Chapter 12 – Large Computer Systems

12.1. A possible program is:

```
LOOP    Move          0,STATUS
        Move          CURRENT,R1
        Move          R1,R2
        Move          R1,Rnet
        Shift_right   Rnet
        Add           Rnet,R2          Add current value from left
        Move          R1,Rnet
        Shift_left    Rnet
        Add           Rnet,R2          Add current value from right
        Move          R1,Rnet
        Shift_up      Rnet
        Add           Rnet,R2          Add current value from below
        Move          R1,Rnet
        Shift_down    Rnet
        Add           Rnet,R2          Add current value from above
        Divide        5,R2             Average all five values
        Move          R2,CURRENT
        Subtract      R2,R1
        Absolute      R1
        Subtract      EPSILON,R1
        Skip_if≥0
        Move          1,STATUS

        {Control processor ANDs all STATUS flags and exits LOOP if result is 1;
        otherwise, LOOP is repeated.}

END     LOOP
```

12.2. Assume that each bus has 64 address lines and 64 data lines. There are two cases to consider.

i) For uncached reads, each read with a split-transaction bus requires $2T$, consisting of $1T$ to send the address to memory and $1T$ to transfer the data to the processor. Using a conventional bus, it takes $6T$ because of the $4T$ delay in reading the contents of the memory. Therefore, 3 conventional buses would give approximately the same performance as the split-transaction bus.

ii) For cached reads, it is necessary to consider the size of the cache block. Assume that this size is 64 bytes; therefore, it takes 8 clock cycles to transfer an entire block over the bus.

Using a split-transaction bus it is possible to use all cycles to transfer either read requests (addresses) or data; therefore, it takes $9T$ per read (not in consecutive clock cycles!). Using a conventional bus each read takes $13T$ (consecutive clock cycles). Thus, 4 of these 13 cycles are wasted waiting for the memory response. This means that in this case also it would be necessary to use 3 conventional buses to obtain approximately the same performance.

12.3. The performance would not improve by a factor of 4, because some bus transactions involve uncached reads and writes. Since uncached accesses involve only one word of data, they use only one quarter of the 4-word wide bus. Of course, the overall performance would depend on the ratio of cached and uncached accesses.

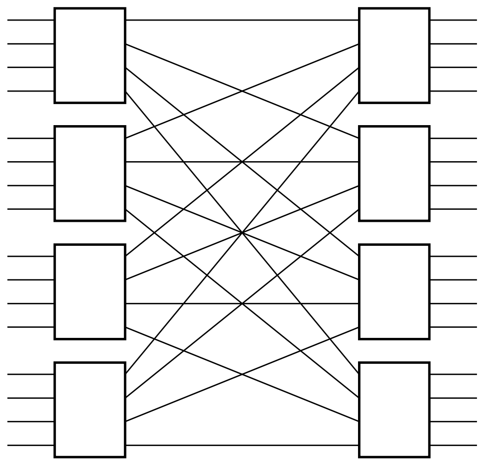
12.4. Assume n is a power of 2 because of the form of the shuffle network.

Crossbar cost = n^2.

Shuffle network cost = $2(n/2)\log_2 n$.

Solving for smallest n satisfying $n^2 \geq 5[2(n/2)\log_2 n]$ where n is a power of 2, gives $n \geq 5\log_2 n$. At n = 16, inequality is not satisfied. At n = 32, inequality is satisfied. Therefore, the smallest n is 32.

12.5. The network is



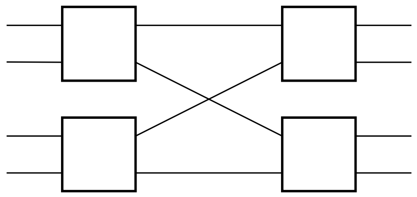
Note that the definition of the shuffle pattern must be generalized in such a way that for each source input there is a path (in fact, exactly one path) to each destination output.

Cost of network built from 2×2 switches is $(n/2)\log_2 n$.

Cost of network built from 4×4 switches is $4(n/4)\log_4 n = n(\log_2 n/\log_2 4) = (n/2)\log_2 n$.

Therefore, the cost of the two types of networks is the same.

Blocking probability: The 4×4 switch is a nonblocking crossbar, and can be built from 2×2 switches as



But this is a blocking network. Therefore, the blocking probability of a large network built from 4×4 switches is lower than one built from 2×2 switches.

12.6. Program structure:

Sequential segment S_1 (k time units)
PAR segment P_1 (1 time unit)
Sequential segment S_2 (k time units)
PAR segment P_2 (1 time unit)
Sequential segment S_3 (k time units)

$T_1 = 3k + 2k$
$T_n = 3k + 2\lceil (k/n) \rceil$
Speedup = $(5k)/(3k + 2\lceil (k/n) \rceil)$

Limiting value for speedup is 5/3. This shows that the sequential segments of a program severely limit the speedup when the sequential segments take about the same time to execute as the time taken to execute the PAR segments on a single processor.

12.7. The n-dimensional hypercube is symmetric with respect to any node. The distance between nodes x and y is the number of bit positions that are different in their binary addresses. The number of nodes that are k hops away from any particular node is $\binom{n}{k}$. Therefore, the average distance a message travels is

$$[\sum_{k=1}^{n} k \cdot \binom{n}{k}]/(2^n - 1)$$

which simplifies to $[2^{n-1} \cdot n]/(2^n - 1)$, and is less than $(1 + n)/2$, as can be verified by trying a few values. For large n, the average distance approaches $n/2$.

12.8. When a Test-and-Set instruction “fails,” that is, when the lock was already set, the task should call the operating system to have its task name queued and to allow some other task to execute. When the task holding the lock wishes to release the lock (set it to 0), the task calls the operating system to do so, and then the operating system dequeues and runs one of the waiting tasks which is then the one owning the lock. If no task is waiting, the lock is cleared (= 0) to the free state.

12.9. The details of how either invalidation or updating can be implemented are described in Section 12.6.2, and the advantages/disadvantages of the two techniques can be deduced directly from that discussion. In general, it would seem that invalidation and write-back of dirty variables results in less bus traffic and eliminates potentially wasted cache updating operations. However, cache hit rates may be lowered by using this strategy. Updating associated with a write through policy may lead to higher hit rates and may be simpler to implement, but may cause unacceptably high bus traffic and wasted update operations. The details of how reads and writes on shared cached blocks (lines) are normally interleaved from distinct processors in some class of applications will actually determine which coherence strategy is most appropriate.

12.10. No. If coherence controls are not used, a shared variable in cache B may not get updated/invalidated when it is written to in cache A while A's processor has mutually exclusive access. Later, when B's processor acquires mutually exclusive access, the variable will be incorrect.

12.11. In Figure 12.18, both threads continuously write the same shared variable *dot_product*; hence, this is done serially. In Figure 12.19, each thread updates its local variable *local_dot_product*, which is done in parallel. Therefore, if very large vectors are used (so that the actual computation of the dot product dominates the processing time), the program in Figure 12.19 may give almost twice as good performance as the program in Figure 12.18.

12.12. It is only necessary to create 3 new threads (rather than just one in Figure 12.19), and assign processing of one quarter of each vector to each thread.

12.13. The only significant modification is for the program with *id* = 0 to send one quarter of each vector to programs with *id* = 1, 2, 3. Having completed the dot-product computation, each program with $id > 0$ sends the result to the program with $id = 0$, which then prints the final result.

12.14. Overhead in creating a thread is the most significant consideration. Other overhead is encountered in the lock and barrier mechanisms. Assume that the thread overhead is 300 times greater than the execution time of the statement that computes the new value of the dot product for a given value of k. Also, assume that the overhead for lock and barrier mechanisms is only 10 times greater.

Then, as a rough approximation, the vectors must have at least $320 \times 2 = 640$ elements before any speedup will be achieved.

12.15. The dominant factor in message passing is the overhead of sending and receiving messages. Assume that the overhead of either sending or receiving a message is 1000 times greater than the execution time of the statement that computes the new value of the dot product for a given value of k. Then, since there are 3 send and 3 receive messages involved, the vectors will have to have at least $1000 \times 6 = 6000$ elements before any speedup is achieved.

Note that we have assumed that the overhead of 1000 is independent of the size of the message – as a first order approximation.

12.16. The shared-memory multiprocessor can emulate the message-passing multicomputer easier than the other way around. The act of message-passing can be implemented by the transfer of (message) buffer pointers or complete (message) buffers between the two communicating processes that otherwise only operate in their own assigned area of main memory. A multicomputer system can emulate a multiprocessor by considering the aggregate of all of the local memories of the individual computers as the shared memory of the multiprocessor. Access from a computer to a nonlocal component of the shared memory can be facilitated by passing messages between the two computers involved. This is a cumbersome and slow process.

12.17. The situation described is possible. Consider stations A, B, and C, situated at the left end, middle, and right end of the bus, respectively. Station A starts to send a message packet of $0.25\,\tau$ duration to destination station B at time t_0. The packet is observed and copied into station B during the interval $[t_0 + 0.5\tau, t_0 + 0.75\tau]$. Just before $t_0 + \tau$, station C begins to transmit a packet to some other station. It immediately collides with A's packet, and the garbled signal arrives back at station A just before $t_0 + 2\tau$.

12.18. (a) The F/E bit is tested. If it is 1 (denoting ”full”), then the contents of BOXLOC are loaded into register R0, F/E is set to 0 (denoting “empty”), and execution continues with the next sequential instruction. Otherwise (i.e., for [F/E] = 0), no operations are performed and execution control is passed to the instruction at location WAITREC.

(b) In the multiprocessor system with the mailbox memory, each one-word message is sent from T_1 to T_2 by using the single instructions:

```
SEND    PUT    R0,BOXLOC,SEND        (1)
```

and

```
REC     GET    R0,BOXLOC,REC         (2)
```

in tasks T_1 and T_2, respectively, assuming that [F/E] = 0 initially.

In the system without the mailbox memory, replace (1) in task T_1 with the sequence:

```
WLOCK    TAS.B     WRITE
         BMI       WLOCK
         MOV.W     R0,LOC
         CLR.B     READ
```

and replace (2) in task T_2 with the sequence:

```
RLOCK   TAS.B    READ
        BMI      RLOCK
        MOV.W    LOCK,R0
        CLR.B    WRITE
```

Let the notation V(7) stand for bit b_7 of byte location V. Ordinary word location LOC represents the data field of mailbox location BOXLOC, and the combination of WRITE(7) and READ(7) represents the F/E bit associated with BOXLOC.

In particular, [WRITE(7)] = 0 means that LOC is empty, and [READ(7)] = 0 means that LOC is full.

Initially, when LOC is empty, the settings must be [WRITE(7)] = 0 and [READ(7)] = 1. Note that when the instruction MOV.W is being executed in either task, we have [WRITE(7)] = [READ(7)] = 1, indicating that LOC is either being filled or emptied. Also note that it is never the case that [WRITE(7)] = [READ(7)] = 0. This solution works correctly for the general case where a number of tasks pass data through LOC.

For the case suggested in the problem, with a single task T_1 and a single task T_2, the following sequences are sufficient. In T_1, use:

```
TESTW   TST.B    FULL
        BNE      TESTW
        MOV.W    R0,LOC
        MOV.B    #1,FULL
```

In T_2 use:

```
TESTR   TST.B    FULL
        BEQ      TESTR
        MOV.W    LOC,R0
        CLR.B    FULL
```

In this case, FULL plays the role of the F/E bit of BOXLOC (with [FULL] = 0 initially), and the TAS instruction is not needed.