Parallel Processing for Financial Valuation Problems

Les Clewlow

Research Fellow
Financial Options Research Centre

April 1991
Revised: January 1993

We would like ot acknowledge helpful comments from participants of a seminar held at Bridge Information Systems on 27 February 1992. Any errors remian our own.

Funding for this work was provided by corporate members of the Financial Options Research Centre: Bankers Trust, LIFFOE, London Clearing House, London Fox, Midland Global Markets, Mitsubishi Finance International, Nomura Bank International, UBS Phillips and Drew.

Financial Options Research Centre
Warwick Business School
University of Warwick
Coventry
CV4 7AL
Phone: 0203 523606

FORC Preprint: 93/22

Parallel Processing for Financial Valuation Problems

ABSTRACT

The major disadvantage of serial computers is that the set of computations or tasks required to solve a given problem must be performed sequentially. Some or all of these may be computationally independent and could therefore be performed simultaneously. Performing the computations sequentially is obviously sub-optimal. The solution is a computer which can perform many computations simultaneously. This is the recent and rapidly developing field of parallel processing.

In this paper we introduce parallel processing terminology and concepts and examine some examples of applying these techniques to financial valuation problems with particular emphasis on contingent claim related problems.

Keywords: Parallel Processing, Valuation, Efficiency

1 Introduction

The implementation of parallel processing has been achieved in various ways. Probably the most obvious is to have a multiplicity of functional units. If we have more than one processing unit (PU) or processing element (PE) then any subset of tasks which can be performed simultaneously can each be assigned a separate PE and thus processed simultaneously.

Pipelining was the first technique to be used commercially to improve the performance of computers. Figure 1 depicts one type of pipelining, called instruction pipelining, which has been used. The idea is that individual operations, in this case instruction execution, can be further divided into a set of simpler sequential operations. Thus instruction execution can be divided into instruction fetch, instruction decode, operand fetch and execution. Each of these operations is performed by a separate sub-unit of the processing unit. In a standard processing unit, once a sub-unit has completed its task, it will be idle until execution of the next instruction. With pipelining the sub-units operate continuously, after completing its task for one instruction it can immediately begin its task for the next instruction. This leads to a reduction in total execution time as depicted in Figure 1.

We can overlap processing and input/output operations. We want to avoid

using the processing units for simple tasks such as input and output of data to and from storage devices. These devices are likely to be very slow in comparison with the processing units which would therefore waste large amounts of time waiting. The answer is to provide simple input/output (I/O) processors with direct memory access (DMA) channels. The I/O processors can then perform any I/O operations at the same time as the main processors are operating.

Another approach to easing the I/O bottle-neck caused by the relative slowness of mass storage devices is hierarchical memory systems. Figure 2 shows a typical system. Firstly there is a high speed cache between the processor and main memory. The contents of the cache will depend on the cache management policy, but will typically be a copy of a block of main memory which has a high probability of being accessed next. For example the most recently accessed block of main memory. The main memory will be organised on a virtual memory basis in which the most recently accessed blocks of data are kept in main memory. If access to a block not in main memory is requested the requesting process is blocked or suspended. A ready-to-run process is then started while the requested block is fetched from the next level of the memory system. This arrangement is duplicated through all levels of the hierarchy. When a block of memory is modified the update policy determines when this is copied through to the higher levels which can be done

concurrently with the running processes. This system allows the processes to read and write at the speed of the cache as often as possible.

Finally we have multiprogramming and time-sharing. These are essentially methods of using uniprocessor resources more efficiently. Figure 3 illustrates the processing structure for normal sequential, multiprogrammed and time-shared execution of three programs, P1, P2 and P3. With multiprogramming we simply allow the uniprocessor to service the next program while the current program is involved in non-processor I/O operations. With time-sharing we allocate fixed or variable (in Figure 3 it is fixed) time slices to each program. The processor then services each program in turn. This allows even more efficient computation and I/O concurrency. This approach is most effective for multiuser systems. Since the time between time slices allocated to a given program is small compared to human response time, individual users do not notice the phases when their program is waiting for processor service.

2 Architectural Schemes

There is a very wide range of different architectures both in commercial systems and research systems. Many classification schemes have been proposed in an attempt to unify the various approaches, facilitate comparison between them and

matching of the appropriate architecture to a given problem.

In 1966 Michael Flynn [Flynn,1966] proposed a scheme based on the multiplicity of instruction streams and data streams. Here a stream refers to a sequence of related items. The four categories are,

- SISD - Single instruction stream, single data stream
- SIMD - Single instruction stream, multiple data streams
- MISD - Multiple instruction streams, single data stream
- MIMD - Multiple instruction streams, multiple data streams

These are depicted diagrammatically in Figure 4. The standard uniprocessor falls into the SISD category. SIMD architectures are also called *array computers* or *array processors*, the multiple processors all execute the same set of instructions but on different sets of data. This approach is applicable to finite difference algorithms for example. The MIMD category represents the most general form of parallel architecture. Here each processor can perform different operations on different items of data.

A scheme, introduced by Tse-yun Feng in 1972 [Feng,1972], classifies architectures by their maximum parallelism or the maximum number of bits which can be processed in a single clock cycle. This is represented by the product of the

word length and the *bit slice length.* The word length is the number of bits used to represent a single word of data and the bit slice length is the number of bits from different words being processed simultaneously. For example an architecture with 64 bit words, four pipelines and eight stages in each pipe would be represented as (64,32). Figure 5 shows a graph of some existing computer systems under Feng's scheme. Although this scheme allows comparison of the parallelism of various architectures, any inferences on their relative speeds would in general be misleading.

Wolfgang Händler developed a classification in 1977 [Händler,1977] which captures certain details of the architecture which the previous two schemes do not. The scheme is based on three architectural levels, the processor control unit (PCU), the arithmetic logic unit (ALU) and the bit level circuit (BLC). At the PCU level we consider the number of processors and the number that are pipelined. At the ALU level we the number of ALU's per PCU and the number that are pipelined. At the BLC level we consider the word length and the number of pipeline stages. An architecture is thus classified by six parameters. Figure 6 presents this graphically.

The contents of conventional computer memory is accessed by specifying the unique address of the data item. An alternative, associative array memories, are

content- addressable. In order to retrieve data a pattern is specified, the memory then returns all data items matching the pattern. The hardware to achieve this is considerably more complicated and expensive than for standard memory. This type of architecture is mainly used where very fast storage and retrieval of large amounts of data is required. For example in image processing, computer vision and artificial intelligence.

In a conventional computer computations are performed in the strict order they are specified in the list of instructions or program. In dataflow computers computation ordering is *data-driven*, that is it is based on data or operand availability.

3 Network structures

One of the most crucial factors in the design of parallel processing machines is the interconnection network between the processors and the memory modules. The topology of the network can be characterised by the routing functions. Let each of the N PE's in the network have a unique address in the set $A = \{0, ..., N-1\}$. Then the routing functions are bijections from A to A. The network may be static or dynamic. Figure 7 shows some typical static networks. Dynamic networks may be single-stage or multi-stage. A single-stage network has N input selectors

(IS) and N output selectors (OS) (Figure 8). The IS are 1 to D demultiplexers and the OS are M to 1 multiplexers where $1 \leq D \leq N$ and $1 \leq M \leq N$. A single-stage network is also called a *recirculating network* because data may have to recirculate through the single stage in order to reach their destinations. The number of recirculations needed depends on the connectivity of the single-stage, the higher the connectivity the lower the number of recirculations. The limit is the crossbar network in which $D = M = N$ which only requires one pass through the stage. Its cost however is $O(N^2)$. Multi-stage networks are capable of connecting an arbitrary input terminal to an arbitrary output terminal. They are either *blocking*, *rearrangeable* or *nonblocking*. In blocking networks, simultaneous connection of more than one input/output pair may result in conflicting use of network links. Examples of this class are the *data-manipulator*, *Omega*, *flip*, *n-cube* and *baseline*. A network is rearrangeable if existing connections can always be rearranged so that a new input/output pair can be connected. The *Benes* network is an example of this class. The nonblocking network can connect all possible input/output pairs without blocking. The *Clos* and *crossbar* networks are examples of this class.

4 Parallel language features

There are two approaches to producing programs for parallel machines. The algorithm can be coded for a standard sequential computer and the compiler can be constructed to search for potential parallelism within the sequential code. This approach is normally adopted for sequential code which already exists. The design of compilers which can perform this task is extremely complicated and they will typically compile orders of magnitude slower than standard sequential compilers. The second approach is to add parallel constructs to the high- level language so that the parallelism within the algorithm can be made explicit. The compiler then has only to convert these to the machine dependent instruction streams. The Bernstein condition [Bernstein,1966] determines whether sequentially organised processes can be executed in parallel. It can be stated as follows,

Let I_i be the set of all variables which task T_i reads.

Let O_i be the set of all variables which task T_i writes.

Then for the sequential tasks T_1 and T_2 to be executable in parallel we require,

$I_1 \cap O_2 = \emptyset$

$I_2 \cap O_1 = \emptyset$

These are general conditions which essentially state that the sets of variables used in the two tasks must not intersect in anyway. This is sufficient but not necessary since in certain cases it may be possible for the tasks to be performed in parallel even though the variable sets do intersect.

Parallel languages are generally block structured languages with extensions to the block and loop structures to allow specific reference to the parallelism [Dijkstra,1968]. Some examples are,

begin

S_0 ;

PARbegin S_1 ; S_2 ; ... ; S_n ; **PARend**

S_{n+1} ;

end

Here the S_n are statements and the **PARbegin, PARend** block indicates that the statements within can be assigned to different processors to be executed in parallel.

$s = n/p$;

PARfor $i = 1$ to p **do**

for $j = (i-1)s + 1$ **to** si **do**

begin

$C(j) = 0$;

for $k = 1$ **to** n **do**

$C(j) = C(j) + A(j,k)B(k)$;

end

This algorithm is performing the matrix multiplication $C = A * B$ where A is $n * n$ and B is $n * 1$ and the number of processors p is less than n such that $s = n/p$.

There may be cases where one task requires temporary use of a variable which is used by another task. This is called a *critical section* and the variable must be declared as **shared**,

var v : **shared** V ;

var w : **shared** W ;

PARbegin

csect v **do** P ;

csect w **do** Q ;

PARend

V and W are variable types and we have stated that the block P is a critical section for v and similarly Q for w. The critical sections will have exclusive use

of the critical variable and any other processes attempting to use it will be forced into a wait state until the critical section is completed.

5 Parallel algorithms for SIMD machines

In this section we give examples of finance problems suitable for solution on SIMD machines.

5.1 Term structure estimation

The term structure of interest rates is defined by the discount factors on a collection of default-free pure discount bonds with different times to maturity. The term structure is required to value interest rate dependent and derivative securities. Steeley [1991] has shown how the term structure can be estimated as a linear combination of B-splines by least squares regression. Assume we have m default free bonds of price P_i, paying a coupon $C_{i,j}$ at time $t_j, j = 1, ..., n; n > m$. Then we have

$$P_i = \sum_{j=1}^{n} C_{i,j} d_{t_j} \tag{1}$$

If replace the discount factors d_{t_j} with a continuous approximation

$$d(t) = \sum_{l=1}^{L} \alpha_l f_l(t) \tag{2}$$

we obtain

$$P_i = \sum_{l=1}^{L} \alpha_l \sum_{j=1}^{N} C_{i,j} f_l(t) \tag{3}$$

In matrix notation we have a linear regression problem

$$P = D\alpha \tag{4}$$

where P is the vector of gross price observations, α is the vector of approximation coefficients and D is the matrix of summed products of cash flows ($C_{i,j}$) and evaluated B-splines ($f_l(t)$). The solution to this is

$$\hat{\alpha} = (D'D)^{-1}D'P \tag{5}$$

This requires matrix multiplication and inversion. Matrix inversion can be achieved by LU-decomposition and back substitution which can be decomposed into parallel operations on sub-matrices. Matrix multiplication can be decomposed extremely efficiently into parallel operations. Consider the multiplication $D'D$, this requires L^2m cumulative multiplications so on a SISD machine requires

$O(L^2m)$ time. A parallel algorithm for L processing elements for this matrix multiplication can be specified as follows,

for $i = 1$ **to** L

begin

PARfor $j = 1$ **to** L $c_{i,j} = 0$;

for $k = 1$ **to** m

PARfor $j = 1$ **to** L

$$c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j} \ ;$$

end

This algorithm requires $O(mL)$ time. If we have mL processors then the outermost for loop can be parallelised also and we obtain an $O(m \log_2 L)$ time algorithm.

5.2 Asian options by fast Fourier transform

Asian call options pay the difference if positive between the arithmetic average of the price over a specified period and the previously agreed strike price. Carverhill and Clewlow (1990) showed how Asian options could be valued by fast Fourier transforms. Consider the formula for the average A,

$$A \equiv \frac{1}{n}[X_1 + ... + X_n] \tag{6}$$

where the X_i are the values of the underlying asset at times t_i. Now if the underlying asset is lognormally distributed then we can write

$$X_i = \exp(Y_0 + Z_0^1 + ... + Z_{i-1}^i) \tag{7}$$

where Y_0 and the increments Z_i^{i+1} are all independent and normal, with mean $((r-s) - \frac{1}{2}\sigma^2)(\frac{T}{n})$ and variance $\sigma^2(\frac{T}{n})$, where T is the time to maturity of the option. Thus,

$$A = \frac{1}{n}[\exp(Y_0 + Z_0^1) + \exp(Y_0 + Z_0^1 + Z_1^2) + ... \exp(Y_0 + Z_0^1 + Z_1^2 + ... + Z_{n-1}^n)] \tag{8}$$

$$A = \frac{1}{n}\exp Y_0[\exp Z_0^1(1 + \exp Z_1^2(1 + \exp Z_2^3(1 + ...(1 + \exp Z_{n-1}^n)...)] \tag{9}$$

We can obtain the probability density of the average from the densities of the normal distributions $Z_0^1, ..., Z_{n-1}^n$ by the following inductive procedure,

$$\tilde{A}_{n-1} = Z_{n-1}^n, \tag{10}$$

$$\tilde{A}_{i-1} = Z_{i-1}^i + \ln(1 + \exp(\tilde{A}_i),\ i = n-1, ..., 1, \tag{11}$$

$$A_0 = Y_0 - \ln(n) + \tilde{A}_0 \tag{12}$$

$$A = \exp A_0 \tag{13}$$

The density $f_{\log(1+\exp \tilde{A}_i)}$ is obtained from the density $f_{\tilde{A}_i}$ by the transformation

$$f_{\log(1+\exp X)}(x) = \frac{\exp(x)}{\exp(x)-1}\log(\exp(x)-1) \tag{14}$$

We can obtain the density for the sum of this and the normal density Z_{i-1}^{i} by convolution of the two densities. The convolution can be performed efficiently by fast Fourier transforming the densities, multiplying the results and then fast Fourier transforming the result back again.

When performing the Fourier transform numerically we strictly perform the discrete Fourier transform (DFT). Consider $s(k)$, $k = 0, ..., M-1$ to be M samples of a time series, the DFT $x(j)$, $j = 0, ..., M-1$ is defined as,

$$x(j) = \sum_{k=0}^{M-1} s(k)e^{2\pi ijk/M} \ , \ j = 0, ..., M-1 \tag{15}$$

Now let $f(m) = s(m)$ and $g(m) = s(m+M/2)$ for $m = 0, ..., M/2-1$. The DFT of the M-point sequence $s(k)$, $k = 0, ..., M-1$ can be computed in terms of the two $M/2$-point sequences $\{f(m)+g(m)\}$ and $\{(f(m)-g(m)).e^{2\pi im/M}\}$, $m = 0, ..., M/2$. Now consider a SIMD machine with $N = M/2$ processing elements arranged as an $\log_2(M/2)$-dimensional unit cube. If PE_k initially contains $s(k)$ and $s(k + M/2)$ then the $M/2$ butterfly operations at each of $\log_2 M$ stages computes the DFT as shown in Figure 9. (Hertz (1990) describes an efficient fast Fourier transform algorithm for the popular Connection Machine).

6 Parallel algorithms for MIMD machines

A parallel algorithm for a MIMD machine is a set of concurrent processes or tasks which must interact to synchronise and exchange data. If the processes synchronise at specific interaction points the algorithm is called synchronous. These interaction points divide the algorithm into stages. Processes may have to wait at the interaction points for slower processes. The speed of the algorithm is therefore determined by the slowest process or task. Another method of synchronisation is for the processes to communicate via dynamically updated global variables in shared memory. These algorithms are called asynchronous because processes do not necessarily have to wait for other processes. We now describe some finance problems suitable for solution on MIMD machines.

6.1 Stochastic volatility or interest rate models

The framework for developing a continuous time model with multiple state variables is well established (Garman (1976), Cox, Ingersoll and Ross (1985)). For stochastic volatility or interest rate models we have two state variables, the underlying asset and the volatility/interest rate. Consider a model where the state variables follow the random processes,

$$ds_i = \mu_i dt + \sigma_i dz_i, i = 1, 2 \tag{16}$$

The general pricing equation for a contingent claim which depends on the s_i is

$$\frac{\partial C}{\partial t} = rC - \sum_i (r - \alpha_i) s_i \frac{\partial C}{\partial s_i} - \frac{1}{2} \sum_i \sum_i \sigma_{i,j} \frac{\partial^2 C}{\partial s_i \partial s_j} \tag{17}$$

where α_i is the convenience yield for the asset i which is related to its price of risk λ_i by

$$\lambda_i \sigma_i = \mu_i - (r - \alpha_i) \tag{18}$$

This partial differential equation must typical be solved by finite difference techniques as there will not be analytical solution.

We approximate the partial differentials by finite differences on a grid of space mesh size h and time mesh size k,

$$\frac{\partial C}{\partial t} = \frac{C(s_i, s_j, t + k) - C(s_i, s_j, t)}{k} \tag{19}$$

$$\frac{\partial C}{\partial s_i} = \frac{C(s_i + h, s_j, t) - C(s_i - h, s_j, t)}{2h} \tag{20}$$

$$\frac{\partial^2 C}{\partial s_i \partial s_j} = \frac{\begin{array}{c} C(s_i + h, s_j + h, t) - C(s_i - h, s_j + h, t) \\ -C(s_i + h, s_j - h, t) + C(s_i - h, s_j - h, t) \end{array}}{4h^2} \tag{21}$$

We can rewrite this equation in terms of the mesh points,

$$C(l,m,n-1) = C(l,m,n) -$$

$$k\text{fn}\left(\begin{array}{c} C(s_i+h, s_j+h, t), C(s_i, s_j+h, t), C(s_i-h, s_j+h, t), \\ C(s_i+h, s_j, t), C(s_i, s_j, t) + C(s_i-h, s_j, t), \\ C(s_i+h, s_j-h, t), C(s_i, s_j-h, t), C(s_i-h, s_j-h, t) \end{array}\right) \tag{22}$$

where (l,m) represents the position (s_i, s_j) and n represents the time step.

The solution is then obtained by repeatedly solving for the value of the option on the space mesh at time step $n-1$ from the values at time step n from the known values at maturity back to the present.

We now assign a contiguous subset of the space mesh to each processor. Each processor then computes the nth step for each space mesh point in its subset. When all processors have completed this operation the mesh variables are updated with the new values. This procedure is continued until the value of the option at the present time is obtained.

In partitioning the algorithm into tasks for each of the processors we try to obtain tasks whose execution times are similar. For example tasks of the same complexity will have execution times which are independent and identically distributed.

6.2 Risk management of large portfolios

Finally a more general application would be an investor with a large portfolio of assets and derivative instruments which he wants to hedge. Computing the deltas and gammas for the multiple constituents of the portfolio will be a time consuming process. However the computations for each constituent will in general be independent, we can therefore partition the computations as in the previous section and assign each subset to a separate processor in a MIMD machine. Communication of intermediate results between processors where necessary is also possible within the general MIMD architecture. With this approach we can obtain a speed-up roughly equal to the number of processors. The speed-up and efficiency of a parallel solution are considered in more detail in the next section.

7 Performance of parallel processing machines

The maximum theoretical speed increase achievable by a parallel computer with N identical processor is at most N time faster than a computer with a single identical processor working on the same problem. But because of practical problems such as inter-processor communications, shared memory access conflicts and synchronisation of the multiple processors the maximum theoretical speed-up is never achieved. The speed-up (S) is formally defined as the ratio of the time taken by

a single processor over the time taken by N identical processors. The efficiency (η) is defined as the speed-up over the number of processors. Thus a machine with N processors which achieves a speed-up of N will have a perfect efficiency of 1, if its speed-up is less than N its efficiency will be less than 1. Minsky's conjecture states that the lower-bound on the speed-up is $\log_2 N$. We can calculate an expected speed-up with a simple computational model. Assume the problem divides into n computational modes and the ith mode can utilise i processors, all modes are equally probable (probability is then $1/n$), take unit time on a single processor and we have at least n processors. The expected time required to solve the problem is then

$$T_n = \sum_{i=1}^{n} \frac{1}{n}\frac{1}{i} \leq (\ln(n) + 1)/n \tag{23}$$

Therefore the speed-up is,

$$S \geq n/(\ln(n) + 1) \tag{24}$$

For a more practical example, consider a pipeline with k stages processing n tasks. If each stage takes unit time then a non-pipelined processor will take nk time units. For a pipelined processor the first k time units are required to fill the pipeline and thereafter results emerge every time unit, therefore the time taken is

$k + (n-1)$. The speed-up is thus,

$$S = nk/(k + (n-1)) \tag{25}$$

and the efficiency is therefore,

$$\eta = k/(k + (n-1)) \tag{26}$$

8 Summary

In this paper we have introduced parallel processing concepts. We have shown how they can be applied to a wide variety of financial valuation problems. Application of these techniques results in significant improvements in the speed of solution of the problem. Implementation of the techniques described is straightforward. Commercial parallel processing machines are provided with standard libraries for matrix operation, fast Fourier transforms and partial differential equation solution. For the more general problems characterised by portfolio risk management the major task is to partition the problem into computationally independent subsets. The system will then provide ways of assigning these subsets to the available processors. With parallel processing machines now providing from hundreds to thousands of processors the potential benefits cannot be denied.

References

Bernstein, A.J. , "Analysis of Programs for Parallel Processing", IEEE Transactions in Electronics and Computing E- 15, October 1966, pp. 746-757.

Carverhill, A. and Clewlow, L.J. , "Flexible Convolution", RISK Magazine 3, March 1990, pp. 25-29.

Cox, J.C., Ingersoll, J.E. and Ross, S.A. , "An Intertemporal General Equilibrium Model of Asset Prices", Econometrica 53, March 1985, pp. 363-384.

Dijkstra, E.W. , "Cooperating Sequential Processes", Programming Languages, F.Genuys (ed.), Academic Press, 1968, pp. 43-112.

Feng, T-Y. , "Some Characteristics of Associative/Parallel Processing", Proceedings 1972 Sagamore Computing Conference, Syracuse University, 1972, pp. 5-16.

Flynn, M.J. , "Very High Speed Computing Systems", Proceedings IEEE 54, 1966, pp. 1901-1909.

Garman, M. , "A General Theory of Asset Valuation Under Diffusion State Processes", Working paper no. 50, Graduate School of Business Administration, University of California, Berkeley, 1976.

Handler, W. , "The Impact of Classification Schemes on Computer Architecture", Proceedings 1977 International Conference on Parallel Processing, pp. 7-15.

Hertz, P. , "An algorithm for the fast Fourier transform on a Connection Machine", Computers in Physics 4(1), Jan/Feb 1990, pp. 86-90.

Steeley, J.M. , "Estimating the Gilt-Edged Term Structure: Basis Splines and Confidence Intervals", Journal of Business Finance and Accounting 18(4), June 1991, pp. 513-529.

Figure captions

Figure 1 : Instruction pipelining

Figure 2 : Hierarchical memory system

Figure 3 : Multiprogramming and time-sharing

Figure 4 : Flynn's classification scheme

Figure 5 : Some existing systems under Feng's classification scheme

Figure 6 : Handler's classification scheme

Figure 7 : Static networks

Figure 8 : Single stage network

Figure 9 : SIMD FFT computation

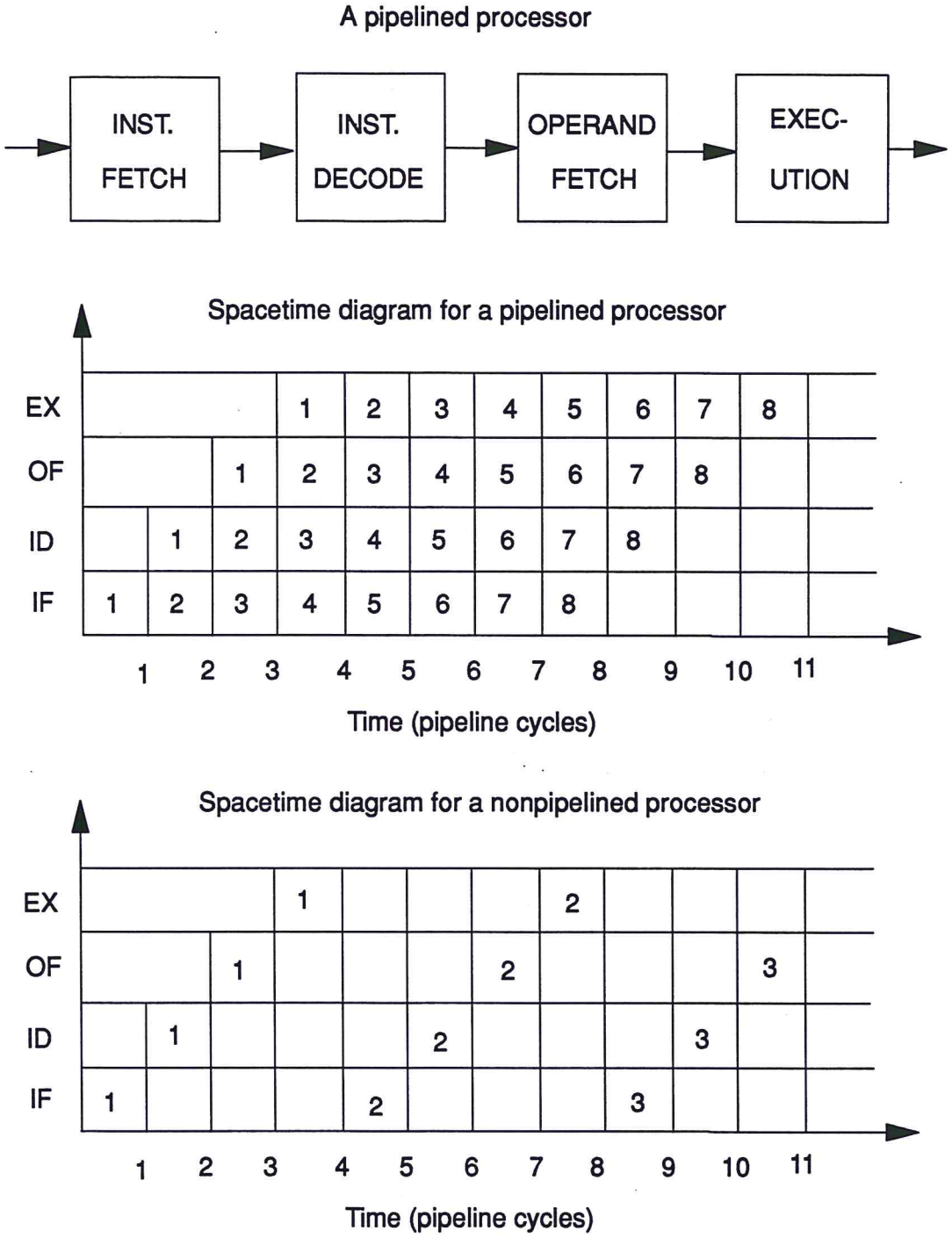


Figure 1: Instruction pipelining

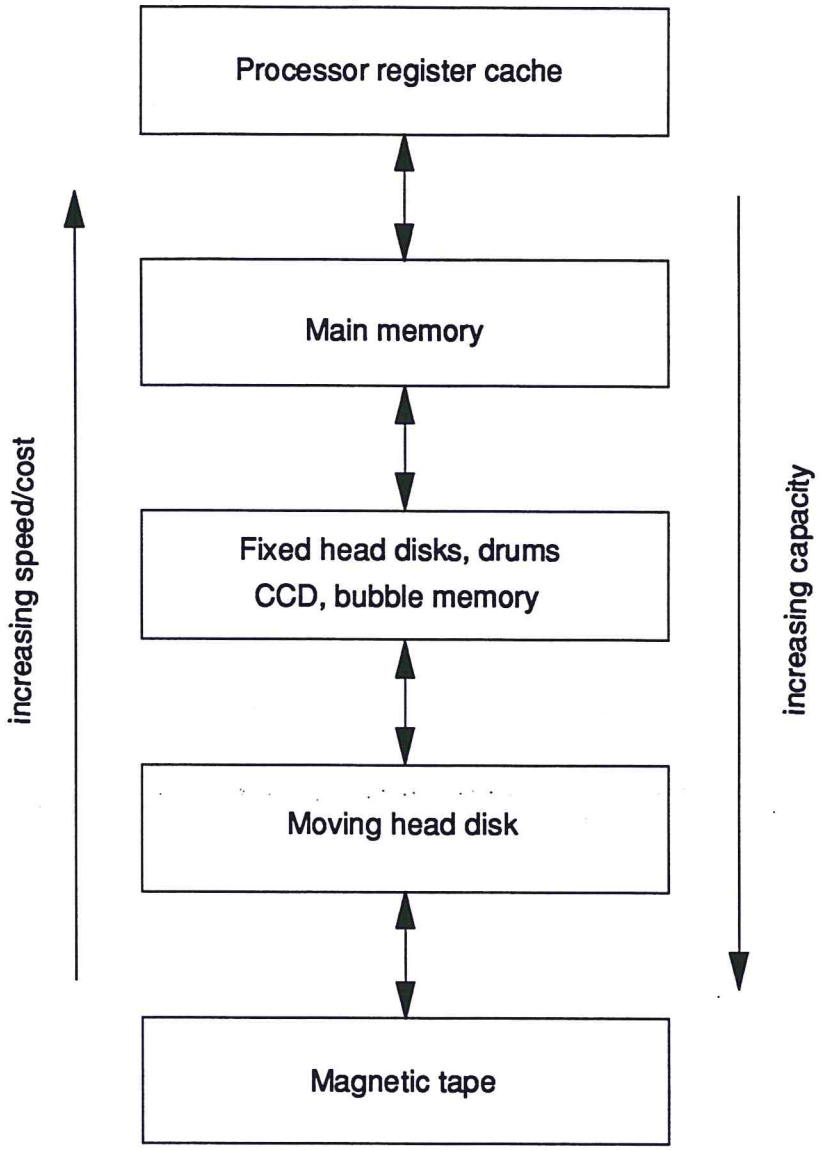


Figure 2: Hierarchical memory system

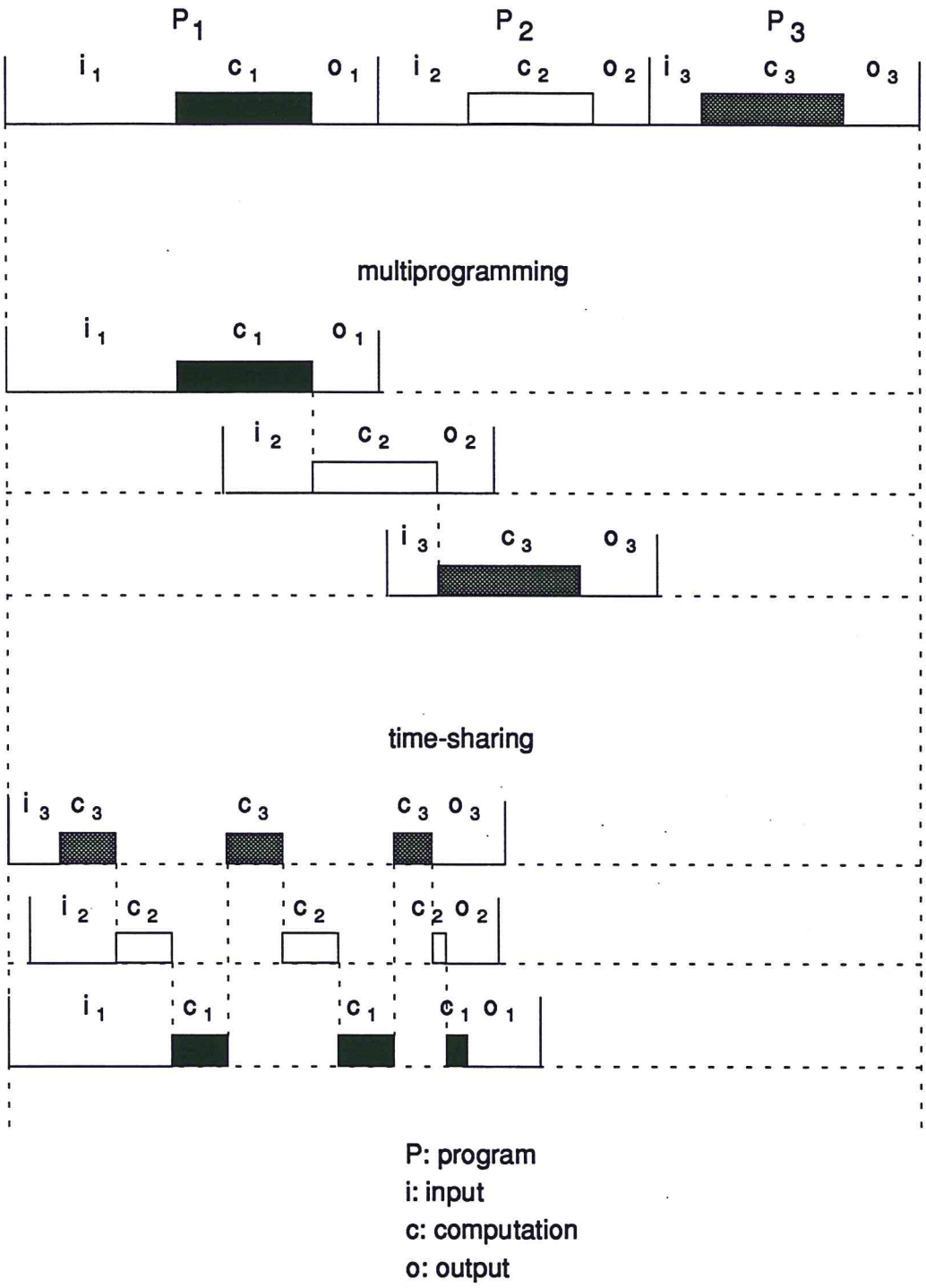
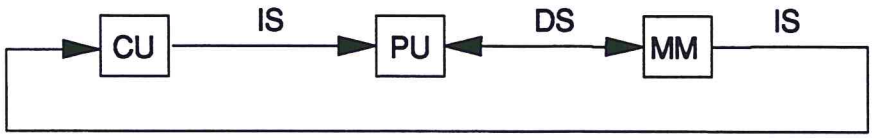
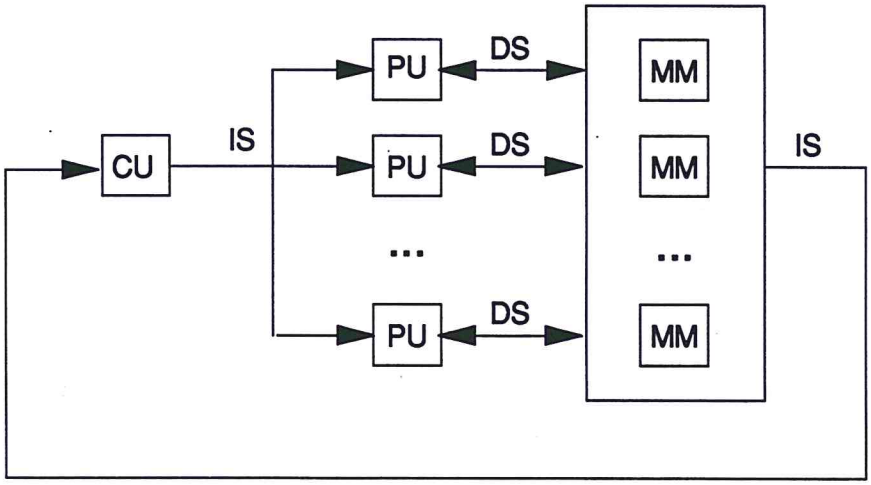


Figure 3: Multiprogramming and time-sharing

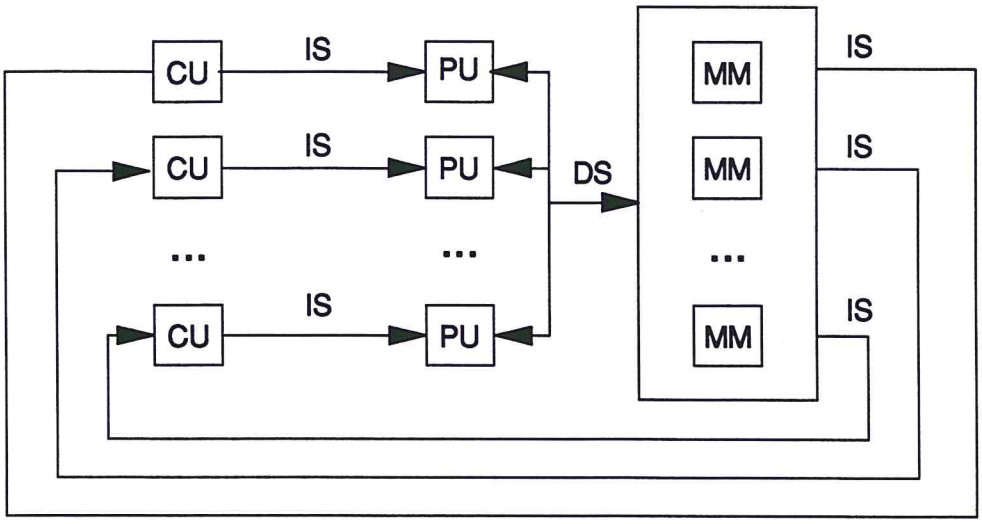


(a) SISD

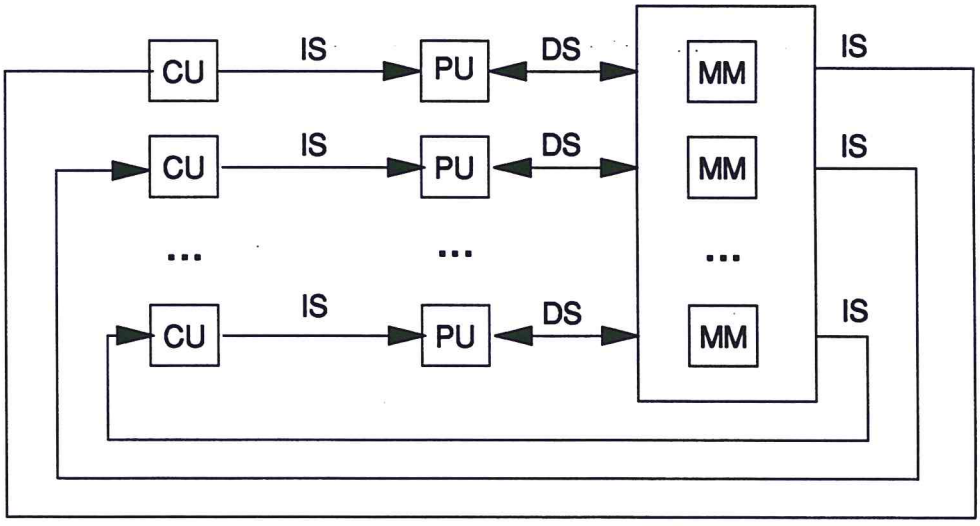


(b) SIMD

Figure 4: Flynn's classification scheme



(c) MISD



(d) MIMD

Figure 4: Flynn's classification scheme

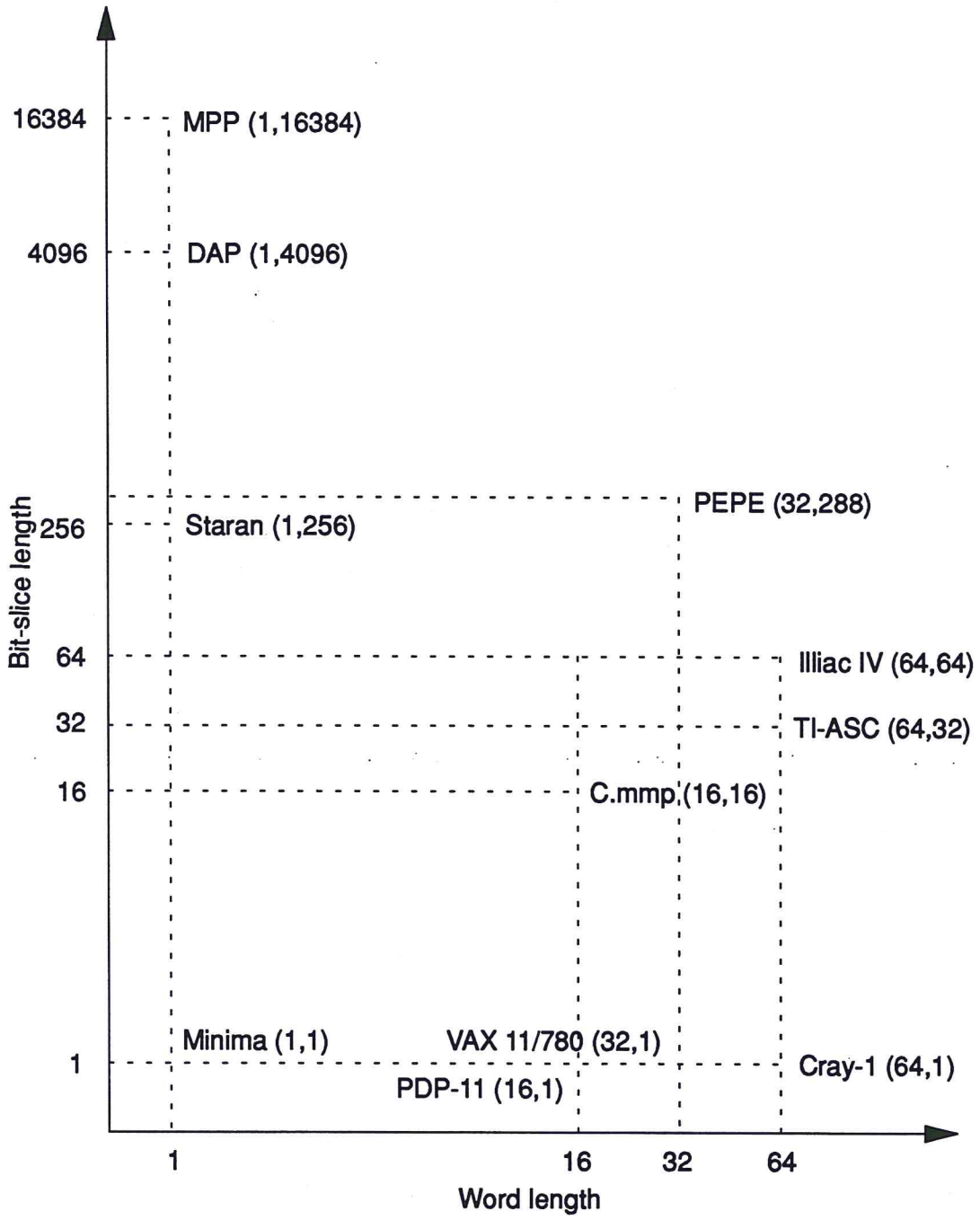


Figure 5: Some existing systems under Feng's classification scheme

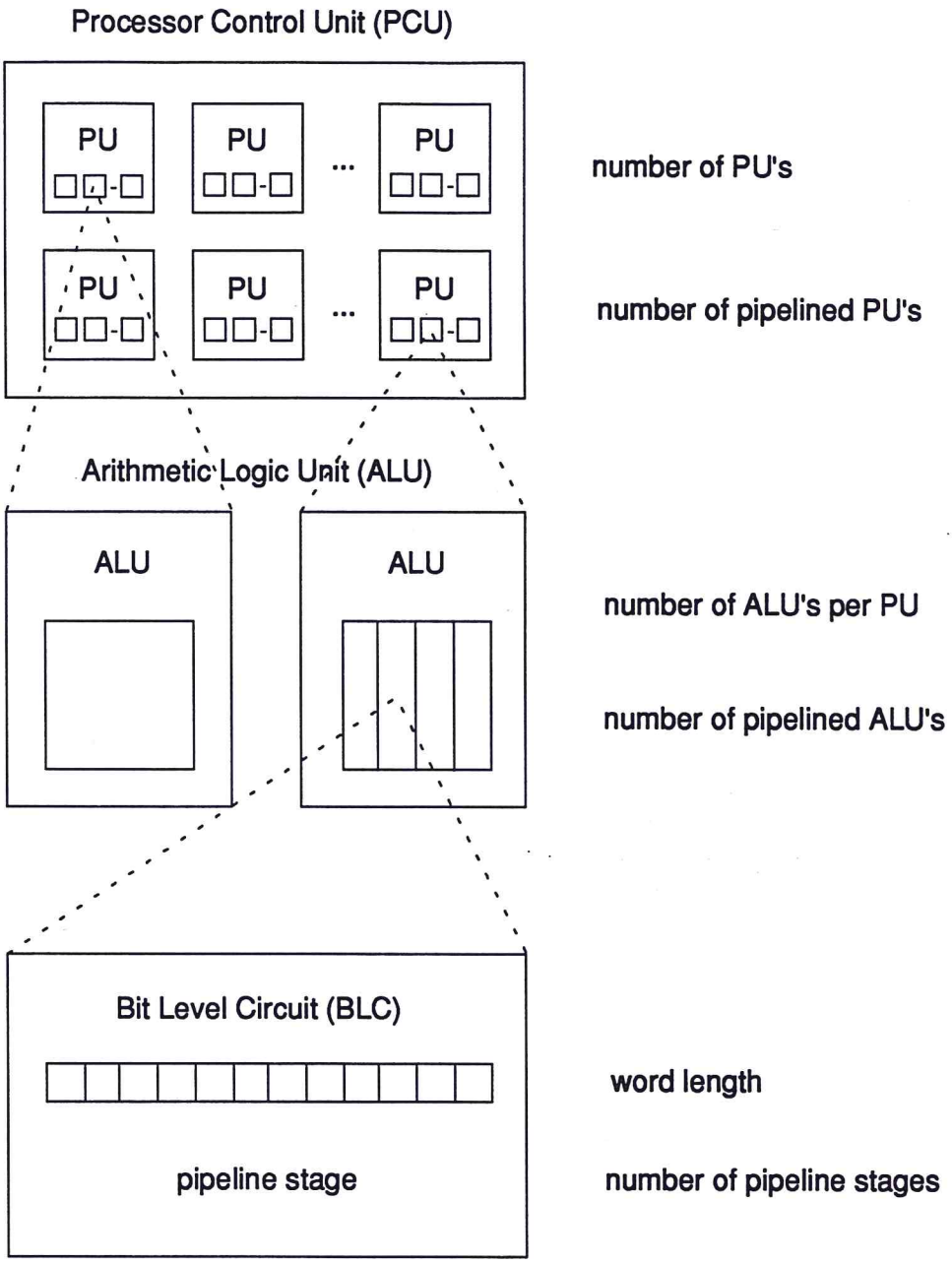


Figure 6: Handler's classification scheme

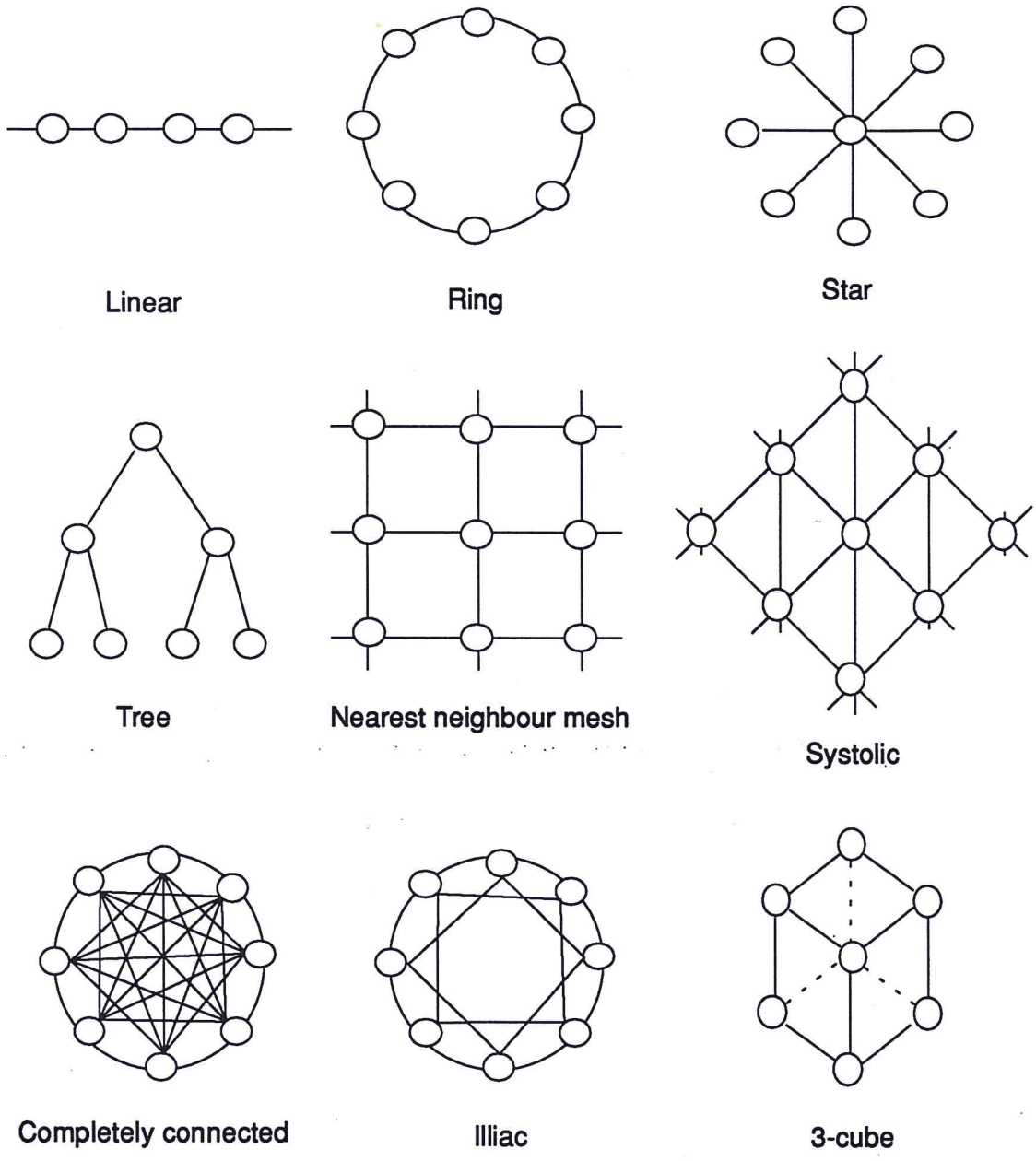


Figure 7: Static networks

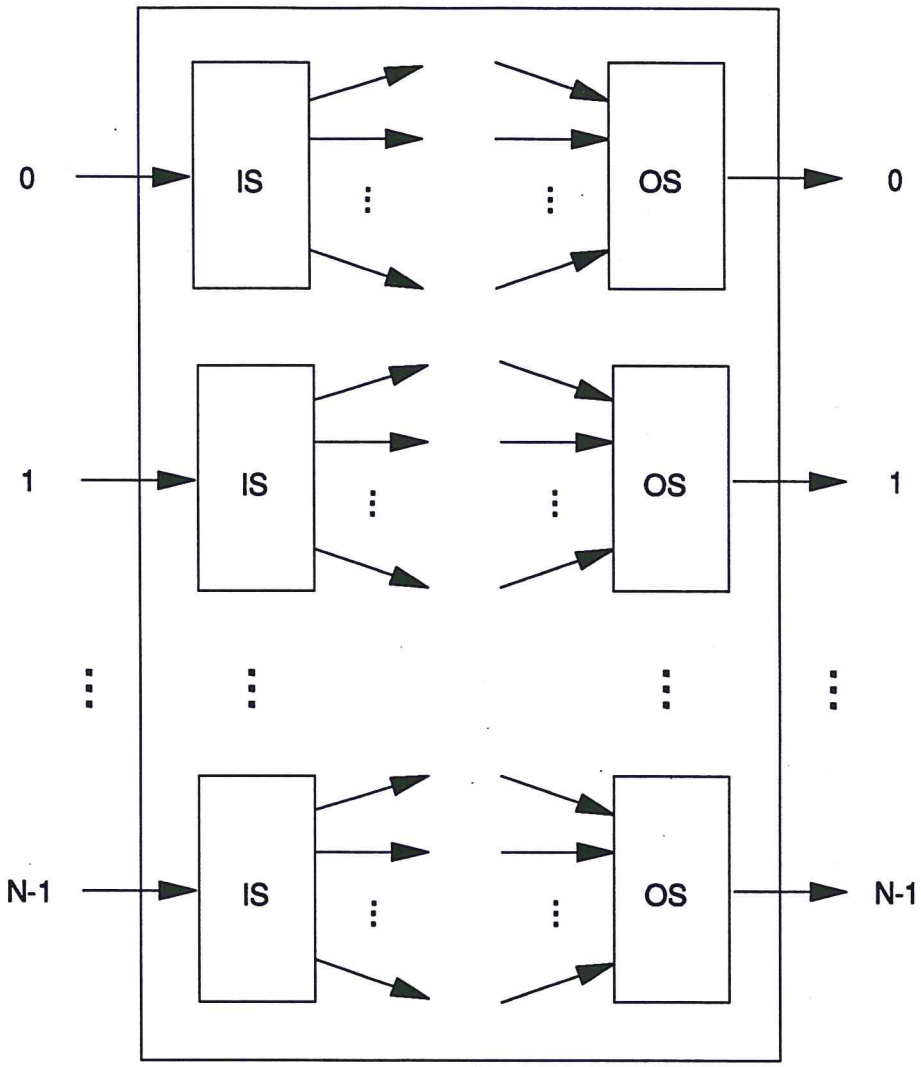


Figure 8: Single stage network

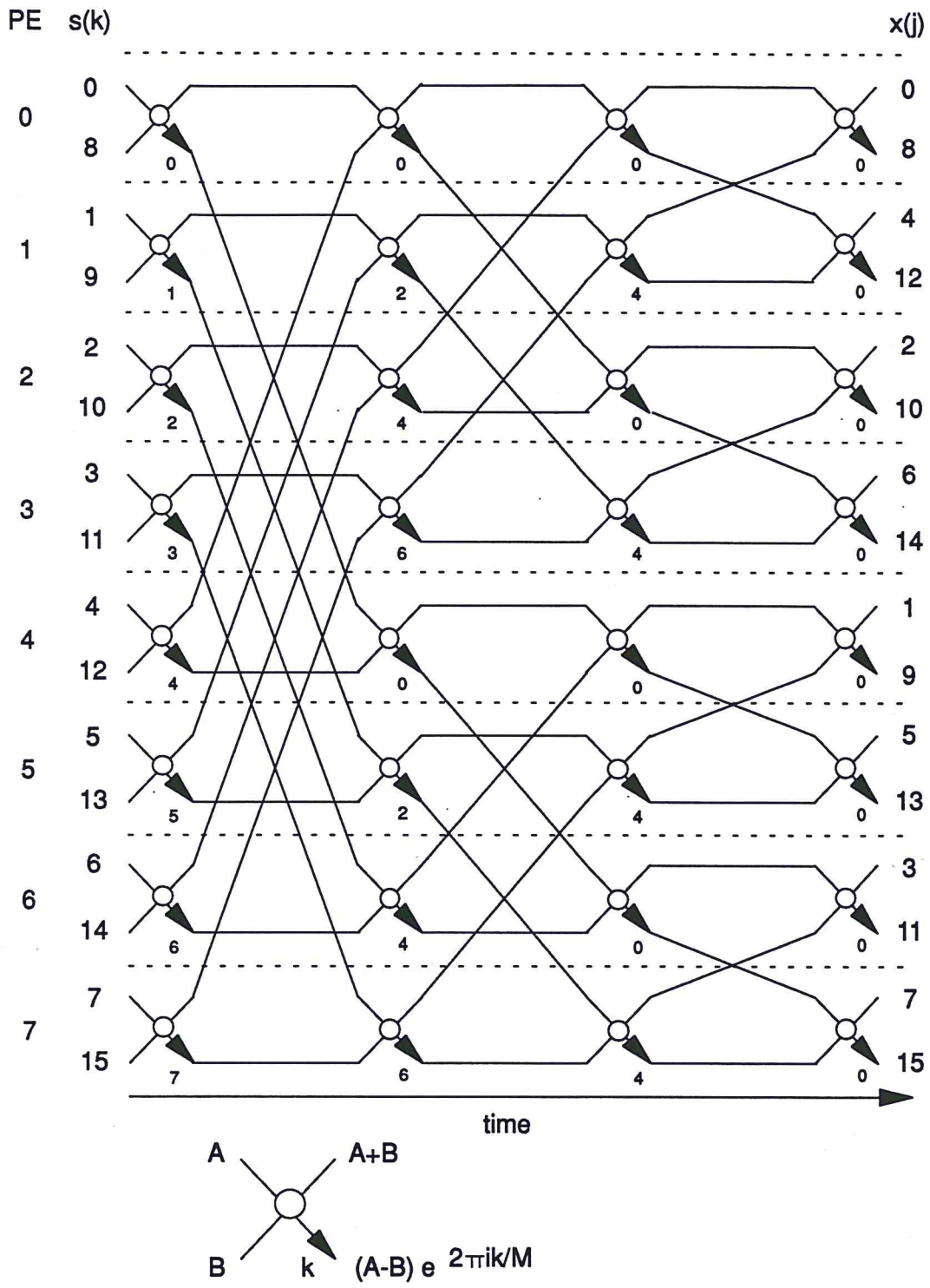


Figure 9: SIMD FFT computation