SOLUTIONS MANUAL TO ACCOMPANY HWANG

ADVANCED COMPUTER ARCHITECTURE

PARALLELISM

SCALABILITY

PROGRAMMABILITY

HWANG-CHENG WANG
University of Southern California

JUNG-GEN WU
National Tiawan Normal University

McGraw-Hill, Inc.
New York St. Louis San Francisco Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan Montreal
New Delhi San Juan Singapore Sydney Tokyo Toronto

Solutions Manual to Accompany Hwang
ADVANCED COMPUTER ARCHITECTURE
Parallelism, Scalability, Programmability

ISBN 0-07-031623-6

234567890 HAM HAM 9098765

Contents

Foreword

Dr. Hwang-Cheng Wang and Dr. Jung-Gen Wu have timely produced this Solutions Manual. I believe it will benefit many instructors using the *Advanced Architecture: Parallelism, Scalability, Programmability* (ISBN 0-07-031622-8) as a required textbook.

Drs. Wang and Wu have provided solutions to all the problems in the text. Some of the solutions are unique and have been carefully worked out. Others contain just a sketch of the underlying principles or computations involved. For such problems, they have provided references which should help instructors find out more information in relevant sources.

The authors have done an excellent job in putting together the solutions. However, as with any scholarly work, there is always room for improvement. Therefore, instructors are encouraged to communicate with us regarding possible refinement to the solutions. Comments or errata can be sent to Kai Hwang at the University of Southern California. They will be incorporated in future printings of this Solutions Manual. Sample test questions and solutions will also be included in the future to make it more comprehensive.

Finally, I want to thank Dr. Wang and Dr. Wu and congratulate them for a difficult job well done within such a short time period.

Kai Hwang

Preface

This Solutions Manual is intended for the exclusive use of instructors only. Reproduction without permission is prohibited by copyright laws.

The solutions in this Manual roughly fall in three categories:

- For problem-solving questions, detailed solutions have been provided. In some cases alternative solutions are also discussed. More complete answers can be found in the text for definition-type questions.
- For research-oriented questions, a summary of the ideas in key papers is presented. Instructors are urged to consult the original and more recent publications in literature.
- For questions that require computer programming, algorithms or basic computation steps are specified where appropriate. Example programs can often be obtained from on-line archives or libraries available at many research sites.

Equations and figures in the solutions are numbered separately from those in the text. When an equation or a figure in the text is referenced, it is clearly indicated. Code segments have been written in assembly and high-level languages. Most codes should be self explanatory. Comments have been added in some places to help understanding the function performed by each instruction.

We have made tremendous effort to improve the correctness of the answers. But a few errors might have been undetected, and some factors might have been overlooked in our analysis. Moreover, several questions are likely to have more than one valid solution; solutions for research-oriented problems are especially sensitive to progress in related areas. In the light of these, we welcome suggestions and corrections from instructors.

Acknowledgments

We have received a great deal of help from our colleagues and experts during the preparation of this Manual. Dr. Chi-Yuan Chin, Myungho Lee, Weihua Mao, Fong Pong, Dr. Viktor Prasanna, and Shisheng Shang have contributed solutions to a number of the problems. Chien-Ming Cheng, Cho-Chin Lin, Myungho Lee, Jih-Cheng Liu, Weihua Mao, Fong Pong, Stanley Wang, and Namhoon Yoo have generously shared their ideas through stimulating discussions. We are indebted to Dr. Bill Nitzberg and Dr. David Black for providing useful information and pointing to additional references. Finally, our foremost thanks go to Professor Kai Hwang for many insightful suggestions and judicious guidance.

H.C. Wang
J.G. Wu

Chapter 1

Parallel Computer Models

Problem 1.1

$$\text{CPI} = \frac{45 \times 1 + 32 \times 2 + 15 \times 2 + 8 \times 2}{45 + 32 + 15 + 8} = \frac{155}{100} = 1.55 \text{ cycles/instruction.}$$

$$\text{MIPS rate} = 10^{-6} \times \frac{40 \times 10^6 \text{cycles/sec}}{1.55 \text{ cycles/instruction}} = 25.8\text{MIPS.}$$

$$\text{Execution time} = \frac{(45000 \times 1 + 32000 \times 2 + 15000 \times 2 + 8000 \times 2)\text{cycles}}{(40 \times 10^6)\text{cycles/s}} = 3.875 \text{ ms.}$$

The execution time can also be obtained by dividing the total number of instructions by the MIPS rate:

$$\frac{(45000 + 32000 + 15000 + 8000)\text{instructions}}{25.8 \times 10^6 \text{ instructions/s}} = 3.875 \text{ ms.}$$

Problem 1.2 Instruction set and compiler technology affect the length of the executable code and the memory access frequency. CPU implementation and control determines the clock rate. Memory hierarchy impacts the effective memory access time. These factors together determine the effective CPI, as explained in Section 1.1.4.

Problem 1.3

(a) The effective CPI of the processor is calculated as

$$\text{CPI} = \frac{15 \times 10^6 \text{ cycles/sec}}{10 \times 10^6 \text{ instructions/sec}} = 1.5 \text{ cycles/instruction.}$$

(b) The effective CPI of the new processor is

$$(1 + 0.3 \times 2 + 0.05 \times 4) = 1.8 \text{ cycles/instruction.}$$

Therefore, the MIPS rate is

$$\frac{30 \times 10^6 \text{ cycles/sec}}{1.8 \text{ cycles/instruction}} = 16.7 \text{ MIPS}.$$

Problem 1.4

(a) Average CPI $= 1 \times 0.6 + 2 \times 0.18 + 4 \times 0.12 + 8 \times 0.1 = 2.24$ cycles / instruction.

(b) MIPS rate $= 40/2.24 = 17.86$ MIPS.

Problem 1.5

(a) False. The fundamental idea of multiprogramming is to overlap the computations of some programs with the I/O operations of other programs.

(b) True. In an SIMD machine, all processors execute the same instruction at the same time. Hence it is easy to implement synchronization in hardware. In an MIMD machine, different processors may execute different instructions at the same time and it is difficult to support synchronization in hardware.

(c) True. Interprocessor communication is facilitated by sharing variables on a multiprocessor and by passing messages among nodes of a multicomputer. The multicomputer approach is usually more difficult to program since the programmer must pay attention to the actual distribution of data among the processors.

(d) False. In general, an MIMD machine executes different instruction streams on different processors.

(e) True. Contention among processors to access the shared memory may create hot spots, making multiprocessors less scalable than multicomputers.

Problem 1.6 The MIPS rates for different machine-program combinations are shown in the following table:

	Machine		
Program	Computer A	Computer B	Computer C
Program 1	100	10	5
Program 2	0.1	1	5
Program 3	0.2	0.1	2
Program 4	1	0.125	1

Various means of these values can be used to compare the relative performance of the computers. Definition of the means for a sequence of positive numbers $a_1, a_2, ..., a_n$ are summarized below. (See also the discussion in Section 3.1.2.)

(a) Arithmetic mean: $AM = (\sum_{i=1}^{n} a_i)/n$.

(b) Geometric mean: $GM = (\prod_{i=1}^{n} a_i)^{1/n}$.

(c) Harmonic mean: $HM = n/\left[\sum_{i=1}^{n}(1/a_i)\right]$.

In general,

$$AM \geq GM \geq HM. \tag{1.1}$$

Based on the definitions, the following table of mean MIPS rates is obtained:

	Computer A	Computer B	Computer C
Arithmetic mean	25.3	2.81	3.25
Geometric mean	1.19	0.59	2.66
Harmonic mean	0.25	0.20	2.1

Note that the arithmetic mean of MIPS rates is proportional to the inverse of the harmonic mean of the execution times. Likewise, the harmonic mean of the MIPS rates is proportional to the inverse of the arithmetic mean of execution times. The two observations are consistent with Eq. 1.1.

If we use the harmonic mean of MIPS rates as the performance criterion (i.e., each program is executed the same number of times on each computer), computer C has the best performance. On the other hand, if the arithmetic mean of MIPS rates is used, which is equivalent to allotting an equal amount of time for the execution of each program on each computer (i.e., fast-running programs are executed more frequently), then computer A is the best choice.

Problem 1.7

- An SIMD computer has a single control unit. The other processors are simple slave processors which accept instructions from the control unit and perform an identical operation at the same time on different data. Each processor in an MIMD computer has its own control unit and execution unit. At any moment, a processor can execute an instruction different from the other processors.
- Multiprocessors have a shared memory structure. The degree of resource sharing is high, and interprocessor communication is carried out via shared variables in the shared memory. In multicomputers, each node typically consists of a processor and local memory. The nodes are connected by communication channels which provide the mechanism for message interchanges among processors. Resource sharing is light among processors.
- In UMA architecture, each memory location in the system is equally accessible to all processors, and the access time is uniform. In NUMA architecture, the access time to a memory location depends on the proximity of a processor to the memory location. Therefore, the access time is nonuniform. In NORMA architecture, each processor has its own private memory; no memory is shared among processors. Each processor is allowed to access its private memory only. In COMA architecture, such as that adopted by KSR-1, each processor has its private cache, which together constitutes the global address space of the system. It is like a NUMA with cache in place of memory. A page of data can be migrated to a processor upon demand or be replicated on more than one processor.

Problem 1.8

(a) The total number of cycles needed on a sequential processor is $(4+4+8+4+2+4) \times 64 = 1664$ cycles.

(b) Each PE executes the same instruction on the corresponding elements of the vectors involved. There is no communication among the processors. Hence the total number of cycles on each PE is $4+4+8+4+2+4 = 26$.

(c) The speedup is 64 with a perfectly parallel execution of the code.

Problem 1.9

Because the processing power of a CRCW-PRAM and an EREW-PRAM is the same, we need only focus on memory accessing. Below, we prove that the time complexity of simulating a concurrent write or a concurrent read on an EREW-PRAM is $O(\log n)$. Before the proof, we assume it is known that an EREW-PRAM can sort n numbers or write a number to n memory locations in $O(\log n)$ time.

(a) We present the proof for simulating concurrent writes below.

1. Create an auxiliary array A of length n. When CRCW processor P_i, for $i = 0, 1, ..., n-1$, desires to write a datum x_i to a location l_i, each corresponding EREW processor P_i writes the ordered pair (l_i, x_i) to location $A[i]$. These writes are exclusive, since each processor writes to a distinct memory location.
2. Sort the array by the first coordinate of the ordered pairs in $O(\log n)$ time, which causes all data written to the same location to be brought together in the output.
3. Each EREW processor P_i, for $i = 1, 2, ..., n-1$, now inspects $A[i] = (l_j, x_j)$ and $A[i-1] = (l_k, x_k)$, where j and k are values in the range $0 \le j, k \le n-1$. If $l_j \neq l_k$ or $i = 0$, then processor P_i, for $i = 0, 1, ..., n-1$, writes the datum x_j to location l_j in global memory. Otherwise, the processor does nothing. Since the array A is sorted by first coordinate, only one of the processors writing to any given location actually succeeds, and thus the write is exclusive.

This process thus implements each step of concurrent writing in the common-CRCW model in $O(\log n)$ time.

(b) We present the proof for simulating concurrent reads as follows:

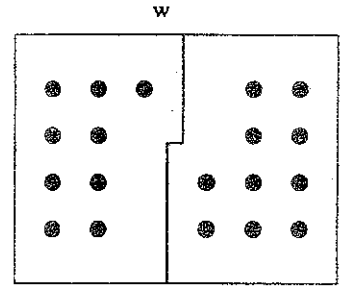
1. Create an auxiliary array A of length n. When CRCW processor P_i, for $i = 0, 1, ..., n-1$, desires to read a datum from a location l_i, each corresponding EREW processor P_i writes the ordered three-tuple (i, l_i, x_i) to location $A[i]$, in which the x_i is an arbitrary number. These writes are exclusive, since each processor writes to a distinct memory location.
2. Sort the array A by the second coordinate of the three-tuple in $O(\log n)$ time, which causes all data read from the same location to be brought together in the output.

3. Each EREW processor P_i, for $i = 1, 2, ..., n-1$, now inspects $A[i] = (j, l_j, x_j)$ and $A[i-1] = (k, l_k, x_k)$, where j and k are values in the range $0 \leq j, k \leq n-1$. If $l_j \neq l_k$ or $i = 0$, then processor P_i, for $i = 0, 1, ..., n-1$, reads the datum from location l_j in global memory. Otherwise, the processor does nothing. Since the array A is sorted by the second coordinate, only one of the processors reading from any given location actually succeeds, and thus the read is exclusive.
4. Each EREW processor P_i that read a datum stores the datum to the third coordinate in $A[i]$, and then broadcasts it to the third coordinate of $A[j]$'s for $j = i+1, i+2, ...,$ and $l_j = l_i$. This takes $O(\log n)$ time.
5. Sort the array A by the first coordinate of the ordered three-tuple in $O(\log n)$ time.
6. Each EREW processor P_i reads data in the third coordinate from $A[i]$. These reads are exclusive, since each processor reads from a distinct memory location.

This process thus implements each step of concurrent reading in the common-CRCW model in $O(\log n)$ time.

Problem 1.10 For multiplying two n-bit binary integers, there are $2n$ bits of input, and $2n$ bits of output.

Suppose the circuit in question, in the grid model, is a rectangle of height h and width w as shown in the following diagram:



Assume without loss of generality that $h \leq w$. and there is at most one word along each grid line. It is possible to divide the circuit by a line as shown in the figure. This line runs between the grid lines and runs vertically, except possibly for a single jog of one grid unit. Most importantly, we can select the line so that at least $2n/3$ of the output bits (i.e., $1/3$ of the output bits) are emitted on each side. We select the line by sliding it from left to right, until the first point at which at least $2n/3$ of the output bits are output to the left of the line.

If no more than $4n/3$ of these bits are output to the left, we are done. If not, start from the top, considering places to jog the line back one unit to the left. We know that if the line jog at the every top, fewer than $4n/3$ of the bits are emitted to the left, and if the line jogs at the very bottom, more than $2n/3$ are. Thus, as no single grid point

can be the place where as many as $n/3$ of the bits are emitted, we can find a suitable place in the middle to jog the line. There, we shall have between $2n/3$ and $4n/3$ of the output bits on each side.

Now assume without loss of generality that at least half of the input bits are read on the left of the line, and let us, by renumbering bits, if necessary, assume that these are x_k, x_{2k}, ..., x_{nk}. Suppose also that output bits y_{i_1}, y_{i_2}, ..., $y_{i_{2n/3}}$ are output on the right. We can pick values so that $y_{i_1} = x_k$, $y_{i_2} = x_{2k}$, and so on. Thus information regarding the $2n/3$ input bits, x_k, x_{2k}, ..., $x_{2kn/3}$, must cross the line.

We may assume at most one wire or circuit element along any grid line, so the number of bits crossing the line in one time unit is at most $h+1$ (h horizontal and one vertical, at the jog). It follows that $(h+1)T \geq 2n/3$, or else the required $2n/3$ bits cannot cross the line in time. Since we assume $w \geq h$, we have both $hT = \Omega(n)$ and $wT = \Omega(n)$. Since $wh = A$, we have $AT^2 = \Omega(n^2)$ by taking the product. That is, $AT^2 \geq kn^2$.

Problem 1.11

(a) Since the processing elements of an SIMD machine read and write data from different memory modules synchronously, no access conflicts should arise. Thus any PRAM variant can be used to model SIMD machines.

(b) The processors in an MIMD machine can read the same memory location simultaneously. However, writing to a same memory location is prohibited. Thus the CREW-PRAM can best model an MIMD machine.

Problem 1.12

(a) The memory organization changed from UMA model (global shared memory) to NUMA model (distributed shared memory).

(b) The medium-grain multicomputers use hypercube as their interconnection networks, while the fine-grain multicomputers use lower dimensional k-ary n-cube (e.g., 2-D or 3-D torus) as their interconnection networks.

(c) In the register-to-register architecture, vector registers are used to hold vector operands, intermediate and final vector results. In the memory-to-memory architecture, vector operands and results are retrieved directly from the main memory by using a vector stream unit.

(d) In a single threaded architecture, each processor maintains a single thread of control with limited hardware resources. In a multithreaded architecture, each processor can execute multiple contexts by switching among threads.

Problem 1.13

Assume the input is $A(i)$ for $0 \leq i \leq n-1$, and n is a power of 2.

```
/* Program in PE(i) */
```

$l \leftarrow n$
Repeat
  $l \leftarrow l/2$
  If $(i < l)$ **then begin**
    Read $A(i)$ from shared memory
    Read $A(i+l)$ from shared memory
    Compute $Max(A(i), A(i+l))$
    Store into $A(i)$
    end
Until $(l = 1)$

At the end of the computation, $A(0)$ has the result.

Problem 1.14 Assume a CRCW PRAM model in which the sum of the values in multiple processors is written to a memory location. The array elements of each matrix and the processors are numbered 0 through $n-1$ in each dimension.

(a) The matrix multiplication program with the use of n PEs is specified below:

for $i = 0, n-1$ **do**
  for $j = 0, n-1$ **do**
    forall PE$(k), k = 0, n-1$ **do**
      Read $A(i,k)$
      Read $B(k,j)$
      Compute $C(i,j) = A(i,k) \times B(k,j)$
      Store $C(i,j)$

The concurrent stores by all n processors will result in the sum of the product terms being written to the memory location for $C(i,j)$. Each of the step in the innermost loop is done in $O(1)$ time. Therefore, the time required is $O(n^2)$.

(b) The matrix multiplication program with the use of n^2 PEs is specified below:

forall PE$(k,j), 0 \leq j,k \leq n-1$ **do**
  Read $B(k,j)$
for $i = 0, n-1$ **do**
  forall PE$(k,j), 0 \leq j,k \leq n-1$ **do**
    Read $A(i,k)$
    Compute $C(i,j) = A(i,k) \times B(k,j)$
    Store $C(i,j)$

Note that in each i-loop, $A(i,k)$ is read concurrently by n processors $PE(k,j), 0 \leq j \leq n-1$. In $O(1)$ time, one row of matrix C is computed, and the total time complexity of the algorithm is $O(n)$.

Problem 1.15

(a) Tera
(b) CM-2
(c) KSR-1
(d) EM-5
(e) VPP500
(f) Paragon
(g) Dash
(h) RP3

Chapter 2

Program and Network Properties

Problem 2.1

(a) The amount of computation involved in a software process.

(b) The time measure of the communication overhead incurred between machine subsystems.

(c) A statement S2 is flow-dependent on statement S1 if an execution path exists from S1 to S2 and at least one output of S1 feeds in as input to S2.

(d) Statement S2 is antidependent on statement S1 if S2 follows S1 in program order and the output of S2 overlaps the input to S1.

(e) Two statements are output-dependent if they produce the same variable.

(f) I/O dependence occurs when the same file or I/O device is requested by two statements.

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

(h) Resource dependence is concerning the conflicts in using shared resources, such as integer units, floating-point units, registers, and memory areas, among parallel events.

(i) Bernstein's condition states that two processes can execute in parallel if their input sets I_1, I_2 and output sets O_1, O_2 satisfy the following conditions:

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$
$$O_1 \cap O_2 = \phi$$

(j) For each time period, the number of processors used to execute a parallel program is defined as the degree of parallelism.

Problem 2.2

(a) The number of edges incident on a node.

(b) The maximum number of distinct links between any two nodes.

(c) When a given network is cut into two equal halves, the minimum number of edges along the cut is called the bisection width.

(d) Static networks are formed of point-to-point direct connections which will not change during program execution.

(e) Dynamic networks are implemented with switched channels, which are dynamically configured to match the communication demand in user programs.

(f) A network which can handle all possible connections without blocking is called a nonblocking network.

(g) Multicast corresponds to a mapping from a subset to another (many to many). Broadcast is a one-to-all mapping.

(h) Difference between mesh and torus is the connections at the boundary nodes (Fig. 2.17). It leads to different network diameters.

(i) A network is symmetric if the topology is the same looking from any node in the network.

(j) A multistage network consists of multiple stages of interconnected switches. A number of switches are used in each stage. Interstage connection patterns are specified between the switches in adjacent stages. The switches can be dynamically set to establish connections between different input and output pairs.

(k) A crossbar network is a single-stage network. It provides a dynamic connection between any source and any destination without blocking. An $n \times m$ crossbar network requires $n \times m$ crosspoint switches.

(l) A digital bus is a collection of wires and connectors for data transactions among processors, memory modules, and peripheral devices attached to the bus. The bus is used for only one transaction at a time between a source and a destination. In case of multiple requests, bus arbitration logic must be used to allocate or deallocate the bus.

Problem 2.3

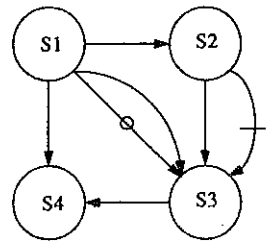
(a) A control-flow computer uses a program counter to sequence the execution of instructions in a program. In a dataflow computer, the execution of an instruction is driven by data availability. In a reduction computer, the computation is triggered by the demand for an operation's result.

(b) See Table 2.1 in the text.

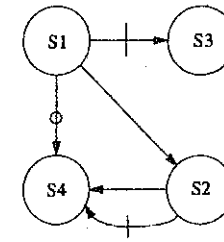
(c) In string reduction, each demander gets a separate copy of the expression for its own evaluation. In graph reduction, each demander is given a pointer to the result of the reduction.

Problem 2.4 The dependence graphs are shown in the following diagrams:

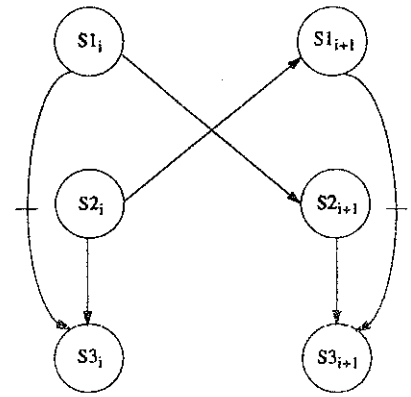
(a)



(b)

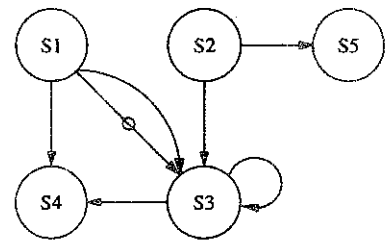


(c) In the following dependence graph, the subscript is used to indicate the iteration:



Problem 2.5

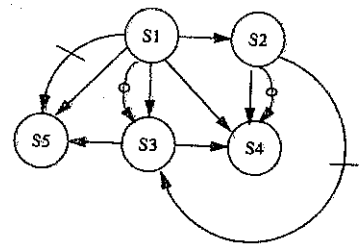
(a) Dependence graph:



(b) There are storage dependences between instruction pairs (S2, S5) and (S4, S5). There is a resource dependence between S1 and S2 on the load unit, and another

between S4 and S5 on the store unit.

(c) There is an ALU dependence between S3 and S4, and a storage dependence between S1 and S5.



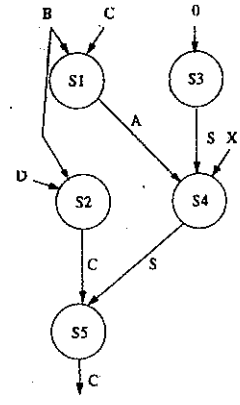
Problem 2.6 The input and output sets for the instructions are enumerated below:

$$\begin{aligned} I_1 &= \{B, C\}, & O_1 &= \{A\}, \\ I_2 &= \{B, D\}, & O_2 &= \{C\}, \\ I_3 &= \phi, & O_3 &= \{S\}, \\ I_4 &= \{S, A, X(I)\}, & O_4 &= \{S\}, \\ I_5 &= \{S, C\}, & O_5 &= \{C\}. \end{aligned}$$

Using Bernstein's conditions, we find that

- S1 and S3 can be executed concurrently, because $I_1 \cap O_3 = \phi$, $I_3 \cap O_1 = \phi$, and $O_1 \cap O_3 = \phi$.
- S2 and S3 can be executed concurrently, because $I_2 \cap O_3 = \phi$, $I_3 \cap O_2 = \phi$, and $O_2 \cap O_3 = \phi$.
- S2 and S4 can be executed concurrently, because $I_2 \cap O_4 = \phi$, $I_4 \cap O_2 = \phi$, and $O_2 \cap O_4 = \phi$.
- S1 and S5 cannot be executed concurrently, because $I_1 \cap O_5 = \{C\}$.
- S1 and S2 cannot be executed concurrently, because $I_1 \cap O_2 = \{C\}$.
- S1 and S4 cannot be executed concurrently, because $I_4 \cap O_1 = \{A\}$.
- S2 and S5 cannot be executed concurrently, because $I_5 \cap O_2 = O_5 \cap O_2 = \{C\}$.
- S3 and S4 cannot be executed concurrently, because $I_4 \cap O_3 = O_4 \cap O_3 = \{S\}$.
- S3 and S5 cannot be executed concurrently, because $I_5 \cap O_3 = \{S\}$.
- S4 and S5 cannot be executed concurrently, because $I_5 \cap I_4 = I_5 \cap O_4 = \{S\}$.

The program can be reconstructed as shown in the flow graph below:

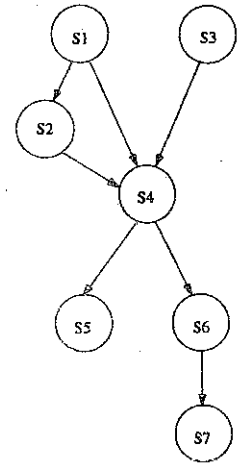


Problem 2.7 The input and output sets for the instructions are as follows:

$$
\begin{aligned}
I_1 &= \{B, C\}, \quad O_1 = \{A\}, \\
I_2 &= \{D, E\}, \quad O_2 = \{C\}, \\
I_3 &= \{G, E\}, \quad O_3 = \{F\}, \\
I_4 &= \{A, F\}, \quad O_4 = \{C\}, \\
I_5 &= \{G, C\}, \quad O_5 = \{M\}, \\
I_6 &= \{L, C\}, \quad O_6 = \{A\}, \\
I_7 &= \{E, A\}, \quad O_7 = \{A\}.
\end{aligned}
$$

Similar to Problem 2.6, we can use Bernstein's conditions to determine statements that can be executed in parallel. The result is shown below: S1 || S3, S1 || S5, S2 || S3, S2 || S7, S3 || S5, S3 || S6, S3 || S7, S5 || S6, and S5 || S7.

However, Bernstein's conditions are not sufficient for this problem; the precedence relations as shown in the following diagram have to be taken into account:



It is clear that S1, S2, and S3 must be executed before S4. Moreover, S5, S6, and S7 must be executed after S4. This consideration prohibits parallel execution among the two groups of statements. Thus, the statements that can be executed in parallel are: S1 || S3, S2 || S3, S5 || S6, and S5 || S7.

The parallel code is shown below:

Cobegin
  S1, S3
Coend
S2
S4
Cobegin
  S5, S6
Coend
S7

Problem 2.8

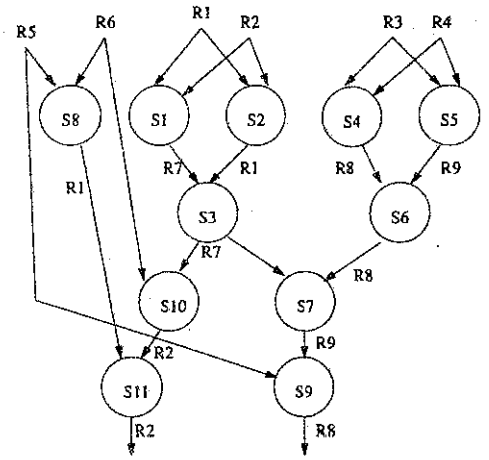
(a) Assume the variables A, B, C, D, E, F, X, Y, and Z are stored in registers R1, R2, R3, R4, R5, R6, R7, R8, and R9, respectively. To minimize the number of registers used, we assign intermediate results $A + B$ to R7, $A - B$ to R1, $C + D$ to R8, $C - D$ to R9, and $X - F$ to R2. Suppose an assembly language instruction has the format:

opcode source1, source2, destination

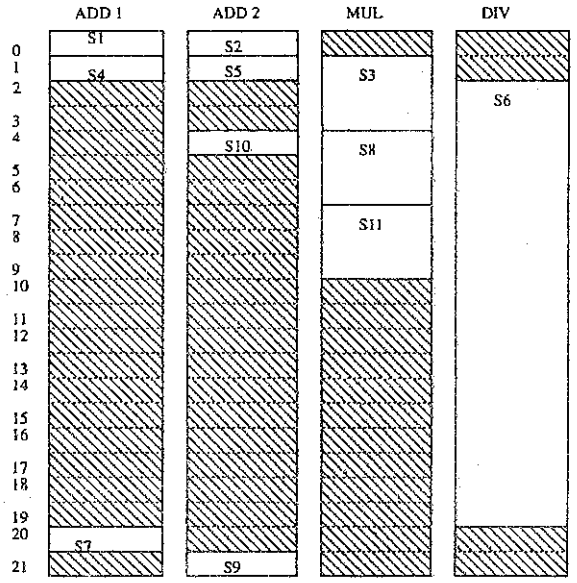
The corresponding assembly program is shown below:

S1:	add	R1,R2,R7
S2:	subtract	R1,R2,R1
S3:	multiply	R7,R1,R7
S4:	add	R3,R4,R8
S5:	subtract	R3,R4,R9
S6:	divide	R8,R9,R8
S7:	add	R7,R8,R9
S8:	multiply	R5,R6,R1
S9:	subtract	R5,R9,R8
S10:	subtract	R7,R6,R2
S11:	multiply	R2,R1,R2

(b) Dependence graph corresponding to the assembly code:

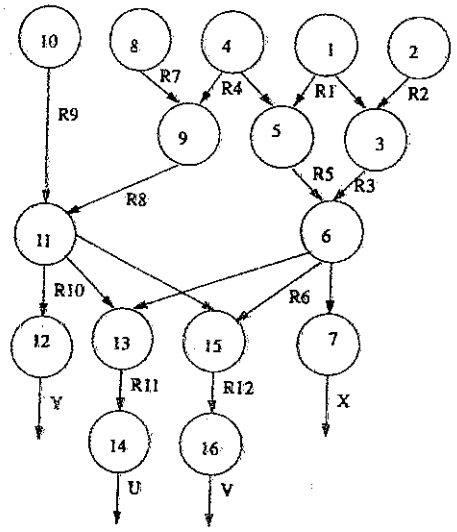


(c) An optimal schedule is shown in the following diagram. Using this schedule, the program can be executed in 22 cycles.

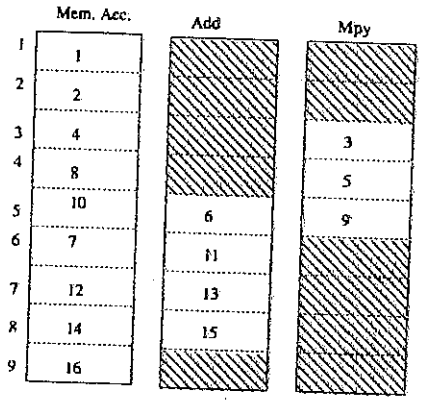


Problem 2.9 The program graph and optimal schedule are shown below:

(a) Program graph:



(b) Minimum-time execution:



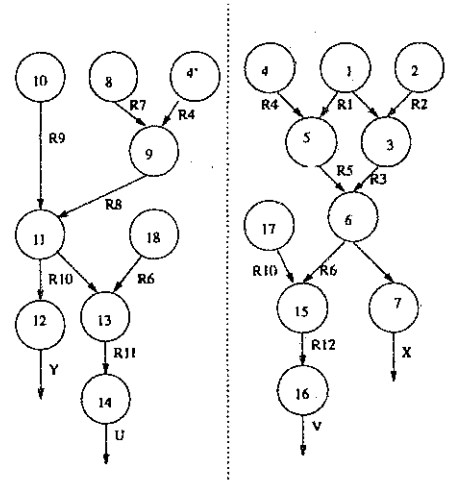
Problem 2.10 The partition of the program and the scheduling for execution on two processors are discussed in the follows.

(a) The program is divided into two parts for execution by the two processors. Sub-programs executed by processors 1 and 2 are specified in the following diagram:

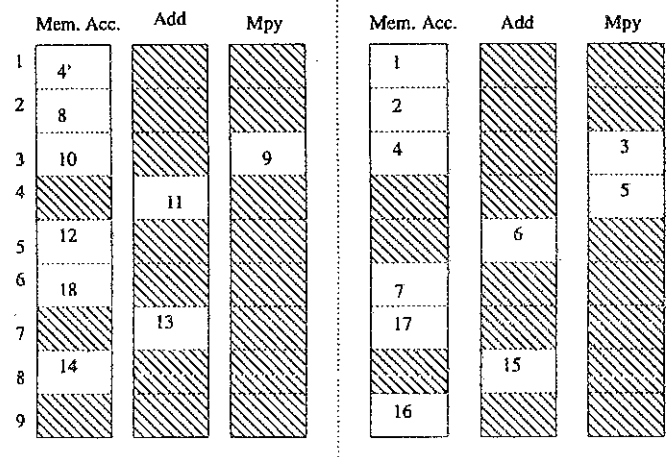
	Processor 1	
4':	Load	R4, D
8:	Load	R7, C
9:	Mul	R8, R7, R4
10:	Load	R9, E
11:	Add	R10,R8, R9
12:	Store	Y, R10
18:	Load	R6, X
13:	Add	R11, R6, R6, R10
14:	Store	U, R11

	Processor 2	
1:	Load	R1,A
2:	Load	R2,B
3:	Mul	R3, R1, R2
4:	Load	R4,D
5:	Mul	R5, R1, R4
6:	Add	R6, R3, R5
17:	Load	R10,Y
7:	Store	X,R6
15:	Sub	R12,R6,R10
16:	Store	V, R12

The program flow graph is shown below:



(b) An optimal schedule for the parallel execution of the program is as follows:



Problem 2.11

(a) To design a direct network for a 64-node multicomputer, we can use

- A 3D torus with 4 nodes along each dimension. The relevant parameters are: $d = 3\lfloor r/2 \rfloor = 6$, $D = 3\lfloor r/2 \rfloor = 6$, and $l = 3N = 192$. Also, $d \times D \times l = 6912$.
- A 6-dimensional hypercube. The relevant parameters are: $d = n = 6$, $D = n = 6$, and $l = n \times N/2 = 6 \times 64/2 = 192$. We have $d \times D \times l = 6912$.
- A CCC with dimension $k = 4$. The relevant parameters are: $d = 3$, $D = 2k - 1 + \lfloor k/2 \rfloor = 2 \times 4 - 1 + \lfloor 4/2 \rfloor = 9$, and $l = 3N/2 = 96$. The value of $d \times D \times l$ is 2592.

If the quality of a network is measured by $(d \times D \times l)^{-1}$, then a CCC is better than a 3-D torus or a 6-cube. A 3-D torus and a 6-cube have the same quality.

(b)

- The torus and hypercube have similar network properties and are treated together. We have $\sum_{i=1}^{D} = \frac{(1+6)\times 6}{2} = 21$. Denote by $P(i)$ the probability of information interchange between nodes at a distance i. Then we have

$$P(1) = \frac{6}{21}, P(2) = \frac{5}{21}, P(3) = \frac{4}{21}, P(4) = \frac{3}{21}, P(5) = \frac{2}{21}, P(6) = \frac{1}{21}.$$

Therefore, the mean internode distance is

$$1 \times \frac{6}{21} + 2 \times \frac{6}{21} + 3 \times \frac{4}{21} + 4 \times \frac{3}{21} + 5 \times \frac{2}{21} + 6 \times \frac{1}{21} = \frac{56}{21} = \frac{8}{3}.$$

- For the CCC, we have $\sum_{i=1}^{D} = \frac{(1+9)\times 9}{2} = 45$. The probabilities of internode communication for distance i are

$$P(1) = \frac{9}{45}, P(2) = \frac{8}{45}, P(3) = \frac{7}{45}, P(4) = \frac{6}{45}, P(5) = \frac{5}{45}, P(6) = \frac{4}{45},$$

$$P(7) = \frac{3}{45}, P(8) = \frac{2}{45}, P(9) = \frac{1}{45}.$$

Hence, the mean internode distance is

$$1\times\frac{9}{45}+2\times\frac{8}{45}+3\times\frac{7}{45}+4\times\frac{6}{45}+5\times\frac{5}{45}+6\times\frac{4}{45}+7\times\frac{3}{45}+8\times\frac{2}{45}+9\times\frac{1}{45} = \frac{165}{45}.$$

In conclusion, the mean internode distance of 4-CCC is greater than that of 6-cube and 3-D torus. 6-cube and 3-D torus have identical mean internode distance. The similarity of the 6-cube and 3-D torus in the above is more than incidental. In fact, it has been shown [Wang89] that when $k = 4$ (as is the case for this problem), a k-ary n-cube is exactly a $2n$-dimensional binary hypercube.

Problem 2.12

(a) It should be noted that we are looking for nodes that can be reached from N_0 in exactly 3 steps. Therefore, nodes that can be reached in 1 or 2 steps have to be excluded.

- For an 8×8 Illiac mesh, they can be calculated by the equation $(a+b+c)$ mod 64, where a, b, and c can be $+1, -1, +8$, or -8. There are 20 combinations (4 if a, b, and c are all different; 12 if two of them are equal; 4 if $a = b = c$). However, 8 of the combinations contain the pair $+1$ and -1 or the pair $+8$ and -8, making them reachable in one step. Such nodes have to be eliminated from the list. Hence, 12 nodes can be reached from N_0 in three steps. The addresses of these nodes are 3, 6, 10, 15, 17, 24, 40, 47, 49, 54, 58, and 61.

- For a binary 6-cube, the binary address $a_5 ... a_1 a_0$ of a node reachable in three steps from N_0 has exactly three 1's. There are 20 possible combinations (C(6,3)). The addresses of these nodes are 7, 11, 13, 14, 19, 21, 22, 25, 26, 28, 35, 37, 38, 41, 42, 44, 49, 50, 52, and 56.
- The nodes reachable in exactly three steps can be determined as follows. List all 6-bit numbers which contain three 1s. There are 20 such numbers. First take 1's complement of each number and then add 1 to each of the resulting numbers. (Equivalently, the new numbers are obtained by subtracting each of the original numbers from 64.) If a new number has three or four 1s in its binary representation and the 1s are separated by at least one 0, then both nodes whose addresses are the original number and the new number can be reached in exactly three steps. (The last point of the rule is due to the fact that clustered 1s can always be replaced by two 1s.) The addresses of these nodes are 11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 43, 45, 51, and 53.

(b) The upper bound on the minimum number of routing steps needed to send data from any node to another for an 8×8 Illiac mesh is 7 $(= \sqrt{64} - 1)$, for a 6-cube is 6, and for a 64-node barrel shifter is 3 $(= \log_2 64/2)$.

(c) The upper bound on the minimum number of routing steps needed to send data from any node to another is 31 for a 32×32 Illiac mesh, 10 for a 10-cube, and 5 for a 1024-node barrel shifter.

Problem 2.13 Part of Table 2.4 in the text is duplicated below:

Network Characteristics	Bus System	Multistage Network	Crossbar Switch
Minimum latency for unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processor	$O(w/n)$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring Complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching Complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one to one at a time.	Some permutations and broadcast, if network unblocked	All permutations, one at a time.
Remarks	Assume n processors on the bus; bus width is w bits.	$n \times n$ MIN using $k \times k$ switches with line width of w bits.	Assume $n \times n$ crossbar with line width of w bits.

Problem 2.14

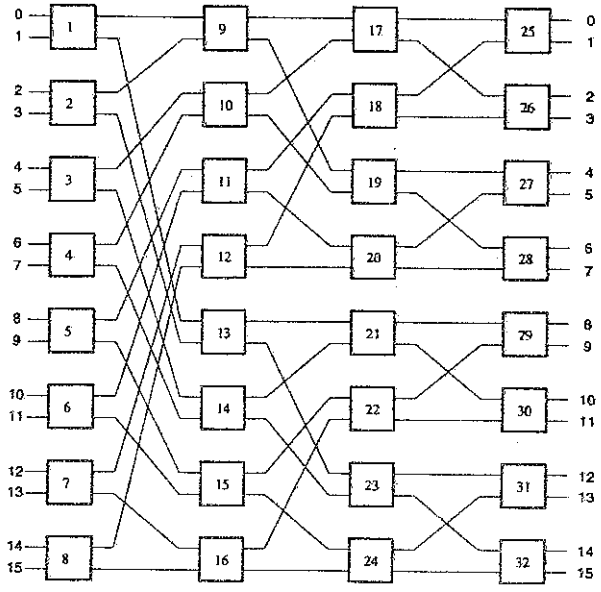
(a) For each output terminal, there are 4 possible connections (one from each of the input terminals), so that there are $4 \times 4 \times 4 \times 4 = 256$ legitimate states.

(b) 48 $(= 16 \times 3)$ 4×4 switch modules are needed to construct a 64-input Omega network. There are 24 $(= 4 \times 3 \times 2 \times 1)$ permutation connections in a 4×4 switch module. Therefore a total of (24^{48}) permutations can be implemented in a single pass through the network without blocking.

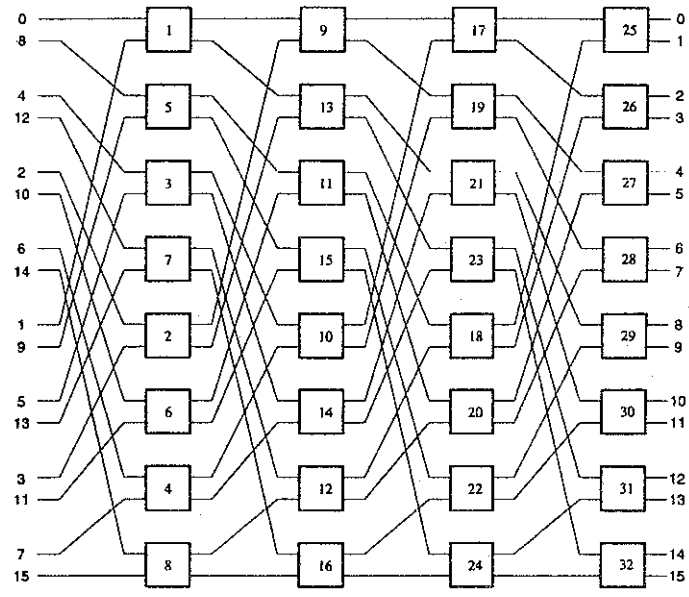
(c) The total number of permutations of 64 inputs is 64!. So the fraction is $24^{48}/64! \approx 1.4 \times 10^{-23}$.

Problem 2.15

(a) We label the switch modules of a 16×16 Baseline network as below.

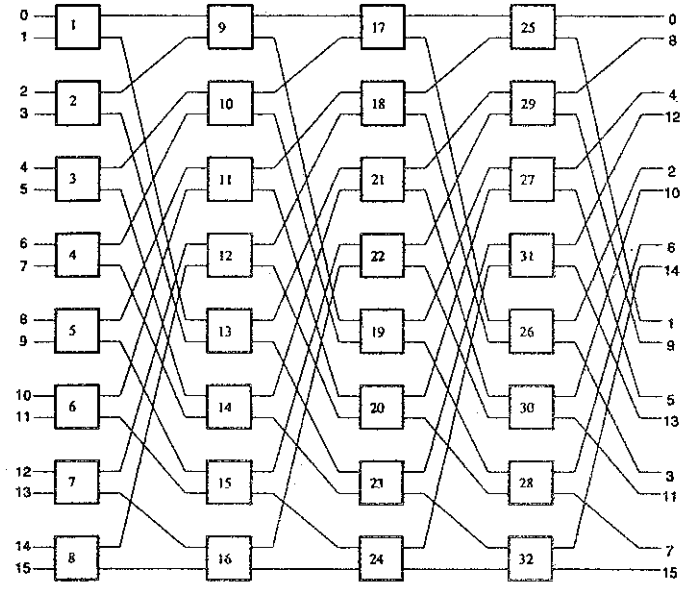


Then, we change the positions of some switch modules, the Baseline network becomes:



which is just an Omega network.

(b) If we change the positions of some switch modules in the Baseline network, it becomes:



which is just the Flip network.

(c) Since both the Omega network and the Flip network are topologically equivalent to the baseline network, they are topologically equivalent to each other.

Problem 2.16

(a) k^n.

(b) $n\lfloor k/2 \rfloor$.

(c) $2k^{n-1}$.

(d) $2n$.

(e)
- A k-ary 1-cube is a ring with k nodes.
- A k-ary 2-cube is a 2-D $k \times k$ torus.
- A mesh is a torus without end-around connections.
- A 2-ary n-cube is a binary n-cube.
- An Omega network is the multistage network implementation of shuffle-exchange network. Its switch modules can be repositioned to have the same interconnection topology as a binary n-cube.

(f) The conventional torus has long end-around connections, but the folded torus has equal-length connections. (See Figure 2.21 in the text).

(g)
- The relation

$$B = 2wN/k$$

will be shown in the solution of Problem 2.18. Therefore, if both the number of nodes N and wire bisection width B are constants, the channel width W will be proportional to k:

$$w = B/b = Bk/(2N).$$

The latency of a wormhole-routed network is

$$T_{WH} = \frac{L}{w} + \frac{F}{w}D,$$

which is inversely proportional to w, hence also inversely proportional to k. This means a network with a higher k will have lower latency. For two k-ary n-cube networks with the same number of nodes, the one with a lower dimension has a larger k, and hence a lower latency.

- It will be shown in the solution of Problem 2.18 that the hot-spot throughput is equal to the bandwidth of a single channel:

$$\Theta_{HS} = k/2.$$

Low-dimensional networks have a larger k, hence a higher hot-spot throughput.

Problem 2.17

(a) In a tree network, a message going from processor i to processor j goes up the tree to their least common ancestor and then back down according to the least significant bits of j. Message traffic through lower-level (closer to the root) nodes is heavier than that of higher-level nodes. The lower-level channels in a fat tree has a greater number of wires, and hence a higher bandwidth. This will prevent congestion in the lower-level channels.

(b) The capacity of a universal fat tree at level k is

$$c_k = \min(\lceil n/2^k \rceil, \lceil w/2^{2k/3} \rceil).$$

- If $k > 3\log(n/w)$, $\lceil n/2^k \rceil < \lceil w/2^{2k/3} \rceil$. Therefore, $c_k = \lceil n/2^k \rceil = (n+1)/2^k$, which is 1, 2, 4, ..., for $k = \log(n+1), \log(n+1) - 1, \log(n+1) - 2, ...$
- If $k \leq 3\log(n/w)$, then $\lceil n/2^k \rceil \geq \lceil w/2^{2k/3} \rceil$. Hence $c_k = \lceil w/2^{2k/3} \rceil$, which is ..., $w/8^{2/3}, w/4^{2/3}, w/2^{2/3}$, for $k = ..., 3, 2, 1$.
- Initially, the capacities double from one level to the next toward the root, but at levels less than $3\log(n/w)$ away from the root, the channel capacities grow at the rate of $\sqrt[3]{4}$.

Problem 2.18

(a) A k-ary n-cube network has N nodes where $N = k^n$. Assume k is even. If the network is partitioned along one dimension into two parts of equal size, the "cross section" separating the two parts is of size N/k. Corresponding to each node in the cross section, there are two wires, one being the nearest-neighbor link and the other wraparound link in the original network. Therefore, the cross section contains $b = 2N/k$ wires each w bits wide, giving a wire bisection width $B = bw = 2wN/k$. The argument also holds for k odd, although the partitioning is slightly more complex.

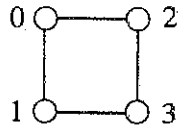
(b) The hot-spot throughput of a network is the maximum rate at which message can be sent from one specific node P_i to another specific node P_j. For a k-ary n-cube with deterministic routing, the hot-spot throughput, Θ_{HS}, is equal to the bandwidth of a single channel w. From (a), $w = kB/(2N)$. Therefore,

$$\Theta_{HS} = kB/(2N),$$

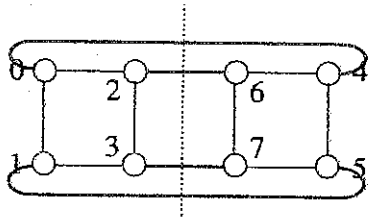
which is proportional to k for a fixed B.

Problem 2.19

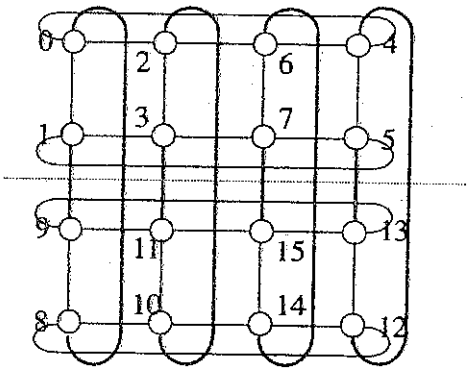
(a) Embedding of an $r \times r$ torus in a hypercube is shown in the following diagrams for $r = 2$ and 4, respectively ((a) and (c)). As can be seen, if the nodes of a torus are numbered properly, we obtain inter-node connections identical to those of a hypercube (nodes whose numbers differ by a power of 2 are linked directly).



(a)



(b)

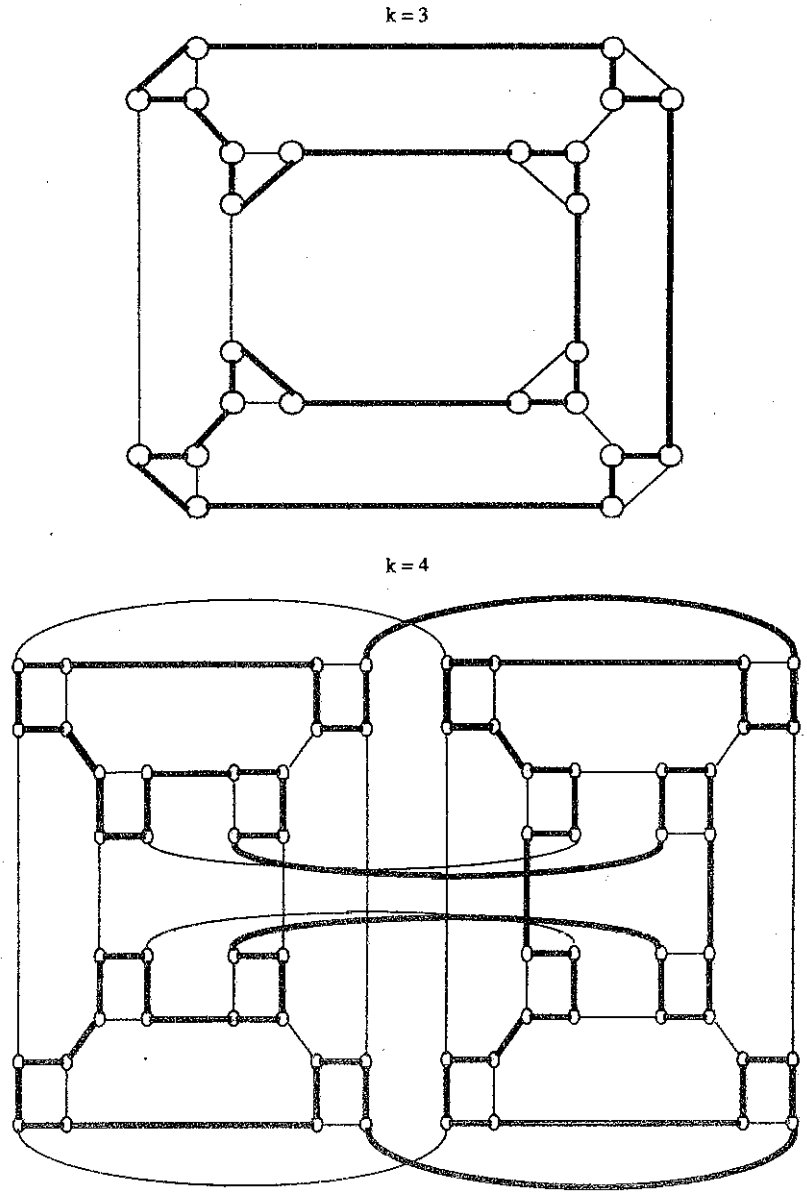


(c)

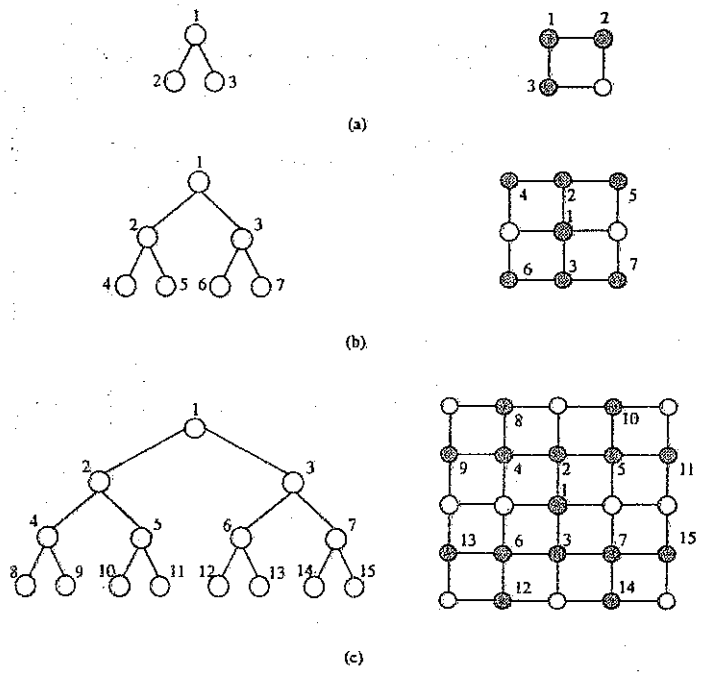
A $2r \times 2r$ torus can be constructed from $r \times r$ tori in two steps. In step one, a $2r \times r$ torus is built by combining an $r \times r$ with its "mirror" image (in the sense of node numbering) and connecting the corresponding nodes, as shown in diagram (b). In step 2, the $2r \times r$ torus is combined with its mirror image to form a $2r \times 2r$ torus. In this manner, a torus can be fully embedded in a hypercube of dimension d with $2^d = r^2$ nodes.

In general, it has been shown that any $m_1 \times m_2 \cdots \times m_t$ torus, where $m_i = 2^{p_i}$ can be embedded in a hypercube of dimension $d = p_1 + p_2 + \cdots + p_t$ with the proximity property preserved using binary reflected gray code for the mapping [Chan86].

(b) Embedding of a ring on a CCC is equivalent to finding a Hamiltonian cycle on the CCC. In the following figure, the embedding of rings on CCCs for $k = 3$ and 4, respectively, is shown. It is easy to first consider the embedding of a ring on a binary hypercube by treating the cycle at each vertex of the hypercube as a supernode. This step can be carried out easily and there are several possible ways to embed a ring on a hypercube. Then, whenever a supernode on the embedded ring is visited, all the nodes in the corresponding cycle are linked.



(c) Embedding of a complete balanced tree in a mesh is shown in the following diagram for trees of different heights. In general, the root of a tree is mapped to the center node of a mesh, and leaf nodes are mapped to outlying mesh nodes. The process is recursive. Suppose a tree of height $l \geq 3$ has been embedded in an $r \times r$ mesh. When embedding a tree of height $l + 1$, an $(r + 2) \times (r + 2)$ mesh is needed, with the new leaf nodes mapped to the boundary nodes of the augmented mesh. This is illustrated for $l = 3$.



Problem 2.20

(a) A hypernet combines the hierarchical structure of a tree in the overall architecture and the uniform connectivity of a hypercube in its building blocks.

(b) By construction, a hypernet built with identical modules (buslets, treelets, cubelets, etc.) has a constant node degree. This is achieved by a systematic use of the external links of each cubelet when building larger and larger systems.

(c) The hypernet architecture was proposed to take advantage of localized communication patterns present in some applications such as connectionist neural networks. The connection structure of hypernets gives effective support for communications between adjacent lower-level clusters. Global communication is also supported, but the bandwidth provided is lower. Algorithms with commensurate nonuniform communication requirements among different components are suitable candidates for implementation on hypernets.

I/O capability of a hypernet is furnished by the external links of each building block. As a result, I/O devices can be spread throughout the hierarchy to meet I/O demand. Fault tolerance is built into the hypernet architecture to allow graceful degradation. Execution of a program can be switched to a subnet in case of node or link failures. The modular construction also facilitates isolation and subsequent replacement of faulty nodes or subnets.

Chapter 3

Principles of Scalable Performance

Problem 3.1

(a) CPI $= 1 \times 0.6 + 2 \times 0.18 + 4 \times 0.12 + 12 \times 0.1 = 2.64$ cycles / instruction.

(b) MIPS rate $= \frac{4 \times 40 \times 10^6 \text{cycles/s}}{2.64 \text{cycles / instruction}} = 60.60\text{MIPS}.$

(c) When a single processor is used, the execution time is $t_1 = 200000/17.86 = 1.12 \times 10^4$ μs. When four processors are used, the time is reduced to $t_4 = 220000/60.60 = 3.63 \times 10^3$ μs. Hence the speedup is $11.2/3.63 = 3.08$ and the efficiency is $3.08/4 = 0.77$.

Problem 3.2

(a) If the vector mode is not used at all, the execution time will be

$$0.75T + 9 \times 0.25T = 3T.$$

Therefore, effective speedup $= 3T/T = 3$. Let the fraction of vectorized code be α. Then $\alpha = 9 \times 0.25T/3T = 0.75$.

(b) Suppose the speed ratio between the vector mode and the scalar mode is doubled. The execution time becomes

$$0.75T + 0.25T/2 = 0.875T.$$

The effective speedup is $3T/0.875T = 24/7 = 3.43$.

(c) Suppose the speed for vector mode computation is still nine times as fast as that for scalar mode. To maintain the effective speedup of 3.43, the vectorization ratio α must satisfy the following relation:

$$\frac{1}{\frac{\alpha}{9} + \frac{1-\alpha}{1}} = \frac{24}{7}.$$

Solving the equation, we obtain $\alpha = 51/64 \approx 0.8$.

Problem 3.3

(a) Suppose the total workload is W million instructions. Then the execution time in seconds is

$$T = \frac{\alpha W}{nx} + \frac{(1-\alpha)W}{x}.$$

Therefore, the effective MIPS rate is

$$\frac{W}{T} = \frac{nx}{\alpha + n(1-\alpha)} = \frac{nx}{n-(n-1)\alpha}.$$

(b) Substituting the given data into the expression in (a), we have

$$\frac{16 \times 4}{16 - 15\alpha} = 40,$$

which can be solved to give $\alpha = 24/25 = 0.96$.

Problem 3.4 Assume the speed in enhanced mode is n times as fast as that in regular mode, the harmonic mean execution time T is calculated as

$$T(\alpha) = \alpha/R + (1-\alpha)/(nR),$$

where R is the execution rate in regular mode.

(a) If α varies linearly between a and b, the average execution time is

$$T_{avg} = \frac{\int_a^b T(\alpha)d\alpha}{(b-a)} = \frac{(n-1)(b+a)+2}{2nR}.$$

The average execution rate is

$$R_{avg} = \frac{1}{T_{avg}} = \frac{2nR}{(n-1)(b+a)+2},$$

and the average speedup factor is

$$S_{avg} = \frac{R_{avg}}{R} = \frac{2n}{(n-1)(b+a)+2}.$$

(b) If $a \to 0$ and $b \to 1$, then

$$S_{avg} = 2n/(n+1).$$

Problem 3.5

(a) The harmonic mean execution rate in MIPS is

$$R = \frac{1}{\sum_{1=1}^{n} f_i/R_i}.$$

The arithmetic mean execution time is

$$T = \sum_{i=1}^{n} f_i/R_i.$$

(b) Given $f_1 = 0.4, f_2 = 0.3, f_3 = 0.2$, $f_4 = 0.1$, and $R_1 = 4$ MIPS, $R_2 = 8$ MIPS, $R_3 = 11$ MIPS, $R_4 = 15$ MIPS, the arithmetic mean execution time is $T = 0.4/4 + 0.3/8 + 0.2/11 + 0.1/15 = 0.162$ μs per instruction.

Several factors cause R_i to be smaller than $5i$. First, there might be memory access operations which take extra machine cycles. Second, when the number of processors is increased, more memory access conflicts arise, which increase the execution time and lower the effective MIPS rate. Third, part of the program may have to be executed sequentially or can be executed by only a limited number of processors simultaneously. Finally, there is an overhead for processors to synchronize with each other. Because of these overheads, R_i/i typically decreases with i.

(c) Given a new distribution $f_1 = 0.1, f_2 = 0.2, f_3 = 0.3$, and $f_4 = 0.4$ due to the use of an intelligent compiler, the arithmetic mean execution time becomes $T = 0.1/4 + 0.2/8 + 0.3/11 + 0.4/15 = 0.104$ μs per instruction.

Problem 3.6 Amdahl's law is based on a fixed workload, where the problem size is fixed regardless of the machine size. Gustafson's law is based on a scaled workload, where the problem size is increased with the machine size so that the solution time is the same for sequential and parallel executions. Sun and Ni's law is also applied to scaled problems, where the problem size is increased to the maximum memory capacity.

Problem 3.7

(a) The total number of clock cycles needed is

$$C_0 = \sum_{i=1}^{1024} (2 + 2i) = 2 \times 1024 + 1024 \times 1025 = 1,051,628.$$

(b) If consecutive outer loops are assigned to a single processor, the workload is not balanced and the parallel execution time is dominated by that on processor 32. The clock cycles needed is

$$\begin{aligned} C_1 &= \sum_{i=993}^{1024} (2 + 2i) \\ &= (993 + 1025) \times 16 \\ &= 64,608. \end{aligned}$$

The speedup is

$$S_1 = \frac{1051648}{64608} \approx 16.28.$$

(c) To balance the load, we divide the outer loop into 64 chunks, each consisting of 16 iterations. Each processor is allocated a pair of chunks in a fold-over manner. That is, processor 1 is allocated the first and the last chunks, processor 2 the second and second to the last chunks, and so on. Thus, we have the following modified code:

```
      Doall ℓ = 1, 32
            Do 10 I = (ℓ − 1) × 16 + 1, ℓ × 16
               SUM(I) = 0
               Do 20 J = 1, I
20                SUM(I) = SUM(I) + J
10          Continue
            Do 30 I = (64 − ℓ) × 16 + 1, (64 − ℓ + 1) × 16
               SUM(I) = 0
               Do 40 J = 1, I
40                SUM(I) = SUM(I) + J
30          Continue
      Endall
```

(d) Suppose the overhead associated with flow control is neglected. The number of cycles required for the computation in processor $\ell, 1 \le \ell \le 32$, is

$$\begin{aligned} C_2 &= \sum_{i=(\ell-1)\times 16+1}^{\ell\times 16} (2\times i+1) + \sum_{i=(64-\ell)\times 16+1}^{(64-\ell+1)\times 16} (2\times i+1) \\ &= \{[(\ell-1)\times 16 + 2 + \ell\times 16 + 1] + \\ &\quad [(64-\ell)\times 16 + 2 + (64-\ell+1)\times 16 + 1]\} \times 16 \\ &= 2054 \times 16 = 32,864. \end{aligned}$$

The speedup in this case is

$$S_2 = \frac{1051648}{32864} = 32.$$

Problem 3.8

(a) An example program is shown below. Assume α, β, γ are the base addresses of A, B, C, respectively, which point to the first element of the individual arrays. Also assume only a small number of registers are available. The notation M(addr) stands for the value stored in memory location addr.

```
        Mov    R1, 0      Initialize R1 index i
        Mov    R5, 0      Initialize R5 = i × n
        Mov    R7, 0      Initialize R7 = offset of C_ij
Loop1:  Mov    R2, 0      Reset R2 − index j
```

Loop2:	Mov	R3, 0	Reset R3 = index k
	Mov	R4, 0	Reset R4 = $k \times n$
	Mov	R6, R5	Reset R6 = $i \times n$
	Mov	R11, 0	Initialize R11 = value of C_{ij}
Loop3:	Add	R4, R2	Compute offset for B_{kj}
	Load	R9, M(R4 + β)	Fetch B_{kj}
	Load	R10, M(R6 + α)	Fetch A_{ik}
	Mul	R10, R9	$A_{ik} \times B_{kj}$
	Add	R11, R10	Update C_{ij}
	Inc	R3	Increment k
	Inc	R6	Increment offset for A_{ik}
	Add	R4, n	Compute $k \times n$
	Cmp	R3, n	Check limit for k
	Jnz	Loop3	Loop until limit is reached
	Store	M(R7 + γ), R11	Store value of C_{ij}
	Inc	R2	Increment j
	Inc	R7	Increment offset for C_{ij}
	Cmp	R2, n	Check limit for j
	Jnz	Loop2	Loop until limit is reached
	Inc	R1	Increment i
	Add	R5, n	Compute $i \times n$
	Cmp	R1, n	Check limit for i
	Jnz	Loop1	Loop until limit is reached

(b) From the above code, the number of instructions is

$$I = 10n^3 + 9n^2 + 5n + 3.$$

For timing analysis, the following number of cycles is assumed for different types of instructions:

- Add, Mul, Cmp, Jnz: 2 cycles.
- Load, Store: 4 cycles.
- Mov, Inc: 1 cycle.

Based on the above assumptions, we obtain the following serial execution time:

$$T_s = (22n^3 + 14n^2 + 8n + 3) \text{ cycles.}$$

The average cycles per instruction can be calculated as

$$\text{CPI} = \frac{T_s}{I} = \frac{22n^3 + 14n^2 + 8n + 3}{10n^3 + 9n^2 + 5n + 3} \text{cycles / instruction.}$$

Asymptotically, CPI is close to 2.2 when n is large.

(c) If the clock rate is 40 MHz, a rough estimation for the MIPS rate is

$$\frac{40}{2.2} = 18.2 \text{ MIPS.}$$

(d) Matrix A is partitioned into blocks by the row and matrix B by the column, as shown in the following diagram:

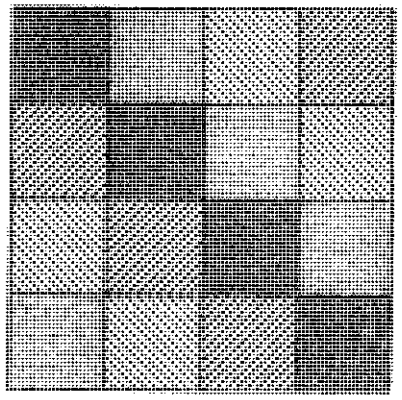
$$\begin{array}{|c|}\hline A_{1,\bullet} \\ \hline A_{2,\bullet} \\ \hline \cdots \\ \hline A_{N-1,\bullet} \\ \hline A_{N,\bullet} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|}\hline B_{\bullet,1} & B_{\bullet,2} & \cdots & B_{\bullet,N} \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline C_{1,1} & C_{1,2} & \cdots & C_{1,N} \\ \hline C_{2,1} & C_{2,2} & \cdots & C_{2,N} \\ \hline \cdots & \cdots & \cdots & \cdots \\ \hline C_{N-1,1} & C_{N-1,2} & \cdots & C_{N-1,N} \\ \hline C_{N,1} & C_{N,2} & \cdots & C_{N,N} \\ \hline \end{array}$$

Each block $A_{i,\bullet}$ represents the row vectors $A_{(i-1)n/N+1,\bullet}$ through $A_{in/N,\bullet}$. Similar notations are used for the block submatrices B. The multiplication of one A block of size $(n/N) \times n$ and a B block of size $n \times (n/N)$ yields one subblock of size $(n/N) \times (n/N)$ in the product matrix C.

The amount of time required for the computations in each processor is $22(n/N)^2n + 14(n/N)n + 8n/N + 3$, provided each processor is identical to the uniprocessor used in (a) and memory access conflicts are ignored. Each processor needs to compute N such subblocks. Thus, the total parallel execution time is $(22n^3/N + 14n^2 + 8n + 3N)$ cycles. The potential speedup is $(22n^3 + 14n^2 + 8n + 3)/(22n^3/N + 14n^2 + 8n + 3N) \approx N$ when n is large.

(e) The matrix is partitioned as in part (d). Initially, node i has submatrix $A_{i,\bullet}$ and $B_{\bullet,i}$ for $i = 1..N$. In the first step, each node computes a subblock C_{ii} of matrix C. After that, the nodes exchange subblocks of B in the following manner: Node 1 sends its B block to node 2, node 2 sends its B block to node 3, ..., Node N sends its B block to node 1. Then each node computes a subblock $C_{i,i+1}$ except node N which computes $C_{N,1}$. The process is repeated until all the subblocks of C are computed in N steps. If the initial distribution of B to the nodes is not counted, the number of message-passing steps is $N - 1$.

The sequence of computations is illustrated in the following diagram for $N = 4$, with different shades indicating subblocks computed in different steps. Each block corresponds to a C subblock of size $(n/4) \times (n/4)$.



(f) Assume each message consists of a single element of matrix B, which is 8 bytes for double-precision floating-point numbers. The message sending operation for node i in step j $(1 \leq j \leq N-1)$ can be specified as follows:

```
/* Process sending messages */
        jj = j + i - 1
        if (jj == N+1) jj = 1 do
        for k = (jj-1) * n / N + 1 to jj * n / N do
              for l = 1 to n do
                    if (i == N) send(1, B(l,k), 8);
                    else send(i+1, B(l,k), 8)
              enddo
        enddo
```

The first parameter of the send instruction is the destination node ID, the second is the element of B to be transmitted, and the third is the length of the message. There should also be code in the node receiving the message; it is similar to the sending counterpart. For simplicity the code is not shown. By symmetry, each node must execute both sending and receiving instructions.

The execution time t_{mc} can be divided into the time for arithmetic operations (t_a) and that for communication (t_c), assuming there is no overlap between the two types of operations. The time for arithmetic operations is identical to that on a shared-memory multiprocessor, which is $t_a = 22n^3/N + 14n^2 + 8n + 3N$. The total number of message-passing operations is $(N-1) \times (n/N) \times n$. Thus,

$$t_{mc} = (22n^3/N + 14n^2 + 8n + 3N + 100(N-1) \times (n/N) \times n) \text{ cycles.}$$

Therefore, the speedup is

$$\frac{22n^3 + 14n^2 + 8n + 3}{22n^3/N + 14n^2 + 8n + 3N + 100(N-1) \times (n/N) \times n}$$

Note that different assumptions for this problem will lead to different speedup results. It is also possible to use other matrix multiplication algorithms such as those described in the text.

Problem 3.9

(a) Arithmetic mean execution time of each machine is calculated as follows:

- Machine A: $(1 + 1000 + 500 + 100)/4 = 400.25$ s.
- Machine B: $(10 + 100 + 1000 + 800)/4 = 477.5$ s.
- Machine C: $(20 + 20 + 50 + 100)/4 = 47.5$ s.

(b) Harmonic mean MIPS rates:

- Machine A: $100/400.25 = 0.25$ MIPS.
- Machine B: $100/477.5 = 0.21$ MIPS.
- Machine C: $100/47.5 = 2.1$ MIPS.

(c) In terms of harmonic mean execution rate, Machine C is higher than Machine A, which is in turn higher than Machine B. See also the discussion in Problem 1.6.

Problem 3.10

(a) The total execution time in serial execution is

$$T(1) = \sum_{i=1}^{m^*} W_i^* / \Delta.$$

In parallel execution with n processors, the execution time is

$$T(n) = \sum_{i=1}^{m^*} \frac{W_i^*}{i\Delta} \lceil \frac{i}{n} \rceil.$$

Therefore, the speedup for the fixed-memory model is

$$S_n^* = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \lceil \frac{i}{n} \rceil}.$$

If (i) $W_i^* = 0$ for $i \neq 1$ and $i \neq n$, (ii) $W_1^* = W_1$, (iii) $W_n^* = G(n)W_n$, then

$$S_n^* = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n/n}.$$

(b) When $G(n) = 1$, i.e., problem size remains fixed when memory size is increased, Amdahl's law is obtained:

$$S_n = \frac{W_1 + W_n}{W_1 + W_n/n}.$$

(c) When $G(n) = n$, i.e., problem size increases in direct proportion to memory size increase (which in turn is proportional to the number of processors), Gustafson's law is obtained:

$$S_n' = \frac{W_1 + nW_n}{W_1 + W_n}.$$

(d) Let $W_1 = \alpha, W_n = 1 - \alpha$. The relation $S_n \leq S_n'$ follows from the definitions. Thus,

$$S_n = \frac{\frac{\alpha}{n} + 1 - \alpha}{\alpha + (1 - \alpha)} = \frac{\alpha + n(1 - \alpha)}{n} \leq \alpha + n(1 - \alpha) = S_n'.$$

We now show $S_n^* > S_n'$. Assume $G(n) = g \geq n$.

$$S_n' = \frac{\alpha + n(1 - \alpha)}{\alpha + \frac{n(1-\alpha)}{n}}, \tag{3.1}$$

$$S_n^* = \frac{\alpha + g(1 - \alpha)}{\alpha + \frac{g(1-\alpha)}{n}}. \tag{3.2}$$

Let $\beta = (1-\alpha)/n$. Eqs. 3.1 and 3.2 can be rewritten as

$$S'_n = \frac{\alpha + n^2\beta}{\alpha + n\beta} \tag{3.3}$$

and

$$S^*_n = \frac{\alpha + gn\beta}{\alpha + g\beta}. \tag{3.4}$$

Consider three different cases:

1. $\alpha = 1, \beta = 0$. $S'_n = S^*_n = 1$.
2. $\alpha = 0, \beta = 1$. $S'_n = S^*_n = n$.
3. $0 < \alpha, \beta < 1$. We have

$$\begin{aligned} S'_n - S^*_n &= (\alpha + n^2\beta)(\alpha + g\beta) - (\alpha + gn\beta)(\alpha + n\beta) \\ &= (n^2\alpha\beta + g\alpha\beta) - (gn\alpha\beta + n\alpha\beta) \\ &= (n^2 + g - gn - n)\alpha\beta \\ &= (n-1)(n-g)\alpha\beta \leq 0 \qquad \text{for } n \geq 1, g \geq n, \end{aligned}$$

which completes the proof.

Problem 3.11 In a reasonable execution environment, the workload and execution times should satisfy the following conditions:

1. At most n instructions can be executed by n processors simultaneously:

$$T(n) \leq O(n) \leq nT(n). \tag{3.5}$$

2. $O(n)$ should be at most n times as large as $O(1)$:

$$O(1) \leq O(n) \leq nO(1) \Longrightarrow 1 \leq R(n) \leq n. \tag{3.6}$$

3. With n processors, the execution time is reduced by at most a factor of n:

$$T(n) \leq T(1) \leq nT(n) \Longrightarrow 1 \leq S(n) \leq n. \tag{3.7}$$

4. When only one processor is used, it is fully utilized:

$$T(1) = O(1). \tag{3.8}$$

(a) Equation 3.7 and $E(n) = S(n)/n$ imply

$$1/n \leq E(n) \leq 1 \Longrightarrow 1 \leq 1/E(n) \leq n. \tag{3.9}$$

Also, $U(n) = E(n)R(n) \geq E(n)$ by Eq. 3.6. From Eq. 3.5, we have

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)} \leq 1 \Longrightarrow R(n) \leq 1/E(n). \tag{3.10}$$

The proof is completed by combining the inequalities.

(b) The result is obtained by combining Eqs. 3.6, 3.9, and 3.10.

(c)

$$\begin{aligned} Q(n) &\equiv \frac{S(n)E(n)}{R(n)} \\ &= \frac{T(1)}{T(n)}\frac{T(1)}{nT(n)}\frac{O(1)}{O(n)} \\ &= \frac{T(1)}{T(n)}\frac{T(1)}{nT(n)}\frac{T(1)}{O(n)} \qquad \text{(from Eq. 3.8)} \\ &= \frac{T^3(1)}{nT^2(n)O(n)}. \end{aligned}$$

(d) The following inequalities can be easily shown to hold for $n \geq 1$:

$$1/n \leq (n+3)/(4n) \leq (n+3)(n+\log_2 n)/(4n^2) \leq 1.$$

$$1 \leq (n+\log_2 n)/n \leq 4n/(n+3) \leq n.$$

Note that when $n = 1$, all the inequalities become equalities. The inequalities are also verified by Fig. 3.4 in the text.

Problem 3.12

(a) The best known sequential algorithm for sorting s numbers has a complexity of $O(s \log s)$. On an EREW-PRAM, sorting of s numbers can be carried out in $O(\log s)$ time using s processors, and the speedup is $O(s)$. See [JaJa92]. Complexities of sorting on machines of different topologies are summarized below:

- Linear array. Odd-even transposition sort can be used to sort n elements with n processors in $O(n)$ time units. We can choose $n = s$, resulting in a time complexity of $O(s)$. The speedup is $O(\log s)$ and the scalability is $O(\log s/s)$.
- 2D mesh. With n processors, swap of two elements at diagonally opposite corners of the mesh require $O(n^{1/2})$ time units. This puts a limit on the lower bound of any sorting algorithm. This bound can be achieved by using Batcher's bitonic merge sort. We can choose $n = s$ for a time complexity of $O(s^{1/2})$. The speedup is $O(s^{1/2} \log s)$ and the scalability is $O(\log s/s^{1/2})$.
- 3D mesh. As in 2D mesh, the need to exchange two elements located at the opposite diagonal of a 3D mesh sets a bound on the complexity. With n processors, $O(n^{1/3})$ time units are required. We can choose $n = s$, resulting in a time complexity of $O(s^{1/3})$. The speedup is $O(s^{2/3} \log s)$ and the scalability is $O(\log s/s^{1/3})$.
- Hypercube. Bitonic merge sort can be implemented easily on a hypercube because the algorithms requires comparison of elements whose indices differ in exactly one bit, which coincides with the interconnection pattern of a

hypercube. The complexity is $O(\log^2 n)$. We can choose $n = s$, leading to a complexity of $O(\log^2 s)$. The speedup is $O(s/\log s)$ and the the scalability is $O(1/\log s)$.

- Omega network. Intuitively, sorting a sequence of s numbers is equivalent to a permutation of the sequence to the right order. Since an input sequence can be in any order, to sort any given sequence is equivalent to realizing all possible permutations on the network. It is known that in order to realize all permutations of s numbers on an Omega network, at most $\log s$ passes are needed (Chapter 2). Each pass goes through $\log s$ stages. Therefore the total complexity is $\log^2 s$. In practice, using bitonic merge sort to sort s elements on an Omega network takes $O(\log^2 s)$ compare-exchange and shuffle steps. The speedup is $O(s/\log s)$ and the scalability is $O(1/\log s)$.

The results are tabulated below. For details, see Chapter 4 of [Quinn87] and references therein. [Leighton92] also contains extensive discussion of sorting on various parallel architectures.

Metrics	Machine Architecture				
	Linear array	2-D mesh	3-D mesh	Hypercube	Omega Network
$T(s,n)$	s	$s^{1/2}$	$s^{1/3}$	$\log^2 s$	$\log^2 s$
$S(s,n)$	$\log s$	$s^{1/2}\log s$	$s^{2/3}\log s$	$s/\log s$	$s/\log s$
$\Phi(s,n)$	$\log s/s$	$\log s/s^{1/2}$	$\log s/s^{1/3}$	$1/\log s$	$1/\log s$

(b) Based on the results of Example 3.7 and those in the above table, hypercube and Omega network have the best scalability, followed by 3D mesh, 2D mesh, and linear array in that order. Generally, the higher connectivity a network has, the higher scalability it can achieve. Another important factor that affects scalability is how well the communication pattern of an algorithm matches the interconnection topology of an architecture, as illustrated by the scalability of the two algorithms on hypercube and Omega network.

Problem 3.13

(a) • Computer 1:

$$R_1 = 100/1 = 100 \text{ Mflops.}$$

$$R_2 = 100/1000 = 0.1 \text{ Mflops.}$$

$$R_a = 0.5 \times 100 + 0.5 \times 0.1 = 50.05 \text{ Mflops.}$$

$$R_g = 100^{0.5}0.1^{0.5} = 3.16 \text{ Mflops.}$$

$$R_h = 1/(0.5/100 + 0.5/0.1) = 0.2 \text{ Mflops.}$$

- Computer 2:

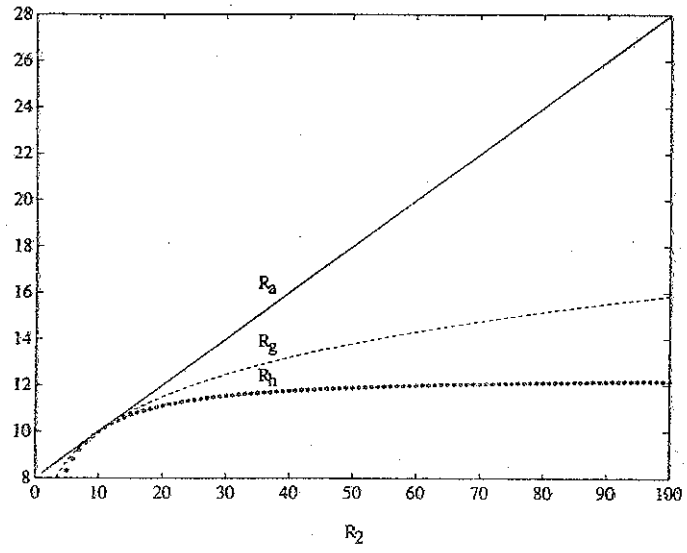
$$R_1 = 100/10 = 10 \text{ Mflops.}$$
$$R_2 = 100/100 = 1 \text{ Mflops.}$$
$$R_a = 0.5 \times 10 + 0.5 \times 1 = 5.5 \text{ Mflops.}$$
$$R_g = 10^{0.5}1^{0.5} = 3.16 \text{ Mflops}$$
$$R_h = 1/(0.5/10 + 0.5/1) = 1.82 \text{ Mflops.}$$

- Computer 3:

$$R_1 = 100/20 = 5 \text{ Mflops.}$$
$$R_2 = 100/20 = 5 \text{ Mflops.}$$
$$R_a = 0.5 \times 5 + 0.5 \times 5 = 5 \text{ Mflops.}$$
$$R_g = 5^{0.5}5^{0.5} = 5 \text{ Mflops.}$$
$$R_h = 1/(0.5/5 + 0.5/5) = 5 \text{ Mflops.}$$

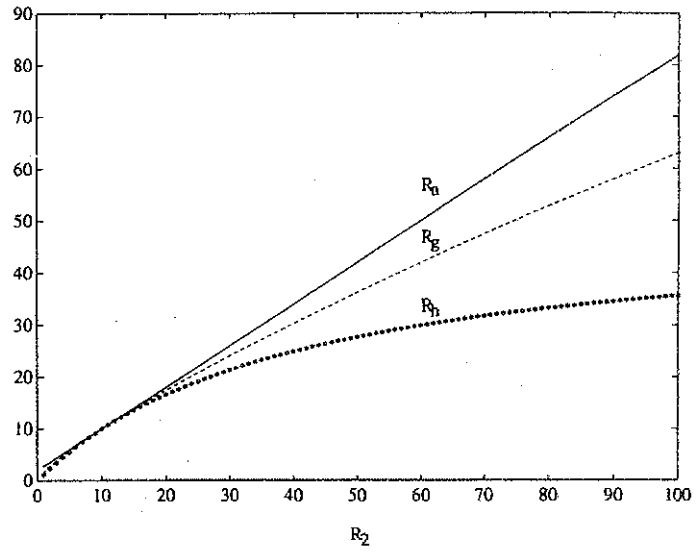
(b)

$$\begin{aligned} R_a &= 0.8 \times 10 + 0.2R_2 = 8 + 0.2R_2 \text{ Mflops.} \\ R_g &= 10^{0.8}R_2^{0.2} \text{ Mflops.} \\ R_h &= 1/(0.8/10 + 0.2/R_2) = 10R_2/(0.8R_2 + 2) \text{ Mflops.} \end{aligned}$$



(c)

$$\begin{aligned} R_a &= 0.2 \times 10 + 0.8R_2 = 2 + 0.8R_2 \text{ Mflops.} \\ R_g &= 10^{0.2}R_2^{0.8} \text{ Mflops.} \\ R_h &= 1/(0.2/10 + 0.8/R_2) = 10R_2/(0.2R_2 + 8) \text{ Mflops.} \end{aligned}$$



(d) Suppose harmonic mean MIPS rate is used as the criterion to compare the relative performance of the three machines. For machine 1, the value is

$$R_h^{(1)} = \frac{1}{f_1/100 + f_2/0.1} = \frac{100}{1000 - 999f_1}.$$

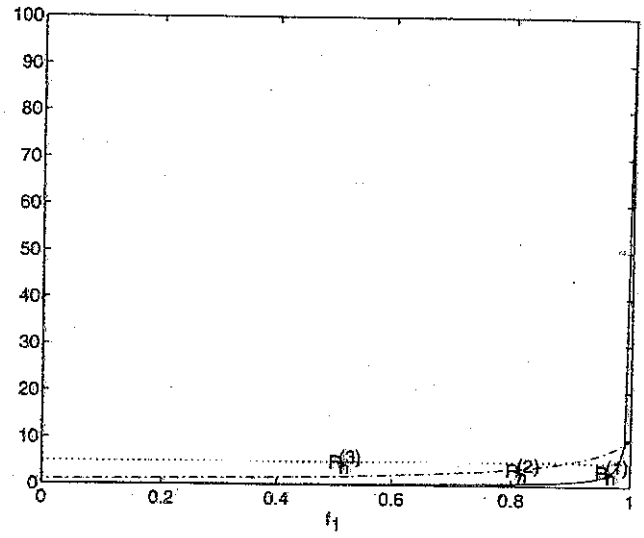
Similarly, for machines 2 and 3, the values are

$$R_h^{(2)} = \frac{1}{f_1/10 + f_2/1} = \frac{10}{10 - 9f_1}$$

and

$$R_h^{(3)} = \frac{1}{f_1/5 + f_2/5} = \frac{5}{1} = 5.$$

respectively. We can plot $R_h^{(1)}, R_h^{(2)}, R_h^{(3)}$ as a function of f_1. The following diagram shows the variation of the harmonic mean MIPS rate for the three machines with respect to f_1.



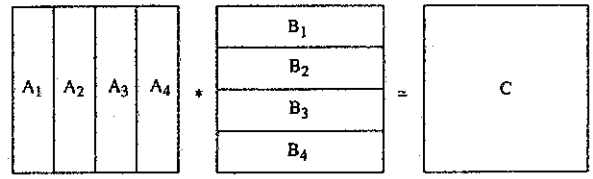
It is seen that $R_h^{(1)}$ is very sensitive to the value of f_1, $R_h^{(2)}$ varies slowly with f_1, and $R_h^{(3)}$ remains constant independent of the value of f_1. For most values of f_1, $R_h^{(3)}$ has a larger value than $R_h^{(1)}$ and $R_h^{(2)}$. But when f_1 is close to 1, $R_h^{(1)}$ and $R_h^{(2)}$ surpass $R_h^{(3)}$. A large value of f_1 means that most of the time is spent on the high MIPS benchmark for machines 1 and 2, leading to a high harmonic mean MIPS rate for the two machines.

Problem 3.14 The communication cost of a data exchange between two directly connected nodes is modeled by $\alpha + \beta m$, where α is the time required to set up a channel, β is the time to transmit one word over the communication channel, and m is the amount of data exchanged.

(a) In Fox-Otto-Hey algorithm, each processor in a $\sqrt{n} \times \sqrt{n}$ torus is assigned s^2/n elements of matrices A and B. The algorithm requires a total of $\sqrt{n}$ iterations. During each iteration, a subblock of A is broadcast along the row of the torus. Therefore, in each iteration, the time taken for communication is $\sqrt{n}(\alpha + s^2\beta/n)$. Since $\sqrt{n}$ iterations are required, the total time for communicaiton is $(n\alpha + s^2\beta)$. However, if the matrix subblocks are sent in a pipelined fashion (such as wormhole routing), the second term is reduced by a factor of $2/\sqrt{n}$ (for details, please see [Fox87]), resulting in $(n\alpha + 2s^2\beta/\sqrt{n})$ for communication overhead.

If the torus is embedded in a hypercube and a sophisticated one-to-all broadcast scheme such as the one in [Johnsson89] is used, the communication time can be further reduced to $(\log n\alpha + 2s^2\beta/\sqrt{n})$. Therefore the total communication overhead on n processors is $n \log n\alpha + 2s^2\sqrt{n}\beta$.

(b) Berntsen's algorithm is designed to take advantage of the higher connectivity in hypercube computers. Matrices A and B are partitioned into 2^k strips by column and by row, respectively, as follows:



The product matrix C is computed as

$$C = \sum_{i=1}^{2^k} A_i B_i.$$

The hypercube is divided into 2^k subcubes, each comprising 2^{2k} nodes. The first step of the algorithm involves the computation of $C_i = A_i B_i$, which is carried out in each subcube using Cannon's algorithm [Cannon69]. See solution of Problem 8.12 for an example. The communication overhead in this step is[1]

$$T_1 = 2^k(2\alpha + 2\frac{s^2}{n}\beta).$$

In the second step, C_i in the subcubes are summed together using a "cascade sum" algorithm [Berntsen90]. This step requires communications among subcubes with an overhead

$$T_2 = k\alpha + 2^k\frac{s^2}{n}\beta.$$

The total communication overhead is $n(T_1 + T_2)$. Using the relation $k = (\log n)/3$, $2^k = n^{1/3}$, the complexity of the communication overhead is proved.

(c) In Dekel-Nassimi-Sahni algorithm, multiplication is performed on a hypercube with $2^{3q} = s^3$ nodes. Each node r in the hypercube is identified by a 3-tuple (i, j, k), $1 \le i, j, k \le 2^q = s$. At the beginning, elements A_{jk} and B_{jk} are stored in node $(0, j, k)$. At the end, C_{jk} is also stored in node $(0, j, k)$.

The algorithm consists of three phases. In the first phases, A_{jk} and B_{jk} are replicated on nodes $(i, j, k), 1 \le i \le s$. Then the element of A at node (i, j, i), which is A_{ji}, is replicated on nodes $(i, j, k), 1 \le k \le s$. Similarly, the element of B at node (i, i, k), which is b_{ik}, is replicated on nodes $(i, j, k), 1 \le j \le s$. This phase involves the movement of data among neighboring nodes along different dimensions of the hypercube, and the total amount of data moved is $2s^3$, giving a total communication time of $(4nq + 2s^3)(\alpha + \beta)$

In the second phase, all the nodes compute $A_{jl} \times B_{lk}$ locally. This phase does not involve any communicaiton overhead. In the last phase, the final value of C_{jk}

[1] In general, the communication cost of multiplying two matrices of sizes $n_1 \times n_2$ and $n_2 \times n_1$ on a $t \times t$ torus is $2t\alpha + (2n_1n_2/t)\beta$ using Cannon's algorithm. Since a torus can be embedded in a hypercube with the same number of nodes, the complexity is identical on the hypercube.

is obtained by summing the product terms along the i dimension. This step incurs a communication overhead of $(nq + s^3)(\alpha + \beta)$. Therefore, the communication overhead of the entire algorithm is

$$(5nq + 3s^3)(\alpha + \beta) = (\frac{5}{3} n \log n + 3s^3)(\alpha + \beta).$$

Problem 3.15 The parallel speed is defined as

$$R_N = \frac{W}{T_N}$$

from which speedup can be written as

$$S_N = \frac{R_N}{R_1} = \frac{W}{T_N R_1}.$$

Here the workload parameter for R_1 is omitted since it is assumed to be independent of workload. Based on the definition of isoefficiency, we have

$$\begin{aligned} \frac{W}{NT_N R_1} &= \frac{W'}{N'T_{N'}R_1} \Longrightarrow \\ \frac{W}{NT_N} &= \frac{W'}{N'T_{N'}} \end{aligned}$$

and the isospeed condition is obtained.

Chapter 4

Processors and Memory Hierarchy

Problem 4.1

(a) Processor design space is a coordinated space with the x and y axes representing clock rate and CPI, respectively. Each point in the space corresponds to a design choice of a processor whose performance is determined by the values of the coordinates.

(b) The time required between issuing two consecutive instructions.

(c) The number of instructions issued per cycle.

(d) The number of cycles required for the execution of a simple instruction, such as add, move, etc.

(e) Two or more instructions attempt to use the same functional unit at the same time.

(f) A coprocessor is usually attached to a processor and performs special functions at a fast speed. Examples are floating-point and graphical coprocessors.

(g) Registers which are not designated for special usage, as opposed to special-purpose registers such as base registers or index registers.

(h) Addressing mode specifies how the effective address of an operand is generated so that its actual value can be fetched from the correct memory location.

(i) In the case of a unified cache, both data and instructions are kept in the same cache. In split caches, data and instructions are held in separate caches.

(j) Hardwired control: Control signals for each instruction are generated by proper circuitry such as delay elements. Microcoded control: Each instruction is implemented by a set of microinstructions which are stored in a control memory. The decoding of microinstructions generates appropriate signals to control the execution of an instruction.

Problem 4.2

(a) Virtual address space is the memory space required by a process during its execution to accommodate the variables, buffers, etc., used in the computations.

(b) Physical address space is the set of addresses assigned to the physically available memory words.

(c) Address mapping is the process of translating a virtual address to a physical address.

(d) The entirety of a cache is divided into fixed-size entities called blocks. A block is the unit of data transfer between main memory and cache.

(e) Multiple levels of page tables used to translate a virtual page number into a page frame number. In this case, some tables actually store pointers to other tables, similar to indirect addressing mode. The objective is to deal with a large memory space and facilitate protection.

(f) Hit ratio at level i of the memory hierarchy is the probability that a data item is found in M_i.

(g) Page fault is the situation in which a demanded page cannot be found in the main memory and has to be brought in from the disk.

(h) A hash function maps an element in a large set to an index in a small set. Usually it treats the input element as a number or a sequence of numbers and performs arithmetic operation on it to generate the index. A suitable hash function should map the input set uniformly into the output set.

(i) An inverted page table contains entries that record the virtual page number associated with each page frame that has been allocated. This is contrary to a direct mapping page table.

(j) The strategies used to select page or pages resident in the main memory to be replaced in case such needs arise.

Problem 4.3

(a) A windowing system divides the register file on a machine into groups which are assigned to different processors. There is usually overlap among the register sets to provide a fast communication mechanism among cooperating procedures for parameter passing and to allow fast context switching. The use of a large number of GPRs allows less frequent memory accesses and speeds up program execution.

(b) A large register file and a large data cache both serve the purpose of reducing memory traffic. From implementation point of view, the same chip area can be used for either a large register file or a large data cache. From programming point of view, registers can be manipulated by program code, but cache is transparent to the user. In fact, data cache is primarily involved in load/store operations. The addressing of a cache involves address translation and is more complicated than

that of a register file.

Reservation stations and reorder buffers are used in superscalar machines to facilitate instruction lookahead and internal data forwarding which are needed to schedule multiple instructions through multiple pipelines simultaneously.

(c) In most RISC processors, the integer unit executes load, store, integer, bit, and control transfer functions. It also fetches instructions for the floating-point unit in some systems. The floating-point unit performs various arithmetic operations on floating-point numbers. The two units can operate concurrently.

Problem 4.4

(a) The comparison is tabulated below:

Item	CISC	RISC
Instruction format	16–64 bits per instruction	fixed (32-bit) format
Addressing modes	12–24	limited to 3–5 (mostly register-based, except load/store)
CPI	2–15, on the average 5	< 1.5, very close to 1

(b)

- Advantages of separate caches:
  1. Double the bandwidth because two complementary requests are serviced at the same time.
  2. Simplify logic design as arbitration between instruction and data accesses to the cache is simplified or eliminated.
  3. Access time is reduced because data and instruction can be placed close to the functional units which will access them. For instance, instruction cache can be placed close to the instruction fetch and decode units.
- Disadvantages of separate caches:
  1. Complicate the problem of consistency because data and instruction may coexist in the same cache block. This is true if self-modifying code is allowed or when data and instructions are intermixed and stored in the same cache block. To avoid this would require compiler support to ensure that instruction and data are stored in different cache blocks.
  2. May lead to inefficient use of cache memory because the working set size of a program varies with time and the fraction devoted to data and instruction also varies. Hence, the sum of data cache size and instruction cache size is usually larger than the size of a unified cache. As a result, the utilization of instruction cache and/or data cache is likely to be lower.

For separate caches, dedicated data paths are required for both instruction and data caches. Separate MMUs and TLBs are also desirable for separate caches to shorten the time of address translation. A higher memory bandwidth should be used for separate caches to support the increased demand.

In actual implementation, there is tradeoff between the degree of support provided and the resulting hardware complexity.

(c)
- Instruction issue: Scalar RISC processor issues one per cycle; superscalar RISC can usually issue more than one per cycle.
- Pipeline architecture: In an m-issue superscalar processor, up to m pipelines may be active in any base cycle. A scalar processor is equivalent to a superscalar processor with $m = 1$.
- Processor performance: An m-issue superscalar can have a performance m times that of a scalar processor, provided both are driven by the same clock rate, no dependence relation or resource conflicts exist among instructions.

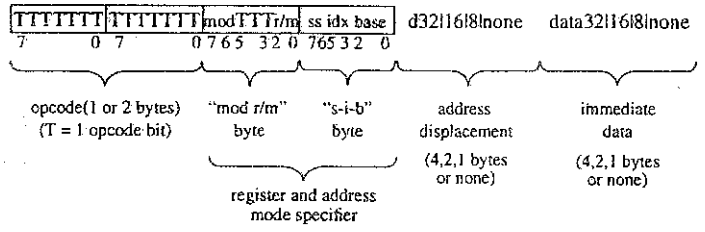
(d) Both superscalar and VLIW architectures employ multiple functional units to allow concurreut instruction executions. Superscalar requires more sophisticated hardware support such as large reorder registers and reservation tables iu order to make efficient use of the system resources. Software support is needed to resolve data dependences and improve efficiency.

In VLIW, instructions are compacted by compiler which explicitly packs together instructions which can be executed in concurrency based on heuristics or run-rime statistics. Because of the explicit specification of parallelism, the hardware and software support at run time is usually simplified. For instance, the decoding logic can be simple.

Problem 4.5 Only a single pipeline in scalar CISC or RISC architecture is active at a time, exploiting parallelism at microinstruction level. Operation requirement is simple. In a superscalar RISC, multiple pipelines can be active simultaneously. To do so requires extensive hardware and software support to effectively exploit instruction parallelism. In a VLIW architecture, multiple pipelines can be active at the same time. Sophisticated compilers are needed to compact irregular codes into a long instruction word for concurrent execution.

Problem 4.6

(a) i486 is a CISC processor. The following diagram shows the general instruction format. A few variations also exist for some instructions.



Data format:

- Byte (8 bits): 0–255
- Word (16 bits): 0–64K
- DWord (32 bits): 0–4G
- 8-bit integer (8 bits): 10^2
- 16-bit integer (8 bits): 10^4
- 32-bit integer (8 bits): 10^9
- 65-bit integer (8 bits): 10^{19}
- 8-bit unpacked BCD (1 digit): 0–9
- 8-bit packed BCD (2 digits): 0–9
- 80-bit packed BCD (18 digits): $\pm 10^{\pm 18}$
- Single-precision real (24 bits): $\pm 10^{\pm 38}$
- Double-precision real (53 bits): $\pm 10^{\pm 308}$
- Extended-precision real (64 bits): $\pm 10^{\pm 4932}$
- Byte string, Word string, DWord string, Bit string to support ASCII data types.

(b) There are 12 different modes whereby the effective address (EA) can be generated:

- register mode
- immediate mode
- direct mode: EA $\leftarrow$ displacement
- register indirect or base: EA $\leftarrow$ (base register)
- based with displacement: EA $\leftarrow$ (base register) + displacement
- index with displacement: EA $\leftarrow$ (index register) + displacement
- scaled index with displacement: EA $\leftarrow$ (index register) $\times$ scale + displacement
- based index: EA $\leftarrow$ (base register) + (index register)
- based scaled index: EA $\leftarrow$ (base register) + (index register) $\times$ scale
- based index with displacement: EA $\leftarrow$ (base register) + (index register) + displacement
- based scaled index with displacement: EA $\leftarrow$ (base register) + (index register) $\times$ scale + displacement
- relative: New_PC $\leftarrow$ PC + displacement. (used in conditional jumps, loops, and call instructions)

(c) Instruction categories:

- data transfer: MOV dst, src
- arithmetic: ADD dst, src
- logic, shift, and rotate:

  AND dst, src
  SHL dst, count
  ROL dst, count

- string comparison: CMPS sdst, ssrc
- bit manipulation: BT dst, bit
- control transfer: JMP addr
- high-level language support: LEAVE (procedure exit)
- protection support: LSL dst, src (load segment limit)
- floating-point operation: FADD tmp
- floating-point control: FINIT (initialize FPU)

(d) HLL support instructions:

```
BOUND reg, addr        ; check if (addr) ≤ (reg) ≤ (addr + S)
ENTER imm16, imm8      ; make stack imm16 bytes at nesting level imm8
LEAVE
SETcc byte             ; set byte on cond, reset byte to 0 otherwise
```

Assembly directives: commands to the assembler, not executable. For instance the following directives define a data segment:

```
DATA1    SEGMENT
  ...
DATA1    ENDS
```

(e) Interrupt, debugging, and testing features.

- Interrupt: i486 can handle up to 256 different interrupts, 32 of which are reserved for Intel, the others can be designed by users. The starting address of an interrupt service routine is called an interrupt vector. The interrupt vectors are stored in an interrupt vector table (IVT). When an interrupt occurs, relevant register values are pushed onto a stack. The interrupt number is used by CPU to retrieve the corresponding interrupt vector from the IVT. After the interrupt service routine is executed, the program resumes execution of the interrupted instruction.
- Two types of test are available:
  1. built-in self test: tests nonrandom logic realized by PLAs, control ROM, TLB, and on-chip cache.
  2. external tests: can be performed on TLB and on-chip cache.
- Three types of on-chip debugging aids are provided:
  1. code execution breakpoint opcode: can be inserted at any desired breakpoint.
  2. single-step capability.
  3. code and data breakpoint capability provided by debug registers.

(f) 80486 allows the execution of 8086 application programs in two modes:

- Real mode: This has the same base architecture as the 8086 but the program is allowed to access the 32-bit register set of the 80486. Default operand size is 16 bits. Paging is not allowed in this mode. The maximum memory size is limited to 2^{20} = 1 Mbyte.

- Virtual mode: This mode allows 8086 application programs to take full advantage provided by i486. In this mode, i486 can execute any 8086, 80286, and 80386 software. Also, paging allows more flexible address mapping. The linear address space available is 2^{32} = 4 Gbytes and virtual address space is 2^{46} bytes.

(g) By setting PG bit (bit 31) of control register CR0 to 0, paging is disabled. This can be controlled by software. When paging is disabled, the linear address generated by segmentation mechanism is the same as the physical memory address and can be used directly to access the data from memory. Paging is used to cope with external fragmentation problem, but it also slows down the system. Applications which have stable memory requirement throughout the execution may use this feature (paging disable) to improve efficiency.

(h) By selecting a segment size of 4 Gbytes, the entire linear address space becomes a single segment, which essentially disables the segmentation mechanism. In this case, segment offset, linear address, and physical address are all identical. Segmentation provides a logical view of the memory space and facilitates protection and sharing of data. If the system is dedicated to a single application program which requires huge memory space, segmentation can be disabled.

(i) Four levels of protections, called privilege levels, are provided:

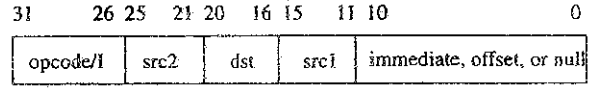
level 0 (PL = 0): kernel	most privileged
level 1 (PL = 1): system services	
level 2 (PL = 2): OS extensions	
level 3 (PL = 3): applications	least privileged

Data stored in a segment with PL = p can be accessed only by code with PL $\leq$ p. A code segment with PL = p can be called only by a task executing with PL $\leq$ p.

(j) Intel i586 has been renamed to Pentium. No detailed information on the processor is available yet.

Problem 4.7

(a) (1) The general format of an instruction in the i860 is shown below:



There are several variants to this format. Floating-point instructions also have a similar format but provide bit fields for specifying precision of operands, pipeline mode, and dual-instruction mode.

Data formats supported include

- Load/store references 8, 16, 32, 64, 128-bit operands.
- Integer operations are performed on 32-bit operands.

- Integer arithmetic operations support 8 and 16-bit operands by sign-extending the operands to 32 bits
- Floating point numbers follow IEEE 754 Standard (see Chapter 6)
- Graphical pixels of 8, 16, and 32 bits are supported. However, regardless of pixel size, i860 always operates on 64 bits of pixels at a time.

(2) Four basic addressing modes are supported:

- Offset: absolute address into the first or last 32 Kbytes of the logical address space.
- Register: operand in a CPU register.
- Register indirect + offset: EA ← const + (reg).
- Register indirect + index: EA ← (reg1) + (reg2)

(3) Instruction categories:

- Load/store instructions:

```
ld.X      ; load integer
```

- Register-to-register move instructions:

```
ixfr      ; transfer integer to F-P register
```

- Integer arithmetic instructions:

```
addu      ; add unsigned
```

- Shift instructions:

```
shl       ; shift left
```

- Logical instructions:

```
andnot    ; logical AND NOT
```

- Control-transfer instructions:

```
intovr    ; software trap on integer overflow
```

- System control instructions

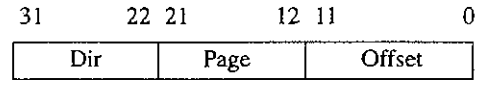
```
flush     ; cache flush
```

(b) i860XP executes hardware snooping instructions, whereas in the previous generation, i860XR, multiprocessor cache consistency requires software to avoid cacheing shared writable data.

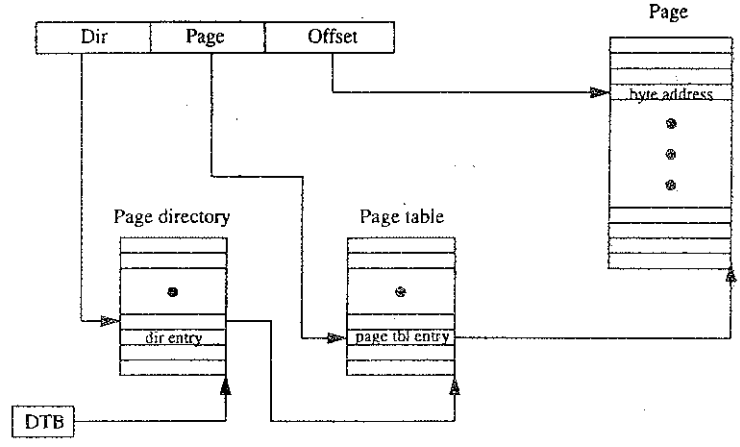
(c) Dual operation mode refers to the simultaneous execution, under the supervision of floating-point control unit, of floating-point operations in the adder and multiplier. Such operations can be specified by dual-operation instructions such as Sub-Multiply or Add-Multiply.

Dual instruction mode refers to the capability for the integer unit and floating-point unit to execute instructions in parallel. Programmers can specify dual-instruction mode by using assembler directives or by explicitly modifying the op-code mnemonics.

(d) i860 has a virtual address space of 2^{32} bytes. Translation of virtual address to physical address is optional and is in effect only when ATE (Address Translation Enable) bit in the directory base register is set to 1 by the operating system. The format of virtual address is as follows:

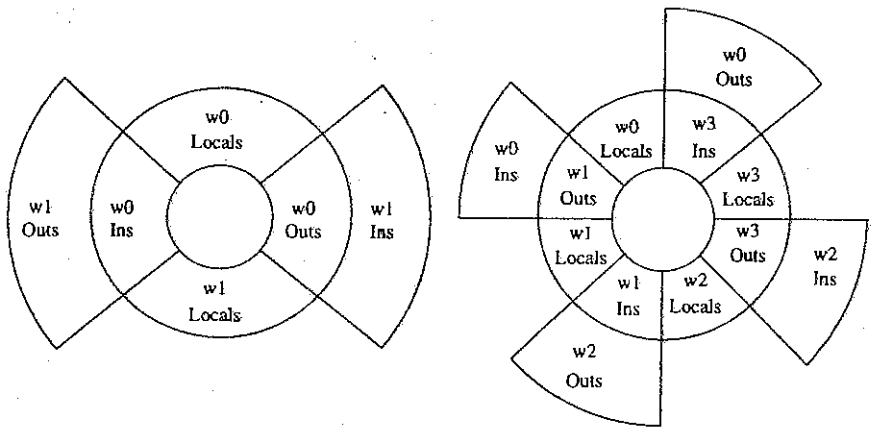


The address translation mechanism uses the Dir field as an index into a page directory, which is pointed to by the DTB (directory table base) field of the directory base register. The Page field is used as an index into the page table determined by the page directory. The offset field is used to select a byte within the page determined by the page table. Each page has a size of 4K bytes. This address translation is illustrated in the following diagram:



Problem 4.8

(a) The allocation of registers is shown in the left-hand side of the following diagram when the total number of registers N is 40.



(b) If $N = 72$, the registers can be organized into 4 windows as shown in the right-hand side of the diagram. Note that in both figures, the eight globally shared registers are not shown.

(c) The scalability of SPARC architecture refers to the number of register windows that can vary with different SPARC implementations.

(d) A calling procedure can pass parameters to a subroutine by writing them into the OUT registers which overlap with the IN registers of the subroutine. Likewise, the results obtained by the subroutine can be passed back by leaving them in the OUT registers which are the IN registers of the calling procedure.

Problem 4.9

(a) Two situations may cause pipelines to be underutilized: (i) the instruction latency is longer than one base cycle, and (ii) the combined cycle time is greater than the base cycle.

(b) Dependence among instructions or resource conflicts among instructions can prevent simultaneous execution of instructions.

Problem 4.10

(a) Vector instructions perform identical operations on vectors of length usually much larger than 1. Scalar instructions operate on a number or a pair of numbers at a time.

(b) Suppose the pipeline is composed of k stages and the vector is of length N. The first output is generated in the k-th cycle. Afterward, an additional output is generated in each cycle. The last result comes out of the pipeline in cycle $(N + k - 1)$. Using a base scalar machine, it takes Nk cycles. Thus the speedup is $Nk/(N + k - 1)$.

(c) If m-issue vector processing is employed, each vector is of length N/m. Therefore, the execution time is $(N/m + k - 1)$ cycles. If only parallel issue is used, the execution time is $(N/m)k$. Thus, the speed improvement is

$$\frac{N/m + k - 1}{(N/m)k} = \frac{Nk}{N + m(k-1)}.$$

Problem 4.11

(a) The average cost is

$$c = \frac{c_1 s_1 + c_2 s_2}{s_1 + s_2}.$$

For c to approach c_2, the conditions are $s_2 >> s_1$ and $c_2 s_2 >> c_1 s_1$.

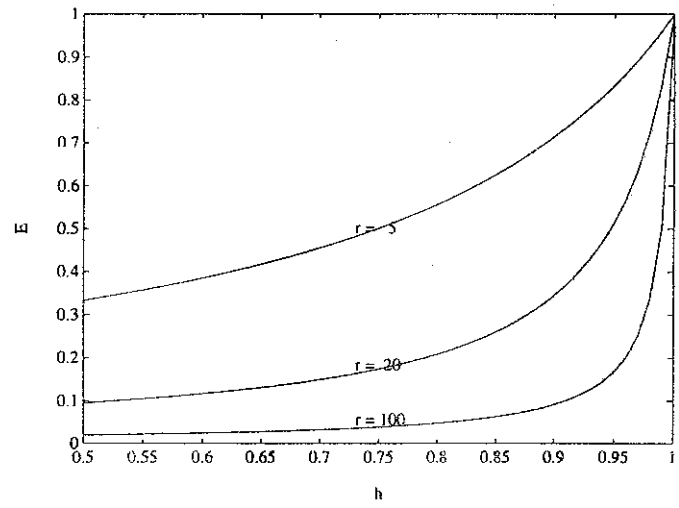
(b) The effective access time is

$$t_a = \sum f_i t_i = h_1 t_1 + (1 - h_1) h_2 t_2 = h t_1 + (1 - h) t_2.$$

(c) If $t_2 = r t_1$. Then $t_a = (h + (1 - h)r)t_1$.

$$E = t_1/t_a = 1/(h + (1 - h)r).$$

(d) The plottings are shown in the following diagram:



(e) If $r = 100$, we have $E = 1/(h + (1 - h) \times 100) > 0.95$. Solving the inequality, we obtain the condition

$$h > \frac{94}{94.05} \approx 99.95\%.$$

Problem 4.12

(a) The average access time is

$$t_a = h_1 t_1 + (1 - h_1) h_2 t_2 = h t_1 + (1 - h) 10 t_1 = (10 - 9h) t_1.$$

If $h = 0.7$, then $t_a = 3.7 t_1 = 74$ ns. If $h = 0.9$, then $t_a = 1.9 t_1 = 38$ ns. If $h = 0.98$, then $t_a = 1.18 t_1 = 23.6$ ns.

(b) The average byte cost is

$$\begin{aligned} c &= \frac{c_1 s_1 + c_2 s_2}{s_1 + s_2} = \frac{20 c_2 s_1 + c_2 \times 4}{s_1 + 4000} \\ &= \frac{20 \times 0.2 s_1 + 0.2 \times 4000}{s_1 + 4000} = \frac{4 s_1 + 800}{s_1 + 4000}. \end{aligned}$$

For $s_1 = 64, 128$, and 256, the average cost is 0.26, 0.32, and 0.43, respectively.

(c) For the three design choices, the product of average access time and average cost is 19.24, 12.16, and 10.15, respectively. Therefore, the third option is the best choice.

Problem 4.13 In a system with private virtual memory, processors communicate with each other through message passing. The latency depends on the interconnection topology and channel bandwidth. As the system grows larger, latency becomes longer. There is no data sharing among the processors. Therefore, data coherence is not a problem. Data may migrate from one node to another. But once a data reaches a destination node, it becomes private data to that node.

In implementation, message passing is facilitated by a pair of commands (send and receive) or through remote procedure calls. Since message passing is essentially I/O operations, it is much more expensive than local memory accesses. As a result, applications which can be partitioned into tasks requiring little interaction with each other are suitable for implementation on such machines.

In a system with globally shared virtual memory space, the private memory associated with individual processors forms a uniform address space visible to all processors. Data can be shared as in a shared-memory multiprocessor. Access latency may vary, depending on the physical memory location of the data. Some systems allow replication of data to reduce the latency. Data can be migrated from one processor to another in pages or other logical units upon demand. If replicated data can be written by multiple processors, data coherence will be an issue which needs to be addressed.

Actual implementation differs. Some systems allow read-only data to be duplicated [Li89]. Others allow replication of writable data as well. In either case, mechanisms must be provided to track the location of each data unit (page or object) and enable fast transportation of data. The complexity can grow rapidly with the system size. In spite of a globally shared address space, access time still varies with the actual location of the data. Therefore, applications with good spatial and temporal localities would be suitable candidates. Applications with less regular communication patterns can also be implemented, although the performance is likely to be degraded. In general, since read

operations are much cheaper than write operations, applications with high read/write ratios are particularly suited.

Problem 4.14

(a) Inclusion property refers to the property that information present in a lower-level memory must be a subset of that in a higher-level memory.

(b) Coherence property requires that copies of an information item be identical throughout the memory hierarchy.

(c) Write-through policy requires that changes made to a data item in a lower level memory be made to the next higher level memory immediately.

(d) Write-back policy postpones the update at level $(i + 1)$ memory until the item is replaced or removed from level i memory.

(e) Paging divides virtual memory and physical memory into pages of fixed sizes to simplify memory management and alleviate fragmentation problem.

(f) Segmentation divides the virtual address space into variable-sized segments. Each segment corresponds to a logical unit. The main purpose of segmentation is to facilitate sharing and protection of information among programs.

Problem 4.15

(a) LRU page replacement gives the following result, with * at the bottom of a column indicating a page fault:

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1
—	1	0	0	2	1	7	6	7	0	1	2	0	3	0	4	5
—	—	1	1	0	2	1	1	6	7	0	1	2	2	3	0	4
—	—	—	—	—	0	2	2	1	6	7	7	1	1	2	3	0
*	*	*			*	*		*		*		*		*	*	*
5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3	
1	5	2	4	5	6	7	6	7	2	4	2	7	7	3	2	
4	1	5	2	4	5	5	5	6	7	7	4	2	2	7	7	
0	4	1	1	2	4	4	4	5	6	6	6	4	4	4	4	
	*			*	*			*	*			*				

In each cycle, the most recently referenced page is brought to the top page frame. As a result, the top row traces out the original page reference stream. The hit ratio using LRU is 16 / 33.

(b) In circular FIFO scheme, the page frames are organized as a circular queue Q. The page frames in are referenced as $Q(I), 0 \leq i \leq 3$. A pointer P is used in conjunction with the usage bits $U(I), 0 \leq i \leq 3$ to decide which page is to be replaced in case of a page fault. Initially the pointer points to the first free page frame $(P = 0)$

and the usage bits are all set to 0 ($U(0:3) = 0$). The behavior rules are specified below:

- Page fault on page J:

  While $(U(P)! = 0)$ do {
  $U(P) = 0$;
  $P = (P + 1) \bmod 4$; }
  $Q(P) = J$;
  $U(P) = 1$;
  $P = (P + 1) \bmod 4$;

- Page hit on a page resident in page frame J:

  $U(J) = 1$;
  if $(P == J)$ $P = (P + 1) \bmod 4$;

The update to the pointer in the event of a page hit is to avoid replacing the page immediately in an ensuing page fault.

Based on the initial conditions and behavior rules, it is easy to write a program to trace the contents of the page frames in response to the reference stream. In the following table, we show the evolution of the arrays and the pointer. An asterisk (*) at the end of a row indicates that a page fault has occurred for a particular page reference.

$Q(0)$	$Q(1)$	$Q(2)$	$Q(3)$	$U(0)$	$U(1)$	$U(2)$	$U(3)$	P	
1	—	—	—	1	0	0	0	1	*
1	0	—	—	1	1	0	0	2	*
1	0	2	—	1	1	1	0	3	*
1	0	2	—	1	1	1	0	3	
1	0	2	—	1	1	1	0	3	
1	0	2	7	1	1	1	1	0	*
6	0	2	7	1	0	0	0	1	*
6	0	2	7	1	0	0	1	1	
6	0	2	7	1	1	0	1	2	
6	0	1	7	1	1	1	1	3	*
6	0	1	2	0	0	0	1	0	*
6	0	1	2	0	1	0	1	0	
3	0	1	2	1	1	0	1	1	*
3	0	1	2	1	1	0	1	2	
3	0	4	2	1	1	1	1	3	*
3	0	4	5	0	0	0	1	0	*
1	0	4	5	1	0	0	1	1	*
1	0	4	5	1	0	0	1	1	
1	2	4	5	1	1	0	1	2	*
1	2	4	5	1	1	1	1	3	
1	2	4	5	1	1	1	1	0	
6	2	4	5	1	0	0	0	1	*
6	7	4	5	1	1	0	0	2	*

6	7	4	5	1	1	0	0	2	
6	7	4	5	1	1	0	0	2	
6	7	2	5	1	1	1	0	3	*
6	7	2	4	1	1	1	1	0	*
6	7	2	4	1	1	1	1	0	
6	7	2	4	1	1	1	1	0	
3	7	2	4	1	0	0	0	1	*
3	7	2	4	1	0	0	0	1	
3	7	2	4	1	0	1	0	1	
3	7	2	4	1	0	1	0	1	

For this particular reference stream, the hit ratio is 16 / 33, which is the same as that for the LRU scheme. However, the contents of the page frames are somewhat different for the two schemes.

Note that different behavior rules have been proposed in literature, which may give rise to slightly different results.

Problem 4.16

(a) Temporal locality refers to the property that recently used data or instructions are likely to be reused in the near future. Spatial locality refers to the property that a process tends to access data or instructions stored in consecutive locations. Sequential locality is related to the observation that the execution order of instructions tends to follow the sequential program order.

(b) Working set is the subset of addresses or pages referenced within a given time window or a given number of most recent references. It approximates the program locality property. Pages in the working set are considered actively used and should reside in main memory. If the window size is large, the resident pages may encompass several locality regions and the size of working set is likely to grow. A large window size should improve hit ratio. But in a multiprogrammed environment, keeping a large number of pages in the memory for each process will exhaust the page frames and cause thrashing. On the other hand, a small window size gives rise to a small working set and may lower hit ratio because actively used pages shift with time.

(c) 90-10 rules: It has been empirically observed that 90% of the execution time is spent on approximately 10% of the code. The rule reflects program locality as a short segment of code gets executed repeatedly and the data accessed tend to be contiguous elements of a large array structure.

Problem 4.17

(a) The effective access time is

$$t_{eff} = t_1 h_1 + t_2(1 - h_1)h_2 = t_1 h_1 + t_2(1 - h_1) = 0.95t_1 + 0.05t_2.$$

(b) The total cost is $c = c_1 s_1 + c_2 s_2$.

(c) 1. We have the following inequality:

$$0.01 \times 512 \times 1024 + 0.0005 \times s_2 \leq 15,000.$$

Therefore s_2 cannot exceed 18.6 Mbytes.

2. The following inequality is obtained:

$$20 \times 0.95 + 0.05 \times t_2 \leq 40.$$

Hence, $t_2 \leq 420$ ns.

Problem 4.18

Attributes	Symbolic processing	Numeric processing
Data objects	Lists, relational databases, scripts, semantic nets, frames, blackboards, objects, production systems.	Integer, floating-point numbers, vectors, matrices.
Common operations	Search, sort, pattern matching, filtering, contexts, partitions, transitive closures, unification, text retrieval, set operations, reasoning.	Add, subtract, multiply, divide, matrix multiplication, matrix-vector multiplication, reduction operations like dot product of vectors, etc.
Memory requirements	Large memory with intensive access pattern. Addressing is often content-based. Locality of reference may not hold.	Great memory demand with intense access. Access pattern usually exhibits high degree of spatial and temporal localities.
Communication patterns	Message traffic varies in size and destination; granularity and format of message units change with applications.	Message traffic and granularity are relatively uniform. Proper mapping can restrict communication to largely between neighboring processors.
Algorithm Properties	Nondeterministic, possibly parallel and distributed computations. Data dependences may be global and irregular in pattern and granularity.	Typically deterministic. Amenable to parallel and distributed computations. Data dependence is mostly local and regular.
Input/Output requirements	Inputs can be graphical and audio as well as from keyboard; access to very large on-line databases.	Large data sets usually exceed memory capacity. Fast I/O is highly desirable.
Architecture Features	Parallel update of large knowledge bases, dynamic load balancing; dynamic memory allocation; hardware-supported garbage collection; stack processor architecture; symbolic processors.	Can be pipelined vector processor, MIMD, or SIMD processors using various memory and interconnection structures. Systolic array is suitable for certain types of computations.

Chapter 5

Bus, Cache, and Shared Memory

Problem 5.1

(a) Maximum bus width is $8 \times 20 = 160$ Mbytes/s.

(b) Memory access time is defined as the time a memory request is received by the memory unit to the time at which all the requested information has been made available at the memory output terminals (Hayes, 1988).

At first, it takes 50 ns (1 bus cycle) for the address to be transmitted to the memory module. After the data is ready on the memory output port, it takes 50 ns to transfer one word to the processor. In the worst case, the four words are accessed one by one separately. The total amount of time for a processor to access one word from the memory is

$$50 \text{ ns} + 100 \text{ ns} + 50\text{ns} = 200 \text{ ns}$$

during which the bus cannot be used by other processors. Thus, the effective bandwidth is

$$8 \text{ bytes}/200 \text{ ns} = 40 \text{ Mbytes/s},$$

which is one-fourth of the maximum bus bandwidth.

If the memory addresses are interleaved, so that access of the four words can be performed simultaneously. It takes 50 ns to transmit the address to the memory module, 100 ns to get the data ready in the latches. Then it takes four bus cycles to transfer the four words to the requesting processor. Therefore, the total time required is $50 + 100 + 4 \times 50 = 350$ ns. Thus, the effective bus bandwidth is

$$4 \times 8 \text{ bytes}/350 \text{ ns} = 91.4 \text{ Mbytes/s}.$$

(c) Any of the arbitration schemes discussed in the text can be used. The decision is based on the desired performance and circuit complexity.

(d) 40 address lines and 64 data lines are needed. In order to limit the total number of signal lines to 104, the address lines can serve as low-order data lines by use of multiplexers. The other lines carry control signals such as bus request, bus grant, reset data sync, address sync, data ack, arbitration, read/write, etc. For a description of the functionality of each signal line, consult the specification of standard buses.

(e) At least 21 slots are needed, one for each processor board, one for each memory board, and one for the bus controller.

Problem 5.2 In a daisy-chained arbitration scheme, there is only a central arbiter. One bus request line is connected to all processors. A single bus grant line is connected to processors in a daisy-chain manner, which means that a processor will acquire the bus only if none of the processors closer to the arbiter requests to use the bus. As a result, the scheme works with a fixed priority based on the proximity of processors to the arbiter.

The advantage is its simplicity in installation; additional processors can be added to an existing chain by sharing the same set of arbitration lines. The simplicity also makes it feasible to install more than one set of the request and grant lines to improve system reliability.

The disadvantage is the violation of fairness principle by fixed priority assignment. Also, it takes a long time for the bus-grant signal to propagate along the chain. As a result, the number of processors that can be effectively supported is small.

If a distributed arbiter scheme is used, each processor has its own arbiter to which a unique arbitration number (AN) is assigned. When two or more processors request to use the bus simultaneously, the arbiters bid by sending ANs to a shared bus request/grant (SBRG) line whose logic selects the maximum among the ANs and leaves it on the line. Subsequently, each arbiter compares its AN with that on the SBRG in parallel. Only the request from the arbiter whose AN matches that on SBRG will be sustained. After the present transaction is finished, the selected processor will seize the bus.

The advantage of using distributed arbiters is the flexibility of implementing various priority schemes and fast arbitration time.

The disadvantage lies in the complex arbitration structure which increases the implementation cost.

Problem 5.3

(a) Assume low-order interleaving is adopted in the organization of the memory modules. Further assume that requests to all memory modules are equally likely. Therefore, the probability of a request by processor P_i to any memory module M_j is p/m, independent of i and j. The probability of no request to M_j from P_i is $1 - p/m$. Hence, the probability of no request from any of the processors is $(1 - p/m)^n$, and the probability of at least one request from any processor is $1 - (1 - p/m)^n$. If $b > m$, all the requests can be satisfied by the bus system. Therefore, the memory

bandwidth is estimated as follows:

$$BW = \sum_{j=1}^{m} (1 - (1 - p/m)^n) = m\,(1 - (1 - p/m)^n)\,.$$

When n is large,

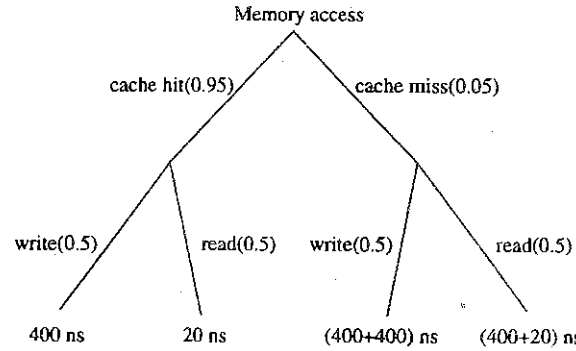
$$(1 - p/m)^n = \left(1 - \frac{(np)/m}{n}\right)^n \approx e^{-np/m}.$$

The memory bandwidth is thus $m(1 - e^{-np/m})$.

(b) Memory bandwidth BW_b is the expected number of busy memory modules or successful memory accesses in a multibus system with b buses. np is the expected number of memory requests generated by the processors. In general, not all memory requests can be satisfied because of conflicts arising from (i) more than one request being made to the same memory module, and (ii) inability of available bus capacity to accommodate all the requests. The presence of conflicts means that not all the expected memory access requests can be successful. Therefore, $BW_b < np$. However, as the authors showed in the paper, through proper choice of the design parameters, most of the memory requests from processors can be satisfied, i.e., $BW_b/np \longrightarrow 1$.

Problem 5.4 For this problem, it is assumed that each cache miss (read or write) leads to the replacement of a block, which can be occupied or empty, in the cache to make room for the missing block.

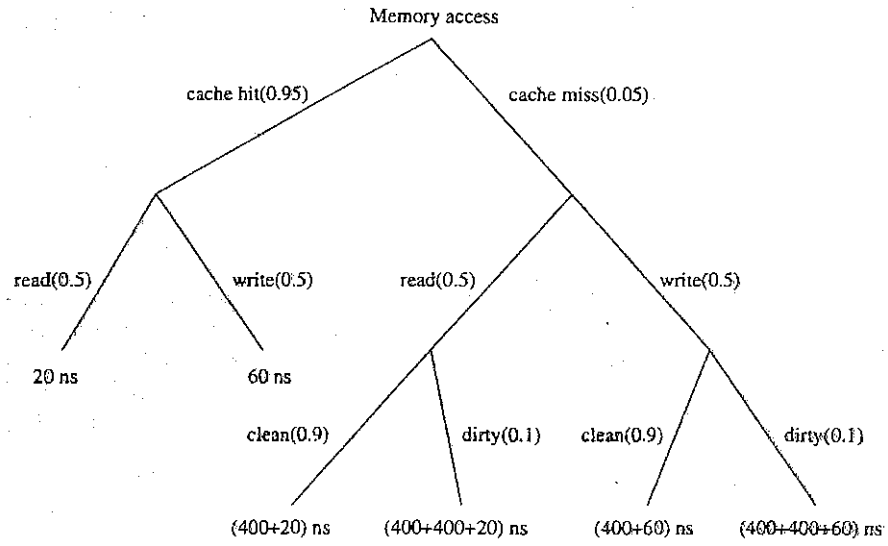
(a) Write-through scheme:



Effective memory access time:

$$\begin{aligned} t_{WT} &= 0.95 \times (0.5 \times (400 + 60) + 0.5 \times 20) + 0.05 \times \\ &\qquad (0.5 \times (400 + 400) + 0.5 \times (400 + 20)) \\ &= 0.95 \times 210 + 0.05 \times 61 \\ &= 230(\text{ns}). \end{aligned}$$

(b) Write-back scheme:



Effective memory access time:

$$\begin{aligned} t_{WB} &= 0.95 \times (0.5 \times 20 + 0.5 \times 60) + 0.05 \times (0.5 \times \\ &\quad (0.9 \times (400 + 20) + 0.1 \times (400 + 400 + 20)) + 0.5 \times \\ &\quad (0.9 \times (400 + 60) + 0.1 \times (400 + 400 + 60))) \\ &= 0.95 \times 40 + 0.05 \times 480 \\ &= 62(\text{ns}). \end{aligned}$$

(c) The memory access time per instruction is

$$\begin{cases} 0.2 \times 230\text{ns} = 46 \text{ ns} & \text{for write-through.} \\ 0.2 \times 62\text{ns} = 12.4 \text{ ns} & \text{for write-back.} \end{cases}$$

Therefore, effective execution time per instruction is

$$\begin{cases} 0.1\ \mu\text{s} + 46 \text{ ns} = 0.146\ \mu\text{s} & \text{for write-through.} \\ 0.1\ \mu\text{s} + 12.4 \text{ ns} = 0.1124\ \mu\text{s} & \text{for write-back.} \end{cases}$$

The effective MIPS rate for each processor is

$$\begin{cases} 1/0.146 = 6.85 & \text{for write-through.} \\ 1/0.1124 = 8.90 & \text{for write-back.} \end{cases}$$

The upper bound of MIPS rate for the multiprocessor system is

$$\begin{cases} 16 \times 6.85 = 109.6 & \text{for write-through.} \\ 16 \times 8.90 = 142.3 & \text{for write-back.} \end{cases}$$

The above upper bounds are obtained by considering only the memory access time. In fact, it is difficult to achieve the upper bounds since the processors may not be fully utilized due to data dependence or resource conflicts among instructions.

Problem 5.5

(a) Low-order interleaving refers to the organization of the memory in which the least significant (low-order) bits of memory address are used to select the memory module and the rest (high-order bits) indicate the offset of a word within the selected module.

(b) When data blocks in a cache are tagged and indexed by physical memory address, it is a physical address cache. In contrast, a virtual address cache does not wait for the physical address to be generated and is accessed by virtual memory address. It offers improved efficiency by overlapping cache access with physical address translation. The disadvantage is potential aliasing problem which entails frequent cache flushing.

(c) In a shared memory, if the update to a memory location is observed by all processors at the same time, then the memory access is atomic. If the update is not necessarily observed by all processors simultaneously, the memory access is nonatomic.

(d) Memory bandwidth is the maximum rate at which data can be transferred to or from the memory. It is determined by memory cycle time, bus width, and memory organization. Effective data transfer rate between memory and processors may be lower due to conflicts. Fault tolerance of the memory system is the capability to continue operation with a lower bandwidth when one or more memory modules fail.

Problem 5.6

(a) In a write-through cache, an update to a cache block causes the corresponding memory block to be updated immediately. In a write-back cache, the update to the memory block is postponed until the cache block is replaced.

(b) Data which are globally shared among several processors and whose values may be updated can be tagged as noncacheable. Instructions, private data, and globally shared readable data are tagged as cacheable. This distinction is an alternative approach used to avoid cache inconsistency problem.

(c) Private caches are those attached to individual processors; shared caches are shared among processors, much like shared memory modules. These two types of caches can coexist in a system. For example, shared cache can be used as second-level cache in a multilevel cache system.

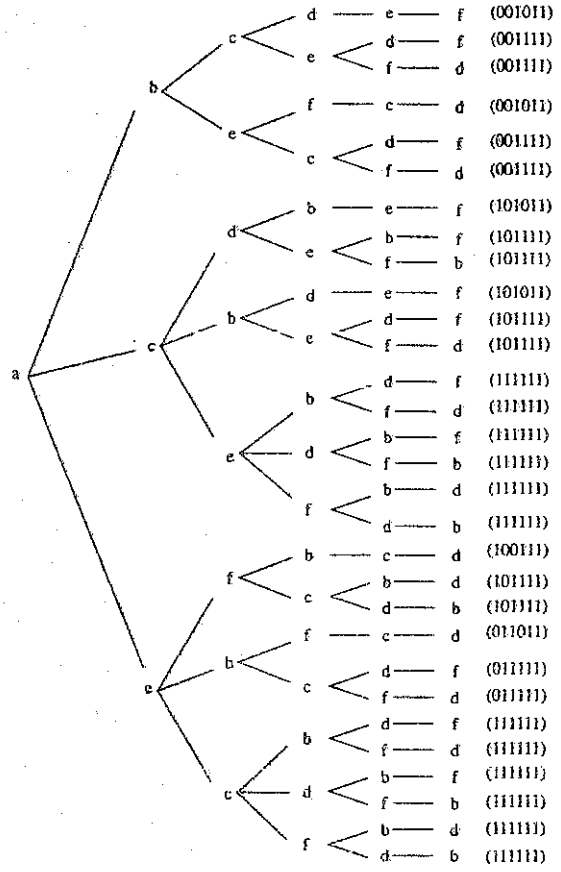
(d) Cache flushing is used to deal with aliasing problem in a virtual address cache. Cache flushing policies determine when flushing should be performed and the level at which flushing takes place (page, segment, context, etc.). These policies are

closely related to operating system design.

(e) Cache hit ratio is affected by factors including cache capacity and block size. A large cache improves hit ratio. For a fixed cache size, there is an optimal block size at which hit ratio peaks. A small block size does not take full advantage of locality properties A large block size, on the other hand, may load unneeded data into the cache. In a set-associative cache organization, the number of sets and set size can also affect hit ratio.

Problem 5.7

(a) In order to preserve individual program orders, the first statement executed must be a or c or e. Consider the case where a is executed first. A tree can be constructed, each branch of which traces out an execution sequence that preserves individual program orders. The tree in the following diagram shows the interleavings of instructions and the corresponding output for each interleaving.



Notice that many different sequences generate the same output. There are 30 possible interleavings of the instructions which preserve individual program orders. A similar tree can be constructed if c or e is the root (the first statement executed). Thus a total of 90 different program-order execution sequences are obtained.

In general, if p processors are used, each executing m_i statements, $i = 1, ..., p$, then the total number of program-order executions is

$$\frac{(\sum_{i=1}^{p} m_i)!}{\prod_{i=1}^{p}(m_i!)}, \tag{5.1}$$

which is identical to the number of possible ways to merge the elements from p sorted lists.

(b) The 720 different execution orders cannot generate all the different combinations. For example, the combination 001100 cannot be generated by any execution order. The 11 pair at the center of the output requires that two of the assignment statements (a, c, and e) be executed before the second output statement. Hence at least two of the three variables already have value 1 before the last output statement is executed, rendering it impossible to generate the last pair of 0s.

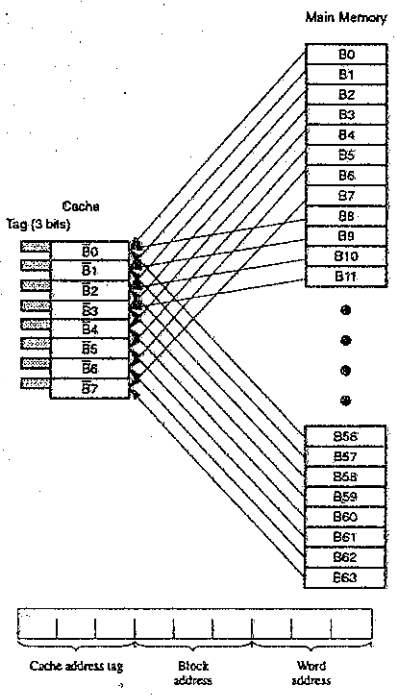
In fact, out of the 64 possible combinations, only 50 can be generated by the six statements executed in any order. Many different execution orders generate identical output sequences, as can be seen in (a).

(c) The sequence 011001 can only be generated by either of the following two execution orders: *cfbeda* and *edafbc*. Note, however, that neither of the two execution orders preserves individual program orders. Therefore, if individual program orders have to be preserved, then the sequence cannot be generated. For a more formal proof, refer to [Dubois88].

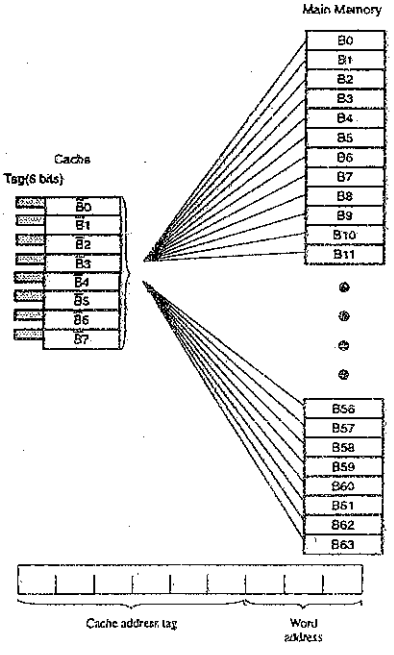
(d) Take as an example the sequence 001100 which can not be generated if memory accesses are atomic. Suppose each processor executes sequentially, but the change to variable values is not immediately observed by all the other processors. Consider the order of execution *abcedf* which does not violate program order of each individual program. First, the pair 00 is generated by b. Then d will produce the pair 11, provided processor 2 has observed the changes made by processors 1 and 3. Finally, processor 3, which has not observed the changes to A and B by the other processors, executes f and prints out 00.

Problem 5.8 The main memory blocks are numbered 0 to 63, the cache block frames are numbered 0 to 63. The mappings are shown in (a) through (d). In each case, the address format and cache tag are also shown.

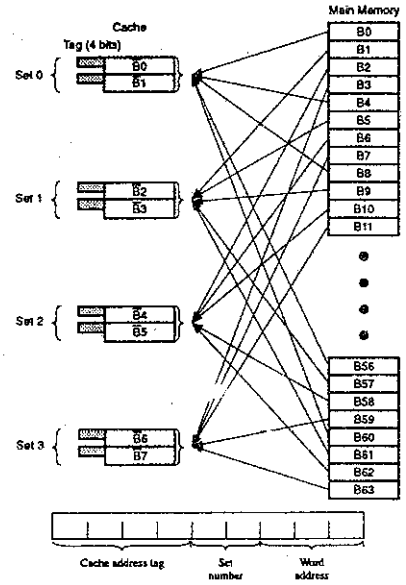
(a) Direct mapping:



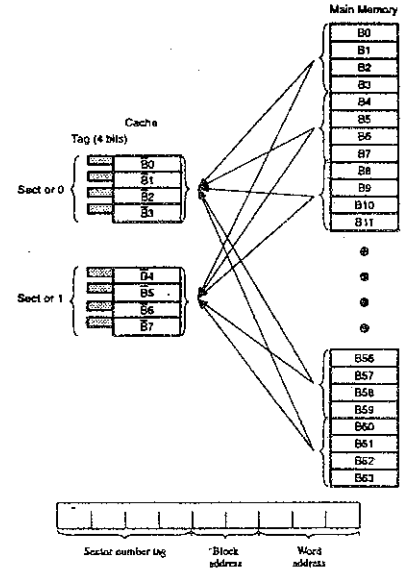
(b) Fully associative mapping:



(c) Set-associative mapping:

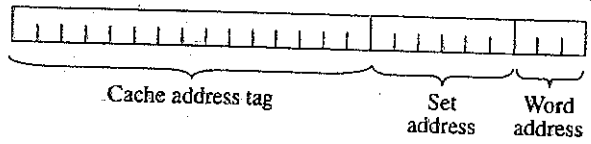


(d) Sector mapping:



Problem 5.9

(a) Each set of the cache consists of $256/8 = 32$ block frames, and the entire cache has $16 \times 1024/256 = 64$ sets. Similarly, the memory contains $1024 \times 1024/8 = 131072$ blocks. Thus, the memory address format is as shown in the following figure:

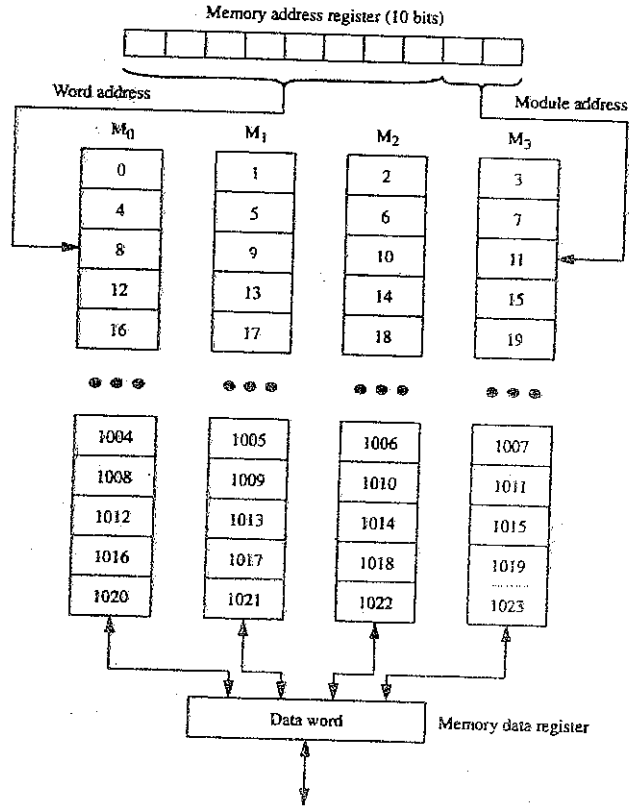


A block B of the main memory is mapped to a block frame in set F of the cache if $F = B \bmod 64$.

(b) The effective memory access time for this memory hierarchy is $50 \times 0.95 + 400 \times (1 - 0.95) = 47.5 + 20 = 67.5$ ns.

Problem 5.10

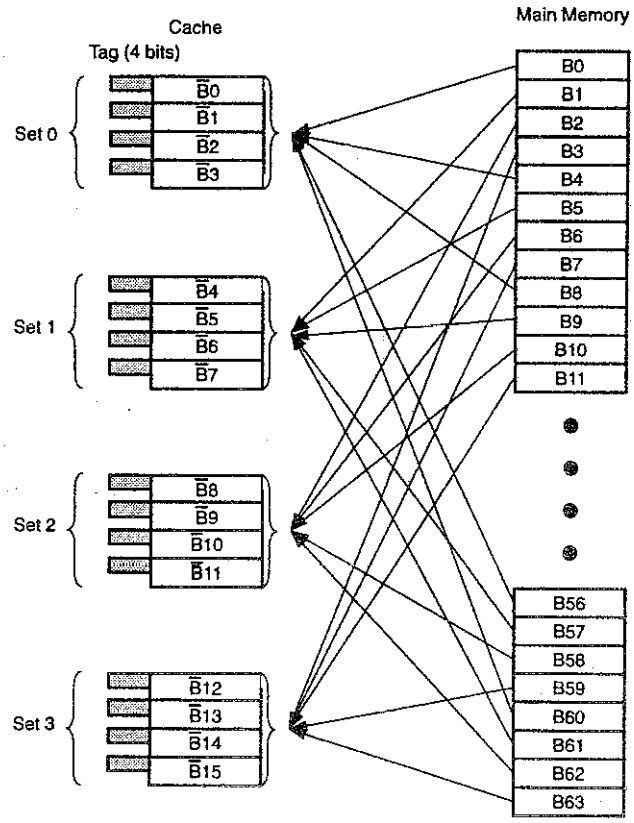
(a) The address assignment is shown in the following diagram:



(b) There are 1024 / 16 = 64 blocks in the main memory, and 256 / 16 = 16 block frames in the cache.

(c) 10 bits are needed to address each word in the main memory: 2 for selecting the memory module and 8 for the offset of a word within the module. 6 bits are required to select a word in the cache: 2 bits to select the set number and 4 bits to select a word within a block. Besides, each block frame needs a 4-bit address tag to determine the block resident in it.

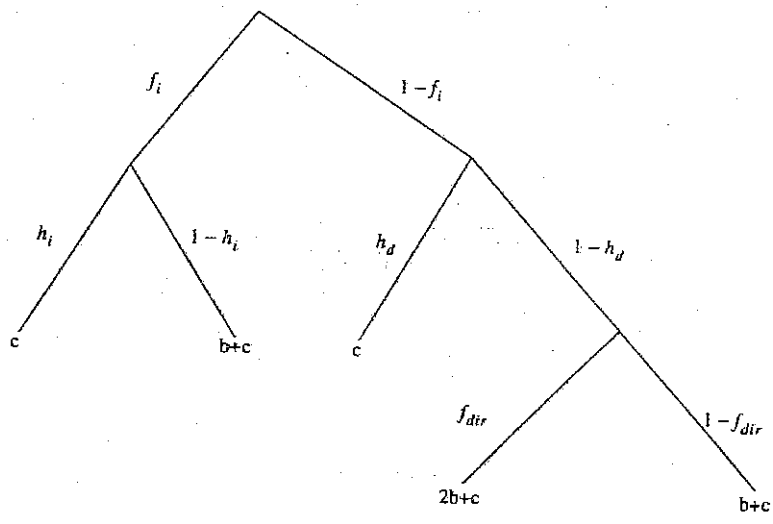
(d) The mapping of memory blocks to the block frames in cache is shown in the following diagram:



After the set in which a memory block can be mapped into is identified, the address tag of the block frames in that set is compared by associative search with the physical memory address to determine if the desired block is in cache.

Problem 5.11

(a) Based on the given data, the following access tree is obtained:



We have the following expression for the average access time:

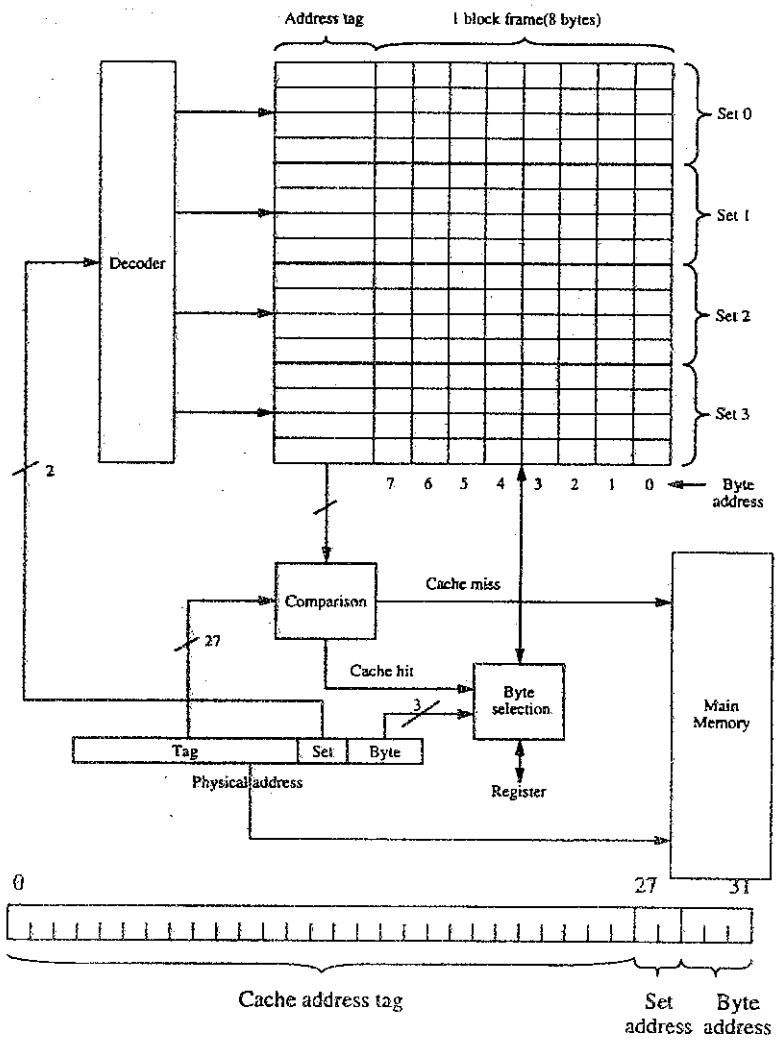
$$t_a = f_i(h_i c + (1-h_i)(b+c)) + (1-f_i)(h_d c + (1-h_d)((b+c)(1-f_{dir}) + (2b+c)f_{dir})).$$

(b) If the extra time taken by invalidation propagation is taken into account, the average access time is

$$t'_a = t_a + (1-f_i)f_{inv}i.$$

Problem 5.12

(a) Cache organization and the relation between physical address and cache address are shown in the following diagrams.



(b) $(000010AF)_{16} = (00000000000000000001000010\underline{10}1111)_2$. From the address mapping shown in the above diagram, it is clear that the address can be assigned to any block frame in set 1.

(c) Bits 27 and 28 in $(FFFF7Axy)_{16}$ must be 0 and 1, respectively, in order for the address to be mapped to the same set as $(000010AF)_{16}$. In other words, the least significant bit of x must be 0 and the most significant bit of y must be 1. The other bits can be either 0 or 1. Therefore, x can be any of the hexadecimal digits $\{0, 2, 4, 6, 8, A, C, E\}$, and y can be any element of $\{8, 9, A, B, C, D, E, F\}$.

Problem 5.13

(a) The effective CPI for each processor can be computed as

$$\text{CPI} = mt + \frac{1}{x} = \frac{1 + mtx}{x}.$$

Therefore, the total MIPS rate of a system with p processors is

$$\text{MIPS} = \frac{p}{\text{CPI}} = \frac{px}{1 + mtx}.$$

(b) Using the expression derived in (a), we obtain the following equation:

$$\frac{32x}{1 + 0.4x} = 56.$$

The equation is solved to give $x = 35/6 = 5.83$ in MIPS.

(c) Substituting the given performance data into the equation in (a), the following MIPS rate is obtained:

$$\frac{32 \times 2}{1 + 1.6 \times 1 \times 2} = \frac{64}{4.2} = 15.24 \text{ MIPS}.$$

Problem 5.14

(a) The effective access time for each memory access is

$$t_a = f_i(1 - h_i)t_m + f_d(1 - h_d)t_m.$$

The CPI in μs can be estimated as

$$\text{CPI} = mt_a + \frac{1}{x} + \alpha t_s = \frac{x(mt_a + t_s) + 1}{x}.$$

The effective MIPS of the entire system is thus

$$\text{MIPS} = \frac{p}{\text{CPI}} = \frac{px}{x(mt_a + t_s) + 1}.$$

(b) Using the data given, we have the following values:

$$t_a = 0.5 \times (1 - 0.95) \times 0.5 + 0.5 \times (1 - 0.7) \times 0.5 = 0.0875$$

$$\text{CPI} = 0.4 \times 0.0875 + \frac{1}{5} + 0.05 \times 5 = 0.485.$$

And finally,

$$\frac{p}{0.485} = 25.$$

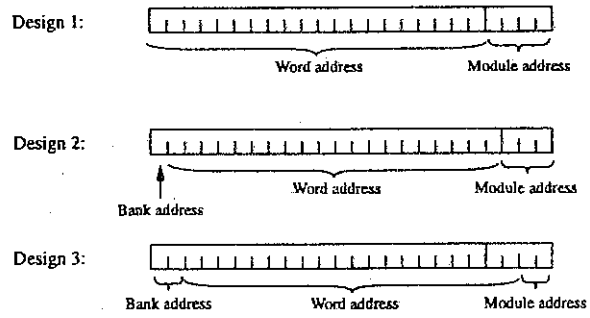
Hence, the number of processors needed is $p = 13$.

(c) The cost of the cache is $4.7 \times 16 \times (32 + 64) = 7219.2$. Hence, the total amount of money allowed for the shared memory is 17781.8, and the memory capacity in Mbytes is

$$C_m = \frac{17781.8}{0.4 \times 1024} \approx 43.4.$$

Problem 5.15

(a) The address formats are shown in the following diagrams for the different design choices:



(b) In case one memory module fails, the memory bandwidth is as follows:

- Design 1: 0.
- Design 2: 8 words per access.
- Design 3: 12 words per access.

(c) In a fault-free situation, Design 1 offers the highest memory bandwidth in the case of vector access. But the entire memory system can be crippled by a single memory module failure. The other two designs offer more graceful degradation in case of module failures, although the bandwidth is not as high as Design 1 under fault-free conditions.

Problem 5.16

(a) All strides except multiples of 17: 80M words per second; strides of multiples of 17: 20M words per second.

(b) All strides except multiples of 4: 80M words per second; strides of multiples of 8: 20M words per second; strides of multiples of 4 but not 8: 40M words per second.

Problem 5.17

(a) Using the formula in Eq. 5.1 of Problem 5.7, there are 20 execution interleaving orders that preserve individual program orders. Trees similar to that given in Problem 5.7 can be constructed. These possible interleaving orders are: *abcdef*, *abdcef*, *abdecf*, *abdefc*, *adbcef*, *adbecf*, *adbefc*, *adebcf*, *adebfc*, *adefbc*, *dabcef*,

dabecf, *dabcfe*, *daebcf*, *daebfc*, *daefbc*, *deabcf*, *deabfc*, *deafbc*, *defabc*.

(b) If program order is preserved and atomic memory accesses are assumed, the following 4-tuple output combinations can be obtained: 0111, 1011, and 1111.

(c) Suppose program order is preserved and nonatomic memory accesses are assumed. Then before c is executed, A has been set to 1 by a. Similarly, C is set to 1 before it is printed by f. Because of nonatomic memory accesses, the value of D in c and that of B in f are uncertain. Therefore the output can be 1xx1 or x11x, depending on the instruction interleaving. Here the don't care bit x can be either 0 or 1. Possible combinations are 1001, 1011, 1101, 1111, 0110, 0111, and 1110. (Pattern 1111 appears in both cases and is shown only once).

Problem 5.18

(a) Hardware complexity and implementation cost is reflected in the mechanism to determine whether a given block is in cache after the block address has been decided. Direct mapping has the lowest cost, since a simple modulus operation is sufficient. Fully associative mapping has the highest cost, since an associative search on all block frames is needed. The relative cost for set-associative and sector mappings depends on the implementation. In set-associative mapping, associative search is needed within each set; in sector mapping, it is needed to determine the sector. For a fixed cache size, the size of each set or sector will make a difference in cost.

(b) In direct mapping, block replacement is rigid and trivial. The other schemes allow similar flexibility in the design of replacement algorithms. For instance, all the replacement algorithms discussed in the text can be implemented with any of the three mappings. In the case of fully associative mapping, the algorithms are applied to the entire cache. In set-associative or sector mapping, only a subset of the cache block frames are examined in the application of the replacement algorithms.

(c) Effects of block mapping policy on the hit ratio:

- Direct mapping: Hit ratio is strongly affected by the reference pattern. If the reference pattern leads to uniform distribution of working set in the cache, hit ratio will be high. But if two or more blocks mapped to the same block frame are referenced alternately, the hit ratio will drop sharply.
- Fully associative mapping: Hit ratio is essentially independent of the reference pattern. Hit ratio should be high except in the rare case of anomalous lack of localities in references.
- Set-associative mapping: On the average, hit ratio should be higher than direct mapping and lower than fully associative mapping. Thrashing is still possible, but with a lower probability than direct mapping.
- Sector mapping: Hit ratio is sensitive to the reference pattern. Because of the mapping scheme adopted, when a block in a sector is replaced, the other blocks in the same sector are invalidated, which effectively reduces the number of valid blocks resident in the cache. This is likely to have an

adverse effect on the hit ratio.

(d)

- For the effect of block size on the cycle count and hit ratio, see the discussion in pages 236–238 and Fig. 5.14 in the text.
- Set number and associativity: For a fixed cache size, the two parameters are inversely proportional to each other. When the cache number is small, it behaves more like a fully associative cache. When the number of sets is large, its behavior is close to that of direct mapping and the hit ratio is expected to become lower. The actual performance is dependent on the characteristics of application programs.
- Cache size: With a larger cache, more data and instructions can be held in the cache, which improves both hit ratio and cycle count.

Problem 5.19

(a) A memory manager performs several functions:

- It keeps track of the memory space being used by individual processes and their IDs.
- It determines which processes to be loaded into memory when memory space is freed.
- It allocates and deallocates memory space as needed.

(b) Suppose a new block needs to be brought into memory. In nonpreemptive allocation, the incoming block can only be placed in a free memory block. In a preemptive allocation scheme, the incoming block is allowed to be placed in a block currently occupied by another process. Nonpreemptive scheme is easier to implement but preemptive scheme can make better use of memory space.

(c) In a swapping system, an entire process (instructions and data) is swapped between main memory and disk. In other words, a process is either resident in memory or forced out of it in entirety. Examples are PDP-11 and early UNIX systems.

(d) In a demand paging system, individual pages rather than entire processes can be swapped between main memory and disks independently. A page is brought into memory only when it is demanded. Demand paging has been implemented in recent releases of UNIX system.

(e) Hybrid memory systems use a combination of swapping and demand paging in managing the memory system. Examples include VAX/VMS and UNIX System V.

Problem 5.20

(a) Lamport's definition of sequential consistency (SC) gets rid of the concept of a global clock and relies solely on the ordering of events. The concepts of program

order and memory order form the foundation of various memory consistency models developed subsequently. The conditions given by Dubois et al. are sufficient but not necessary conditions for implementing SC. The definition is centered around the abstract notion of memory operations in one processor being "performed with respect to other processors". Sindhu et al.'s definition is more formal. A set of axioms based on the mathematical notions of total and partial ordering are used to rigorously specify the behavior of memory systems that satisfy SC. It also defines an atomic swap operation for the implementation of test-and-set which is used to guarantee mutually exclusive entry to critical sections. The similarity among the three SC models is the total ordering of memory events and the obedience of program order within each processor.

(b) The DSB model of weak consistency (WC) imposes SC on synchronization operations only. Other store and load operations are allowed to proceed without waiting for the completion of one another as required by SC model. This allows a higher degree of parallelism to be realized. TSO model imposes program order only on store-store (write after write) operations. Load operations do not have to be visible to the shared memory provided they can be satisfied by a corresponding store operation in the write buffer. TSO model also allows a load operation to bypass write operations.

(c) PSO is derived from TSO by distinguishing store operations performed by an individual processor. In TSO, all the store operations in a processor have to be carried out in program order. But in PSO, only two types of store operations need to be performed in program order: (1) Store operations explicitly separated by a *store barrier* (Stbar) in the program; (2) Store operations performed to the same memory location. In other words, stores which are to different memory locations and are not separated by Stbars are allowed to be executed out of program order. As such, the write buffer in each processor is no longer a FIFO queue. This is similar to DSB weak consistency model and is likely to increase parallelism. The drawback is that the programmer has to determine where strict program order has to be followed and inserts Stbars in those places.

Chapter 6

Pipelining and Superscalar Techniques

Problem 6.1

(a)

$$\text{Speedup} = \frac{nk}{k+(n-1)} = \frac{15000 \times 5}{5+(15000-1)} = \frac{75000}{15004} = 4.9986.$$

(b)

$$\text{Efficiency} = n/[k+(n-1)] = 15000/15004 = 0.9997.$$

$$\text{Throughput} = nf/[k+(n-1)] = 15000 \times 25 \times 10^6 \text{ (instructions/s)}/15004 = 24.99 \text{ MIPS}.$$

Problem 6.2

(a) The clock frequency of DEC Alpha is 150MHz. Comparing it with the 25MHz in a base machine, the superpipeline degree is 6. Alpha issues two instructions every cycle. Therefore, its superscalar degree is 2.

(b) Alpha has a huge virtual address space. Virtual addresses are 64-bit long. Alpha provides instructions for synchronization and cache coherence. This makes it suitable for building multiprocessor systems. However, the scalability of multiprocessor systems is lower than that of multicomputer systems.

Problem 6.3

(a) The superpipelined structure has extra startup overhead and higher branch penalty. See the original paper by Jouppi and Wall (1989).

(b) Under steady state, a superpipelined machine of degree n and a superscalar machine of degree n both can execute n instructions simultaneously. The superpipelined machine outputs one result every $1/n$ clock cycle, while the superscalar machine outputs n results every n clock cycles.

Problem 6.4 The performance cost ratio can be expressed as

$$PCR = \frac{1}{(t/k + d)(c + kh)}.$$

Maximizing PCR is the same as minimizing its inverse. Let k_0 be the optimal number of pipeline stages. Then

$$\frac{\partial}{\partial k}\left(\frac{1}{PCR}\right)\Big|_{k_0} = 0.$$

whence,

$$-\frac{t}{k_0^2}(c + k_0 h) + (\frac{t}{k_0} + d)h = 0.$$

After some simplification, we get

$$\frac{ct}{k_0^2} = dh,$$

and

$$k_0 = \sqrt{\frac{ct}{dh}}.$$

Note

$$\frac{\partial^2}{\partial k^2}\left(\frac{1}{PCR}\right)\Big|_{k_0} = \frac{2tc}{k_0^3} > 0.$$

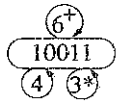
Therefore, at k_0, $1/PCR$ is minimum and PCR is maximum.

Problem 6.5 Lower bound of MAL = the maximum number of checkmarks in any row of the reservation table. Upper bound of MAL = the number of 1's in the initial collision vector plus 1. Detailed proof can be found in the paper by Shar (1972).

Problem 6.6

(a) Forbidden latencies: 1, 2, and 5. Initial collision vector: (10011).

(b) State transition diagram:



(c) MAL = 3

(d) Throughput = $\frac{1}{3\tau}$ = 16.67 million operations per second (MOPS).

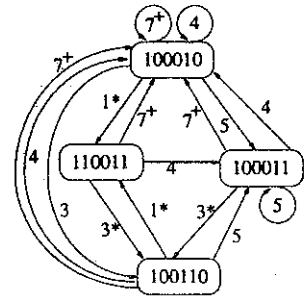
(e) Lower bound of MAL = 2. The optimal latency is not achieved.

Problem 6.7

(a) Reservation table:

	1	2	3	4	5	6	7
S1	X						X
S2		X		X			
S3			X				
S4				X		X	
D					X		

(b) State transition diagram:



(c) Simple cycles: (4), (5), (7), (3,1), (3,4), (3,5,4), (3,5,7), (1,7), (5,4), (5,7), (3,7), (1,3,4), (1,3,5,4), (1,3,5,7), (1,3,7), (1,4,3), (1,4,4), (1,4,7), (5,3,4), (5,3,7), (5,3,1,7)
Greedy cycle: (1,3)

(d)

$$\text{MAL} = \frac{1+3}{2} = 2$$

(e)

$$\text{Throughput} = \frac{1}{2\tau}$$

Problem 6.8

(a) We can complete the computation in $N+11$ clock cycles by the following sequence:

- cycle 1: Compute $A_1 + 0$. Feed A_1 to X and 0 to Y. Connect X and Y to the inputs of the adder
- cycle 2: Compute $A_2 + 0$. Feed A_2 to X and 0 to Y.
- cycle 3: Compute $A_3 + 0$. Feed A_3 to X and 0 to Y.
- cycle 4: Compute $A_4 + 0$. Feed A_4 to X and 0 to Y.
- cycle 5: Compute $A_1 + A_5$. Switch the lower switch to feed Z to the lower input of S1 from now on, and feed A_5 to the upper input.
- cycle 6: Compute $A_2 + A_6$. Feed A_6 to the upper input of S1.
- cycle 7: Compute $A_3 + A_7$. Feed A_7 to the upper input of S1.
- cycle 8: Compute $A_4 + A_8$. Feed A_8 to the upper input of S1.
- cycle 9: Compute $A_1 + A_5 + A_9$. Feed A_9 to the upper input of S1.
- cycle 10: Compute $A_2 + A_6 + A_{10}$. Feed A_{10} to the upper input of S1.
- cycle 11: Compute $A_3 + A_7 + A_{11}$. Feed A_{11} to the upper input of S1.
- cycle 12: Compute $A_4 + A_8 + A_{12}$. Feed A_{12} to the upper input of S1.
- .....
- cycle $N-3$: Compute $A_1 + A_5 + A_9 + ... + A_{N-3}$. Feed A_{N-3} to the upper input of S1.
- cycle $N-2$: Compute $A_2 + A_6 + A_{10} + ... + A_{N-2}$. Feed A_{N-2} to the upper input of S1.
- cycle $N-1$: Compute $A_3 + A_7 + A_{11} + ... + A_{N-1}$. Feed A_{N-1} to the upper input of S1.
- cycle N: Compute $A_4 + A_8 + A_{12} + ... + A_N$. Feed A_N to the upper input of S1.
- cycle $N+1$: Store Z $(= A_1 + A_5 + ... + A_{N-3})$ to R and switch the upper switch to input R to the upper input of S1 from now on.
- cycle $N+2$: Compute $A_1 + A_5 + A_9 + ... + A_{N-3} + A_2 + A_6 + A_{10} + ... + A_{N-2}$.
- cycle $N+3$: Store Z $(= A_3 + A_7 + ... + A_{N-1})$ to R.
- cycle $N+4$: Compute $A_3 + A_7 + A_{11} + ... + A_{N-1} + A_4 + A_8 + A_{12} + ... + A_N$.
- cycle $N+5$:
- cycle $N+6$: Store Z $(= A_1 + A_2 + A_5 + A_6 + ... + A_{N-2} + A_{N-1})$ to R.
- cycle $N+7$:
- cycle $N+8$: Compute $A_1 + A_5 + A_9 + ... + A_{N-3} + A_2 + A_6 + A_{10} + ... + A_{N-2} + A_3 + A_7 + A_{11} + ... + A_{N-1} + A_4 + A_8 + A_{12} + ... + A_N$.
- cycle $N+9$:
- cycle $N+10$:
- cycle $N+11$:
- cycle $N+12$: Result output from Z which is the sum of all elements of A.

(b) The N values are fed sequentially to a nonpipelined adder. Therefore, Nk cycles are needed. The speedup is

$$S_4(N) = \frac{N \times k}{N + 11}.$$

For $N = 64$ and $k = 4$

$$S_4(64) = \frac{64 \times 4}{64 + 11} = 3.41.$$

Among the $N+11$ cycles, 8 cycles ($N+1, N+3, N+5, N+6$, $N+7, N+9, N+10$, and $N + 11$) issue useless instructions. Therefore, $N + 3$ useful add instructions are performed. The efficiency is

$$\eta_4(N) = \frac{N + 3}{N + 11}.$$

$$\eta_4(64) = 67/75 = 0.89.$$

(c)

$$S_4(\infty) = 4.$$

$$\eta_4(\infty) = 1.$$

(d)

$$S_4(N_{1/2}) = S_4(\infty)/2.$$

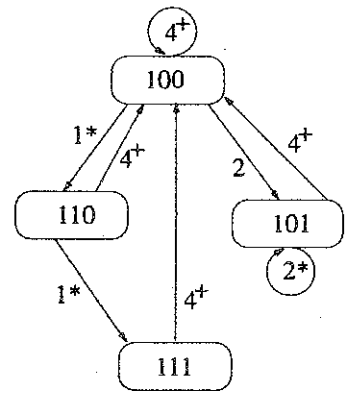
$$\frac{4(N_{1/2})}{N_{1/2} + 11} = 2.$$

$$N_{1/2} = 11.$$

Problem 6.9

(a) Forbidden latency: 3; collision vector: (100).

(b) Static transition diagram is shown below:



(c) Simple cycles: (2), (4), (1,4), (1,1,4), and (2,4); greedy cycles: (2) and (1,1,4)

(d) Optimal constant latency cycle: (2) ; MAL = 2.

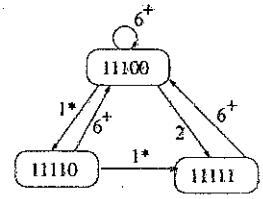
(e)

$$\text{Throughput} = \frac{1}{2 \times 20 \text{ ns}} = 25 \text{ MOPS}.$$

Problem 6.10

(a) Forbidden latencies: 3, 4, 5 ; collision vector: (11100).

(b) State transition diagram is shown below:



(c) Simple cycles: (1,1,6), (2,6), (6), and (1,6).

(d) Greedy cycle: (1,1,6).

(e) MAL = 1 + 1 + 6/3 = 2.67.

(f) minimum allowed constant cycle: (6).

(g) Maximum throughput =

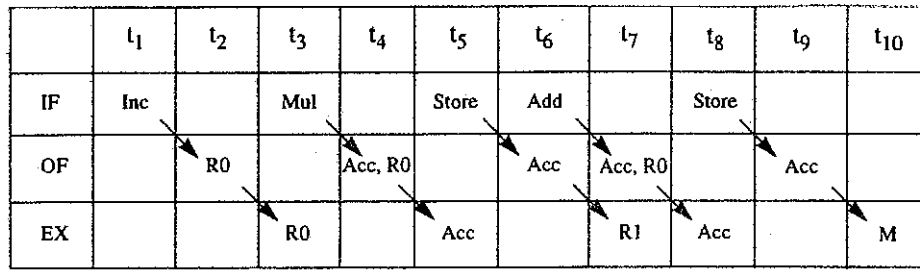
$$\frac{1}{\text{MAL} \times \tau} = 3/(8\tau).$$

(h) $1/(6\tau)$.

Problem 6.11 The three pipeline stages are referred to as IF, OF, and EX for instruction fetch, operand fetch, and execution, respectively. The following diagram shows the sequence of execution:

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
IF	Inc	Mul	Store	Add	Store		
OF		R0	Acc, R0	Acc	Acc, R0	Acc	
EX			R0	Acc	R1	Acc	M

At t_3, $O(I_1) \cap I(I_2) = \{R0\} \longrightarrow$ RAW hazard.
At t_4, $O(I_2) \cap I(I_3) = \{Acc\} \longrightarrow$ RAW hazard.
At t_6, $O(I_4) \cap I(I_5) = \{Acc\} \longrightarrow$ RAW hazard.
The following shows a scheduling which avoids the hazard conditions:



Problem 6.12

(a) For the given value ranges of m and n, we know that $mn(N-1) \geq N-1 \geq N-m$. Now, Eq. 6.32 can be rewritten as

$$S(m,n) = \frac{mn(N-1)+mnk}{(N-m)+mnk}.$$

From elementary algebra, we know that the right hand side of the above equation will attain the largest value when the term mnk is smallest. As a result, the value of k should be 1 in order to maximize $S(m,n)$.

(b) Instructional level parallelism limits the growth of superscalar degree.

(c) The multiphase clocking technique limits the growth of superpipeline degree.

Problem 6.13

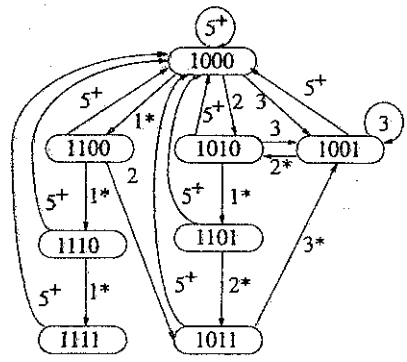
- **Solution 1**

(a) Reservation table:

	1	2	3	4	5	6
S1	X				X	
S2		X				X
S3			X			
S4				X		

(b) Forbidden latency: 4. Collision vector: (1000).

(c) State transition diagram:



(d) Simple cycles: (1,5), (1,1,5), (1,1,1,5), (1,2,5), (1,2,3,5), (1,2,3,2,5), (1,2,3,2,1,5), (2,5), (2,1,5), (2,1,2,5), (2,1,2,3,5), (2,3,5), (3,5), (3,2,5), (3,2,1,5), (,3,2,1,2,5), (5), (3,2,1,2), and (3).

(e) Greedy cycles: (1,1,1,5) and (1,2,3,2).

(f) MAL $= \dfrac{1+1+1+5}{4} = 2.$

(g) Maximum throughput $= 1/(2\tau)$.

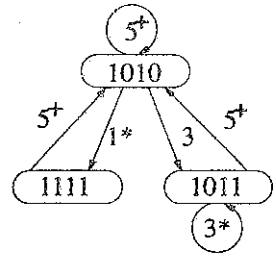
- **Solution 2**

(a) Reservation table:

	1	2	3	4	5	6
S1	X				X	
S2		X		X		X
S3			X		X	
S4				X		

(b) Forbidden latency: 2, 4. Collision vector: (1010).

(c) State transition diagram



(d) Simple cycles: (3), (5), (1,5), and (3,5).

(e) Greedy cycles: (1,5) and (3).

(f) MAL = 3.

(g) Maximum throughput = $1/(3\tau)$.

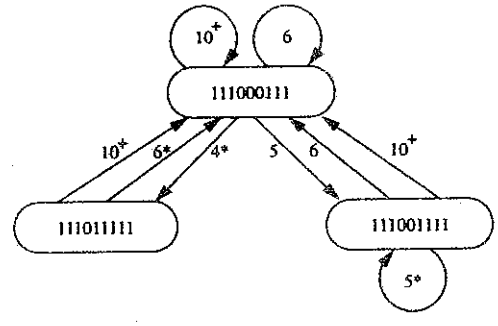
Problem 6.14

(a) The complete reservation table for the composite pipeline is as follows:

	1	2	3	4	5	6	7	8	9	10	11	12
S1	X								X			
S2		X								X		
S3			X	X							X	X
T1					X			X				
T2						X						
T3							X					
U1					X		X					
U2								X				
U3						X						

(b) Forbidden latencies: 8, 1, 7, 9, 3, 2. Collision vector: (111000111).

(c) State transition diagram:



(d) Simple cycles: (5), (6), (10), (4,6), (4,10), (5,6), and (5,10). Greedy cycles: (5) and (4,6).

(e) MAL = 5.

(f) Maximum throughput = $1/(5\tau)$.

Problem 6.15

(a) X needs 400 ($= 4 \times 100$) cycles to execute the program. It takes 16000 ns (400×40 ns). Y needs 104 ($= 5 + [100 - 1]$) cycles to execute the program. It takes 5200 ns.

$$\text{Speedup} = \frac{16000}{5200} = 3.08.$$

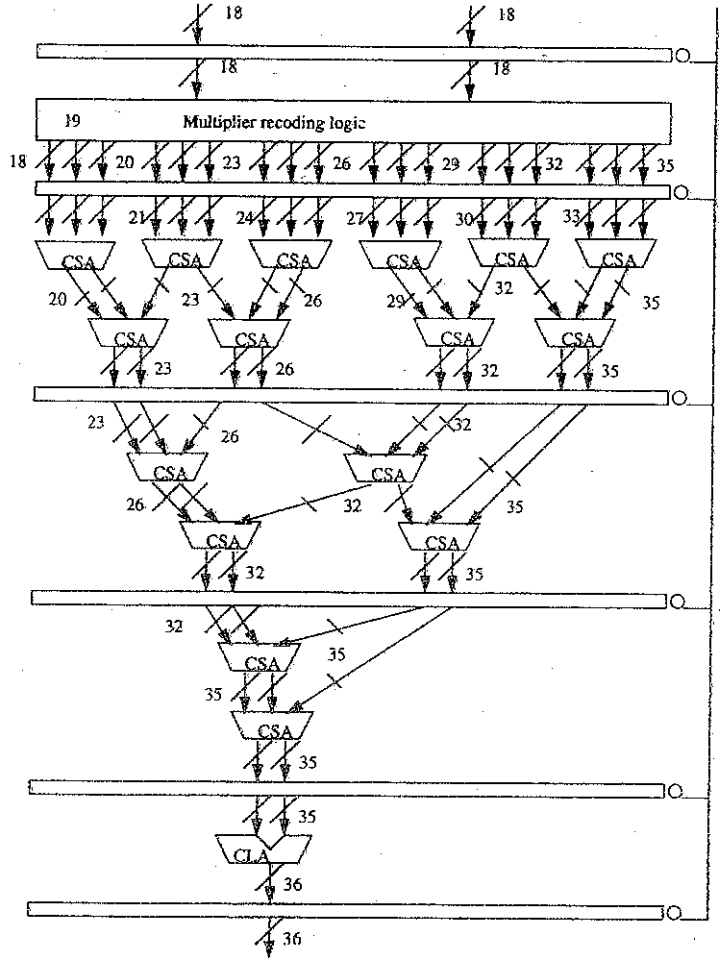
(b) MIPS rates are computed as follows:

$$\text{X}: \frac{100}{16\mu s} = 6.25 \text{ MIPS.}$$

$$\text{Y}: \frac{100}{5.2\mu s} = 19.2 \text{ MIPS.}$$

Problem 6.16

(a) The five-stage multiply pipeline is depicted below:



(b) The maximum clock rate is $\tau = \tau_m + d = 90 + 20 = 110$ ns.

(c) The maximum throughput = 1 / (110 ns) = 9.1 MOPS.

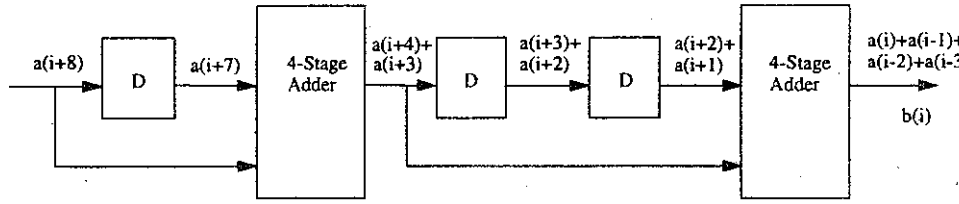
Problem 6.17

(a) 1. Exponent subtract
2. Align
3. Fraction add
4. Normalize

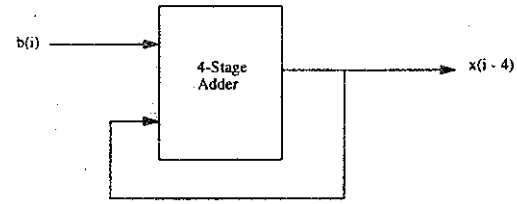
(b) From the solution of Problem 6.8, 111 clock cycles are needed.

Problem 6.18

(a) The composite pipeline:



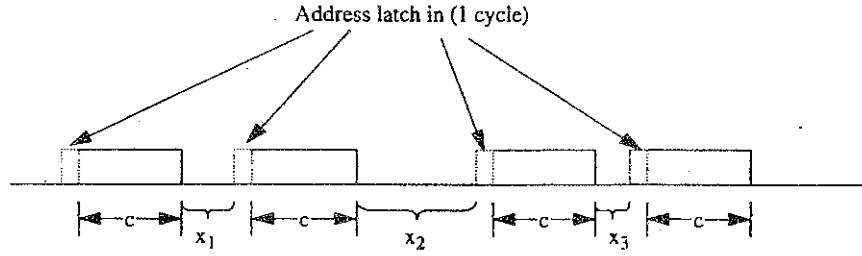
(b) Connection of the third adder:



Chapter 7

Multiprocessors and Multicomputers

Problem 7.1 Since requests are continually generated by the processors during each cycle, the bus never becomes idle. The memory requests are uniformly distributed across all the modules. Thus the probability that a memory module is selected is $1/m$ in each cycle. After a memory module is selected, it will be busy for c cycles. Then it may be reselected or become idle for a number of cycles. The behavior of a memory module can be described by the following diagram:



The idle or waiting period can be modeled by a random variable x which follows a geometric distribution. The mean value of x can be computed as follows:

$$\bar{x} = \sum_{i=0}^{\infty} (1 - \frac{1}{m})^i (\frac{1}{m}) i$$

Let

$$f(z) = \sum_{i=0}^{\infty} (1 - \frac{1}{m})^i (\frac{1}{m}) z^i.$$

From the theory of z-transform, we know that

$$\bar{x} = \left. \frac{df(z)}{dz} \right|_{z=1}.$$

Let $p = 1/m$ and $q = 1 - p$. Then

$$\begin{aligned} f(z) &= p\sum_{i=0}^{\infty}(qz)^i \\ &= \frac{p}{1-qz} \qquad \text{for } qz < 1, \end{aligned}$$

whence,

$$\bar{x} = f'(1) = \frac{pq}{(1-q)^2} = \frac{pq}{p^2} = \frac{q}{p} = m - 1.$$

(a) The memory bandwidth delivered by the bus configuration is

$$B = \frac{mc}{(c+m)\tau}.$$

Using the given values for the variables, we obtain

$$B = \frac{16 \times 4}{(16+4) \times 10 \times 10^{-9}} = 32 \times 10^6 \text{words / sec.}$$

(b) The fraction of time during which a memory module is busy is

$$\frac{c}{c+1+m-1} = \frac{c}{c+m}.$$

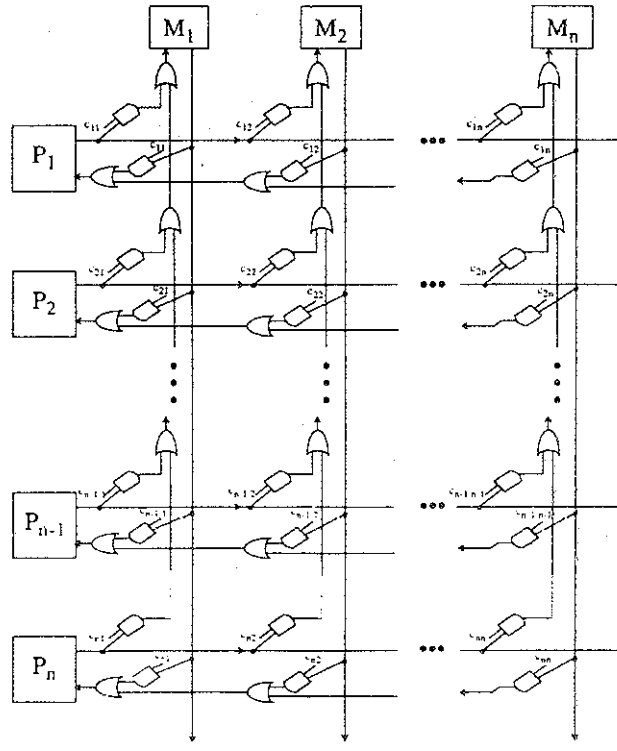
Since there are m independent memory modules, the utilization of all the memory modules is

$$\frac{4 \times 16}{4+16} = 3.2.$$

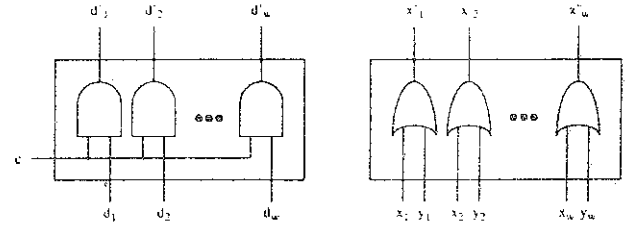
requests per memory cycle.

Problem 7.2

(a) The following diagram shows the crossbar network which connects n processors to n memory modules.

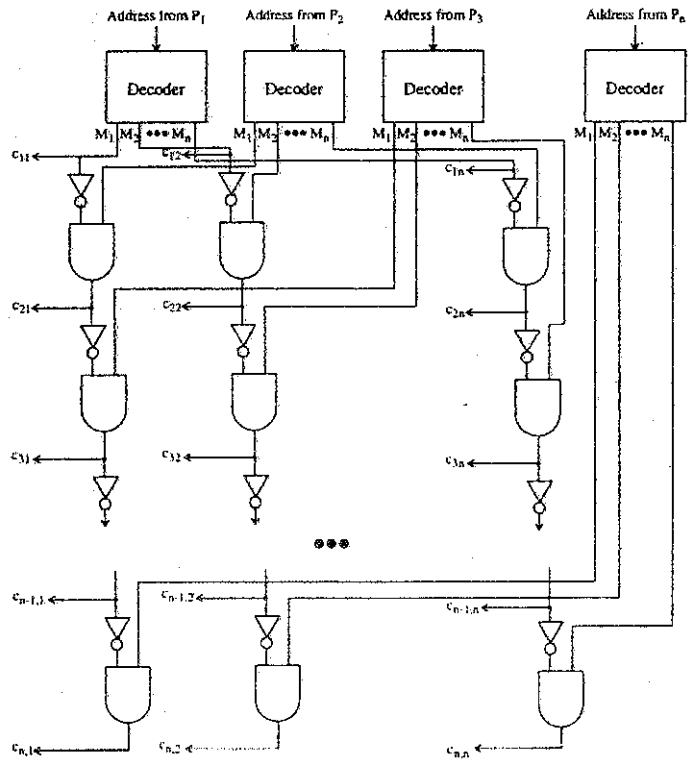


The complexity of the crossbar network can be estimated as follows. At each crosspoint, there are 2 AND gates and 2 OR gates. But in the last row (processor n) and last column (memory module n), we do not need OR gates for the read/write operation. Therefore, there are $2n^2$ AND gates and $2n^2 - 2n$ OR gates. In practice, each AND or OR gate in the diagram consists of w two-input AND or OR gates as shown in the following diagram.



In total, the number of two-input AND gates is $2n^2w$, and the number of two-input OR gates is $(2n^2 - 2n)w$.

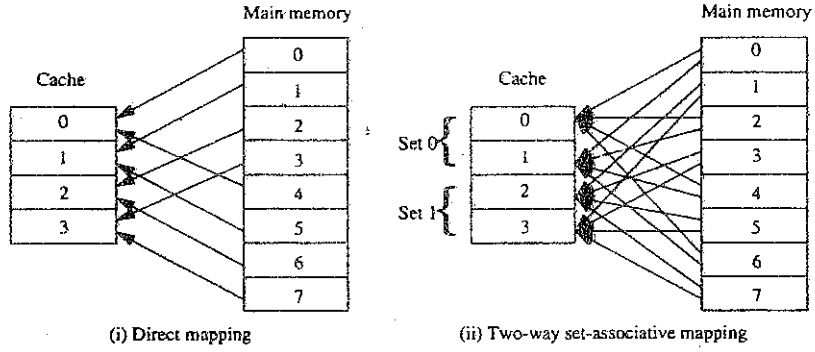
(b) The schematic diagram of the arbiter is shown below:



In case of conflicting requests to access the same memory module, the arbiter will grant priority to the processor with the smallest number. There are $(n-1)$ two-input AND gates along each column, leading to a total requirement of $n(n-1)$ such gates.

Problem 7.3

(a) The mappings are shown in the following figure.



(i) Direct mapping

(ii) Two-way set-associative mapping

(b) The results are shown in the following two tables; the first table corresponds to direct mapping and the second two-way set-associative mapping. In each table, an arrow connecting the same block numbers indicates that the corresponding access takes more than one cycle due to read/write misses or bus contention. In any case, at most 3 cycles are required to complete an access in the case of a read/write miss coupled with bus contention. The subscript associated with a block indicates the state of that block (R for read-only, and W for read-write.)

	cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	block trace	0	→	0	0	0	1	→	1	1	4	→	4	3	→	3	3	5	→	5	5	5
	frame 0	—	0_R	0_W	0_W	0_W	0_R	0_R	—	4_R	4_R	4_R	4_R	4_R	4_R	4_R	4_R	4_R
	frame 1	—	—	—	—	—	1_W	1_W	1_W	1_W	1_W	1_W	1_W	—	—	5_R	5_W	5_W
P1	frame 2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	frame 3	—	—	—	—	—	—	—	—	—	—	3_R	3_R	3_R	3_R	3_R	3_R	3_R
	cache miss	*				*			*		*			*				
	bus in use	*		*		*			*		*			*	*		*	

	block trace	2	→	2	2	0	→	0	0	7	→	7	5	→	5	5	5	7	7	0
	frame 0	—	—	—	—	—	—	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R
	frame 1	—	—	—	—	—	—	—	—	—	—	—	5_R	5_R	5_R	5_R	—	—		
P2	frame 2	—	—	2_R	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W
	frame 3	—	—	—	—	—	—	—	—	—	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W
	cache miss	*				*				*		*								
	bus in use		*		*		*			*		*				*				

	cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	block trace	0	→	0	0	0	1	→	1	1	4	→	4	3	→	3	3	5	→	5	5	5
	frame 0	—	0_R	0_W	0_W	0_W	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R
	frame 1	—	—	—	—	—	—	—	—	4_R	4_R	4_R	4_R	4_R	4_R	4_R	4_R	4_R
P1	frame 2	—	—	—	—	—	1_W	1_W	1_W	1_W	1_W	1_W	1_W	—	—	5_R	5_W	5_W
	frame 3	—	—	—	—	—	—	—	—	—	—	3_R	3_R	3_R	3_R	3_R	3_R	3_R
	cache miss	*				*			*		*			*				
	bus in use	*		*		*			*		*			*	*		*	

	block trace	2	→	2	2	0	→	0	0	7	→	7	5	→	5	5	5	7	7	0
	frame 0	—	—	2_R	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W	2_W
	frame 1	—	—	—	—	—	—	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R	0_R
P2	frame 2	—	—	—	—	—	—	—	—	—	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W	7_W
	frame 3	—	—	—	—	—	—	—	—	—	—	—	5_R	5_R	5_R	5_R	—	—		
	cache miss	*				*				*		*								
	bus in use		*		*		*			*		*				*				

For the given page reference patterns, the hit ratio is 6/11 for P1 and 7/11 for P2 with either cache organization. The major difference is the contents of block frames in the caches due to different ways of mapping between memory and cache. As can be seen, a memory block can possibly reside in more cache block frames with the set-associative organization, which generally improves hit ratio.

Problem 7.4 A valid schedule must satisfy the following two conditions:

(1) It does not violate the dependence relation specified in the diagram.
(2) It does not cause resource conflicts. In other words, no processor or memory module can be allocated to more than one segment at a time.

There are many possible ways to schedule the program segments without violating the above conditions. A systematic approach is to use list scheduling as discussed in [Adam74]. The heuristics is to identify a critical path based on the memory latency and schedule segments on the critical path first under the data dependence and resource constraints.

P_3 is demanded the most among all processors and is busy for 20 time steps. Moreover, none of the memory modules are requested more than 20 times, as shown in the following table:

Memory module	Access frequency
M_1	12
M_2	18
M_3	15
M_4	12
M_5	8
M_6	5
Total	70

Therefore, the demand on P_3 precludes the possibility of a scheduling that can finish the task in less than 20 time steps. In the following table, we show one possible scheduling. Each cell of the table contains a pair of numbers x, y with x corresponding to the instruction and y the memory module requested. According to condition (2) in the above, the value of y should be different in cells on the same row.

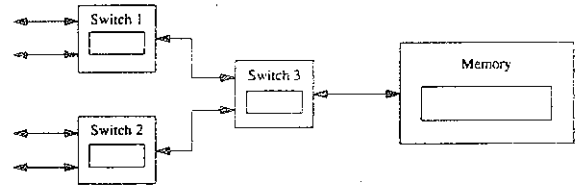
Time step	P_1	P_2	P_3	P_4
1	11,2		3,3	11,1
2		2,2	11,5	3,3
3		11,4	2,2	1,1
4	2,1	9,3	1,5	
5	1,1		9,4	2,2
6		10,3	6,1	9,4
7		8,2	10,4	6,3
8		6,2	8,3	10,4
9	10,1	4,3	7,6	7,5
10		13,1	4,2	4,4
11	4,5	14,4	5,2	13,6
12	16,2	5,6	14,5	14,4
13	5,1	16,2	20,3	16,4
14	17,1	20,3	16,2	20,4
15		12,2	12,6	12,5
16	22,3	18,2	18,5	
17	21,2	22,1	15,3	21,4
18	15,3		19,2	22,6
19	19,2	15,3		19,1
20		19,2	23,5	23,3
21	23,1	23,2	24,3	24,4

Based on the scheduling, 21 time steps are required and the average memory bandwidth is $70/21 = 3.33$ words per time step. There are other schedules that yield an identical bandwidth. For instance, the pair 11,4 on row 3 can be moved to the same column on row 1.

Note that in the above scheduling, an additional condition is satisfied; that is, once a segment is scheduled, it is continuously scheduled in consecutive time steps until completion without being disrupted. If the condition is relaxed, it is possible to obtain a scheduling with a total of 20 time steps.

Problem 7.5

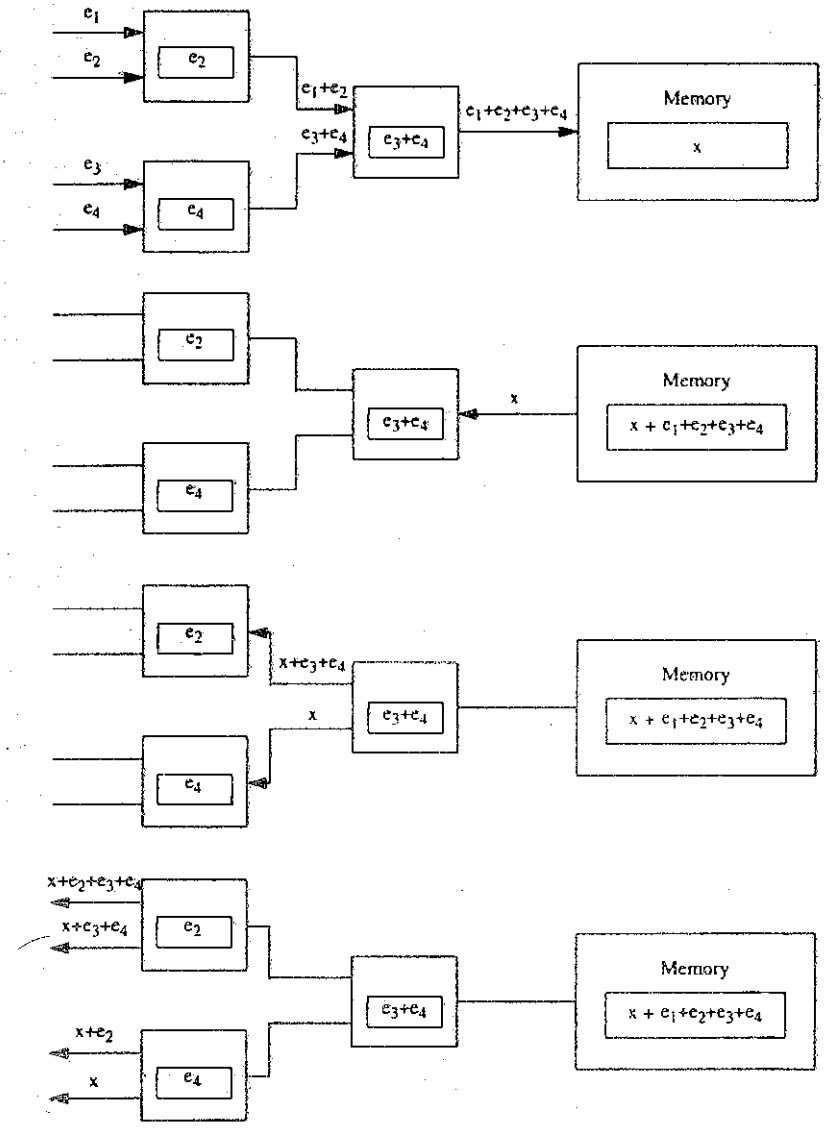
(a) Three switch cells are needed to combine the inputs as shown in the following diagram:



Each switch box is able to perform the following functions (see [Stone90], p. 348):

- Match the addresses on upper and lower inputs.
- Add the two increments.
- Save one of the increments.
- Match a returning value for Fetch&Add to the saved increment.

(b) The following figure shows a possible scenario of the data transfer between processors and a certain memory module (hot spot).



The final content of the memory location is the same regardless of the serialization of the increments e_i. But the increment saved in the buffer of each switch cell can be different, resulting in different values being returned to different processors.

Problem 7.6 The m-way shuffle of n objects, where $n = mk$ is defined by the following mapping:

$$f : i \longrightarrow \left(mi + \left\lfloor \frac{i}{k} \right\rfloor \right) \bmod n$$

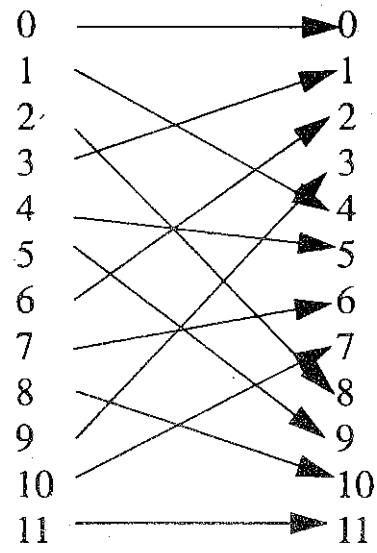
for $i = 0, 1, 2, ..., n-1$.

In general, i can be written as $i = \ell_1 k + \ell_2$ with $0 \leq \ell_2 < k$. The m-way shuffle can be alternatively defined by the following expression:

$$f(\ell_1 k + \ell_2) = \ell_2 m + \ell_1.$$

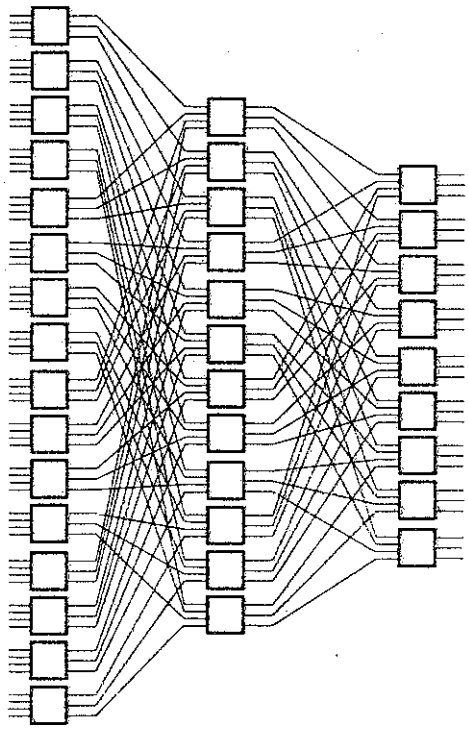
For example, the number $1 = 0 \times k + 1$ is mapped to $1 \times m + 0 = m$, and $n - 1 = mk - 1 = (m-1)k + (k-1)$ is mapped to $m(k-1) + (m-1) = mk - 1 = n - 1$. Similar mapping can be carried out for other numbers. A graphical rendition of the m-way shuffle is illustrated in the next subproblem.

(a) The four-way shuffle of 12 objects is shown in the following diagram:



The 12 objects are divided into four piles of 3 objects each. Then the first object in each pile is selected, followed by the second object in each pile, and so on.

(b) A Delta network with 64 inputs and 27 outputs implemented with 4×3 switching modules is shown in the following diagram:



(c) Suppose the n stages are numbered 1 through n from input side to output side. The number of switch modules needed in the first stage is a^{n-1}, generating a total of $a^{n-1}b$ outputs, which in turn require $a^{n-2}b$ switch modules in the second stage, and so on. In general, $a^{n-i}b^{i-1}$ switch modules are required in stage i. Therefore, the total number of switch modules is

$$\begin{aligned} N &= \sum_{i=1}^{n} a^{n-i}b^{i-1} = a^{n-1}\sum_{i=0}^{n-1}\left(\frac{b}{a}\right)^i \\ &= \begin{cases} \dfrac{a^n - b^n}{a-b} & \text{if } a \neq b \\ na^{n-1} & \text{if } a = b \end{cases} \end{aligned}$$

The interstage connection pattern is a-shuffle. For instance, suppose we apply the formula to the design in part (b). The number of 4×3 modules is $\frac{4^3-3^3}{4-3} = 37$, and the interstage connection pattern is 4-shuffle.

(d) Suppose the destination address d is expressed in base b: $d = (d_{n-1}d_{n-2}...d_1d_0)_b$. Then d_i is used to control the setting of the switch modules in stage $n-i$. Specifically, the connection at the output port of a module is determined by the value of d_i.

(e) Omega networks and Delta networks use the same interstage connection pattern. Omega networks uses only switch modules with the same numbers of inputs and

outputs. Also, in Omega networks, the inputs are shuffled before they are connected to the input ports of the switch modules in the first stage. Thus, an Omega network can be simulated by a Delta network in which $a \times a$ switch modules are used and the inputs are a-shuffled before entering the network.

Problem 7.7

(a) It is convenient to determine the number of legitimate states from the perspective of output terminals of a switch module. A legitimate state is defined as a connection pattern in which an output terminal is connected to exactly one input terminal. Two or more input terminals connected to one output terminal simultaneously cause conflicts and should be prohibited. On the contrary, an input terminal can be connected to more than one output terminal at the same time in broadcast mode.

Consider a $k \times k$ module. The problem can be viewed as the selection of k items from a bucket containing k different items. In this model, broadcast from one input to one or more outputs is equivalent to allowing an item to be selected more than once; i.e., an item is put back into the bucket after it is selected. Clearly, the number of possible selections is k^k. Therefore, the number of legitimate states that can be realized by a $k \times k$ switch module is k^k.

For instance, if $k = 4$, a total of $4^4 = 256$ legitimate states are realizable by each switch module.

(b) Since there exists a unique path from each input terminal to each output terminal, it is possible to implement all permutations with proper setting of the switch boxes. Therefore the total number of permutations achievable in one or more passes through the Omega network is 64!. The number of switch modules needed is $64/2 * \log_2 64 = 192$, each of which can realize 2 permutations. Therefore, the fraction of permutations that can be realized in one pass is

$$2^{192}/64! \approx 4.95 \times 10^{-32}.$$

(c) In case 8×8 switch modules are used, each module can realize $8! = 40320$ permutations. There are two stages in the network with 8 switch modules at each stage, giving a total of 16 modules. Thus the total number of permutations that can be implemented in one pass is 40320^{16}, which is approximately 3.85×10^{-16} of the total number of permutations that can be implemented in one or more passes.

(d) A total of $512/8 \times \log_8 512 = 192$ of 8×8 switch modules are used. Therefore the fraction of permutations that can be implemented in one pass is

$$(8!)^{192}/512!$$

which can be estimated by Sterling's formula.

Problem 7.8 From the conditions given, the following relation is obtained:

$$m = (b\lceil l/P \rceil + 1)k, \qquad \text{where } P = \min(n/k, l)$$

If $P = l$, then $m = (b+1)k$ and the degree of multiprogramming k is equal to $m/(b+1)$. If $P = n/k$, then we have the following equality:

$$m = (b\lceil lk/n \rceil + 1)k.$$

In case n divides lk, the following quadratic equation in k is obtained:

$$blk^2 + nk - mn = 0, \tag{7.1}$$

which can be solved to give

$$k_0 = \frac{-n + \sqrt{n^2 + 4mnbl}}{2bl}.$$

Obviously, the expression $blk^2 + nk - mn$ increases monotonically with k. Hence, if $k > k_0$, the demand for synchronization lines will exceed m. Therefore, the degree of multiprogramming should not exceed k_0.

Problem 7.9 An important property of the multilevel bus/cache architecture is that any memory block which has a copy in the level-1 caches also has a copy in the level-2 cache. This inclusion property makes it possible to use level-2 caches as filters to avoid unnecessary traffic on the buses.

Consider the use of a write-broadcast protocol to maintain cache coherence of the system in Fig. 7.3. When a level-1 cache C_{11} writes to a memory block, the updated value is broadcast on the intracluster bus so that the other caches which have a copy of the memory block will update their data.

The updated value is also propagated up to C_{20}, which updates its copy of the memory block. C_{20} then broadcasts the new value on the intercluster bus. If a copy of the block exists in C_{22} (for instance), its value is updated. By the inclusion property, the memory block is likely to be resident in the cluster underneath C_{22}. Therefore, the data is also passed down to the intracluster bus and level-1 caches which have a copy of the memory block also update their values.

The relative merits of write-invalidate and write-broadcast protocols have been studied extensively, through either simulation or analytic approach [Archibald86, Yang89]. See the discussion in the solution for Problem 7.19 below. Most comparison has been conducted on single-level caches, but the results should be applicable to hierarchical caches. Write-broadcast protocols generally exhibit better performance, although actual performance depends on the memory reference patterns. An advantage of write-invalidate protocols is the relatively simple hardware support required.

Problem 7.10

(a) The general trend in the industry is toward open systems which favor commercially available processors over proprietary design. This helps reduce the cost and shorten the time of development. More effort can be focused on high-level design such as interconnection structure and software development.

(b) Increasing scalability is the main motivation.

(c) To avoid the problems of memory contention and/or cache inconsistency (if private memory or cache is used).

(d) To offer more flexibility in using existing multiprocessor software.

Problem 7.11

(a) A message is the logical unit for internode communication. It is often assembled by an arbitrary number of fixed-length packets. It may have a variable length.
A packet is the basic unit of information transmission which contains the destination address for routing purposes.
A flit (flow control digit) is the smallest unit of information that a queue or channel can accept or refuse.

(b) In a store-and-forward network, the basic unit of information flow is a packet. Each node has a packet buffer. A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes. When a packet reaches an intermediate node, it is first stored in the buffer. Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

(c) In wormhole routing scheme, a flit is the basic unit of information flow. Flit buffers are used in the hardware routers attached to nodes. The transmission from the source node to the destination node is done through a sequence of routers. All the flits in the same packet are transmitted in order as inseparable companions in a pipelined fashion. Only the header flit knows where the packet is going. All the data flits must follow the header flit. Different packets can be interleaved during transmission. However, the flits from different packets cannot be mixed up.

(d) A virtual channel is a logical link between two nodes. It is formed with a flit buffer in the source node, a physical channel between them, and a flit buffer in the receiver node. There are more than one virtual channels between two nodes. However, fewer number of physical channels are time-shared by all the virtual channels.

(e) Buffer deadlocks may occur with store-and-forward routing in which no buffers are provided on the channels. A deadlock situation occurs when there is a circular wait among the nodes and the buffers in the nodes are all full. Channel deadlock can occur with wormhole routing when the channels used by different messages enter a circular wait. Both types of deadlocks are illustrated in Example 7.3.

(f) When two packets reach the same node and they request the same outgoing channel, the cut-through routing scheme uses a packet buffer to temporarily store one of the received packets. When the channel becomes available later, the stored packet will be transmitted then.

(g) When two packets reach the same node and they request the same outgoing channel, the blocking policy blocks the second pocket from advancing. However, the

packet is not abandoned.

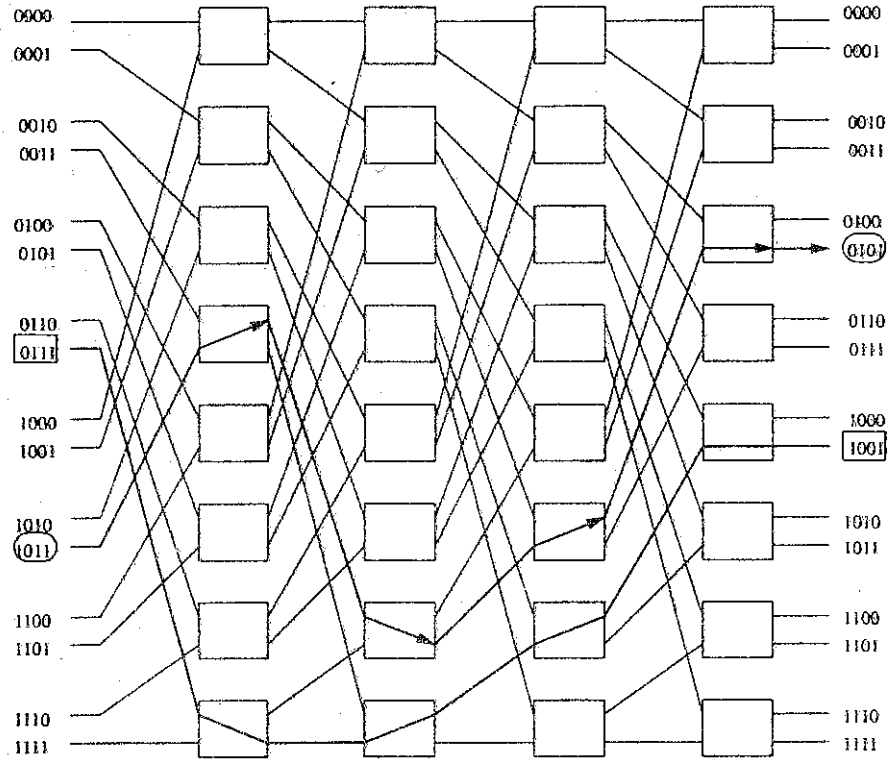
(h) When two packets reach the same node and they request the same outgoing channel, the discard policy simply drops the packet being blocked. Packet retransmission is required when the channel is available later.

(i) In detour flow control, the blocked packet is rerouted to a detour channel. From there, another route may be found to reach the destination node.

(j)
- A virtual network is a network in which all nodes are connected by virtual channels. There are multiple virtual channels between two nodes. Hence, several virtual networks can be formed.
- The nodes in a network can be subdivided into several subsets. The nodes in a subset and their connections form a subnetwork of the original network.

Problem 7.12

(a) A 16 × 16 Omega network using 2 × 2 switches is shown below:



(b) 1011 → 0101 is indicated by → in the above diagram; 0111 → 1001 is indicated by —-. As can be seen, there is no blocking for the two connections.

(c) Each switch box can implement two permutations in one pass (straight or cross). There are $\log_2 16 \times 16/2$ switch boxes. Therefore, the total number of single-pass permutations can be computed as

$$2^{\frac{16}{2} \times \log_2 16} = 2^{32} = 16^8.$$

The total number of permutations is 16!, therefore,

$$\frac{\text{Number of single pass permutations}}{\text{Total number of permutations}} = \frac{16^8}{16!} \approx 2.05 \times 10^{-4}.$$

(d) At most $\log_2 16 = 4$ passes are needed to realize all permutations.

Problem 7.13

(a) A unicast pattern is a one-to-one communication, and a multicast pattern is a one-to-many communication.

(b) A broadcast pattern is a one-to-all communication, and a conference pattern is a many-to-many communication.

(c) The channel traffic at any time instant is indicated by the number of channels used to deliver the message involved.

(d) The communication latency is indicated by the longest packet transmission time involved.

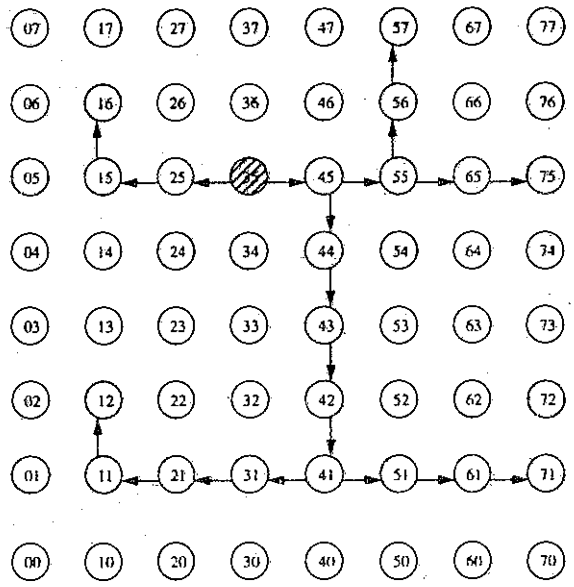
(e) Partitioning of a physical network into several logical subnetworks. In each of the subnetworks, appropriate routing schemes can be used to avoid deadlock.

Problem 7.14

(a) (101101) → (101100) → (101110) → (101010) → (111010) → (011010).

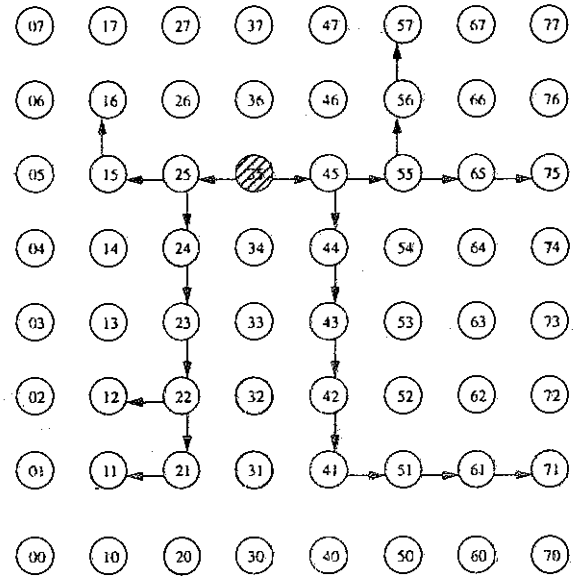
(b) Two optimal routing schemes under different constraints:

- Routing with a minimum number of channels:



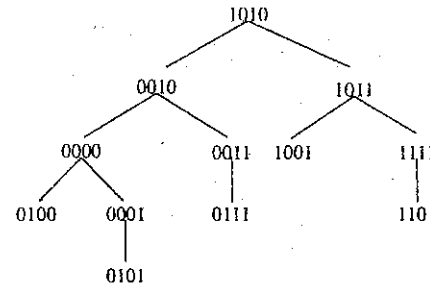
For this routing, traffic = 20, distance = 9.

- Routing with a minimum distance from the source to each destination:

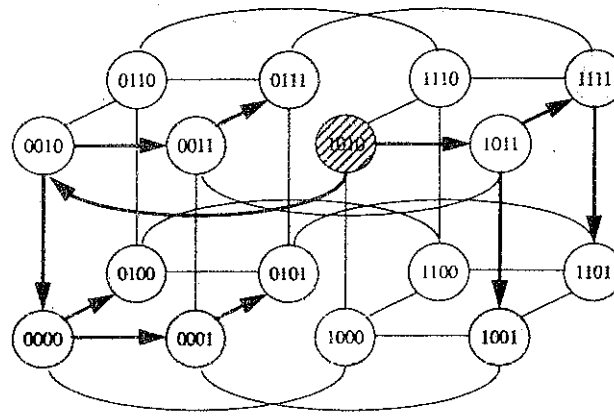


For this routing, traffic = 22, distance = 8.
There are other routes with the same traffic and distance.

(c) The routing is shown in the following tree:



The paths are shown by heavy lines in the following diagram:



Problem 7.15

(a) In a hypercube of dimension n, we denote a node as n_k where k is an n-digit binary number. Node n_k has n output channels, one for each dimension, labeled $c_{0k}, \ldots, c_{(n-1)k}$. The E-cube algorithm routes in increasing order of dimension. A message arriving at node n_k destined for node n_l is routed on channel c_{ik} where i is the position of the least significant bit in which k and l differs. Since messages are routed in order of increasing dimensions, and hence increasing channel subscripts, there are no cycles in the channel dependency graph and E-cube routing is deadlock-free.

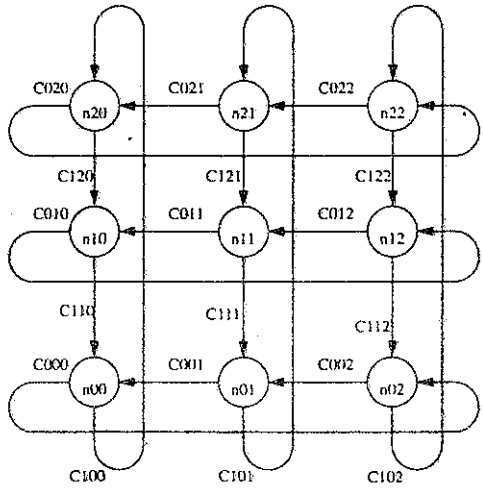
(b) There are four possible X-Y routing patterns corresponding to the east-north, east-south, west-north, and west-south paths chosen. As in the 3×3 mesh shown in Figure 7.37b, we can have two pairs of virtual channels in the Y-dimension. For each of the four routing patterns, no cycle will be formed in the channel dependency graph. Thus, the X-Y routing is deadlock-free.

(c) In a k-ary n-cube, we denote the address of a node by n_j where j is an n-digit radix k number. The ith digit of j represents the node's position in dimension i. For example, the center node in the 3-ary 2-cube below is n_{11}. A channel is

identified by the address of its source node and the dimension it is in. For example, the dimension 0 (horizontal) channel from n_{11} to n_{10} is c_{011}.

To break cycles we divide each channel into an upper and a lower virtual channels. The upper virtual channel of c_{011} is labeled c_{0111}, and the lower virtual channel is labeled c_{0011}. In general, virtual channel subscripts are of the form dvx where d is the dimension, v selects the virtual channel, and x identifies the source node of the channel. To assure that a routing is deadlock free, we restrict it to routing through channels in order of ascending subscripts.

As in the E-cube algorithm we route messages in increasing order of the dimensions, starting with the lowest dimension. In each dimension i, a message is routed in that dimension until it reaches a node whose subscript matches the destination address in the ith position. The message is routed on the upper channel if the ith digit of the destination address is greater than the ith digit of the current node's address. Otherwise, the message is routed on the lower channel. This algorithm routes messages in order of ascending subscripts. Thus, it is deadlock-free.



Problem 7.16

(a) The turn model works by prohibiting a minimum number of turns (change of directions by 90 degrees) to prevent the formation of cycles. With the cycles broken, circular waits are removed and deadlocks are prevented. Formally, routing algorithms developed under this model allow a channel-numbering scheme in which the channels traveled by each packet either increase or decrease monotonically. This type of routing has been shown to be deadlock-free. See also the solution for Problem 7.15.

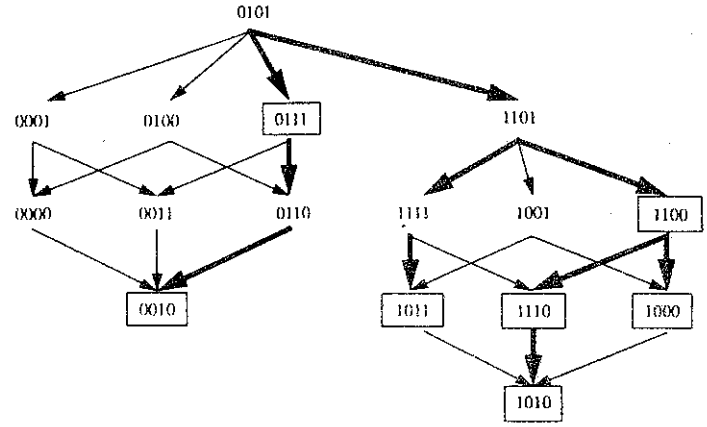
(b) The authors have described three different routing algorithms for use with n-dimensional meshes: all-but-one-negative-first, all-but-one-positive-last, and negative-first. These algorithms specify that a packet should use outgoing channels along certain directions before (or after) the others. As stated in (a), the algorithms

have to be used in conjunction with special channel numbering schemes. For more details, see the paper.

(c) A k-ary n-cube uses a torus connection along each dimension; i.e., each node at the edge of a mesh has a wraparound connection. One way to use the algorithms developed for meshes is to assign to each wraparound channel a number greater (or smaller) than any other channel along that direction in the mesh, depending on the routing algorithm used.

Problem 7.17

(a) In multicast, the objectives are two-fold. One is to send a message to all the destination nodes, and the other is to do so efficiently. A tree can be constructed and used to determine the minimum subtree which covers all the destination nodes. This is illustrated in the following diagram using the multicast pattern in Example 7.8.



The destinations are enclosed in boxes. To cover all the destinations from the source with a minimum number of edges (lowest traffic), the paths indicated by heavy lines are chosen, which are identical to the choice of the greedy algorithm in Example 7.8. The path has a latency of 4 and a traffic of 10. Note that there are other alternatives to some of the nodes/edges selected. For instance, 1001 can be used instead of 1111, and destination 1010 can be reached from 1011 instead of 1110.

(b) The greedy multicast algorithm provides a strategy to deterministically select intermediate nodes (called forward nodes in the paper) between the source and destinations. The selection is based on the distance between the addresses of the source s and a destination d_i, which is the number of 1s in $r_i = s \oplus d_i$, where $\oplus$ stands for bitwise exclusive-OR operation.

The design of the algorithm is such that each intermediate node on the path from s to d_i will reduce the number of 1s in r_i by 1. In fact, the descendant nodes of each intermediate node are chosen according to the number of destination

nodes for which the goal is achieved. Therefore, if initially s and d_i differ in δ bit positions, the message will arrive at d_i in δ steps, which is the minimum possible number of steps on a hypercube.

The authors proved that the greedy algorithm also minimizes network traffic if the number of destinations is 1 or 2, but is slightly inferior to the optimal algorithm when the number is larger than 2.

Problem 7.18 In the write-once protocol, a block may exist in one of four states in a cache:

- Invalid: there is no copy of the block in the cache,
- Valid: an arbitrary number of caches can have this read-only block, and all the copies are identical,
- Reserved: data in the block has been locally modified exactly once since it was brought into the cache and shared memory is updated, and
- Dirty: data in the block has been locally modified more than once since it was brought into the cache and the shared memory is stale.

The write-once protocol is mainly characterized by the introduction of the *Reserved* state. A first-time write to a clean and potentially shared block results in a write-through to memory and it updates the main memory copy as well as the local copy. The local copy becomes *Reserved*, which indicates an exclusive copy in the system and saves subsequent write invalidations.

Each cache has a *buswatcher* which monitors the transactions on the bus. When the bus watcher detects an address on the bus which hits in the local cache with a dirty copy, it intervenes in the bus transaction by asserting the *memorybypass* signal to inhibit the memory from supplying the data. To facilitate rapid access to the address tags and state bit pairs concurrently with accesses to the address tags by the CPU, dual (identical) cache directories are used.

Problem 7.19

There are five states for cached blocks in the Dragon protocol: Invalid, Valid-Exclusive (only cached copy in the system; clean and identical with the memory copy), Shared-Clean, Shared-Dirty (write-back required at replacement) and Dirty (only copy in caches and modified).

The Dragon protocol is a write-broadcast protocol as the Firefly protocol. As long as there exists more than one cached copy, writes are broadcast to other caches. One difference is the updates to shared blocks are also immediately reflected at main memory in the Firefly protocol, while the Dragon protocol introduces the Shared-Dirty state such that memory copy is updated only when the Shared-Dirty copy is replaced. The cache that performed the latest write to the shared block is in the Shared-Dirty state and is responsible for supplying the block on misses in remote caches and for updating main memory on replacement.

In case of write hits on unmodified private blocks the Dragon and the Firefly are able to eliminate unnecessary overhead by changing cache state from Valid-Exclusive to

Dirty without inducing any bus transaction. On the contrary, the write-once protocol requires a single word to be written to main memory.

The distributed write protocols of Dragon and Firefly yields better performance than the write-invalidation of write-once protocol in the handling of shared data. This is because the overhead of distributing written data to all caches having a copy is lower than repeatedly invalidating all other copies and subsequently forcing misses on the next references in those caches where the block was invalidated.

The performance of the Dragon can slightly exceed that of the Firefly protocol because the Firefly broadcasts writes to main memory as well as to other caches. Therefore, the performance of the Firefly may be affected by the long latency of the memory system. But the Dragon gains the performance at the cost of adding one more state Shared-Dirty and it becomes more complex compared to the simplicity of the write-once protocol.

Problem 7.20

(a) When more than one input of a crossbar module wants to use the same output port, the output connection is granted to the input port with the smallest number. In Cedar implementation, there is a priority resolution logic in each output port. An arriving packet waits in the input queue if the output port is already busy or the input is not chosen by the resolution logic. Only when all currently conflicting requests have been resolved will any new request be allowed to enter the arbitration. In this fashion, high-priority input ports will be prevented from starving low-priority ones. In summary, a combination of first-come-first-served queueing principle and a fixed priority based on input port number is used to resolve conflicts. See [Konicek91].

(b) See Fig. 7.10a in the text for a similar connection of a 64×64 network using 8×8 switch modules.

(c) See Fig. 7.10b in the text for a similar connection of a 512×512 network using 8×8 switch modules.

Chapter 8

Multivector and SIMD Computers

Problem 8.1

(a) In the register-to-register architecture, operands and results are retrieved indirectly from the main memory through the use of a large number of vector or scalar registers. In the memory-to memory architecture, source operands, intermediate and final results are retrieved directly from the main memory. More registers are needed in a register-to-register architecture, and higher memory bandwidth is needed in the memory-to-memory architecture.

(b) An SIMD machine with n processors and a pipelined machine with n stages and $1/n$ clock period have the same performance (n results every basic cycle). However, the SIMD machine needs n times of hardware (ALU), and the pipelined machine needs n times of memory bandwidth.

Problem 8.2

(a) The percentage of vector code in a program required to achieve equal utilization of vector and scalar hardware.

(b) The percentage of code in a program which can be vectorized.

(c) A compiler capable of vectorization.

(d) The instructions correspond to the following mappings:

$$f : V_i \rightarrow s_i,$$

or

$$g : V_i \times V_j \rightarrow s_k.$$

(e) A gather instruction fetches the nonzero elements of a sparse vector using indices.

$$f : M \rightarrow V_1 \times V_0.$$

A scatter instruction stores a vector in a sparse vector whose nonzero entries are indexed.

$$f : V_1 \times V_0 \rightarrow M.$$

(f) A sparse matrix is a matrix in which most of the entries are zero. A masking instruction uses a mask vector to compress or expand a vector to a shorter or longer index vector, respectively, corresponding to the following mapping:

$$f : V_0 \times V_m \rightarrow V_1.$$

Problem 8.3

(a) The low-order interleaved memory can be rearranged to allow *simultaneous access*, or *S-access*, as illustrated in Fig. 8.1a. In this case, all memory modules are accessed simultaneously in a synchronized manner. Again the high-order $(n-a)$ bits select the same offset word from each module.

At the end of each memory cycle (Fig. 8.1b), $m = 2^a$ consecutive words are latched in the data buffers simultaneously. The low-order a bits are then used to multiplex the m words out, one per minor cycle. If the minor cycle (τ) is chosen to be $1/m$ of the major memory cycle (θ), then it takes two memory cycles to access m consecutive words.

However, if the access phase of the last access is overlapped with the fetch phase of the current access (Fig. 8.1b), effectively m words take only one memory cycle to access. If the stride is greater than 1, then the throughput decreases, roughly proportionally to the stride.

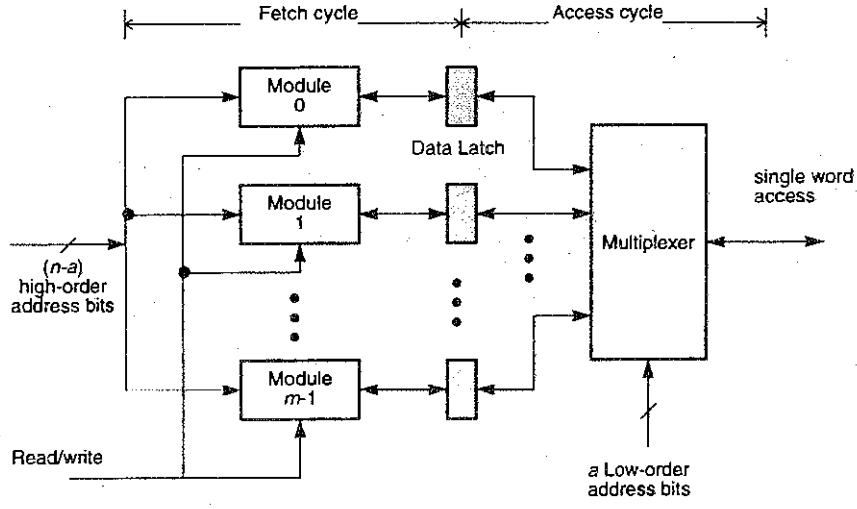
(b) The m-way low-order interleaved memory structure shown in Figs. 8.2a and 8.3 allows m memory words to be accessed concurrently in an overlapped manner. This *concurrent access* has been called *C-access* as illustrated in Fig. 8.3b.

The access cycles in different memory modules are staggered. The low-order a bits select the modules, and the high-order b bits select the word within each module, where $m = 2^a$ and $a + b = n$ is the address length.

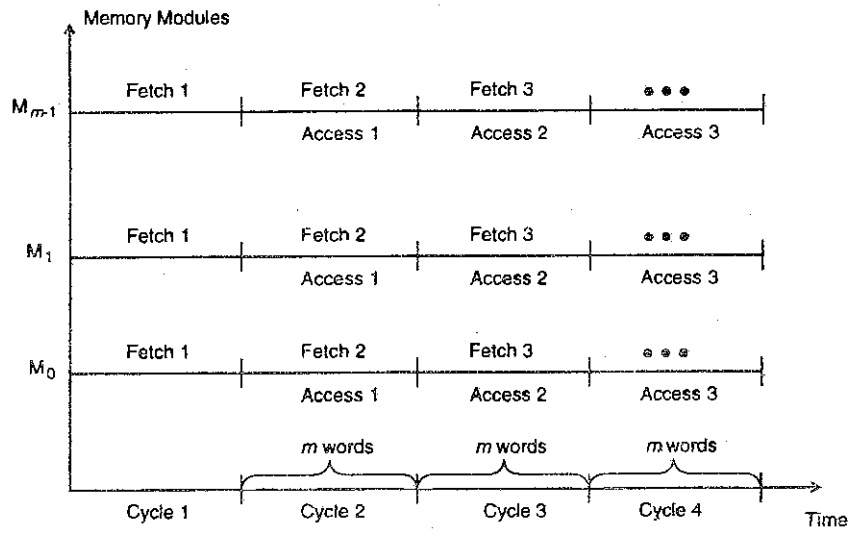
To access a vector with a stride of 1, successive addresses are latched in the address buffer at the rate of one per cycle. Effectively it takes m minor cycles to fetch m words, which equals one (major) memory cycle (θ), as shown in Fig. 8.3b.

If the stride is 2, the successive accesses must be separated by two minor cycles in order to avoid access conflicts. This reduces the memory throughput by one-half. If the stride is 3, there is no module conflict and the maximum throughput (m words) results. In general, C-access will yield the maximum throughput of m words per memory cycle if the stride is relatively prime to m, the number of interleaved memory modules.

(c) A memory organization in which the C-access and S-access are combined is called *C/S-access*. This scheme is shown in Fig. 8.4, where n access buses are used with m interleaved memory modules attached to each bus. The m modules on each bus are m-way interleaved to allow C-access. The n buses operate in parallel to allow

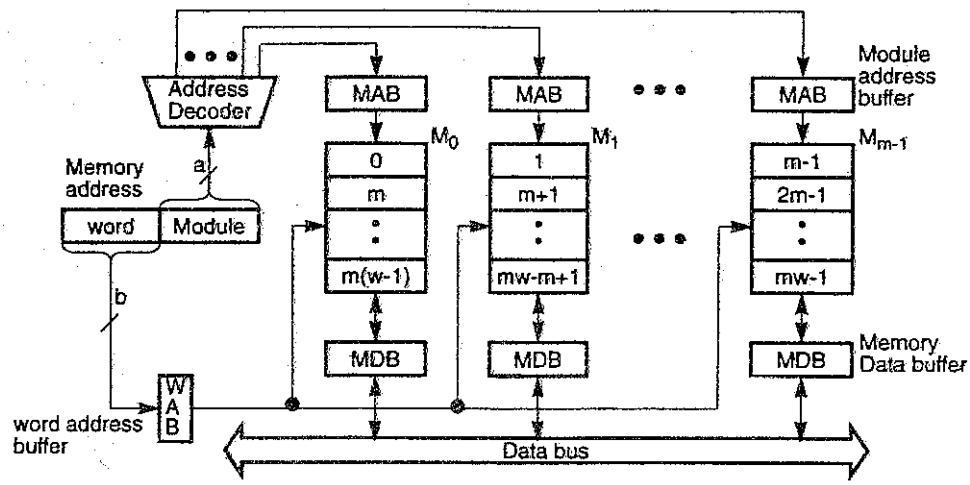


(a) S-access organization for an m-way interleaved memory

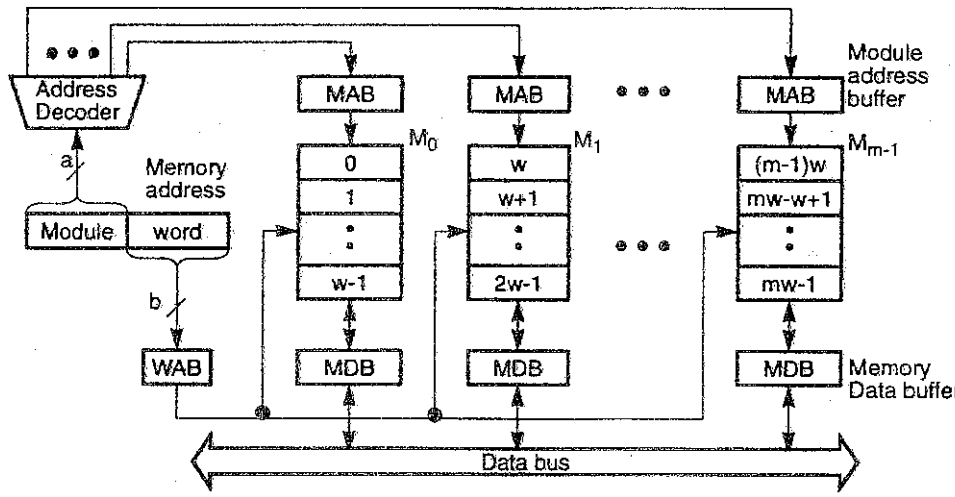


(b) Successive vector accesses using overlapped fetch and access cycles

Figure 8.1 The S-access interleaved memory for vector operands access.

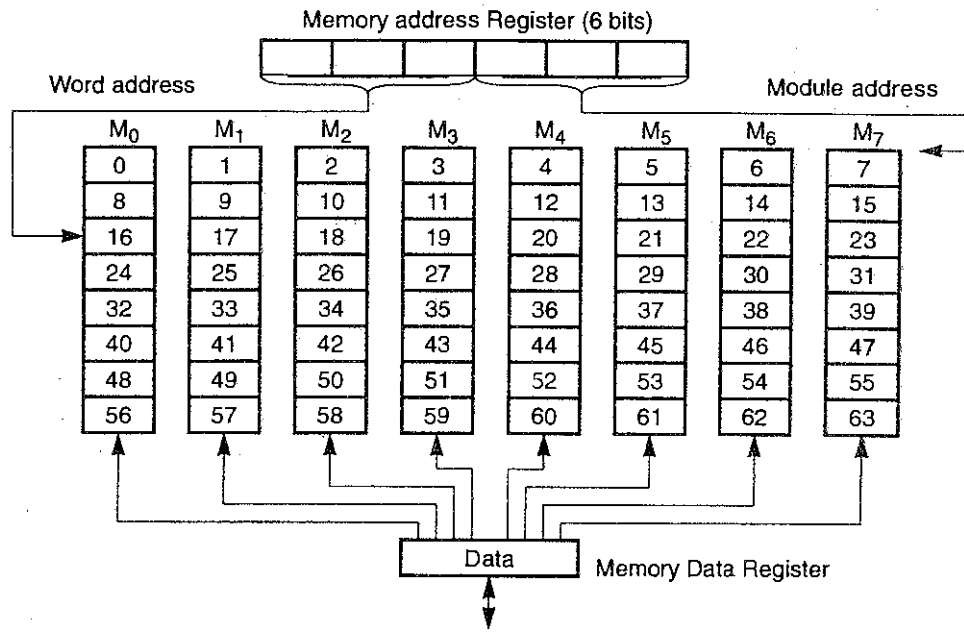


(a) Low-order m-way interleaving (the C-access memory scheme)

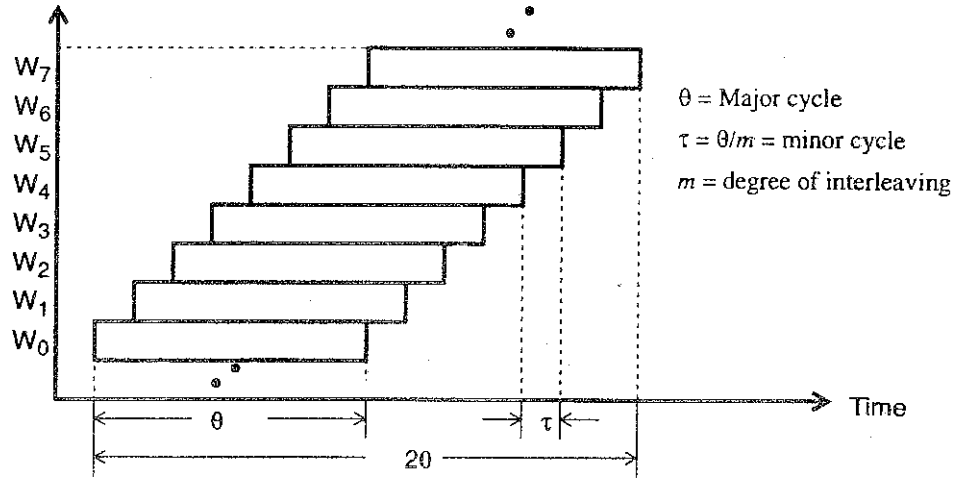


(b) High-order m-way interleaving

Figure 8.2 Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (word addresses shown in boxes).



(a) Eight-way low-order interleaving (absolute address shown in each memory word)



(b) Pipelined access of eight consecutive words in a C-access memory

Figure 8.3 Multiway interleaved memory organization and the C-access timing chart.

S-access. In each memory cycle, at most $m \cdot n$ words are fetched if the n buses are fully used with pipelined memory accesses.

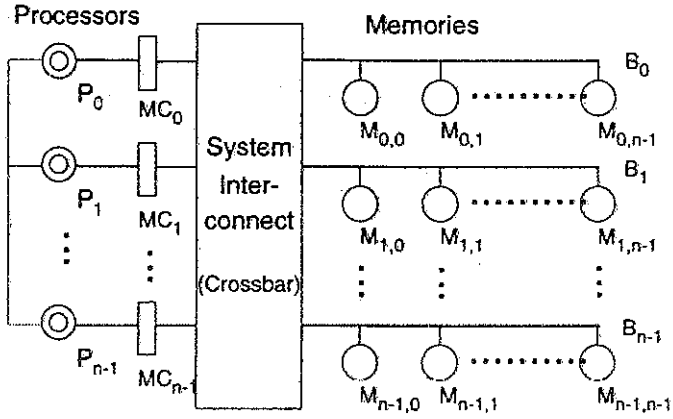


Figure 8.4 The C/S memory organization. (Courtesy of D.K. Panda, 1990)

The C/S-access memory is suitable for use in vector multiprocessor configurations. It provides parallel pipelined access of vector data set with high bandwidth. Special *vector cache* design is needed within each processor in order to guarantee a smooth data movement between the memory and multiple vector processors.

Problem 8.4 The comparison is summarized in the following table:

Class	Architecture	Performance	Cost
Full-scale Supercomputers	multiprocessor multi vector pipeline pipeline chaining	>1 Gflops	\$2 ~ 25 million
High-end mainframes or near Supercomputers	attached vector processor	> 200 Mflops	\$1 ~ 4 million
Minisupercomputers or supercomputing workstations	multicomputer	> 100 Mflops	\$0.1 ~ 1.5 million

Problem 8.5

(a) A composite function of vector operations converted from a looping structure of linked scalar operations

(b)
- The program construct for processing long vectors is called a vector loop. When a vector has a length greater than that of the vector registers, segmentation of the long vector into fixed-length segments is necessary. One segment is processed at a time.
- Pipeline chaining links vector operations following a linear dataflow pattern. Vector registers are used as interfaces between functional pipelines. Continuous data flow is maintained in successive pipelines.

(c) A synchronous program graph in which all nodes have zero delay.

(d) A pipenet is constructed from interconnecting multiple functional pipelines through two buffered crossbar networks which are themselves pipelined.

Problem 8.6

(a) Figure 8.5 shows the CM-2 processor chips with memory and floating-point chips. Each data processing node contains 32 bit-slice data processors, an optional floating-point accelerator, and interfaces for interprocessor communication. Each data processor is implemented with a 3-input and 2-output bit-slice ALU and associated latches and memory interface. This ALU can perform bit-serial full-adder and Boolean logic operations.

The processor chips are paired in each node sharing a group of memory chips. Each processor chip contains 16 processors. The parallel instruction set, called *Paris*, includes nanoinstructions for memory load and store, arithmetic and logical, and control of the router, NEWS grid, and hypercube interface, floating-point, I/O, and diagnostic operations.

The memory data path is 22 bits (16 data and 6 ECC) per processor chip. The 18-bit memory address allows $2^{18} = 256$K memory words (512 Kbytes of data) shared by 32 processors. The floating-point chip handles 32-bit operations at a time. Intermediate computational results can be stored back into the memory for subsequent use. Note that integer arithmetic is carried out directly by the processors in a bit-serial fashion.

(b)
- Special hardware is built on each processor chip for data routing among the processors. The router nodes on all processor chips are wired together to form a Boolean n-cube. A full configuration of CM-2 has 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube.

  Each router node is connected to 12 other router nodes, including its paired node (Fig. 8.5). All 16 processors belonging to the same node are equally capable to send a message from one vertex to any other processor at another vertex of the 12-cube. The following example clarifies this message passing concept.

  On each vertex of the 12-cube, the processors are numbered 0 through 15. The hypercube routers are numbered 0 through 4095 at the 4096 vertices. A processor 5 on router node 7 is thus identified as the 117th processor in the entire system, because $16 \times 7 + 5 = 117$.

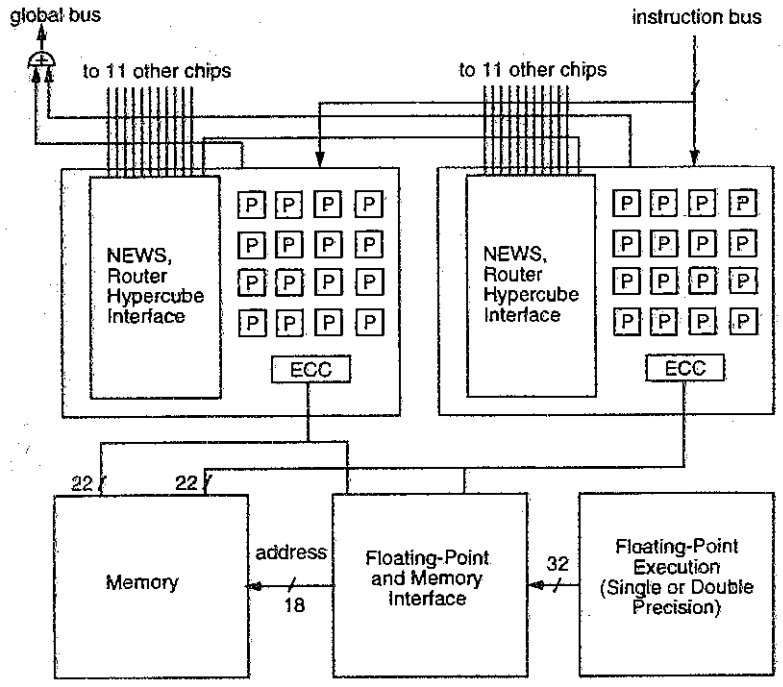


Figure 8.5 A CM-2 processing node consisting of two processor chips and some memory and floating-point chips. (Courtesy of Thinking Machines Corporation, 1990)

Suppose processor 117 wants to send a message to processor 361, which is located at processor 9 on router node 22 ($16 \times 22 + 9 = 361$). Since router node 7 = $(000000000111)_2$ and router node 22 = $(000000010110)_2$, they differ in dimension 0 and dimension 4.

This message must traverse dimensions 0 and 4 to reach its destination. From router node 7, the message is first directed to router node 6 = $(000000000110)_2$ through dimension 0 and then to router node 22 through dimension 4, if there is no contention for hypercube wires. On the other hand, if router 7 has another message using the dimension 0 wire, the message can be routed first through dimension 4 to router 23 = $(000000010111)_2$ and then to the final destination through dimension 0 to avoid channel conflicts.

- Within each processor chip, the 16 physical processors can be arranged as an 8×2, 1×16, 4×4, $4 \times 2 \times 2$, or $2 \times 2 \times 2 \times 2$ grid, and so on. Sixty-four *virtual processors* can be assigned to each physical processor. These 64 virtual processors can be envisioned as forming an 8×8 grid within the chip.

The NEWS grid stands for the fact that each processor has a north, east, west, and south neighbor in the various grid configurations. Furthermore, a subset of the hypercube wires can be chosen to connect the 2^{12} nodes (chips) as a two-dimensional grid of any shape. For instance, 64×64 is one of the possible grid configurations.

Coupling the internal grid configuration within each node with the global grid configuration, one can arrange the processors in NEWS grids of any shapes involving any number of dimensions. This flexible interconnections among the processors make it very attractive for routing data on dedicated grid configurations based on the application requirements.

(c) Besides dynamic reconfiguration in NEWS grids through the hypercube routers, the CM-2 has special built-in hardware support for scanning or spreading across the NEWS grids. These are very powerful parallel operations for fast data combining or spreading throughout the entire array.

Scanning on NEWS grids combines communication and computation. The operation can simultaneously scan in every row of a grid along a particular dimension for the partial sum of that row, or finding the largest or smallest value, or computing bitwise OR, AND, or exclusive OR. Scanning operations can be expanded to cover all elements of an array.

Spreading can send a value to all other processors across the chips. A single-bit value can be spread from one chip to all other chips along the hypercube wires in only 75 steps. Variants of scans and spreads have been built into the Paris instructions for ease of access.

(d)
- In broadcasting, copies of a single item are sent to all processors. In CM-2, this is carried out through the broadcast bus to all data processors at once.
- Global combining allows the front end to obtain the sum, largest value, logical OR, etc., of values, one from each processor.
- Data parallel programming provides the high-level programmer with the illusion of as many processors as necessary; one programs as if there were a processor for every data element to be processed. These are often described as *virtual processors*.

Problem 8.7

(a) The X-Net interconnect directly connects each PE with its eight neighbors in the two-dimensional mesh. Each PE has 4 connections at its diagonal corners, forming an X pattern, similar to the BLITZEN X grid network (Davis and Reif, 1986). A tri-state node at each X intersection permits communications with any of 8 neighbors using only 4 wires per PE.

The connections to the PE array edges are wrapped around to form a two-dimensional torus. The torus structure is symmetric and facilitates several important matrix algorithms and can emulate a one-dimensional ring with two X-Net steps. The aggregate X-Net communication bandwidth is 18 Gbytes/s in the largest MP-1 configuration.

(b) The network provides global communication between all PEs and forms the basis for MP-1 I/O system. The three router stages implement the function of a 1024×1024 crossbar switch. Three router chips are used on each processor board.

Each PE cluster shares an originating port connected to router stage S1 and a target port connected to router stage S3. Connections are established from an originating PE through stages S1, S2, and S3, and then to the target PE. The full MP-1 configuration has 1024 PE clusters, so each stage has 1024 router ports. The router supports up to 1024 simultaneous connections with an aggregate bandwidth of 1.3 Gbytes/s.

(c) 1. Each PE has a 4-bit integer ALU, a 1-bit logic unit, a 64-bit mantissa unit, a 16-bit exponent unit, and a flag unit. All these functional units can be simultaneously active at the same time.

2. The PE array communicates with parallel disk array through the high-speed I/O system, which is essentially implemented by the 1.3 Gbytes/s global router network.

Problem 8.8

(a) A fat tree is more like a real tree in that it gets thicker from the leaves. Processing nodes, control processors, and I/O channels are located at the leaves of the fat tree. A *binary fat tree* was illustrated in Fig. 8.6. The internal nodes are switches. Unlike an ordinary binary tree, the channel capacities of a fat-tree increase as we ascend from leaves to root.

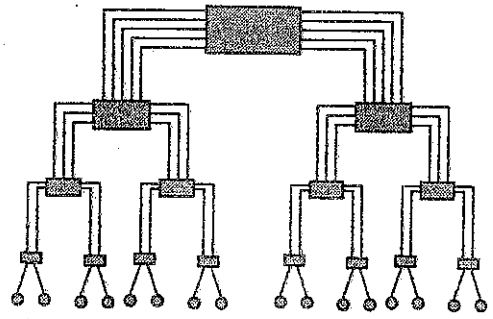


Figure 8.6 Binary fat tree.

The hierarchical nature of a fat tree can be exploited to give each user partition a dedicated subtree, which cannot be interfaced with by any other partition's message traffic. The CM-5 data network actually implemented a 4-ary fat tree as shown in Fig. 8.7. Each of the internal switch nodes is made up of several router chips. Each router chip is connected to 4 child chips and either 2 or 4 parent chips.

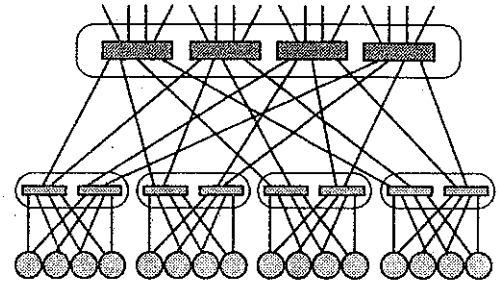


Figure 8.7 CM-5 data network implemented with a 4-ary fat tree (Courtesy of Leiserson et al., Thinking Machines Corporation, 1992)

To implement the partitions, one can allocate different subtrees to handle different partitions. The size of the subtrees varies with different partition demands. The I/O channels are assigned to another subtree, which is not devoted to any user partition. The I/O subtree is accessed as shared system resources. In many ways, the data network functions like a hierarchical system bus, except with no interference among partitioned subtrees. All leaf nodes have unique physical addresses.

(b) The fat tree can be subdivided into several subtrees. Each subtree is assigned to a user partition. Each partition consists of a control processor, a collection of processing nodes, and dedicated portions of the data and control networks.

(c) As shown in Fig. 8.8, the basic control processor consists of a RISC microprocessor (CPU), memory subsystem, I/O with local disks and Ethernet connections and a CM-5 network interface. This is equivalent to a standard off-the-shelf workstation-class computer system. The network interface connects the control processor to the rest of the system through the control network and data network.

Each control processor runs the CMOST, a UNIX-based OS with extensions for managing the parallel processing resources of the CM-5. Some control processors are used to manage computational resources in user partitions. Some others are used to manage I/O resources. Control processors are specialized in managerial functions rather than computational functions. For this reason, high-performance arithmetic accelerators are not needed. Instead, additional I/O connections are more useful in control processors.

(d) As illustrated in Fig. 8.10a, vector units can be added between the memory bank and the system bus as an optional feature. The vector units replace the memory controller in Fig. 8.9. Each vector unit has a dedicated 72-bit path to its attached memory bank, providing a peak memory bandwidth of 128 Mbytes/s per vector unit.

The vector unit executes vector instructions issued to them by the scalar microprocessor and performs all functions of a memory controller, including generation and check of ECC (error correcting code) bits. As detailed in Fig. 8.10b,

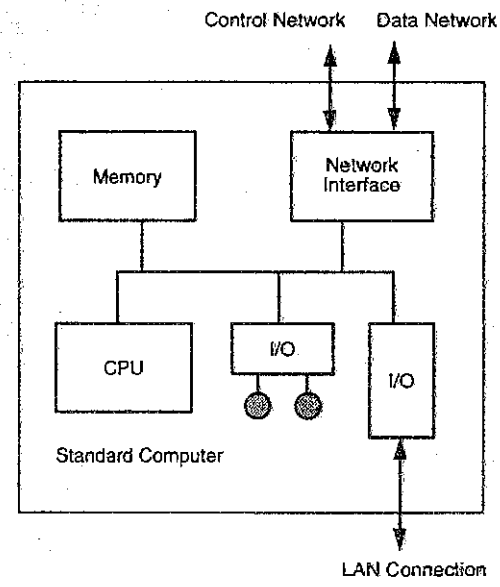


Figure 8.8 The control processor in CM-5. (Courtesy of Thinking Machines Corporation, 1992)

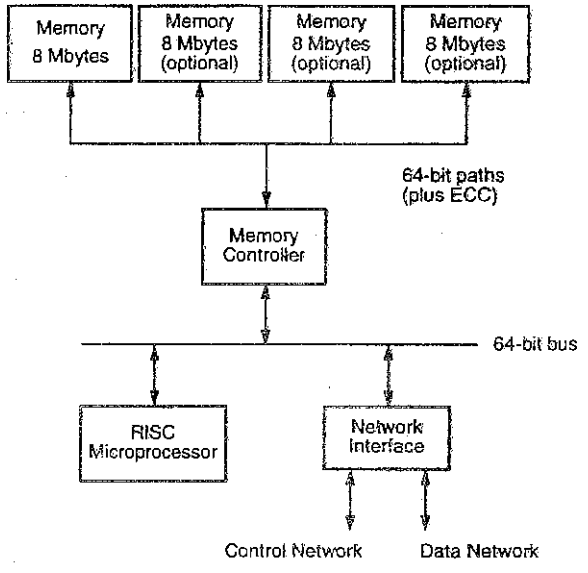
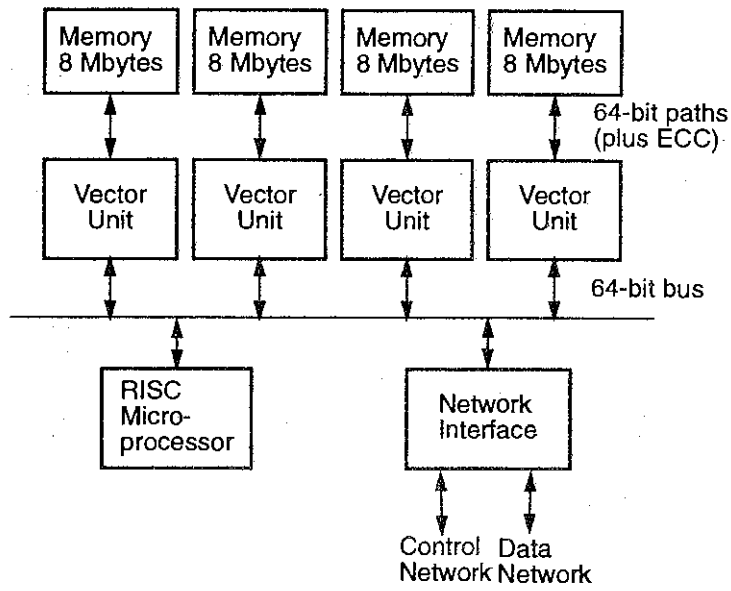
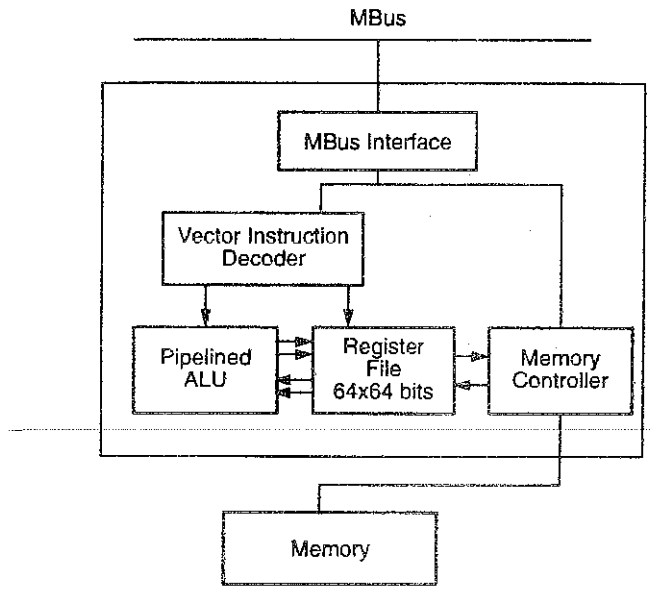


Figure 8.9 The processing node in CM-5. (Courtesy of Thinking Machines Corporation, 1992)



(a) Processing node with vector units



(b) Vector unit functional architecture

Figure 8.10 The processing node with vector units in CM-5. (Courtesy of Thinking Machines Corporation, 1992)

each vector unit has a vector instruction decoder, pipelined ALU and 64 64-bit registers like a conventional vector processor.

Each vector instruction may be issued to a specific vector unit or pairs of units, or broadcast to all four units at once. The scalar microprocessor takes care of address translation and loop control, overlapping them with vector unit operations. Together, the vector units provide 512 Mbytes/s memory bandwidth and 128 Mflops 64-bit peak performance per node.

In this sense, each processing node of CM-5 is itself a supercomputer. Collectively, 16K processing nodes can yield a peak performance of $2^{14} \times 2^7 = 2^{21}$ Mflops = 2 Tflops.

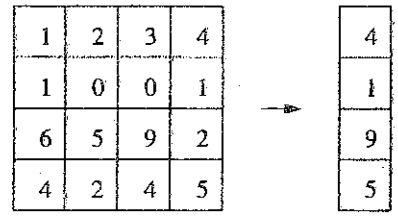
Initially, the SPARC microprocessors are being used in implementing the control processors and processing nodes. As processor technology advances, other new processors may also be combined in the future. The network architectures are designed to be independent of the processors chosen except the network interfaces which may need some minor modifications when new processors are used.

Problem 8.9

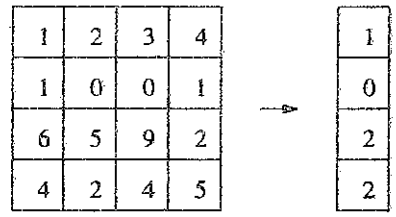
(a) An example of replication: If A and B are arrays and X is a scalar quantity, the statement $A = B + X$ implicitly broadcasts X to all processors so that the value of X can be added to every element of B.

(b) Besides sum-reduction, other important reduction operations include taking maximum or minimum, logical AND, and logical OR. The following are examples of maximum-reduction and minimum-reduction:

Maximum Reduction

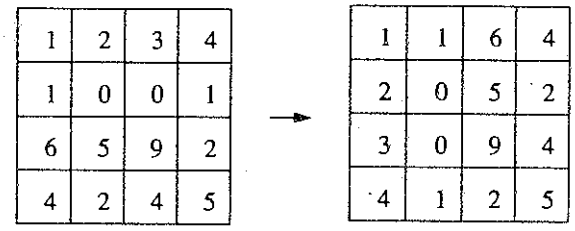


Minimum Reduction



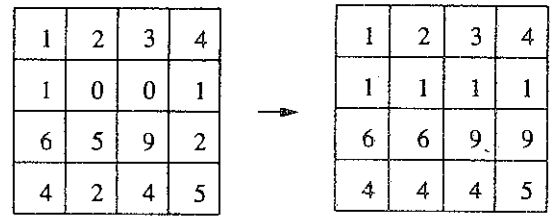
(c) Transposing a matrix, reversing a vector, shifting a multidimensional grid, and FFT butterfly patterns are all examples of permutation. Here is an example of matrix transposition:

Matrix Transposition

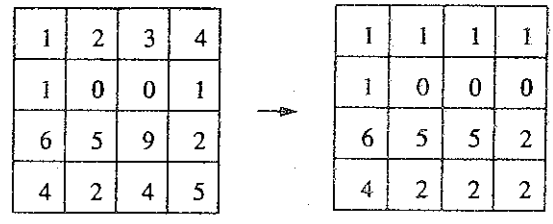


(d) The following are examples of maximum-prefix and minimum-prefix:

Maximum Prefix

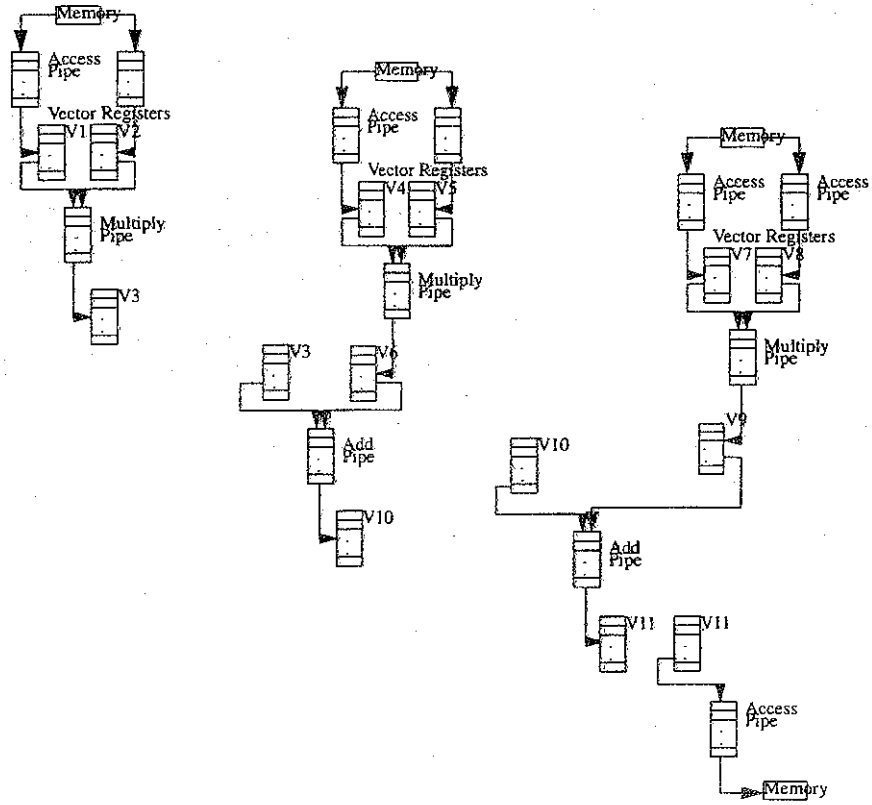


Minimum Prefix

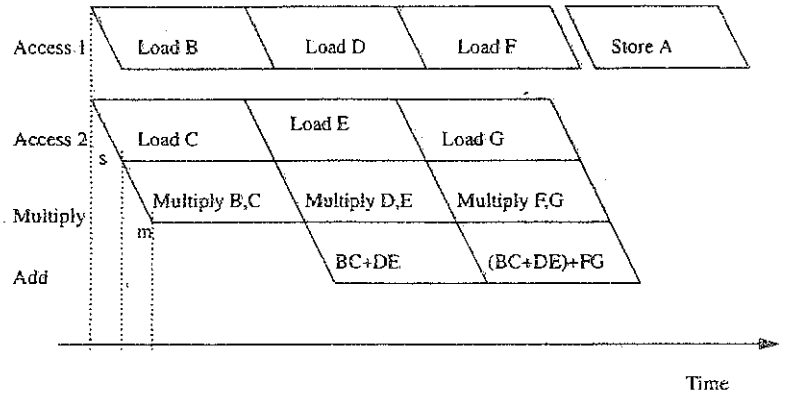


Problem 8.10

(a) Pipeline chaining for CVF execution:



(b) Space-time diagram:

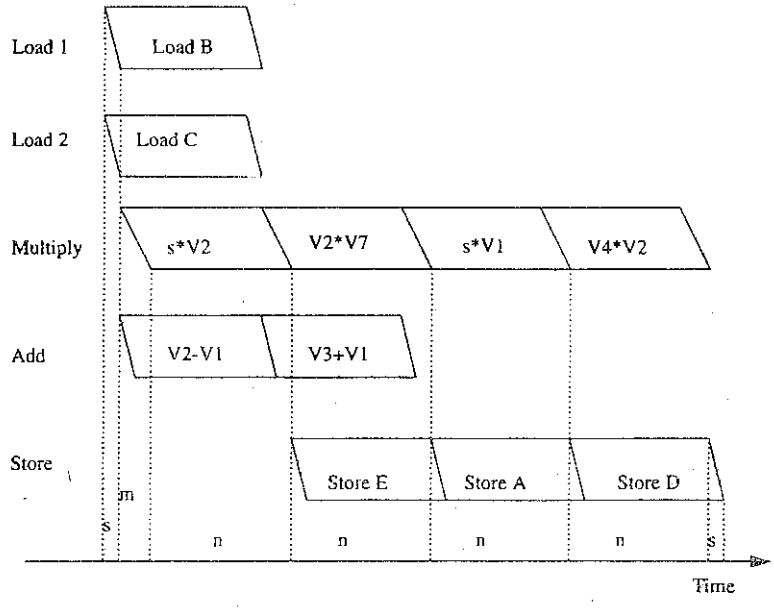


Problem 8.11

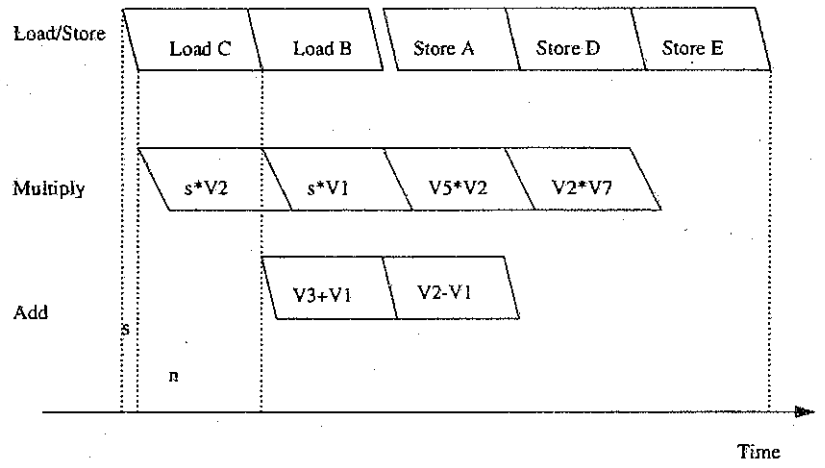
(a) The 11 vector instructions needed to perform the given CVFs on Cray X-MP are shown in the follows:

$$
\begin{aligned}
M(B:B+63) &\rightarrow V1 \\
M(C:C+63) &\rightarrow V2 \\
s \times V2 &\rightarrow V3 \\
V3+V1 &\rightarrow V4 \\
V4 &\rightarrow M(A:A+63) \\
s \times V1 &\rightarrow V5 \\
V5 \times V2 &\rightarrow V6 \\
V6 &\rightarrow M(D:D+63) \\
V2-V1 &\rightarrow V7 \\
V2 \times V7 &\rightarrow V8 \\
V8 &\rightarrow M(E:E+63)
\end{aligned}
$$

(b) Space-time diagram for the execution of the CVF code:



(c) Execution of the CVFs using pipeline chaining on Cray 1:



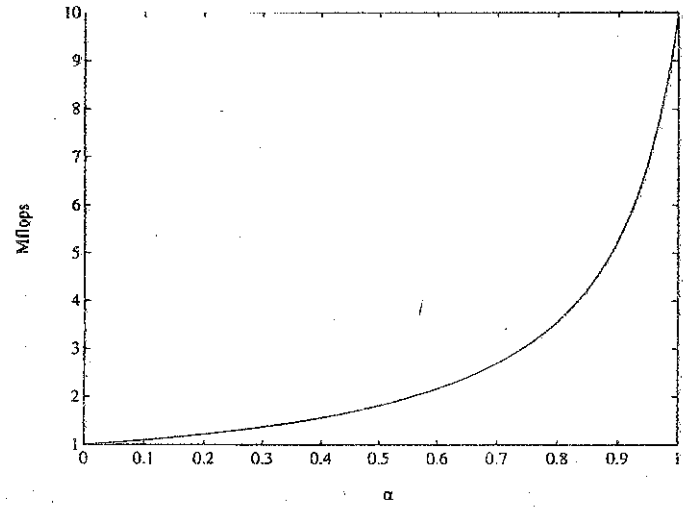
Speedup of Cray X-MP over Cray 1 $= \dfrac{5n+s}{4n+m+2s} \approx 1.25$ for large n.

Problem 8.12

(a) Average execution rate can be computed as

$$\begin{aligned} R_a &= \frac{\alpha + (1-\alpha)}{\alpha/R_v + (1-\alpha)/R_s} \\ &= \frac{10}{10 - 9\alpha} \text{ (Mflops).} \end{aligned}$$

(b) The plot is shown below:



(c) We have

$$\frac{10}{10 - 9\alpha} = 7.5.$$

Hence $\alpha = 26/27 = 0.963$.

(d) With the given data, the following equation is obtained:

$$\frac{R_v}{R_v - 0.7(R_v - 1)} = 2,$$

which can be solved to give $R_v = 3.5$ Mflops.

Problem 8.13

(a) The algorithm to compute the expression in a serial computer is shown below:

$s = A_1 \times B_1$
For i = 2 to 32 **Do**
  $s = s + A_i \times B_i$
Enddo

There are 32 multiply operations and 31 add operations. The number of time units needed is $32 \times 4 + 31 \times 2 = 190$.

(b) The algorithm for the SIMD computer is shown below:

Parfor j = 1 to 8 **Do**
  $s(j) = A_{1j} \times B_{1j}$ /* 1 multiply operation */
  For i = 2 to 4 *Do*
    $s(j) = s(j) + A_{ij} \times B_{ij}$ /* 1 multiply and 1 add operations */
  Enddo
  $s(j) = s(j) + s(j + 1)$ /* 1 routing and 1 add operations */
  $s(j) = s(j) + s(j + 2)$ /* 2 routing and 1 add operations */
  $s(j) = s(j) + s(j + 4)$ /* 4 routing and 1 add operations */
Enddo

There are 4 multiply operations, 6 add operations, and 7 routing operations. The time needed is $4 \times 4 + 6 \times 2 + 7 \times 1 = 35$ cycles.

Problem 8.14

(a) A Cray Y-MP C-90 has 16 processors. Each processor has 2 vector pipelines. Each pipeline has a floating point multiply and an add unit which can operate concurrently. Therefore, two floating point operations can be performed each cycle in a vector pipeline. Total operations performed in a cycle are $16 \times 2 \times 2 = 64$. It has a cycle time 4.2 ns. Hence, the peak performance = (64 floating-point operations) / (4.2 ns) = 15.2 Gflops.

(b) An NEC SX-X has 4 processors. Each processor has 4 sets of vector pipelines. Each set has two add/shift and two multiply/logical pipelines. Total operations performed in a cycle are $4 \times 4 \times 2 \times 2 = 64$. Its cycle time is 2.9 ns. Thus the peak performance = (64 floating-point operations) / (2.9 ns) = 22 Gflops.

(c) Both machines perform 64 floating operations per cycle as explained above.

Problem 8.15

(a) Matrices A and B are both divided into blocks, each of size 8×8. Denote the blocks as $A_{i,j}$ and $B_{i,j}$, respectively, for $0 \leq i, j \leq 7$. Cannon's algorithm for matrix multiplication is used in this problem. The following diagram shows the initial distribution of matrices A and B among the PEs. The submatrix blocks are stored in a skewed manner. The diagonal subblocks of A appear in the first column, those of B appear in the first row.

A_{00}	A_{01}	A_{02}	A_{03}	A_{04}	A_{05}	A_{06}	A_{07}
A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{10}
A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	A_{20}	A_{21}
A_{33}	A_{34}	A_{35}	A_{36}	A_{37}	A_{30}	A_{31}	A_{32}
A_{44}	A_{45}	A_{46}	A_{47}	A_{40}	A_{41}	A_{42}	A_{43}
A_{55}	A_{56}	A_{57}	A_{50}	A_{51}	A_{52}	A_{53}	A_{54}
A_{66}	A_{67}	A_{60}	A_{61}	A_{62}	A_{63}	A_{64}	A_{65}
A_{77}	A_{70}	A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}

B_{00}	B_{11}	B_{22}	B_{33}	B_{44}	B_{55}	B_{66}	B_{77}
B_{10}	B_{21}	B_{32}	B_{43}	B_{54}	B_{65}	B_{76}	B_{07}
B_{20}	B_{31}	B_{42}	B_{53}	B_{64}	B_{75}	B_{06}	B_{17}
B_{30}	B_{41}	B_{52}	B_{63}	B_{74}	B_{05}	B_{16}	B_{27}
B_{40}	B_{51}	B_{62}	B_{73}	B_{04}	B_{15}	B_{26}	B_{37}
B_{50}	B_{61}	B_{72}	B_{03}	B_{14}	B_{25}	B_{36}	B_{47}
B_{60}	B_{71}	B_{02}	B_{13}	B_{24}	B_{35}	B_{46}	B_{57}
B_{70}	B_{01}	B_{12}	B_{23}	B_{34}	B_{45}	B_{56}	B_{67}

Blocks of C are stored in the natural order in PEs as shown below.

C_{00}	C_{01}	C_{02}	C_{03}	C_{04}	C_{05}	C_{06}	C_{07}
C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}
C_{20}	C_{21}	C_{22}	C_{23}	C_{24}	C_{25}	C_{26}	C_{27}
C_{30}	C_{31}	C_{32}	C_{33}	C_{34}	C_{35}	C_{36}	C_{37}
C_{40}	C_{41}	C_{42}	C_{43}	C_{44}	C_{45}	C_{46}	C_{47}
C_{50}	C_{51}	C_{52}	C_{53}	C_{54}	C_{55}	C_{56}	C_{57}
C_{60}	C_{61}	C_{62}	C_{63}	C_{64}	C_{65}	C_{66}	C_{67}
C_{70}	C_{71}	C_{72}	C_{73}	C_{74}	C_{75}	C_{76}	C_{77}

(b) The overall algorithm is specified as follows for each PE:

For $i = 0$ to 7 **Do**
Compute the product of block submatrices of A and B residing in it and add the product to the part of matrix C.
Pass block submatrix of A to its left neighbor in a wraparound fashion using shift operations.
Pass block submatrix of B to its upper neighbor in a wraparound fashion using shift operations.
Enddo

Basically, in step 1 of the ith iteration, $PE_{k,l}$ performs the following computations:

$$C_{k,l} = C_{k,l} + A_{k,(i+j) \bmod 8} B_{(i+j) \bmod 8,l}$$

where j is the initial column index of the block submatrix of A residing in $PE_{k,l}$. It is straightforward to specify the detailed operations for the multiplication of two submatrix blocks in each PE.

Steps 2 and 3 exchange matrix elements among the PEs. In the last iteration, they bring individual submatrices back to the PEs in which they are initially resident. Note that all the PEs perform identical operations on different data, in keeping with the SIMD mode of operation.

(c) The multiplication of 8×8 matrix blocks in each PE and accumulation into C take 8^3 multiplication and 8^3 addition operations. Steps 2 and 3 require 8^2 shift operations each. Therefore, the number of cycles needed in each iteration is $2 \times 8^3 + 2 \times 8^2 = 1152$. The total number of cycles in 8 iterations is 9216. If the shift operations of the last iteration are omitted, 128 cycles can be saved.

(d) If data duplication is allowed, each block submatrix of A is duplicated along the row, and each submatrix of B is duplicated along the column by the following instructions:

For $i = 0$ to 7 **Do**
PEs in column i broadcast submatrices of A to other PEs in the same row.
PEs in row i broadcast submatrices of B to other PEs in the same column.
Enddo

Now, each PE has all the elements needed to compute a subblock of C matrix and no further data movement is required. So the last step is for all PEs to compute the submatrix blocks of C simultaneously. The arithmetic operations are identical to those in (b). Possible saving in execution time comes from the reduction in communication overhead if broadcast operations can be carried out efficiently.

Problem 8.16 The comparison of CM-2 and CM-5 is summarized in the table below; more detailed comparison can be found in relevant manuals.

Machine	Architecture	Operation Mode	Potential Performance	Improvement
CM-2	64K bit-slice processors, Hypercube	SIMD	10 Gflops	
CM-5	16K SPARCs, 4-ary fat tree	SIMD MSIMD Sync. MIMD	2 Tflops	mixture of parallel techniques

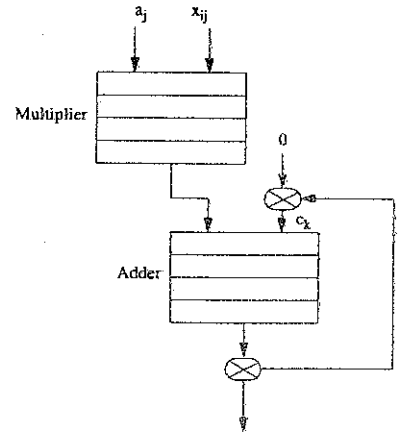
Problem 8.17 The linear combination can be written as

$$\begin{aligned}(y_0, y_1, \ldots, y_{1023}) &= a_0\mathbf{x}_0 + a_1\mathbf{x}_1 + \ldots + a_{1023}\mathbf{x}_{1023} \\ &= a_0 \times (x_{0,0}, x_{1,0}, \ldots, x_{1023,0}) + a_1 \times (x_{0,1}, x_{1,1}, \ldots, x_{1023,1}) + \ldots + \\ &\quad a_{1023} \times (x_{0,1023}, x_{1,1023}, \ldots, x_{1023,1023}) \\ &= (a_0x_{0,0} + a_1x_{0,1} + \ldots + a_{1023}x_{0,1023}, a_0x_{1,0} + a_1x_{1,1} + \ldots + \\ &\quad a_{1023}x_{1,1023}, \ldots a_0x_{1023,0} + a_1x_{1023,1} + \ldots + a_{1023}x_{1023,1023}).\end{aligned}$$

Thus, we have the following equalities:

$$y_i = \sum_{j=0}^{1023} a_j x_{ij}, \qquad \text{for } i = 0, 1, \ldots, 1023. \tag{8.1}$$

(a) From Eq. (8.1) in the above, we see that each element of **y** can be computed separately. Thus, each processor can be used to carry out one-fourth of the computations — processor ℓ computes elements $(\ell - 1) \times 256$ through $\ell \times 256 - 1$. Vector a is replicated in all processors. The multiplier and adder in each processor are chained as shown in the following diagram:



Without loss of generality, consider the operations performed by processor 0. In each processors, two auxiliary vectors are used. C is a vector of 4 elements which

are initialized to 0. Vector $D(0:1023)$ is used to store intermediate results. Let's examine the computation of y_0. The computations are divided into two phases.

In cycle 0, a_0 and x_0 are fed into the multiplier. After 4 cycles, their product appears at one input of the adder. After four more cycles, the value $a_0x_{0,0}$ appears at the output and is routed back to the input port for C (see the diagram). Thereafter, one more product term is added to C_0 in every four cycles. A similar situation holds for the other elements of C. For a description of pipeline chaining for this purpose, see [Hwang84], pp. 279–280.

After all the product terms for y_0 have been accumulated in the adder, elements of vector C have the following values: $C_k = \sum_{i=0}^{255} a_{4i+k}x_{0,4i+k}$ for $k = 0, 1, 2, 3$. Just prior to the arrival of product terms for y_1 at the adder, $C(0:3)$ are stored in $D(4i:4i+3)$ one by one and $C(0:3)$ are reset to zero.

This process is repeated for successive elements of y. In this way, pairs of elements of a and x can be continuously fed into the multiplier in each cycle. Thus, at the end of the first phase, D has 256 "segments" of 4 elements each. This phase takes $256 \times 1024 + 8 - 1 = 262,151$ clock cycles.

In the second phase, each segment in D is summed up to obtain one element of y. This can be done by first generating 256 pairs of partial sums (two elements in each segment are added). Then each pair of the partial sums is added to produce the final result. In the optimal case, the first four add operations can be overlapped with the last four add operations of the first phase. Therefore, the total number of cycles needed for phase 2 is $512 + 256 = 768$. Consequently, the total number of cycles for the multivector system is $262,151 + 768 = 262,919$.

Note that if the two phases of computations are interspersed, then the vector D is not needed. But the timing is not optimal.

(b) On a single processor without vector processing capability, the number of operations is 1024×1024 multiplications and 1024×1023 addition. Each operations takes 4 cycles, giving a total of 8,384,512 cycles. Therefore, the speedup of the multivector system over the single scalar processor is $8,384,512/262,919 = 31.89$, which is close to the theoretical maximum value of 32.

In the above analysis, pipeline startup time has been neglected and a very intelligent scheduler is assumed. Actual performance may be poorer when various overheads are taken into account.

Problem 8.18 Suppose low-order interleaving is used so that consecutive elements of a vector are stored in contiguous memory modules. Without loss of generality, assume that the first element (element 0) of the vector is stored in memory module 0. Let s be the stride of a vector access, and $n_1 = m_1 s$ and $n_2 = m_2 s$ be the indices of two different elements retrieved. Assume $n_1 > n_2$. The memory modules in which the two elements reside are n_1 mod 17 and n_2 mod 17, respectively. Now,

$$(n_1 \bmod 17) - (n_2 \bmod 17) = (n_1 - n_2) \bmod 17$$
$$= (sm_1 - sm_2) \bmod 17$$

$$= \begin{cases} 0, & \text{if } (m_1 - m_2) \bmod 17 = 0 \\ ((s \bmod 17)((m_1 - m_2) \bmod 17)) \bmod 17 \neq 0, & \text{if } (m_1 - m_2) \bmod 17 \neq 0 \end{cases}$$

The second result follows from the given condition $s \bmod 17 \neq 0$. Therefore, if $(m_1 - m_2) \bmod 17 \neq 0$ for any pair m_1 and m_2, there will be no conflicts in memory accesses. Normally, if the elements are accessed in increasing order and at most 17 elements are accessed at a time, then $(m_1 - m_2) \bmod 17 \neq 0$, which ensures conflict-free accesses.

Chapter 9

Scalar, Multithreaded, and Dataflow Architectures

Problem 9.1

(a) The efficiency can be computed as

$$E = \frac{1/R}{1/R + L} = \frac{1}{1 + RL}.$$

(b) The new rate of remote memory requests is $R' = (1 - h)R$. Hence,

$$E = \frac{1}{1 + R'L} = \frac{1}{1 + RL(1 - h)}.$$

(c) If $N \geq N_d = \frac{L}{1/R' + C} + 1$,

$$E_{\text{sat}} = \frac{1}{1 + CR'} = \frac{1}{1 + (1 - h)CR}.$$

If $N \leq N_d$,

$$E_{\text{lin}} = \frac{N/R'}{1/R' + C + L} = \frac{N}{1 + R'C + R'L} = \frac{N}{1 + (1 - h)(L + C)R}.$$

(d) To compute L, we need to compute *mean internode distance* $\overline{D}$. Let P be the probability that a node sends a message to all other nodes with distance i. In reference to Problem 2.11, $\overline{D}$ can be computed as follows:

$$\overline{D} = \sum_{i=1}^{D} i \times P = \sum_{i=1}^{D} i \frac{D - i + 1}{\sum_{k=1}^{D} k}.$$

Since $D = 2\lfloor r/2 \rfloor \approx r$,

$$\overline{D} = \sum_{i=1}^{D} i \frac{r-i+1}{\sum_1^r k} = \frac{r-i+1}{r(r+1)/2} = \frac{2(r-i+1)}{r(r+1)} = \frac{2}{r(r+1)} \sum_{i=1} r[(r+1)i - i^2] = \frac{r+4}{3}.$$

$$L = 2\overline{D}t_d + t_m = \frac{2(r+4)}{3} t_d + t_m = \frac{2(\sqrt{p}+4)}{3} t_d + t_m.$$

$$E_{\text{sat}} = \frac{1/R'}{1/R' + C} = \frac{1}{1 + (1-h)CR}.$$

$$E_{\text{lin}} = \frac{N}{1 + (1-h)R(L+C)} = \frac{N}{1 + (1-h)R([\frac{2(\sqrt{p}+4)}{3} t_d + t_m] + C)}.$$

Problem 9.2 The architectural assumptions and notations used in this problem are similar to those in [Saavedra90]. A deterministic model is adopted in the analysis. Summarized below are basic system parameters to be used:

- N: The *number of threads* or *contexts* that can be executed simultaneously in each processor.
- C: The *context switching overhead* which accounts for the cycles lost in performing context switch in a processor.
- L: The *communication latency* for a processor to access a a remote memory through the network.
- R: The *run length* of a single thread before it issues a memory request or is switched out. Note that the definition here is the inverse of the definition of R in Problem 9.1.
- f: The *coverage factor* for prefetching, defined as the percentage of memory requests successfully prefetched to satisfy the demand of a thread.
- E: The *processor efficiency*, defined as the percentage of time a processor is actively executing a thread.

(a) Effectively, prefetching reduces the memory latency from L to L' ($L' < L$). If a memory request has been prefetched, the time spent on the request equals $V + L'$, where V is the *overhead for prefetching*, which includes the effects of extra instructions inserted to perform prefetching. (Assume software prefetching technique is used.) The processor efficiency E of a single-threaded processor with prefetching can be expressed as

$$E = \frac{R}{f(V + L') + (1-f)L + R}. \tag{9.1}$$

The latency for remote access is reduced from L to $f(V + L') + (1-f)L$.

(b) Based on the reduced latency, two different regions in the efficiency curve can be identified:

$$E = \begin{cases} \dfrac{R}{R+C} & \text{if } N \geq \dfrac{f(V+L')+(1-f)L}{R+C}+1 \\ \dfrac{NR}{f(V+L')+(1-f)L+R+C} & \text{if } N \leq \dfrac{f(V+L')+(1-f)L}{R+C}+1 \end{cases}$$

Problem 9.3 For this problem, the same parameters as defined in Problem 9.2 are used.

(a) The major benefit of release consistency lies in allowing the *read* requests to bypass outstanding *write* requests and allowing *write* requests to be pipelined. Therefore, the processor stalls only for *read* requests or when the write buffer is full. This probability of a write buffer being full is usually very low if the write buffer is large enough in capacity.

Let ω be the probability of a request being a *write*. The processor efficiency with release consistency alone can be expressed as:

$$E = \frac{R}{R+(1-\omega)L+\omega b}, \tag{9.2}$$

where b is a parameter that depends on the buffer capacity, network delay, ω, and the rate at which remote memory accesses are requested by each processor.

(b) With release consistency model, the number of threads needed to completely hide the latency is

$$N_r = \frac{(1-\omega)L+wb}{R+C}+1.$$

Thus, the efficiency is

$$E = \begin{cases} \dfrac{R}{R+C} & \text{if } N \geq N_r \\ \dfrac{NR}{(1-\omega)L+wb+R+C} & \text{if } N \leq \dfrac{f(V+L')+(1-f)L}{R+C}+1 \end{cases}$$

(c) If prefetching and release consistency are both employed, the latency will be further reduced. Combining the results in Eqs. 9.1 and 9.2 above, we obtain the following expression for the effective latency in a single-threaded processor:

$$L_{eff} = f(V+L')+(1-f)((1-\omega)L+\omega b).$$

Thus, the efficiency of a single-threaded processor is

$$E_{st} = \frac{R}{R+L_{eff}}.$$

Based on L_{eff}, we can determine the number of threads needed to fully hide the latency in a multithreaded processor as

$$N_{pr} = \frac{f(V+L')+(1-f)((1-\omega)L+\omega b)}{R+C}+1.$$

Hence, the efficiency in a multiple-threaded processor is

$$E_{mt} = \begin{cases} \dfrac{R}{R+C} & \text{if } N \geq N_{pr} \\ \dfrac{NR}{f(V+L')+(1-f)((1-\omega)L+\omega b)+R+C} & \text{if } N \leq N_{pr} \end{cases}$$

Problem 9.4

(a) We know m processors are attached to each column bus, since there are m row buses in the system. Each generates r requests per second on the bus. Thus, the total request is mr. Suppose each request consists of t bits, assuming a uniform length for all requests. (Alternatively, t can be taken as the average length of each request.) Then the following relation holds:

$$mrt = Ba.$$

Therefore, the memory bandwidth is

$$B = \frac{mrt}{a}.$$

(b) Assume all the buses (row or column) have the same bandwidth. There are $2m$ buses in the system. Hence, the total bus bandwidth is $2mB = 2m^2rt/a$.

(c) There are m^2 processors in the system, each generating r requests per second. If each request uses only a row bus or column bus, then the total bandwidth requirement is m^2rt. This has to be satisfied by the available memory bandwidth, which is $2mB$. Therefore

$$m^2rt \leq 2mB.$$

Hence,

$$r \leq \frac{2B}{mt}.$$

(d) If all the processors send requests that need to go through two buses (one column bus and one row bus), then at a certain instance of time, there would be $m^2r + m^2r = 2m^2r$ requests that need to be serviced by the bus system. Therefore, the total bus bandwidth needed is $2m^2rt$.

(e) The bus bandwidth of the multicube system is designed to allow a bus utilization rate of at most 1 (i.e., $0 < a \leq 1$). In (d), the bus bandwidth requirement represents the maximum bandwidth demand. From the relation

$$2m^2rt \leq 2m^2rt/a,$$

it is concluded that the available bus bandwidth provided by the multicube is adequate.

Problem 9.5

(a) See Fig. 4 in [Hwang91].

(b) Because of the column and row access modes available on the OMP, special instructions are needed. Please refer to the original paper [Hwang91] for a description of the instructions, data distribution, and SPMD program for performing matrix multiplication.

(c) The number of orthogonal memory accesses is $2N^3/n^3 + 2N^2/n + N^2/n^2$. The number of synchronizations is $2N^2/n^2$. For details, see the proof of Lemma 1 in [Hwang91].

(d) Two-dimensional FFT requires $4N^2/n^2$ orthogonal memory accesses and one synchronization. For a description of the SPMD program and complexity analysis, see [Hwang91].

Problem 9.6

(a) The SVM retains the programming paradigm of a tightly-coupled shared-memory multiprocessor, which directly supports data sharing among processes. This promotes portability of programs across systems. In addition, it has the advantages of a distributed memory machine. The large virtual address space allows programs to be much larger in code and data space than the physical memory on an individual node. Moreover, remote memory can be used as an added level of the memory hierarchy between the local memory and disks to improve the performance. Thus, SVM provides such desirable properties as low cost and scalability by getting rid of hardware bottlenecks.

(b) In SVM systems implemented by OS (such as IVY), it is convenient to use the underlying virtual memory page size as the unit of sharing among processes. In hardware-implemented SVM systems (such as Dash), the unit of sharing is usually much smaller, typically the size of a cache block. Some of the differences are listed below:

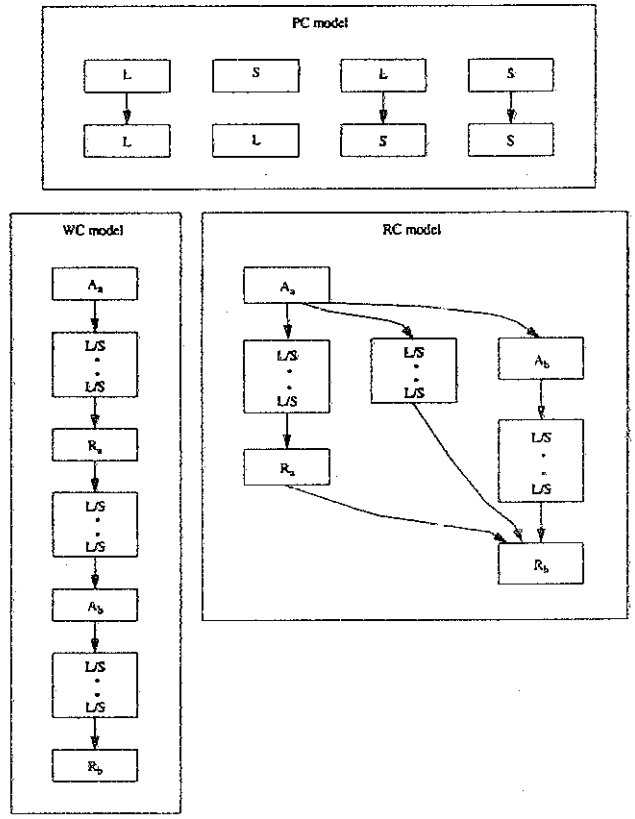
- Page-level sharing is more effective for exploiting locality of references in shared-memory processes. But it is also more susceptible to contention among processes (more than one process trying to access the same page).
- Page-level sharing is more likely to cause false sharing. That is, two processes may access completely different parts of a page.
- The size of directory is much larger if the unit of sharing is cache blocks instead of pages. The storage demand of directory information can be excessive.

Problem 9.7

(a)
- To implement RC, it needs two memory instructions (load-lock and store-unlock) and a lockup-free cache and some kind of scoreboarding to keep track of outstanding requests.

- To implement PC, it needs multiport memory to allow processors to perform out-of-order writes.
- To implement WC, it needs store buffers in each processor with some matching hardware to bypass loads.

(b) Different consistency models impose different constraints on the order of shared memory accesses by each process. The following diagram adapted from [Gharachorloo91] illustrates the event ordering according to PC, WC, and RC models. In the figure, L stands for load, S for store, A for acquire, R for release; an arrow means program order has to be observed; loads and stores in the same block can be executed in any order provided dependence relations are respected. Subscripts to acquire and release operations stand for synchronization variables or memory locations.



Some of the advantages and shortcomings of each model are summarized below. See, for instance, [Gharachorloo91, Mosberger93] for further discussions.

- Advantages of PC: Loads are allowed to overtake store accesses by the same processor if the accesses are to different locations. If a load and a store are to the same memory location, the load can be satisfied by the store

operation, as in TSO or PSO model. Thus, a load never stalls for pending stores.

- Shortcomings of PC: Store operations in each processor have to follow program order, making the chance of a write buffer being full higher, which means the processor is more likely to be stalled.
- Advantages of WC: WC ensures sequential consistency only at synchronization points. Load/store operations between synchronization points can be performed in any order as long as control and data dependences are not violated in each processor. Hence, store operations can be pipelined, leading to improved performance.
- Shortcomings of WC: Processor is stalled at an acquire operation, waiting for previous stores and release to complete. It is also stalled at the first load following a release operation. As a result, in fine-grain computations with frequent synchronizations, WC can perform poorly compared to PC.
- Advantages of RC: The shortcoming of WC for fine-grain computations is eliminated since RC does not block a processor at a load/acquire for previous store/release operations to complete. Independent synchronizations do not need to wait for the completion of each other as shown in the diagram. Therefore, a higher degree of parallelism can be realized.
- Shortcomings of RC: RC requires more complex hardware/software support for implementation (see (a)). Special language construct and compiler support are needed to properly label a program and generate the code for execution in this model.

Problem 9.8

(a)
- It provides the communication, synchronization, and global naming mechanisms required to efficiently support fine-grain, concurrent programming models.
- It extends a conventional microprocessor instruction set architecture with instructions to support parallel processing.
- It provides hardware support for end-to-end message delivery including formatting, injection, delivery, buffer allocation, buffering, and task scheduling.
- It supports a broad range of parallel programming models, including shared-memory, data-parallel, dataflow, actor, and explicit message-passing, by providing low-overhead primitive mechanism for communication, synchronization, and naming. Its communication mechanisms permit a user-level task on one node to send a message to any other node in a 4096-node machine in less than 2 μs.

(b) All messages route first in the X-dimension, then Y, then Z.

(c) The AAU performs all functions associated with memory addressing. It contains the address and ID register to support naming and relocation. It protects memory accesses and implements the translation instructions. It maintains two queues to

buffer incoming messages and schedule the associated tasks.

(d) See Example 9.4.

Problem 9.9

(a) In the VLSI system, networks with many dimensions require more and longer wires than low-dimensional networks. Thus, high-dimensional networks cost more and run more slowly than low-dimensional networks. Under the assumption of constant wire bisection, low-dimensional networks have wide channels, and high-dimensional networks have narrow channels. With wormhole routing method, which is used by most of the second- and third-generation multicomputers, the wider channels provide a lower latency, less contention, and higher hot-spot throughput.

(b) We can treat the router at each node as the stage, and the flit buffer as the stage latch in a superpipelined functional units. Information is transmitted (processed) from a router (stage) to another. The differences are:

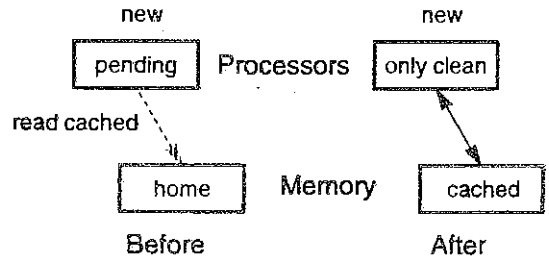
- Most of the pipelined functional units are synchronously operated.
- Pipelined functional units have fixed data flow patterns, but the message passing mechanism may dynamically change its data flow by the routing information in the header flits.

Problem 9.10

(a)

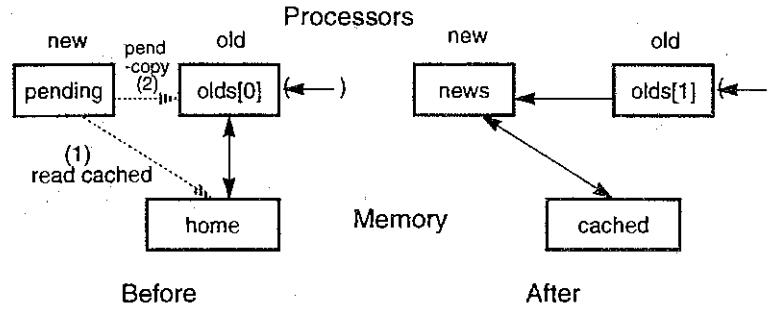
- The memory is initially in the home state (uncached), and all cache copies are invalid. Sharing-list creation begins at the cache where an entry is changed from an invalid to a pending state. When a read-cache transaction is directed from a processor to the memory controller, the memory state is changed from uncached to cached and the requested data is returned.

  The requester's cache entry state is then changed from a pending state to an only-clean state. Sharing-list creation is illustrated in the figure below. Multiple requests can be simultaneously generated, but they are processed sequentially by the memory controller.



- For subsequent memory access, the memory state is cached, and the cache head of the sharing list has possibly dirty data. As illustrated in the figure below, a new requester (cache A) first directs its read-cache transaction to

memory but receives a pointer to cache B instead of the requested data.



A second cache-to-cache transaction, called *prepend*, is directed from cache A to cache B. Cache B then sets its backward pointer to point to cache A and returns the requested data. The dashed lines correspond to transactions between a processor and memory or another processor. The solid lines are sharing-list pointers. After the transaction, the inserted cache A becomes the new head, and the old head (cache B) follows cache A in the chain.

(b) Compared to backplane bus, chained directory provides a greater bandwidth and better scalability. Its cost can be cheaper since snoopy cache controllers are not needed. It allows an invalidation signal to be sent to specific processors instead of broadcasting the signal to all processors. However, it may take a longer time for the signal to reach all the processors involved.

The advantage of a chained directory compared to a full-map directory is the saving in space needed to store directory information. Suppose there are P processors in the system and the number of memory blocks is M. Typically M is proportional to P. If a full-map directory is used, a presence bit is needed to indicate whether a processor has a particular memory block in its cache. The total number of presence bits is $O(MP) = O(P^2)$. On the other hand, if chained directory is used, each block only needs to maintain a pointer to the first processor that caches the block. Each pointer takes $O(\log P)$ bits, thus a total of $O(M \log P) = O(P \log P)$ bits is needed. This saving also makes SCI more scalable than full-map directory.

Compared to full-map directory, chained directory has two disadvantages. First, the time it takes to send an invalidation signal to all processors that have a cache copy of a memory block may be longer when the number of processors is large. The reason is that in full map, the invalidation signal can be sent to all such processors in parallel, whereas with the use of chained directory, the invalidation is propagated through the chain, which can take a long time. Second, the protocol design may be more complicated. Because of the longer delay, race conditions are more likely to arise, which have to be taken into account in protocol design.

Problem 9.11 Different context-switch policies affect the average busy time R.

(a) In switch on cache miss policy, memory access with long latency will be involved. Thus, context switch makes good use of the idle time. The overhead is the time taken to determine whether a cache hit or miss has occurred. If switch on load scheme is used, the aforementioned overhead is eliminated. But R is likely to be smaller than switch on cache miss.

Switch on every instruction interleaves instructions from different contexts on a cycle-by-cycle basis, irrespective of whether a load operation is encountered. The independence among successive instructions can hide pipeline dependences, hence improving pipelined execution efficiency. On the other hand, locality may be jeopardized, which results in a lower cache hit ratio. A scheme which interleaves contexts in blocks of instructions improves the locality of references. But the degree of dependence among successive instructions is higher than that in switch on every instruction scheme. The determination of a suitable block size can be difficult.

(b) Each context-switch scheme has its merits and drawbacks. Thus, more research is needed to determine which one provides the best performance. The choice will depend on other performance parameters as well. For instance, context-switch cost and memory access latency are likely to influence the decision. The behavior of programs should also be taken into account. Both analytical analysis and simulation will be useful in assessing the performance of different models.

Problem 9.12 Dash uses a distributed shared memory architecture which combines the ease of using shared memory and scalability of message-passing systems.

(a) Dash uses an invalidation-based cache coherence protocol. See Fig. 7.15 in the text for the cache states and the events causing transitions from one state to another.

(b) A home cluster maintains the directory and physical memory location of a memory address. Each entry in the directory corresponds to a memory block. It has a presence bit for each processor cache. In addition, a state bit indicates whether the block is uncached, shared, or dirty.

A memory access is satisfied by going through the hierarchy of processor cache, local cluster, home cluster, and finally remote clusters. The directory information makes it possible to send invalidation signals to those processors which have a copy of a memory block instead of broadcasting to all processors. It also helps decide when a memory block needs to be written back to main memory.

(c) See Example 9.5 in the text.

(d) See Example 9.5 in the text.

Problem 9.13

(a) The KSR-1 offers a single-level memory, called *ALLCACHE*. This ALLCACHE design represents the confluence of cache and shared virtual memory concepts that exploit locality required by scalable distributed computing. Each local cache has a capacity of 32 Mbytes (2^{25} bytes). The global virtual address space has 2^{40} bytes.

(b) With ALLCACHE, an address becomes a name, and this name automatically migrates throughout the system and is associated with a processor in a cachelike fashion as needed. Copies of a given cell are made by the hardware and sent to other nodes to reduce access time. A processor can prefetch data into a local cache and poststore data for other cells. The hardware is designed to exploit spatial and temporal locality. When a processor writes to an address, all cells are updated and memory coherence is maintained.

(c) Both systems have distributed main memory, scalable interconnection networks, and directory-based coherence scheme. Dash allows pages to be migrated among processors. DDM has a COMA architecture, which replaces the private memory attached to each node by a huge secondary/tertiary cache, called *attraction memory*. Data blocks can be migrated or duplicated among processors. Processing nodes in both Dash and DDM are clusters of multiple processors. Dash uses a wormhole-routed mesh interconnect, whereas DDM uses a hierarchy of buses. Refer to the papers for more details.

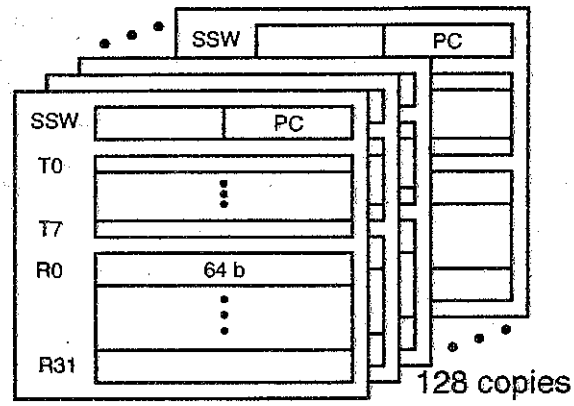
Problem 9.14

(a) Some of the design goals of the Tera architecture are listed below:

- Very high-speed implementations — The architecture should have a short clock period and be scalable to many processors.
- Applicability to a wide spectrum of problems — Programs that do not vectorize well due to a preponderance of scalar operations or too frequent conditional branches should execute efficiently as long as there is sufficient parallelism to keep the processors busy.
- Ease of compiler implementation — The design of the instruction set should simplify the task of the compiler in generating code that can exploit parallelism efficiently.

(b) The interconnection network of one 256-processor Tera system contains 4096 nodes arranged in a $16 \times 16 \times 16$ toroidal mesh; i.e., the mesh "wraps around" in all three dimensions. Of the 4096 nodes, 1280 are attached to the resources comprising 256 cache units and 256 I/O processors. The 2816 remaining nodes do not have resources attached but still provide message bandwidth.

To increase node performance, some of the links are missing. If the three directions are named x, y, and z, then x-links and y-links are missing on alternate z-layers. This reduces the node degree from 6 to 4, or from 7 to 5, counting the resource link. In spite of its missing links, the bandwidth of the network is very large.



Stream Status Word (SSW)
- 32 bit PC (Program Counter)
- Modes (e.g., rounding, lookahead disable)
- Trap disable mask (e.g., data alignment, overflow)
- Condition codes (last four emitted)

No synchronization bits on R0-R31

Target Registers (T0-T7) look like SSWs

(c) Each processor in a Tera computer can execute multiple instruction streams (threads) simultaneously. In the current implementation, as few as 1 or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a thread that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different thread may be issued in each tick without interfering with its predecessors.

When an instruction finishes, the thread to which it belongs becomes ready to execute the next instruction. As long as there are enough threads in the processor so that the average instruction latency is filled with instructions from other threads, the processor is being fully utilized. Thus, it is only necessary to have enough threads to hide the expected latency (perhaps 70 ticks on average); once latency is hidden, the processor is running at peak performance and additional threads do not speed the result.

If a thread were not allowed to issue its next instruction until the previous instruction is completed, then approximately 70 different threads would be required on each processor to hide the expected latency. The lookahead described later allows threads to issue multiple instructions in parallel, thereby reducing the number of threads needed to achieve peak performance.

(d) Each thread has the following states associated with it:

- One 64-bit stream status word (SSW);
- Thirty-two 64-bit general-purpose registers (R0–R31);

- Eight 64-bit target registers (T0–T7).

Context switching is so rapid that the processor has no time to swap the processor-resident thread state. Instead, it has 128 of everything, i.e., 128 SSWs, 4096 general-purpose registers, and 1024 target registers. It is appropriate to compare these registers in both quantity and function to vector registers or words of caches in other architectures. In all three cases, the objective is to improve locality and avoid reloading data.

Program addresses are 32 bits in length. Each thread's current program counter is located in the lower half of its SSW. The upper half describes various modes (e.g., floating-point rounding, lookahead disable), the trap disable mask (e.g., data alignment, floating overflow), and the four most recently generated condition codes.

Most operations have a _TEST variant which emits a condition code, and branch operations can examine any subset of the last four condition codes emitted and branch appropriately. Also associated with each thread are thirty-two 64-bit general-purpose registers. Register R0 is special in that it reads as 0 and output to it is discarded. Otherwise, all general-purpose registers are identical.

The target registers are used as branch targets. The format of the target registers is identical to that of the SSW, though most control transfer operations use only the low 32 bits to determine a new PC. Separating the determination of the branch target address from the decision to branch allows the hardware to prefetch instructions at the branch targets, thus avoiding delay when the branch decision is made. Using target registers also makes branch operations smaller, resulting in tighter loops. There are also skip operations which obviate the need to set targets for short forward branches.

One target register (T0) points to the trap handler which is nominally an unprivileged program. When a trap occurs, the effect is as if a coroutine call to a T0 had been executed. This makes trap handling extremely lightweight and independent of the operating system. Trap handlers can be changed by the user to achieve specific trap capabilities and priorities without loss of efficiency.

(e) The Tera architecture uses a new technique called *explicit-dependence lookahead.* Each instruction contains a 3-bit lookahead field that explicitly specifies how many instructions from this thread will be issued before encountering an instruction that depends on the current one. Since seven is the maximum possible lookahead value, at most 8 instructions and 24 operations can be concurrently executing from each thread.

A thread is ready to issue a new instruction when all instructions with lookahead values referring to the new instruction have completed. Thus, if each thread maintains a lookahead of seven, then nine threads are needed to hide 72 ticks of latency.

(f) The Tera uses multiple contexts to hide latency. The machine performs a context switch every clock cycle. Both pipeline latency (eight cycles) and memory latency are hidden in the HEP/Tera approach. The major focus is on latency tolerance

rather than latency reduction.

With 128 contexts per processor, a large number (2K) of registers must be shared finely between threads. The thread creation must be very cheap (a few clock cycles). Tagged memory and registers with full/empty bits are used for synchronization. As long as there is plenty of parallelism in user programs to hide latency and plenty of compiler support, the performance is potentially very high.

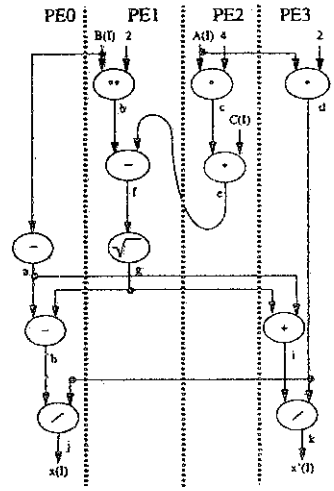
However, these Tera advantages may be embedded in a number of potential drawbacks. The performance can be bad for limited parallelism, as in the case of single-context environments. On the other hand, a large number of contexts (threads) require lots of registers and other hardware resources which in turn implies higher cost and complexity. Finally, the limited focus on latency reduction and cacheing entails a high degree of parallelism and a high memory bandwidth in order to hide latency; both tend to drive up the cost in building the machine.

Problem 9.15

(a) Static dataflow computers do not allow more than one token to reside on the same arc of a dataflow graph. The firing rule for an operator node is that all the input tokens are present and there is no token on the output arc(s). The implementation requires extensive acknowledge signals.

Dynamic dataflow computers allow more than one token to be on the same arc simultaneously. Each token is associated with a tag. When tokens of identical tags are present on all the input arcs of an operator, it is fired.

(b) The root of $A_i x_i^2 + B_i x_i + C_i = 0$ can be computed as $x_i = \dfrac{-B_i \pm \sqrt{B_i^2 - 4A_i C_i}}{2A_i}$. The dataflow graph is shown in the following diagram for any i.



There are 11 nodes, each with two input arcs and one outpnt arc. The output tokens of the nodes are labeled a through k.

(c) Partition of the computations among the PEs is shown in the above diagram. The partition is not unique for achieving a balanced load among processors. Suppose each computation takes one clock cycle. Three of the PEs execute three computations and the fourth one (PE2)executes two computations. The average latency of one iteration is 3 when the computation reaches the steady state. The schedule is shown in the following table with the subscript of each output token corresponding to the loop index i.

PE0	PE1	PE2	PE3
a_1	b_1	c_1	d_1
		e_1	
	f_1	c_2	
	g_1	e_2	
h_1	b_2		i_1
j_1	f_2	c_3	k_1
a_2	g_2	e_3	d_2
h_2	b_3		i_2
j_2	f_3	c_4	k_2
a_3	g_3	e_4	d_3
		⋮	

Problem 9.16

(a) For each $y(i)$, m multiplications and $m-1$ additions need to be performed, giving a total of mn multiplications and $(m-1)n$ additions.

(b) The computations of individual elements of y are independent of each other. Hence, the computations can be partitioned as follows: $y(i)$ are computed by processor 0 for $i = 0..63$, by processor 1 for $i = 64..127$, by processor 2 for $i = 128..191$, and by processor 3 for $i = 192..255$.

(c) Using the above partition, each processor will need to have 67 elements of x and all four elements of w for circular convolution. For instance, processor 0 needs $x(0)$ through $x(63)$ and $x(253)$ through $x(255)$. Note that the extra 3 elements to be fetched into each processor reside in memory modules 29, 30, and 31, respectively. This is a result of how the vector elements are stored and the nature of circular convolution. Therefore, proper interleaving is required to avoid conflicts. This interleaving is facilitated by the assumption of enough registers in each processor so that memory access and arithmetic operations can be performed in separate phases.

The ith elements of x and y are stored in memory module $j = i \bmod 32$. Elements of vector w are stored in a similar fashion in memory modules 0 through 3. With the storage scheme, each memory module stores 8 elements of vector x.

Modules 0 through 28 will be accessed 8 times each. Modules 29 through 31 will be accessed 12 times due to access contentions described in the above. To fetch w into all the four processors takes 4 cycles. The access of 67 elements of x into each of the four processors takes another 67 cycles.

Computations of y can be carried out concurrently in all four processors, each responsible for 64 elements, resulting in a total of $4 \times 64 + 3 \times 64 = 448$ cycles. Finally, the elements of y are stored back to the memory at a rate of 4 elements per cycle, taking 64 cycles. Therefore, the total parallel computing time is

$$t_4 = 4 + 67 + 448 + 64 = 583 \text{ cycles.}$$

(d) If a single processor is used, the following steps are required:

- Fetch w from memory in 1 cycle,
- Fetch x from memory in 64 cycles,
- Compute y in $256 \times (4 + 3) = 1792$ cycles,
- Store y into memory in 64 cycles.

Thus, the execution time by a single processor is

$$t_1 = 1 + 64 + 1792 + 64 = 1921 \text{ cycles.}$$

The speedup using 4 processors over a single processor is

$$t_1/t_4 = 1921/583 \approx 3.3.$$

Problem 9.17

(a) A fine-grain processor typically has a small amount of memory associated with it. In the construction of large-scale computer systems, fine-grain processors match better with fine-grain software parallelism and have cost advantage over medium-grain processors.

(b) In a uniprocessor system, there is only a single address space. Many programs have been developed based on this concept. A single global address space offers a continuity of the perception and can simplify the program development process as the programmer does not need to worry about the message-passing mechanisms on individual machines. It also simplifies data partitioning and dynamic load balancing and improves the portability of programs across machines with different architectures.

(c) Because of high synchronization cost, coarse-grain parallelism necessitates the allocation of a large chunk of computations, such as several iterations, to each processor. As a result, low-level parallelism such as individual iteration or instruction is not fully exploited. From scalability point of view, as the number of processors is increased, it is important to take advantage of such low-level parallelism in order to reduce solution time and improve processor utilization. The consideration favors the use of fine-grain parallelism over medium- or coarse-grain parallelism.

Chapter 10

Parallel Models, Languages and Compilers

Problem 10.1

(a) In synchronous message-passing, the sender and the receiver must be synchronized in time and space. In other words, a communicaiton channel must be established before message passing can convene, much like communication over a telephone line. No buffering is needed on the channel.

In the case of asynchronous message-passing, it is not necessary to coordinate the sender and receiver. A message is delivered to the channel and may be stored in the buffers on the channel or a global mailbox before arriving at the sender. In this scheme, an acknowledge from the receiver is needed to signal the correct receipt of a message.

(b) In synchronous message passing, if a channel cannot be established between two communicating processes, the message will be blocked, which in turn will block the execution of the processes involved. On the other hand, in asynchronous message passing, as long as the channel buffer is sufficiently large, the transmission of messages and execution of processes will not be blocked. Therefore, it offers better resource utilization and potentially shorter communication delays.

(c) Rendezvous is a scheme adopted in Ada for synchronous message passing. In this scheme, a sender or receiver arriving earlier at the rendezvous has to wait for the arrival of the other before they can proceed to exchange messages.

(d) In a name-addressing scheme, a sender or receiver process is identified by the process ID and the node in which it resides. This convention is adopted by Ada. In a channel-addressing scheme, a path is established between a sender and a receiver process by specifying the channels connecting the nodes in which the processes reside.

(e) In asynchronous message passing, the sender and receiver processes are effectively uncoupled from each other via the use of mediaries such as channel buffers or a global mailbox. Through this uncoupling, both processes can execute more freely, leaving the transmission of messages to be handled by the mechanisms provided by the communication channels.

(f) Both interrupt and lost messages can occur in asynchronous message-passing systems. An interrupt message differs from a regular message in that it has to be handled immediately by the receiver process, even though the receiver may not expect to receive it. After it is serviced, the interrupted process can resume its execution.

Lost messages are those directed to a wrong process or node and eventually are lost. It is important to design effective detection and debugging facilities to redirect lost messages to the correct receivers to ensure smooth program execution.

Problem 10.2 The idea is to add fork-join primitives into the code to allow parallel execution of the program. Different concurrent Lisp languages, such as Multilisp, Qlisp, Symmetric Lisp, and Connection Machine Lisp, have different syntax and semantics. A Lisp-like code segment based on concurrent object-oriented model can be found in [Agha90]. In practice, Lisp language available on an accessible machine should be used to write a program to carry out the computations. Performance data can then be collected and analyzed.

Problem 10.3

(a) C* is a data parallel language developed by Thinking Machines. It provides high-level constructs for parallel computing on SIMD machines. Quinn and Hatcher described compiling and various optimization techniques to convert a program written in C* to one in C for execution on SPMD or MIMD machines. Four issues were addressed in their paper:

- how to infer message-passing requirement?
- how to support synchronization requirement?
- how to emulate a large number of PEs efficiently on a machine without hardware support for virtual processors?
- how to minimize message-passing cost?

Several methods to deal with these problems were discussed by the authors, including reduction of synchronization and message-passing. For instance, in order to reduce message-passing cost, instruction and data can be replicated on all nodes. Also, data exchange can be carried out in blocks instead of bytes to reduce startup overhead. Their experiments with Gaussian elimination on an nCube 3200 showed that a C program generated from translation of C* code with message optimization was comparable in quality to a hand-coded C program.

(b) SIMD mode is synchronous in that all active PEs execute the same operations in a lockstep fashion. It is especially suitable for data parallel computations. In SPMD

mode every PE executes the same program in an asynchronous manner. PEs coordinate with each other at synchronization points but otherwise each PE works at its own pace between those points. Synchronization is achieved by message passing among processors. Asynchronous algorithms executed in SPMD mode are prone to time-dependent errors. In contrast, SIMD execution has simple flow control, and the computation results are deterministic regardless of the number of PEs. However, not all applications are suitable for execution in SIMD mode.

(c) See the optimization described in the paper for Gaussian elimination and conduct similar optimization for FFT after an analysis of the program flow.

Problem 10.4

(a) Multiprogramming refers to the interleaved execution of multiple independent programs on a uniprocessor or multiprocessor system through time sharing. Its use is intended to overlap CPU and I/O operations among programs to improve resource utilization.

(b) Multiprocessing is multiprogramming implemented at process level on a multiprocessor. If interprocessor communications are handled at instruction level, the mode of operation is MIMD multiprocessing and exploits fine-grain parallelism.

(c) Multiprocessing, in which interprocessor communication takes place at program, procedural, or subroutine level, is characterized as operating in MPMD mode. In this mode, coarse-grain parallelism is exploited.

(d) When a single program is divided into several interrelated tasks which can be executed concurrently on a multiprocessor, the mode of operation is referred to as multitasking.

(e) Multithreading is a refinement of multitasking and multiprocessing concept. A task can create multiple threads which are executed on one or more processors at the same time. Since threads are lightweight processes with minimum state and register information, context switching is much faster than in multiprogramming.

(f) Program partitioning refers to the decomposition of a large program and data sets into small pieces which can be executed in parallel on multiple processors.

Problem 10.5

(a) 1. A(5,8,1), A(5,9,1), A(5,10,1), A(5,8,2), A(5,9,2), A(5,10,2), A(5,8,3), A(5,9,3), A(5,10,3), A(5,8,4), A(5,9,4), A(5,10,4), A(5,8,5), A(5,9,5), A(5,10,5).

2. B(3,5), B(3,6), B(3,7), B(3,8), B(6,5), B(6,6), B(6,7), B(6,8), B(9,5), B(9,6), B(9,7), B(9,8).

3. C(1,3,4), C(2,3,4), C(3,3,4).

(b) 1. Yes. The number of elements is the same in each dimension of the source and destination arrays.

2. No, because the two arrays have different sizes in the first dimension.
3. No, because the two arrays have different dimensions.
4. Yes.

Problem 10.6

(a) Flow dependence between statements S_1 and S_1 in successive iterations of J-loop. The distance vector is (0,1), and the direction vector is (=, <).

(b) Flow dependence between statements S_1 and S_2. The distance vector is (0,0), and the direction vector is (=,=).

(c) Antidependence between statements S_2 and S_3 in successive I-loop. The distance vector is (−1,0), and the direction vector is (>,=).

Problem 10.7

(a) $S_1 \longrightarrow S_2 \dashv S_3$.

(b) The vectorized code is as follows:

```
A(1:N) = B(1:N)
E(1:N) = C(2:N+1)
C(1:N) = A(1:N) + B(1:N)
```

Note that it is necessary to store the original value of C in E before C is overwritten. Therefore, the order of statements S_2 and S_3 in the original loop is reversed in the vector code. It is also permissible to interchange the first two vector statements, since they are independent.

Problem 10.8

(a)
1. There is flow dependence on variable A between statements S_1 and S_3 in successive iterations of J-loop. The distance vector is (0,1), and the direction vector is (=, <).
2. There is flow dependence on variable E between statements S_3 and S_3 in successive iterations of J-loop. The distance vector is (0,1), and the direction vector is (=,<).
3. There is antidependence on variable C between statements S_1 and S_2 in the same iteration. The distance vector is (0,0), and the direction vector is (=,=).

(b) There is no data dependence among different I-loop iterations. Therefore, they can executed in parallel. The compiler can preschedule the iterations of the I-loop into P processors in contiguous blocks as follows:
processor 1 executes iterations 1, 2, ..., $\lceil N/P \rceil$;
processor 2 executes iterations $\lceil N/P \rceil + 1$, $\lceil N/P \rceil + 2$, ..., $2\lceil N/P \rceil$;

⋮

Alternatively, every Pth iteration can be assigned to the same processor:
processor 1 executes iterations 1, $P+1$, ..., $2P+1$;
processor 2 executes iterations 2, $P+2$, ..., $2P+2$;

⋮

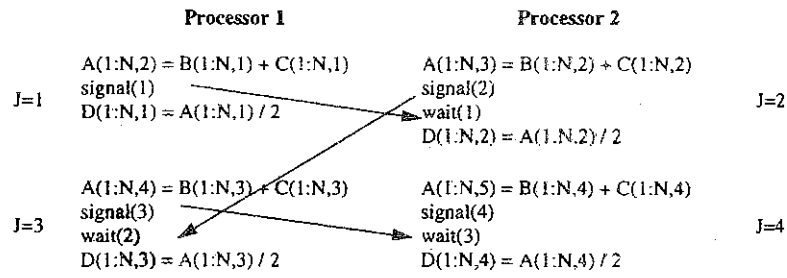
Problem 10.9

(a) The loop can be compiled with the I-loop in vector mode, which will generate stride-1 memory operations.

(b) The loop can be compiled for parallelization in the J-loop as follows:

```
      Doacross J = 1, N
S1:       A(1:N, J+1) = B(1:N, J) + C(1:N, J)
          signal(J)
          if (J > 1) wait (J-1)
S2:       D(1:N, J) = A(1:N, J) / 2
      Endacross
```

The parallel execution is illustrated in the following diagram for the case of two processors.



(c) The code for parallelized I-loop with vectorized J-loop execution is shown below:

```
      Doall I = 1, N
S1:       A(I, 2:N+1) = B(I, 1:N) + C(I, 1:N)
S2:       D(I, 1:N) = A(I, 1:N) / 2
      Endall
```

Problem 10.10

(a) Loop permutation is a transformation which interchanges the order of nesting in a multi-nested loop structure.

(b) Loop reversal reverses the sign of a loop index, including the sign of starting and ending values and any expression involving the particular index.

(c) Loop skewing of I_j by an integer factor f with respect to loop I_i transforms iteration $(p_1, ..., p_{i-1}, p_i, p_{i+1}, \cdots, p_{j-1}, p_j, p_{j+1}, \cdots, p_n)$ to $(p_1, ..., p_{i-1}, p_i, p_{i+1}, \cdots, p_{j-1}, p_j + fp_i, p_{j+1}, \cdots, p_n)$.

The above three transformations can be formulated as elementary matrix operations. See the text for matrix representations and examples.

(d) Loop tiling refers to various techniques of breaking iterations into small blocks to obtain coarser granularity which can reduce synchronization overhead and improve data locality. Typically an n-deep loop is converted into a $2n$-deep loop, where the inner n loops are determined by the tile size used.

(e) Wavefront transformation is a technique to maximize the degree of parallelism in n fully permutable loops with dependences. The idea is to skew the innermost in the nest with respect to each of the other loops and then move the innermost loop to the outermost position. See text for examples.

(f) Locality optimization is used to reduce memory access penalties. The idea is to improve the reuse of data once it is brought into a level of the memory hierarchy which is closer to the processor. Such techniques as loop interchange, instruction and data prefetching, and tiling can be used to achieve the goal.

(g) Software pipelining is the pipelining of successive iterations of a loop in a source program. It is particularly suited to deep hardware pipelines and can be used with either Doall or Doacross loops. Similar to hardware pipelining, it is desirable to minimize instruction initiation latency.

Problem 10.11

(a) In iteration I, $A(I)$ is updated by the value of $A(I + 1)$. The value of $A(I + 1)$ is not updated until the $(I + 1)$st iteration, which has not been executed yet. In general, with forward LCD, the reference to an element always occurs before its value is updated in a later iteration. This type of operations can be vectorized. In effect, the computations in the loop add a scalar constant 3.14159 to each element of A and then shift them forward by one position. In other words, the loop is equivalent to the following vector code:

```
V(1:N) = A(2:N+1) + 3.14159
A(1:N) = V(1:N)
```

(b) The assignment to $A(2)$ in the second iteration depends on the value assigned to $B(2)$ in the first iteration. The compiler can interchange the statements within the loop so that the assignment to B occurs before the assignment to A, as shown below:

```
Do I = 1, N - 1
```

```
    B(I+1) = D(I) * 3.14159
    A(I) = B(I) + C(I)
Enddo
```

The code can be vectorized as follows:

```
B(2:N) = D(1:N-1) * 3.14159
A(1:N-1) = B(1:N-1) + C(1:N-1)
```

Problem 10.12

(a) This program can be vectorized as follows:

```
TEMP(1:N) = A(1:N)
A(2:N+1) = TEMP(1:N) + 3.14159
```

(b) The code cannot be directly vectorized or parallelized because of the carry-around variables S and X. To see this, consider the following parallel code:

```
Doall I = 1, N
    If (A(I) .LE. 0.0) then
        S = S + B(I) * C(I)
        X = B(I)
    Endif
Enddo
```

If all processors are allowed to proceed concurrently, the values of S and X will be nondeterministic. On the contrary, the serial code will give a definite answer for S and X.

However, if intermediate vectors are introduced to store the value of $B(I) * C(I)$, then some vector or parallel processing can be achieved. This is illustrated in the following code for performing the conditional inner-product operation:

```
D(1:N) = 0
where (A(1:N) .LE. 0.0) do
    D(1:N) = B(1:N) * C(1:N)
endwhere
```

See [Wolfe89] for more details. The elements of D can then be summed up in parallel using a binary tree computing structure to obtain S. Alternatively, S can be obtained by a vector reduction operation as follows:

```
S = sum(D(1:N))
```

Similarly, the determination of X in the original loop can be vectorized. Let vector P be initialized so that $P(I) = I$ for $I = 1..N$, and Q is a zero vector. The following vector code yields the desired result:

```
where (A(1:N) .LE. 0.0) do
    Q(1:N) = P(1:N)
endwhere
K = max(Q(1:N))
X = B(K)
```

In the above, *max* is a vector reduction function which finds the maximum value of a vector. Of course, the performance of the vector code will depend on how fast vectors P and Q can be generated. Typically, P and (initial) Q are created at compile time since their elements are fixed. Then the cost can be amortized over a large number of executions of the vector code.

Problem 10.13 Tanenbaum et al. proposed a layered approach to provide a uniform interface for parallel programming. The approach is insensitive to machine architecture and can be used with multiprocessors or multicomputers. Besides architecture-transparency, the other goal is to maintain a good performance in a distributed shared memory system. The two major components of the system are shared objects and reliable broadcasting. An object is an abstract data type with well-defined operations. For instance, an object can be a data structure with read and write operations.

An object that is shared by multiple processes are replicated for each process that needs to access it. When a process performs a read operation on a shared object, it is treated as an operation on a private object and can be done locally with proper synchronization. When a write operation is performed on a shared object, the updated value needs to be sent to other processes via the reliable broadcasting mechanism.

In general, *read* operations occur much more frequently than *write* operations. Therefore, replicating and sharing data objects can be effective. Moreover, the low overhead associated with reliable broadcasting (at most 2 sends for each message) allows the system to scale up in performance. Consult [Tanenbaum92] for more details about the broadcasting protocols and object management schemes.

Chapter 11

Parallel Program Development and Environments

Problem 11.1

(a) In busy wait, a process waiting for an event remains loaded in the context registers of a processor and keeps trying to get into a critical section. In sleep wait, a waiting process is removed from the processor and put in a wait queue. Later on, after the event it is waiting for takes place, the suspended process is awaken and rescheduled.

(b) In sleep wait, a policy is needed to select one of the suspended processes in the wait queue to be revived. The policy must ensure that all suspended processes in the queue are treated fairly. That is, no process should be suspended for an extraordinary amount of time compared to others. For instance, a first-come-first-served revival policy is a fairness policy.

(c) Lock is a mechanism used to implement presynchronization in which a requester process is required to obtain sole access to an atom (shared writable object) before performing an operation to update it. The purpose is to avoid concurrent updates to an object.

(d) Optimistic synchronization (or postsynchronization) allows an atom to be updated before sole access is granted to a requester process. This is achieved in two steps. First, the requester modifies a local version of the object. Second, it checks to see if there has been a concurrent update to the global version. If so, the local update is aborted; otherwise, the global version is updated.

(e) In server synchronization, each atom is associated with an update server. Any process that wishes to perform an atomic operation on an atom has to do so by sending a request to the server. This approach is often adopted in object-oriented systems to provide data encapsulation. The corresponding synchronization environment is often more user friendly as the user does not need to know or worry about the

implementation details of mutual exclusion mechanisms for synchronization. This strategy is adopted in monitors for synchronization and can be implemented as server daemons.

Problem 11.2

(a) Lock is a mechanism used to ensure sole access to a critical section. If a spin lock is used, a process waiting to enter the critical section will keep on trying until it gains access. In the case of a suspend lock, once a process is denied access to the critical section, it is suspended and put in a queue. Suspended processes will be activated one by one when access to the critical section is allowed. Suspend lock allows more efficient use of the processor than spin lock but care must be taken to guard against indefinite waiting for some processes.

(b) Dekker's algorithm for synchronization ensures mutual exclusion and avoids unnecessary waiting. To accomplish this, each process uses a flag to indicate whether it desires to enter the critical section. To achieve mutual exclusion, each process checks whether there is another process in the critical section. If so, it backs off. The following algorithm is described in [Silberschatz88]. It uses an array flag$(0 : n-1)$ to indicate the status of the processes. Each element of the array can assume three values: idle, in, out. A global variable *turn* is used to select a process between 0 and $n-1$. Initially, all the elements of the flag array is set to idle and *turn* can assume any valid value. An auxiliary integer variable j is also used in the algorithm. In this algorithm, each process i, $0 \leq i \leq n-1$, executes the following code:

```
repeat
        flag(i) = out;
        j = turn;
        while j ≠ i do
                if (flag(j) == idle) then
                        j = j+1 mod n;
                else j = turn;
                endif
        enddo
        flag(i) = in;
        j = 0;
        while (j < n) && (j == i || flag(j) != in) j = j+1 mod n;
until j ≥ n;
turn = i;
        critical section
flag(i) = idle;
j = i+1 mod n;
while (j ≠ i && flag(j) == idle) j = j+1 mod n;
turn = j;
exit {critical section }
```

Note that initially it is possible for several processes to set their flags to *in* at the same time. If that happens, all of these processes will be forced to reset their flags to *out*. On the second try, only one of them will be able to enter and set its flag to *in*; the others will be blocked and spin wait. When an incumbent process exits the critical section, it selects the next process to enter the critical section in an orderly manner. This guarantee that any process wishing to enter the critical section will be able to do so after at most $n - 1$ tries.

(c) The generalized Dekker's algorithm can be implemented using Test&Set. Each process is associated with a flag, which can be examined and/or changed by all the processes. In addition, each process has a local variable *key*, which can only be updated by the owing process. A global variable *lock* is used to guard the entrance to a critical section. Initially, all the flags are set to false. Each process i wishing to enter the critical section executes the following code, also adapted from [Silberschatz88]:

```
flag(i) = true;
key = true;
while (flag(i) && key) key = Test&Set(lock); End
      flag(i) = false;
      critical section
j = i + 1 mod n;
while (j ≠ i && flag(j) == false) j = j+1 mod n; End
if (j == i) lock = false;
else flag(j) = false;
endif
```

This code uses atomic Test&Set operation to ensure mutual exclusion of the critical section. The method used to select the next process is similar to the software algorithm in (b).

Problem 11.3

(a) A binary semaphore is a variable which can assume value 1 or 0. It has two associated operations P and V, corresponding to wait and signal. A process wishing to enter a critical section first performs a P operation to see if another process is already in the critical section. If that is the case, it is blocked. When a process leaves the critical section, it performs a V operation, thus allowing a waiting process to be awaken and enter the critical section. A binary semaphore is initialized to 1 to allow the first process to enter the critical section without waiting. It can be implemented in hardware using atomic operations such as Test&Set.

(b) Monitor is a high-level construct that encapsulates shared variables and associated procedures into a module. A monitor consists of (1) local variables, (2) procedures that manipulate local variables, global variables, and parameters passed from calling processes, and (3) initialization of local variables. Only the values of local variables can be changed by the procedures. Also, only one process is allowed to

be in the monitor at a time. Thus mutual exclusion mechanism is embedded in the construct. Monitor relieves individual processes of the need to take care of mutual exclusion in the code and reduces the possibility of errors. For instance, in the use of binary semaphore, if the semaphore is not initialized to 1, processes that wish to enter the critical section will hang up indefinitely. With the use of monitors, the debugging process is simplified by getting rid of inadvertent mistakes.

Problem 11.4 Let the philosophers and forks both be numbered 0 to 4. The fork to the right of philosopher i is fork i and the one to his left is fork $(i-1)$ mod 5.

(a) Let forks(0:4) be the semaphores associated with the forks and all its elements are initialized to 1 at the beginning.

In the fetch protocol, an even-numbered philosopher first picks up the fork to his right and then the one to his left. An odd-numbered philosopher first picks up the fork to his left and then the one to his right. In the release protocol, both forks are put down in a random order.

Fetch protocol

```
if (i mod 2 == 0) then
    P(fork(i));
    P(forks((i−1) mod 5));
else
    P(forks((i−1) mod 5));
    P(forks(i));
endif
```

Release protocol

```
V(forks((i−1) mod 5));
V(fork(i));
```

This protocol allows a philosopher to hold a fork while waiting for the other. Deadlocks are avoided by breaking circular waits among the philosophers which is a necessary condition for deadlock to occur. Based on the protocol, at least one philosopher will be able to eat at any moment. Moreover, a philosopher will pick up the first fork as soon as it becomes available instead of waiting until both forks on his sides are available. This prevents conspiracy between two philosophers to starve a third philosopher seated between them. Therefore, starvation is also avoided.

(b) The above fetch and release protocols can be implemented using monitor as follows:

```
Monitor dining-philosophers
  forks(0:4): condition;

  procedure fetch(i)
    begin
      if (i mod 2 == 0) then
```

```
            wait(fork(i mod 5));
            wait(fork(i−1 mod 5));
        else
            wait(fork(i−1 mod 5));
            wait(fork(i mod 5));
        endif
    end

procedure release(i)
    begin
        signal(fork(i−1 mod 5));
        signal(fork(i mod 5));
    end
```

Problem 11.5 A set of processes are in a state of deadlock when every process in the set is waiting for resources held by another process in the set. According to the definition, we know that the four conditions — hold and wait, no preemption, mutual exclusion, and circular waiting — must hold at the same time to cause a deadlock. If any of the conditions is false, then deadlock can be prevented. For example, if a resource is sharable by more than one process or can be preempted, then there is no need to wait for the resource. Circular waiting is implied in the definition of deadlock. Finally, if a process does not hold resources while waiting for others, these resources can be used by other processes, thereby breaking the stalemate situation.

When all the four conditions hold simultaneously, a deadlock situation will potentially occur. But a deadlock can often be averted by properly revising the resource allocation diagram to eliminate circular waiting.

Deadlock prevention refers to the use of suitable protocols to ensure that at least one of the four necessary conditions for deadlock will not hold and thus the occurrence of deadlock is prevented.

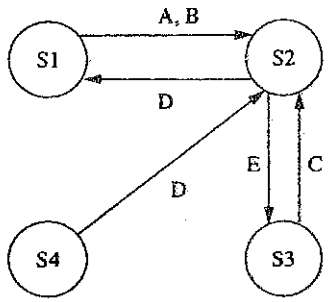
Deadlock avoidance refers to the management of resources so that situations that may lead to deadlock will be avoided. Usually it is achieved by dynamically keeping track of the resources available, allocated, and requested. The operating system closely monitors the usage of resources to avoid deadlocks.

Deadlock detection is a systematic approach for detecting whether a deadlock situation is present. When no deadlock prevention or avoidance measure is employed, deadlocks may occur and need to be detected so that a deadlock recovery algorithm can be invoked.

When a deadlock is detected, a deadlock recovery strategy is used to break it. Two options are often adopted. One is to kill one or more of the deadlocked processes to remove the circular waiting. The other is to preempt some of the resources held by one or more of the deadlocked processes.

Problem 11.6

(a)



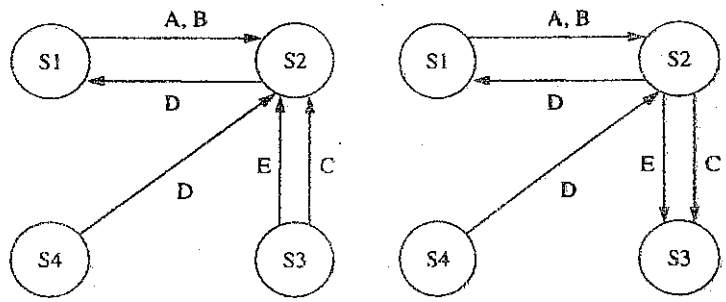
- B and D do not cause deadlock because only after B releases S_4 can D claim S_4. If A is executed before B, there will be no deadlock between A and D, either. But if B is executed before A, then A and D can enter a deadlock, with D holding S_4 and S_2 while waiting for S_1 and A holding S_1 while waiting for S_2.
- C and E can be in deadlock. After C gets S_2 ($P(S_2)$) and E gets S_3 ($P(S_3)$), C claims S_3 and E claims S_2 which can never be satisfied.

(b) If C and E are deadlocked, A, B, and D will be blocked indefinitely. If A and D are deadlocked, C and E will be blocked indefinitely.

(c) It depends on the race conditions. For instance, If C (or E) can secure both S_2 and S_3 before E (or C), it will have all the needed resources. After it finishes execution, both resources are released so that E (or C) can proceed. Thus, deadlock is avoided. Similarly, the deadlock between A and D also depends on race condition.

(d) The deadlock between C and E can be prevented by either of the following two options which alter the resource usage in C and E:
in C: $P(S_3)$; $P(S_2)$; ... or
in E: $P(S_2)$; $P(S_3)$; ...

The resulting resource allocation graphs are shown below; there is no circular wait between C and E now.



Problem 11.7

(a) Suppose on the disk there are n cylinders numbered 0 through $n-1$ starting from the innermost one. An "elevator" algorithm is used in the scheduling. The idea is to continue sweeping in inbound or outbound direction until all requests in that direction have been serviced. Then the sweeping direction is reversed. For details, see [Bic88].

When a request is made and the disk head is busy, the request is put in either of two queues, one (*insweep*) corresponds to inward movement and the other (*outsweep*) to outward movement of the disk head. The queued requests are served according to the position of the destination cylinder.

The scheduler can be implemented by a monitor with conditional wait. See [Silberschatz88]. If a request is put in the *outsweep* queue, the distance between the destination and innermost cylinders (*dest*) is stored with the request. If a request is put in the *insweep* queue, the distance between the destination and outermost cylinders ($n - dest$) is stored. The requests are then serviced in the order determined by the number: the smaller the number, the earlier a request is serviced. Clearly, the motivation for the policy is to reduce the movement of the disk head. The following monitor implementation is adapted from [Bic88].

```
Monitor disk-scheduler
    type direction = (in, out);
    dest, pos: integer;
    dir: direction;
    busy: boolean;
    incount, outcount: integer;
    insweep, outsweep: condition;
    procedure request(dest);
        begin
            if busy
                if (pos < dest) || (pos == dest && dir == out)
                    outsweep.wait(dest);
                    outcount = outcount + 1;
                else
                    insweep.wait(n - dest);
                    incount = incount + 1;
                endif
            else
                busy = true;
                pos = dest;
            endif
        end

    procedure release;
        begin
            busy = false;
```

```
            if dir == out
               if outcount > 0
                    outcount = outcount - 1;
                    outsweep.signal;
               else
                    dir = in;
                    incount = incount - 1;
                    insweep.signal;
               endif
            else
               if incount > 0
                    incount = incount - 1;
                    insweep.signal;
               else
                    dir = out;
                    outcount = outcount - 1;
                    outsweep.signal;
               endif
            endif
        end

    begin
        dir = in; pos = n-1; busy = false;
        incount = 0; outcount = 0;
    end
```

In the above program, the syntax of the signal and wait instructions is slightly changed to accommodate the priority parameter.

(b) A user process can access data on the disk by the following sequence:

```
request(cyl_num);
call driver procedure to transfer the data
release
```

The cyl_num indicates the location on the disk where the requested data resides. It can be generated by the file server from user-specified information.

Problem 11.8 A monitor for a barrier counter can be specified as follows:

```
Monitor barrier-counter
   var counter: integer;
       flag: condition;
   procedure block(n)
       begin
           counter = counter + 1;
           if (counter == n) then
```

```
            begin
                for (i = 1; i < n; i++)
                    signal(flag);
            end
        else
            wait(flag);
        endif
    end

begin
    counter = 0;
end
```

Suppose n processes need to be synchronized at the barrier. Each process calling the procedure in the monitor passes n as a parameter. This arrangement is better than using a global variable since there might be several barrier synchronization points in the program with different numbers of processes. Whenever a process reaches the barrier, the counter is incremented. If a process is not the last one, it is blocked. When the last process arrives, all the $n-1$ blocked processes are awaken one by one.

Problem 11.9

(a) A perfect decomposition is a decomposition of the computation task of an application into a set of processes with several features: (1) the amount of computation in each process is approximately equal; (2) there is little interaction among the processes. When these processes are mapped to the nodes of a multicomputer system, the load on the nodes will be highly balanced, and low communication overhead among the nodes is incurred. As a result, the efficiency tends to be high.

When data decomposition is used, the same code is executed in all the nodes of a multicomputer on different data. In this case, program replication can be used to duplicate the code. Program partitioning is often used in functional decomposition paradigm and requires extensive precedence and flow analysis. In contrast, program replication is straightforward and can be easily used to exploit data parallelism in certain applications.

(b) Different decomposition techniques are compared below:

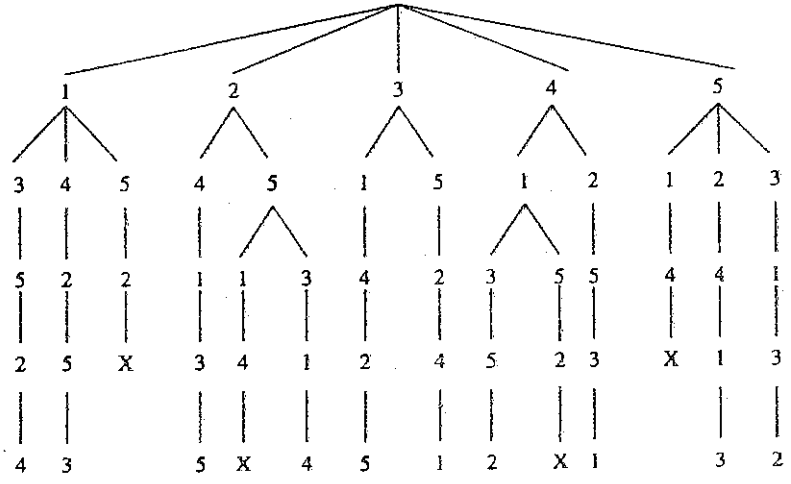
- Domain. Domain decomposition applies to domains and data structures that are regular, as typical in scientific computations. Control decomposition is used with domains that are irregular or may change dynamically, as in artificial intelligence type of applications. Object-oriented programming provides a hierarchical mechanism for data encapsulation and protection. It is used in domains where there is a high degree of sharing of data and code among different processes.
- Algorithms used. Algorithms used in domain decomposition are composed of uniform functional modules which exploit divide-and-conquer parallelism. Algorithms in control decomposition consist of disparate functional mod-

ules, usually corresponding to different phases of computations in an application, such as image understanding. Functional modules are chained to utilize pipeline style of parallelism. Algorithms in object decomposition manipulate objects through creation, destruction, and behavior replacement. They can take advantage of divide-and-conquer and pipeline parallelism.

- Flow of control. Domain decomposition has uniform flow of control, typically consisting of computations in individual nodes, followed by interprocessor communication, during which nodes exchange data and synchronize with each other. In control decomposition, the flow of control is like a pipeline among functional modules. The interaction between functional modules can be more complex and less predictable. The flow of control in object decomposition is effected through the creation of new objects, the change of individual object states, and exchange of messages among objects.

Problem 11.10

(a) An N-queens program searches the tree generated by different ways of placing the queens for legitimate placements. When there is a conflict in some placements, the search of a branch may be terminated prematurely without going down to the leaf nodes. This is illustrated for $N = 5$ in the following diagram. The placements proceed from column to column. Each level in the tree corresponds to one column of the board. Thus, each number in the diagram represents a row position in the corresponding column of the board.



The search of branch 1–5–2 is terminated at the mark X without going further down. A strategy can be developed which prunes unpromising branches to improve performance.

(b) In a parallel program for the N-queens problem, nodes can be assigned to search different branches of the tree, depending on the number of nodes available. In the

above diagram, 12 processors can be used to search the second-level branches. Invalid configurations are discarded once they are detected. The searches are largely independent of each other and the degree of concurrency is high.

A global counter can be used to record the number of legitimate placements. But this is likely to degrade performance. An alternative approach is for each node to maintain a local counter. At the end, the values in the local counters are added. It is informative to actually write the code, run it on a message-passing system, and analyze the performance results.

Problem 11.11

(a) At the beginning, an initial configuration of path is obtained. Cities on the path are distributed among the nodes. In the work of Kallstrom and Thakkar, the nodes are connected in a ring configuration. The algorithm is based on simulated annealing technique for optimization.

During each iteration, pairs of cities are swapped within each node and among neighboring nodes through message passing. When a swapping leads to a shorter path, it is accepted; otherwise, it might be accepted with a certain probability. When the temperature is high, it is likely to accept a configuration which leads to an increase in the path length. As temperature drops, it becomes less and less likely.

Concurrency occurs in the simultaneous swapping and subsequent computation of path lengths in all the active nodes. Global synchronization takes place only at each temperature drop.

(b) Performance analysis should be based on actual implementation of the algorithm on a real machine. Several parameters such as the number of cities, the rate of temperature drop, and the acceptance probability functions can be adjusted to see the effect on performance.

Problem 11.12 Multitasking evolves from macrotasking to microtasking and to autotasking on Cray computers. See [Furtney92].

- Macrotasking requires the modification of a program by inserting explicit calls to library subroutines. The library routines interact with OS and performs two types of functions: task creation and manipulation and synchronization.
- Microtasking is intended to exploit fine-grain parallelism such as that in Do loops. It employs master/slave relationship among CPUs. When the master enters a region amenable to parallel execution, slaves are invoked. Programmers determine where parallelism exists and place directives into programs. Loop iterations are handed out to next available CPU (self scheduling) to achieve good load balancing. Microtasking processes are lightweight (threads) incurring low overhead. The drawback is that programmers need to find opportunities for parallel execution, which can be very tedious.

- Autotasking retains the merits of microtasking and relieves programmers of the uncovering of parallelism. The compiler automatically detects parallelism through dependence analysis and generates code to execute parallel regions on multiple CPUs. Programmers can still add directives to improve performance.

The methods have also been adopted by others such as Alliant and Sequent systems. For application of the techniques, [Hossfeld89] provides several interesting examples. Results in the paper indicate that microtasking is usually more efficient than macrotasking. Results in [Furtney92] suggest that autotasking may lead to better or poorer performance than microtasking, depending on the type of problems tackled and the skill of the programmer.

For a practical tradeoff study of the three different approaches, check the manuals of a target machine, develop programs using the three multitasking techniques, and compare the actual performance.

Problem 11.13 The use of vectorized code in matrix multiplication is to perform inner-product operations. The following code implements the multiplication of matrices A and B and store the result in matrix C. The procedure is identical to that used in the solution for Problem 3.8.

```
Do m = 0, 3
   Doall l = 0, 3
      n = l + m
      if (n .gt. 3) n = n - 4
      Do i = 1, k
         Do j = 1, k
            C(l*k + i, n*k + j) = A(l*k + i, 1:4*k) *
               B(1:4*k, n*k + j)
         Enddo
      Enddo
   Endall
Enddo
```

Problem 11.14 Assume node i $(0 \leq i \leq 31)$ is allocated elements $i \times 32$ through $(i+1) \times 32 - 1$ of vector V. In binary notation, node $(d_4d_3d_2d_1d_0)_2$ holds 32 elements of V, denoted by $(d_4d_3d_2d_1d_0b_4b_3b_2b_1b_0)_2$, where each b_i can be 0 or 1. Each node uses three vector registers R, S, and T, each of size 32. The operation performed by node $(d_4d_3d_2d_1d_0)_2$ is specified in the following pseudo code:

```
R(0 : 31) = V((d4d3d2d1d000000)2 : (d4d3d2d1d011111)2)
Do m = 0, 4
   S(0 : 31) = R(0 : 31)
   Do i = 0, 31     /* i = (b4b3b2b1b0)2 */
      get S((b4b3b2b1b0)2) from the node whose address
      differ in bit (4 - m) and store in T((b4b3b2b1b0)2)
      if (d4-m == 0) then
```

$R(i) = S((b_4b_3b_2b_1b_0)_2) + T((b_4b_3b_2b_1b_0)_2) \times w^{(d_{4-m}\ldots d_3d_4000\ldots0)_2}$

else

$R(i) = T((b_4b_3b_2b_1b_0)_2) + S((b_4b_3b_2b_1b_0)_2) \times w^{(d_{4-m}\ldots d_3d_4000\ldots0)_2}$

endif

Enddo

Enddo

Do $m = 0, 4$

$S(0:31) = R(0:31)$

Do $i = 0, 31$ /* $i = (b_4b_3b_2b_1b_0)_2$ */

$R(i) = S((b_4\ldots b_{m+1}0b_{m-1}\ldots b_0)_2) + S((b_4\ldots b_{m+1}1b_{m-1}\ldots b_0)_2) \times w^{(b_{4-m}\ldots b_3b_4d_0\ldots d_3d_400\ldots0)_2}$

Enddo

Enddo

Note that each binary address has 10 bits. The number of padding 0s in the power of w decreases with m and may reach zero in the last loop. The weighting factors are defined as

$$w^q = \exp\left(j\frac{-2q\pi}{1024}\right), \qquad j = \sqrt{-1}.$$

Each node needs to compute 67 weighting factors. Specifically, node $(d_4d_3d_2d_1d_0)_2$ computes w^q for $q =$

Binary representation for q	Number of combinations
$(d_400000000)_2$	1
$(d_3d_40000000)_2$	1
$(d_2d_3d_4000000)_2$	1
$(d_1d_2d_3d_400000)_2$	1
$(d_0d_1d_2d_3d_400000)_2$	1
$(b_4d_0d_1d_2d_3d_40000)_2$	2
$(b_3b_4d_0d_1d_2d_3d_4000)_2$	4
$(b_2b_3b_4d_0d_1d_2d_3d_400)_2$	8
$(b_1b_2b_3b_4d_0d_1d_2d_3d_40)_2$	16
$(b_0b_1b_2b_3b_4d_0d_1d_2d_3d_4)_2$	32

When a program is written based on the above pseudo code, message-passing instructions need to be added in the node program to carry out data exchanges. At the end of computations, the host program and node program need instructions to return the resnlts in each node to the host.

Problem 11.15

(a) Suppose the image is partitioned into p segments, each consisting of $s = m/p$ rows. Vector *histog* is shared among the processors. Therefore its update has to be performed in a critical section to avoid race conditions. Assume it is possible to protect each element of vector *histog* by a separate semaphore. The following program performs parallel histogramming:

```
Var pixel(0 : m − 1, 0 : n − 1);
Var histog(0 : b − 1): integer;
Var lock(0 : b − 1): [0,1];
histog(0 : b − 1) = 0;
lock(0 : b − 1) = 1;
for k = 0 until p − 1 Doall
    for i = k × s until (k + 1) × s − 1 do
        for j = 0 until n − 1 do
            P(lock(pixel(i, j)));
            histog(pixel(i, j)) = histog(pixel(i, j)) + 1;
            V(lock(pixel(i, j)));
        Enddo
    Enddo
Endall
```

(b) The maximum speedup of the parallel program over the serial program is p, provided there is no conflict in accessing the *histog* vector and the overhead associated with P and V operations is negligible. But in fact, it is difficult to achieve a speedup of p, due to access conflicts and synchronization overhead.

An alternative approach is to associate a local histog vector with each processor, which will obviate the use of critical section when updating values of the individual vectors. At the end of the algorithm, the corresponding values in local histog vectors are summed to obtain the final result. This method should have better performance than the method described in (a). For more details, see [Hwang84] pp. 548–549.

Chapter 12

UNIX, Mach and OSF/1 for Parallel Computers

Problem 12.1

(a) Example application for each of the four concurrency models:

- In a pure parallelism model, a programmer's view of concurrency coincides with hardware parallelism. The model can be used in the design of compilers to generate codes for the parallel execution of loop bodies.
- User concurrency model can be used in applications in which the programs are composed of multiple user-defined tasks queued to be processed in parallel by virtual processors. A coroutine model in which multiple threads of control defined at coroutine level and multiplexed onto a single thread of control at the operating system level is an example of this model. Pipes in UNIX are implemented by coroutines.
- System concurrency models are used in most multithreaded systems, such as Mach. By multiplexing multiple threads on each physical processor, it is possible to context switch to another thread when the current thread becomes blocked.
- Dual concurrency is a combination of user and system concurrency. Applications which comprise multiple coarse-grain user-defined tasks are suitable for this model.

In general, A compute-bound parallel program will typically use pure parallelism. Concurrency models with dedicated physical processors for virtual processors are suitable for fine-grain applications. The reason is that such applications require frequent synchronizations, which can be very expensive if some of the processes involved are idle.

System and dual consistency models are suitable for implementing applications with blocking operations since the high degree of parallelism facilitates efficient

usage of hardware when some of the VPs become blocked. Also, the VPs should be coarse-grained so that the cost of context switch becomes relatively negligible.

(b) When there are more virtual processors than physical processors as in the case of system concurrency and dual concurrency, Mach scheduling can benefit from user input of application-specific information (hints). These hints help decide which virtual processors should be run or suspended. Hints are particularly useful when a thread needs to communicate or synchronize with another thread which is not running. In that situation, the current thread cannot proceed until the communication or synchronization is complete.

If the identity of the non-running thread is not known as in a test-and-set lock mechanism, a discouragement hint is used to indicate that the current thread should give up the processor in hope that the lock-holding virtual processor will be scheduled.

If the identity of the non-running thread is known as in a compare-and-swap synchronization mechanism, a handoff hint will cause the current thread to give up the processor to the specific virtual processor that holds the lock. It is a useful feature which allows a low-priority thread to run ahead of other higher-priority threads.

The two types of hints make it possible to bypass the search of run queues, leading to more efficient concurrency support.

Problem 12.2 Mach ptrace is improved from a similar facility in UNIX to deal with multithreaded execution and concurrent exceptions. It does not require modification of application programs except for the insertion of breakpoints. It also retains much of the user interface provided by UNIX debuggers. Some of the features are summarized below. More details can be found in [Caswell90].

- Memory and register access. Memory accesses are implemented by virtual memory operations vm_read and vm_write. Register accesses are provided by kernel functions thread_get_state and thread_set_state.
- Thread execution control. The *continue* function is implemented by Mach kernel calls *task_resume*, *thread_suspend*, and *thread_resume*. Single-step is implemented by setting the trace bit before performing *task_resume*. Termination is implemented by calling *exit* in the application program.
- Exception handling. An exception causes a message to be sent that identifies the exception, the thread causing the exception, and the task containing the thread. This message is sent to the debugger through a port to which the debugger has receive right. Multiple concurrent exceptions result in messages being queued and examined by the debugger one by one.
- User interface. The user interface of the Mach debugger is extended from that of OSF's gdb. It allows the selection of a current thread which can be examined and modified. Individual threads can also be suspended and resumed to allow fast selection of the desired portion in the debugged application.

- UNIX compatibility. This property is afforded through an internal Mach kernel exception handler *ux_handler* which converts exception messages to signals and directs them to the appropriate threads. Signal compatibility code in Mach ensures that UNIX signals caused by exceptions are sent to the thread that caused the original exception.
- Deadlock avoidance. Deadlock can occur when the execution of an application is continued but it has no runnable threads. The Mach debugger avoids this type of deadlock by making sure that at least one thread of the application is runnable before resuming its execution. Another type of deadlock is caused by a running thread trying to synchronize with suspended threads. Such a deadlock can be interrupted by user intervention.
- Processing of application events. Mach debugger relies on information returned by the *wait* system call and contents of exception messages to discern between two types of events: breakpoints and trace traps. Trace traps do not have to be reported directly to the user by low-level event detection code. In contrast, breakpoint traps are reported so that the user knows which threads have reached breakpoints.

Problem 12.3 Duality of memory and communication is an important feature and key concept in the design of Mach/OS. The idea combines shared memory and message passing and presents a uniform interface for interprocess communication. It enables support for distributed computing and multiprocessing, and achieves the design goals of architecture transparency, compatibility, and portability. Some advantages are discussed below. For more details, see [Young87].

- The use of physical memory as a cacheing device for secondary storage in the external memory management scheme of Mach reduces the amount of I/O operations significantly, thereby improving the performance. This is achieved by data sharing. An application program can create a memory object which represents a large data array and allow other programs to share it through a server message interface. Only a single message exchange with the server is needed to access the array.
- UMA, NUMA, and NORMA architectures can support message passing or shared memory to provide interprocess communication and data sharing. Message passing is implemented in UMA and NUMA architectures by using semaphores or data copy operations. Information sharing on a NORMA architecture can be implemented by a copy-on-reference mechanism.

  Mach provides OS primitives which allow programmers to choose shared memory or message passing as the paradigm for implementing a multithreaded application. Depending on the machine architecture, message passing can be implemented by memory management primitives or vice versa.
- Implications of the memory-communication duality in Mach were illustrated by several applications in [Young87]. For instance, the network file server of UNIX can be emulated by a data manager in the external memory management scheme

of Mach. The network copy-on-reference of process data facilitated by Mach allows more effective process migration. Transaction and database management systems benefit from the external memory management provided by Mach. The tasks of client programs are simplified and the processing is expedited.

Problem 12.4

(a)
- Conventional UNIX kernel can service only one system call from a user process at a time. This measure is needed to protect kernel data structures. However, the serialization of services to requests can impede performance. For instance, execution in the kernel mode can become a bottleneck when multitasking is implemented on a uniprocessor.
- Processes in conventional UNIX do not share address space. When a child process is spawned, the overhead associated with copy-on-write operation can be excessive. Limited resource sharing among processes is a deficiency of conventional UNIX.

(b) A multiprocessor UNIX kernel should address several issues:

- Portability. It should be independent of machine size (number of processors) in a system. It should also hide the architecture from users.
- Compatibility. It should be compatible with existing UNIX systems in user interface, system call conventions, etc.
- Address space. It should be able to handle huge address space to meet technological advances and programming requirement. It should allow cooperating processes to share address space and unrelated processes to have disjoint spaces.
- Dynamic load balancing. It should be able to monitor the load of each processor and dynamically adjust the load among processors.
- Parallel I/O and network access. Parallel I/O should be supported to eliminate potential bottleneck in large data movements. Network functions should be enhanced to facilitate remote access and heterogeneous processing.

Problem 12.5

(a) In a single fixed master kernel, one of the processors is designated as the master, the others are slaves. Only the master can run in kernel mode; slave processors can only run in user modes. When a process needs system service, it makes a request to the master and may be suspended if the master is already busy. The advantage is its simplicity in implementation and in the protection of the kernel data structure. A serious disadvantage is that the master can easily become a bottleneck and the performance is not scalable. Moreover, such a system is not robust. Once the master processor is broken, the entire system will be paralyzed.

(b) In a single floating master kernel, several processors in the system are capable of executing kernel functions but only one is allowed to execute in kernel mode at a time. The entire kernel is protected by a giant lock. A processor wishing to execute in kernel mode must acquire the giant lock before doing so. The advantage is improved fault-tolerance capability over a fixed master kernel. Other than that, little improvement is achieved in terms of performance.

(c) In a single master kernel with assistants, the kernel functions are divided into two parts. Only critical system calls are reserved to the master which can float among several processors with the use of a single giant lock. The other noncritical system calls can be executed on slaves. The division helps offload the burden of the master and reduces the severity of bottleneck. The disadvantage is increased complexity in implementation.

(d) In the multiple cooperative masters model, several masters are allowed to coexist, each of which handles a major kernel subsystem function. Noncritical kernel functions are still executed by slaves. A lock hierarchy with multiple giant locks are used to coordinate operations among the masters. The advantage is that all the kernel functions are spread over several processors, which further improves performance and robustness. The disadvantage is that sophisticated mechanisms must be provided to avoid conflicts among masters.

(e) Multithreaded UNIX kernel is a further refinement of the floating masters concept. Each subsystem function in the multiple cooperative masters model is implemented by multiple threads to maximize the parallelism in kernel operations. The advantage is improved performance scalability with the number of processors in a system. The disadvantage lies in the high development cost and more complex control of the threads.

Problem 12.6

(a) In object-oriented model, messages are treated as objects that are transferred among processors using synchronous or asynchronous message-passing protocols. Multiple tasks with separate address spaces are allowed on each node. When sending a message, the user only needs to specify which process should receive the message without worrying about which node the destination process actually resides in. The OS kernel provides the necessary mechanism to ensure the correct delivery of the message to the right process.

(b) In node-addressed mode, only one task is active on each node and hence can be identified by specifying the node number. When sending a message, the user only needs to specify the destination node. The actual routing is handled by OS kernel. Therefore, the user does not need to know the interconnection structure. An asynchronous communication protocol is used.

(c) In channel-addressed mode, again only one task is active on each node. To exchange a message, a communication path has to be established between the source and

destination nodes before message passing can convene. The user has to explicitly specify the channels constituting the communication path. As a result, the user has to be aware of the system interconnection topology.

The required support from OS kernel varies with the model. For instance, the OS kernel can be quite primitive for channel-addressed model, but rather sophisticated for object-oriented model.

Problem 12.7

(a) Space sharing and time sharing are two ways to partition system resources and allocate them to different tasks. In space sharing, the nodes in a multicomputer system are divided into several disjoint groups, each consisting of a subset of the nodes. Each group can be assigned to a task. In this manner, several tasks can proceed concurrently without interfering with each other. In time sharing, each task can use a multiprocessor system for a certain amount of time. Upon the expiration of time quantum, the active task is suspended and the system is allocated to another task. The two types of sharing can be combined to improve resource utilization.

(b) Cosmic environment (CE) and reactive Kernels (RK) constitute an integrated OS environment which provides a message-passing C programming environment for the hypercube computers developed at Caltech. CE resides on the network host and provides communicaiton support between the host and the node processes through a set of daemons, utility routines, and libraries. RK is a node OS that resides in each node to provide multiprogramming support. In the model, processes are conceived as persistent, active, and stationary. Messages, in contrast, are inert, mobile, and transitory. Processes are dormant and are activated by messages. Together CE and RK provide a uniform communication support between host and node processes regardless of the physical allocation of the processes.

Problem 12.8 Task, thread, message, port, and port set are the basic abstractions in Mach/OS.

(a)
- A task in Mach is a unit for resource allocation. Such resources may include processors, I/O, and memory. A task is a passive entity. It performs no computation but simply provides a framework for executing threads.
- A thread is a lightweight process with an independent program counter and state registers. A UNIX process can be thought of as a task with a single thread. Multiple threads can execute in an environment defined by a task and share the resources allocated to the task. Because of resource sharing, spawning of a thread incurs a lower overhead compared to a conventional UNIX process.
- A port in Mach is logical queue for messages protected by the kernel. In the object-oriented approach adopted by Mach, a port is an object which can be accessed by threads. Application programs communicate with kernel or

server tasks by sending/receiving messages via ports.

- A message is a piece of information exchanged among processes (threads or tasks). It usually consists of a fixed-size header and a variable-size body.
- A port set is composed of a set of ports to which a task may have all or none of the access rights. A port can only belong to a port set and cannot be shared among different port sets.

(b) Three types of parallelism are exploited in Mach/OS kernel:

- Shared-resource multithreading. Parallelism is achieved by executing multiple threads within a task. The threads share resources, communicate and synchronize with each other by shared variables.
- Partially shared-memory multitasking. Parallelism is achieved by creating multiple tasks which interact through a restricted region of shared memory.
- Message-passing multitasking. Parallelism is achieved by creating multiple tasks on distributed processors which communicate through message passing.

Problem 12.9

(a) Thread management refers to the management of threads during their life cycles from creation to destruction. Threads can be dynamically created, suspended, resumed, and destroyed. Depending on the implementation, a thread may be in different states. Threads are scheduled independently by the OS kernel. When a shared memory area is accessed by multiple threads, proper synchronization must be provided to ensure correct operation. When a task is suspended or resumed, all the threads within it are also affected.

(b) Several thread-scheduling policies have been adopted in Mach:

- Time-sharing scheduling. Each thread is assigned a priority and a time quantum. During its execution, the quantum is reduced until it expires. Then another thread with the highest priority is chosen. To prevent possible starvation, priority is adjusted dynamically so that long-waiting jobs will have their priority elevated.
- Interactive scheduling. Using this policy, priority is set in favor of threads that require interactive responses.
- Fixed-priority scheduling. Priority assigned to threads is fixed; it does not change with longevity. This policy risks potential starvation for low-priority threads.

Problem 12.10

(a) Interprocess communication in Mach is effected through ports, which are queues holding messages to be delivered to receiving processes. The communication can be carried out synchronously by streams function or asynchronously by remote procedure calls to provide architecture-independent and network-transparent sup-

port. The mapping between port identifier and actual destination node location is handled by network server tasks.

(b) Virtual memory management in Mach is aimed at providing a large virtual address space through a uniform architecture-independent interface. The unit of memory allocation in Mach is task. A task can request a region of virtual memory in units of pages, and set protection and inheritance attributes, which specify the relationship between the address spaces when child tasks are spawned. Data are stored in memory objects which can be shared among tasks to reduce redundancy and inconsistency. This is facilitated by address mapping mechanisms implemented with the use of various tables. Virtual memory is divided into machine-dependent and machine-independent parts to enhance portability across different hardware systems. More details are provided in the text.

(c) Multitasked clustering in BBN TC2000 allows multiple tasks to be executed on separate, dedicated clusters to reduce context switching and scheduling overhead. The idea exploits space sharing in lieu of time sharing. The size of each cluster can vary depending on the computation demand of individual tasks.

Problem 12.11

(a) OSF/1 evolves from UNIX System V, BSD 4.4, and Mach as depicted in Fig. 12.16a. Many features are borrowed from Mach kernel. For instance, kernel functions such as thread scheduling, memory management, and interprocess communication are similar to those in Mach with a variety of enhancements (for example, the symmetric multiprocessing feature). OSF/1 also has an internal file system switch to facilitate the use of multiple file system supported by System V, BSD 4.4, and network file servers.

(b) POSIX (Portable operating system interface for UNIX) is an attempt to standardize operating systems so that applications conforming to the POSIX standard are portable from one platform to another. In 1985, IEEE defined POSIX standards with FIPS 151-1. It was declared by National Institute of Standards and Technology to be the standard interface for government open systems in 1990. Many vendors have subsequently come up with operating systems that comply with POSIX. OSF/1 compliance with POSIX includes shells, real-time computing, security facilities, transparent file-access support, protocol-independent interprocess communication, etc.

(c) Program development environment contains a set of tools, including editors, compilers, linker, and debugger based on packages developed by Free Software Foundations. Major UNIX shells are also supported. OSF/1 environment supports application program construction through a layered approach, with applications on top of user libraries and system libraries, which in turn are supported by OS kernels. Shared libraries reduce space requirement, improve performance, and lower developing and debugging cost. Separate compilation and dynamic linking helps modular development of application programs. Position-independent code

placement also improves performance, among other benefits.

Problem 12.12

(a) A Pthread is a thread as defined in POSIX standard. Each thread has a single sequential line of control and is intended to carry out a small self-contained job. Motivations for the use of Pthread are enumerated below:

- Use of Pthreads enables cross-development of multiprocessor programs on a uniprocessor system or different platforms.
- A server task can spawn several threads to serve multiple requests. Doing so improves resource utilization with a light overhead. While a thread is blocked, others can be running. On the system with multiple processors, the requests can be services concurrently.
- Independent threads can be executing in different states. Multiple threads allow computation, communication, and I/O activities to be overlapped.
- Multiple threads allow asynchronous events to be handled more efficiently by preventing inconvenient interrupts and avoiding complex flow control.

(b) The database may be shared among several programs. A user program wishing to retrieve/update the database send a request to the server through a communication channel. The server then spawns a thread to serve the request. Since there might be several threads trying to access the database, proper synchronization is needed to prevent simultaneous updates to the data. This is provided by a global lock *db_mutex* which ensures that operations on the database are performed in a critical section. The lock can be envisioned as a semaphore, initialized to 1 at the beginning. Then Pthread_mutex_lock and Pthread_mutex_unlock can be viewed as P and V operations, respectively. See Chapter 11 for more details.

Problem 12.13

(a) LINPACK is a package developed by Jack Dongarra, Jim Bunch, Cleve Moler and Pete Stewart for solving linear equations and linear least squares problems. See [Dongarra79] for a more detailed description of the package and its usage. LINPACK has been widely used as a benchmark to determine the performance of various computer systems. See [Dongarra92].

It can deal with linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. It uses Gaussian elimination with pivoting and Cholesky factorization (for symmetric positive definite matrices) to decompose a matrix. In addition, the package computes the QR (by Householder transform) and singular value decompositions of rectangular matrices and applies them to least squares problems. For a description of the pertinent algorithms, please consult texts on numerical analysis or matrix algebra such as [Dahlquist74] and [Golub89].

(b) Most machines provide vectorization and/or concurrentization support based on extensive dependence analysis. Other optimization techniques such as loop interchange may also be implemented. They also allow user interaction to optionally enable or disable such optimizations. Please check the manuals of machines accessible locally.

(c) Parallel I/O is important to the performance when the data set is large. Without efficient support, I/O can become a bottleneck and degrade overall performance. Parallel I/O is also desirable to support real-time monitoring of program activities for performance tuning or debugging purpose. OS should also support effective program partitioning, scheduling, and synchronization for the parallel execution of LINPACK programs.

Problem 12.14 The degree of compiler support provided by different machines may vary widely. For instance, some systems use very primitive processors which may not be able to perform vector operations, while other systems may have sophisticated processors capable of efficient vector processing. Concurrency support is typically provided through a library of system calls, which manages message passing and other activities. System calls can be linked with user programs at compilation/linkage time. Parallel I/O support is essential so that code and data can be quickly distributed to individual nodes and the results be sent back to the host. Dynamic load balancing provided by the OS will be valuable to the efficient utilization of system resources, especially when the matrix is not regularly structured. Check relevant manuals for more detailed information.

Problem 12.15

(a) If the conservative policy is used, at most $20/4 = 5$ processes can be active simultaneously. Since one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.

(b) To improve the drive utilization, each process can be allocated three tape drives. The fourth one will be allocated on demand. In this policy, at most $\lfloor 20/3 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum is 2.

Bibliography

[Adam74] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems", *Commun. ACM*, 17(12):685–690, 1974.

[Agha90] G. Agha, "Concurrent Object-Oriented Programming", *Commun. ACM*, 33(9):125–141, Sept. 1990.

[Archibald86] J. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Trans. Computer Systems*, 4(4):273–298, Nov. 1986.

[Berntsen90] J. Berntsen, "Communication-Efficient Matrix Multiplication on Hypercubes", *Parallel Computing*, pp. 335–342, 1990.

[Bic88] L. Bic and A. C. Shaw, *The Logical Design of Operating Systems*, 2 ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Cannon69] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. thesis, Montana State University, 1969.

[Caswell90] D. Caswell and D. Black, "Implementing a Mach Debugger for Multithread Applications", *Proc. Winter 1990 USENIX Conf.*, Washington, DC, Jan. 1990.

[Chan86] T.F. Chan and Y. Saad, "Multigrid Algorithms on the Hypercube Multiprocessor", *IEEE Trans. Computers*, 35(11):969–977, 1986.

[Dahlquist74] G. Dahlquist and A. Björck, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, 1974.

[Dongarra79] J. J. Dongarra et al., *LINPACK: Users' Guide*, SIAM, Philadelphia, 1979.

[Dongarra92] J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software", Technical report, Computer Science Department, University of Tennessee, Knoxville, TN, 1992.

[Dubois88] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors", *IEEE Computer*, 21(2), 1988.

[Fox87] G. C. Fox, S. W. Otto, and A. J. Hey, "Matrix Algorithms on Hypercube (I): Matrix Multiplication", *Parallel Computing*, pp. 17–31, 1987.

[Furtney92] M. Furtney, "Parallel Processing at Cray Research, Inc.", in R. H. Perrott (ed.), *Software for Parallel Computers*, pp. 133–154, Chapman & Hall, 1992.

[Gharachorloo91] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors", *Proc. Fourth Int. Conf. Arch. Support for Prog. Lang. and OS*, 1991.

[Golub89] G.H. Golub and C.F. Van Loan, *Matrix Computations*, second ed., The Johns Hopkins University Press, 1989.

[Hossfeld89] F. Hossfeld, R. Knecht, and W. E. Nagel, "Multitasking: Experience with Applications on a Cray X-MP", *Parallel Computing*, 12:259–283, 1989.

[Hwang84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[Hwang91] K. Hwang and C. M. Cheng, "Simulated Performance of A RISC-Based Multiprocessor Using Orthogonal Access Memory", *J. Para. Distri. Computing*, 13:43–57, 1991.

[JaJa92] J. JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[Johnsson89] S. L. Johnsson and C. T. Ho, "Optimal Broadcasting and Personalized Communication in Hypercubes", *IEEE Trans. Computers*, 38(9):1249–1268, Sept. 1989.

[Konicek91] J. Konicek et al., "The Organization of the Cedar System", *Proc. Int. Conf. Parallel Processing*, pp. volume I, 49–56, 1991.

[Leighton92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan-Kaufmann, 1992.

[Li89] K. Li and P. Hudak, "Memory Coherence in Shared-Memory Systems", *ACM Trans. Computer Systems*, pp. 321–359, Nov. 1989.

[Mosberger93] D. Mosberger, "Memory Consistency Models", *Operating Systems Review*, 27(1):18–26, Jan. 1993.

[Quinn87] M. J. Quinn, *Designing Efficient Algorithms for Parallel Commuters*, McGraw-Hill, New York, 1987.

[Saavedra90] R. H. Saavedra and D. E. Culler, "Analysis of Multithreaded Architectures for Parallel Computing", *Proc. ACM Symp. Parallel Algorithms and Architecture*, Greece, July 1990.

[Silberschatz88] A. Silberschatz and J. Peterson, *Operating System Concepts, Alternate Edition*, Addison-Wesley, Reading, MA, 1988.

[Stone90] H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Reading, MA, 1990.

[Tanenbaum92] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal, "Parallel Programming Using Shared Objects and Broadcasting", *IEEE Computer*, 25(8):10–20, 1992.

[Wang89] J. Wang et al., "On the Communication Structures of Hyper-Ring and Hypercube Multicomputers", *J. Computer Sci. Tech.*, 4(1), Jan. 1989.

[Wolfe89] M. J. Wolfe, "Automatic Vectorization, Data Dependence, and Optimizations for Parallel Computers", in Hwang and DeGroot (eds.), *Parallel Processing for Supercomputing and Artificial Intelligence*, Chapter 11, McGraw-Hill, New York, 1989.

[Yang89] Q. Yang, L. N. Bhuyan, and B. Liu, "Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor", *IEEE Trans. Computers*, 38(8):1143–1153, Aug. 1989.

[Young87] M. W. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. Eppinger, J. Chew, W. Bolosky, D. L. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. 11th ACM Symp. Operating System Principles*, pp. 63–76, 1987.