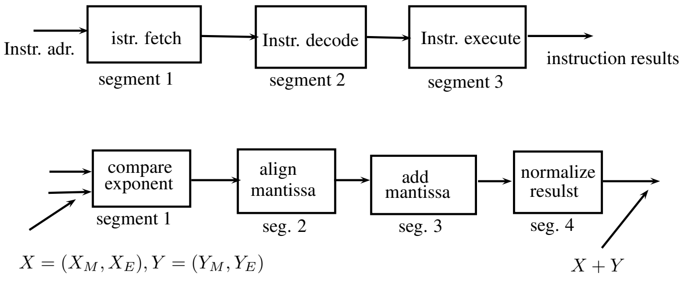

Figure 5: Instruction and data pipeline examples.

Example 0.2.1 *Consider an example to compute:* $A_i * B_i + C_i$*, for* $i = 1, 2, 3, 4, 5$*. Each segment has* r *registers, a multiplier, and an adder unit.*

Solution: Consider the example shown in figure 6.

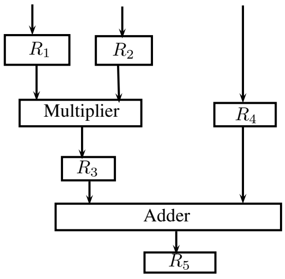

Figure 6: A segment comprising registers and computing elements.

Table 1: Computation of expression $A_i * B_i + C_i$ in space and time in 3-stage pipeline.

Clock pulse number	Segment 1 $R_1,\ R_2$	Segment 2 $R_3,\ R_4$	Segment 3 R_5
1	$A_1,\ B_1$	$-,\ -$	-
2	$A_2,\ B_2$	$A_1 * B_1,\ C_1$	-
3	$A_3,\ B_3$	$A_2 * B_2,\ C_2$	$A_1 * B_1 + C_1$
4	$A_4,\ B_4$	$A_3 * B_3,\ C_3$	$A_2 * B_2 + C_2$
5	$A_5,\ B_5$	$A_4 * B_4,\ C_4$	$A_3 * B_3 + C_3$
6	$-\ -$	$A_5 * B_5,\ C_5$	$A_4 * B_4 + C_4$
7	$-\ -$	$-,\ -$	$A_5 * B_5 + C_5$

$$R_1 \leftarrow A_i, R_2 \leftarrow B_i; \textit{input } A_i, B_i$$
$$R_3 \leftarrow R_1 * R_2, R_4 \leftarrow C; \textit{multiply}$$
$$R_5 \leftarrow R_3 + R_4; \textit{add } C_i \textit{ to product}$$

□

Example 0.2.2 *Computation of expression $A_i * B_i + C_i$ in space and time in 3-stage pipeline.*

Solution: The sequence of steps are represented by the table 1.

Any operator that can be decomposed into a sequence of sub-operations of about the same components can be implemented by pipeline processor. Consider that for a k-segment pipeline with clock cycle time $= t_p$ sec., with total n no. of tasks $(T_1, T_2, \ldots, T_n)$ are required to be executed. T_1 requires time equal to $k.t_p$ secs. Remaining $n-1$ tasks emerge from the pipeline at the rate of one task per clock cycle, and they will be completed in time of $(n-1)t_p$ sec, so total clock cycles required $=$ $k+(n-1)$. For $k=3$ segment and $n=5$ tasks it is $3+(5-1)=7$, as clear from table 1.

0.3 Computational Models

Consider an instruction pipeline unit (segment) that performs the same operation and takes time equal to t_u to complete each task. Total time for n tasks is $n.t_u$. The *speedup* for no. of segments as k and clock period as t_p is:

$$S(n) = \frac{n.t_u}{(k+(n-1))t_p} \tag{2}$$

For large number of tasks, $n >> k-1$, $k+n-1 \approx n$, so,

$$\begin{aligned} S(n) &= \frac{n.t_u}{n.t_p} \\ &= \frac{t_u}{t_p} \end{aligned}$$

Instruction pipelining is similar to use of assembly line in manufacturing plant An instruction's execution is broken in to many steps, which indicates the scope for pipelining. The pipelining requires registers to store data between stages.

Parallel computation with serial section model: It is assumed that at least a fraction f of a given task (computation) cannot be divided into concurrent subtasks. The remaining part $(1-f)$ is assumed to be dividable. For example, f may correspond to data input. Thus, time required to execute the task on n processors is:

$$t_m = f.t_s + (1-f).\frac{t_s}{n} \tag{3}$$

The speedup is therefore,

$$\begin{aligned} S(n) &= \frac{t_s}{f.t_s + (1-f).\frac{t_s}{n}} \\ &= \frac{n}{1+(n-1).f} \end{aligned} \tag{4}$$

So, $S(n)$ is primarily determined by the code section, which cannot be divided. If task is completely serial ($f = 1$), then no speedup can be achieved even by parallel processors.

For $n \to \infty$,

$$S(n) = \frac{1}{f} \tag{5}$$

which is maximum speedup.

Thus, improvement in performance (speed) of parallel algorithm over a sequential is limited not by number of processors but by fraction of the algorithm (code) that cannot be parallelized. This is called *Amdahl's law.* Considering the communication overhead t_c between parallel processing units, we get modified equation of 4, as follows:

$$\begin{aligned} S(n) &= \frac{t_s}{f.t_s + (1-f)(t_s/n) + t_c} \\ &= \frac{n}{f.(n-1) + 1 + n(t_c/t_s)} \end{aligned} \tag{6}$$

For $n \to \infty$,

$$\begin{aligned} S(n) &= \frac{n}{f(n-1) + 1 + n(t_c/t_s)} \\ &= \frac{1}{f + (t_c/t_s)} \end{aligned} \tag{7}$$

Thus, $S(n)$ depends on communication overhead t_c also.

Instruction Pipe-lining: typical stages of pipeline are:

1. FI (fetch instruction)

2. DI (decode Instruction)

3. CO (calculate operands)

4. FO (fetch operands)

5. EI (execute instruction)

6. WO (write operands)

The above stages are self explanatory. To better understand instruction pipelining, we solve the following exercise.

Table 2: Execution of 9 instructions in pipeline.

Time→ Instr.↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I_1	FI	DI	CO	FO	EI	WO								
I_2		FI	DI	CO	FO	EI	WO							
I_3			FI	DI	CO	FO	EI	WO						
I_4				FI	DI	CO	FO	EI	WO					
I_5					FI	DI	CO	FO	EI	WO				
I_6						FI	DI	CO	FO	EI	WO			
I_7							FI	DI	CO	FO	EI	WO		
I_8								FI	DI	CO	FO	EI	WO	
I_9									FI	DI	CO	FO	EI	WO

Example 0.3.1 *There are total 9 different instructions are to be executed. The six stage pipeline can reduce the execution time for 9 instructions to 14 time units, otherwise it is 54 time units.*

Solution: We would execute nine instructions ($I_1 - I_2$) in pipeline, as shown in table 2. We note, that it is done in 14 clock cycles. □

However, in the pipeline executions, the things are not always as we assumed. Following are some things contrary to what we assumed. And, they also need to be considered while designing the pipeline.

The diagram assumes that each instruction goes through 6 stages of pipeline. But, for example, a *load* instruction does not need WO. It is also assumed that there is no memory conflicts. However, FI, FO, WO all require memory access (together), and they appear in a vertical column.

The data value may be in cache, or FO/WO may be null.

Six stages may not be of equal duration, which we have taken each as one cycle.

A conditional branch/interrupt instruction may invalidate several fetches, which we come to know on decoding of the operand. After which stage it should check for conditional branch/interrupt?

And so on.

Hence, all the above factors need to be considered while designing a pipeline.

0.3.1 Factors effecting Instruction Pipe-lining

There is an overhead in each stage of pipeline for data movements buffer to buffer. Amount of control logic needed to handle memory/register dependencies increases with size of pipeline. It needs time for the buffers to operate. This is called *Pipeline Hazard*, and it occurs when pipeline or its portion stalls. There are resource, data, and control hazards.

Resource hazard: Two or more instructions in pipeline require same resource (say ALU or register. (Also called structure hazard).

Data hazards: This is due to conflict in memory access.

Control hazards: It is called branch hazard, and results due to wrong decision in branch prediction.

0.4 Vector Processing

In many computational applications, a problem can be formulated in terms of vectors and matrices. Processing these by a special computer is called *vector processing.* A vector is:

$$V = [V_1 V_2 V_3 \dots V_n] \tag{8}$$

The index for V_i is represented as $V[i]$. A program for adding two vectors A and B of length 100, to produce vector C is given as below for scalar quantities as

```
for(i=0; i < 100; i++)
  c[i]=b[i]+a[i];
```

In machine language we write it as:

```
      mvi i, 0
loop: read A[i]
      add B[i]
      store C[i]
      store i = i +1
      cmp i, 100
      jnz loop
```

This requires the fetching and decoding the same instructions again and again.

In vector processing, it requires to access the arrays A and B, and only counter needs to updated. The vector processing computer eliminates the need of fetching the instructions, and executing them. As they are fetched once only, decoded once only, but executes them 100 times. This allows operations to be specified only as:

$$C(1:100) = A(1:100) + B(1:100)$$

Vector instructions includes the initial address of operands, length of vectors, and operands to be performed, all in one composition instruction. The addition is done with a pipelines floating pointing point adder. It is possible to design vector processor to store all operands in registers in advance. It can be applied in matrix multiplication, for $[l \times m] \times [m \times n]$.

Exercises

1. Design a pipeline configuration to carry out the task to compute:

$$(A_i + B_i)/(C_i + D_i)$$

2. Construct pipeline to add 100 floating point numbers, i.e., find the result of $x_1 \times x_2 \times \ldots x_{100}$.

3. (a) List the advantages of designing a floating point processor in the form of a k-segment pipeline rather than a k-unit parallel processor.

   (b) A floating-point pipeline has four segments S_1, S_2, S_3, S_4, whose delays are 100, 90, 100, and 110 nano-secs, respectively. What is the pipeline's maximum throughput in MFLOPS?

4. It is frequently argued that large (super) computer is approaching its performance limits, and the future advances in large computers will depend on interconnected large number of inexpensive computers together. List the arguments against and favor.

5. List the features to be added in sequential programming languages, to use them in large interconnection of small inexpensive computers.

6. Let $S_1, S_2, \ldots, S_k$ denote the sequence of k-operations on a program. Suppose that execution of these operations on a uni-processor produces the same results regardless of the oder of execution of the k S_i's. Show that this does not imply that the S_i's are parallelizable on a multi-processor.

7. Prove that general problem of determining whether two program segments S_1, S_2 are parallelizable is undecidable by showing that a solution to this problem implies a solution to the halting problem for Turing machine. (Hint: Assume that an algorithm A to determine parallelization exists, and consider applying A to a program containing the statement: **if** Turing machine T halts after at most n steps **then** S_1 **else** S_2).

Bibliography

[1] M. Morris Mano, “Computer System Architecture”, 3nd Edition, Pearson, 2006.

[2] William Stalling, "Computer Organization and Architecture", 8th Edition, Pearson, 2010.

[3] John P. Hayes, “Computer Architecture and Organization”, 2nd Edition, McGraw-Hill International Edition, 1988.