Spring 2000 CDA 4101 Solutions for Homework 8

Problem 1

The routing can be deduced by noting that 4 bits are required to identify the output port number and there are two switches through which each message must pass. Therefore, two bits must be used for each switch. In the case of the connection shown in the notes, assuming the ports are numbered from 0 to 15 starting at the top of the network's input and output sides, then the two leftmost bits in the destination index are used to select one of the four output ports in the first level switch (the first level is the left column of crossbars) and the two rightmost bits are used to select one of the four output ports in the second level switch. Using base 4 labels makes this easy to see. If all switches label their output ports $0, 1, 2, 4$ and the 4 bit destination address is labelled with a two digit base 4 number, e.g., 00 is 0 decimal 33 is 15 decimal, then the left base 4 digit is used to select the output port of the first level switch and the right base 4 digit is used to select the output port of the second level switch.

Consider the identity permutation, (using base 4)

input	$\rightarrow$	*output*
00	$\rightarrow$	00
01	$\rightarrow$	01
02	$\rightarrow$	02
03	$\rightarrow$	03
10	$\rightarrow$	10
11	$\rightarrow$	11
12	$\rightarrow$	12
13	$\rightarrow$	13
20	$\rightarrow$	20
21	$\rightarrow$	21
22	$\rightarrow$	22
23	$\rightarrow$	23
30	$\rightarrow$	30
31	$\rightarrow$	31
32	$\rightarrow$	32
33	$\rightarrow$	33

The switches for this permutation are all set to their identity permutation, i.e., all use the local permutation $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3$ where each switches input and output ports are labelled 0,1,2,3 from top to bottom. It is from this pattern that you can see source of the omega network name, i.e., a particular message traces an omega-like pattern.

The reversal permutation (again using base 4)

input	$\rightarrow$	*output*
00	$\rightarrow$	33
01	$\rightarrow$	32
02	$\rightarrow$	31
03	$\rightarrow$	30
10	$\rightarrow$	23
11	$\rightarrow$	22
12	$\rightarrow$	21
13	$\rightarrow$	20
20	$\rightarrow$	13
21	$\rightarrow$	12
22	$\rightarrow$	11
23	$\rightarrow$	10
30	$\rightarrow$	03
31	$\rightarrow$	02
32	$\rightarrow$	01
33	$\rightarrow$	00

can also be passed without conflict. In this case, all switches are set to the local reversal permutation, i.e., $0 \rightarrow 3, 1 \rightarrow 2, 2 \rightarrow 1, 3 \rightarrow 0$.

The down-end-around-shift can also be implemented without conflict. The permutation is given by

input	$\rightarrow$	*output*
00	$\rightarrow$	33
01	$\rightarrow$	00
02	$\rightarrow$	01
03	$\rightarrow$	02
10	$\rightarrow$	03
11	$\rightarrow$	10
12	$\rightarrow$	11
13	$\rightarrow$	12
20	$\rightarrow$	13
21	$\rightarrow$	20
22	$\rightarrow$	21
23	$\rightarrow$	22

$$30 \rightarrow 23$$
$$31 \rightarrow 30$$
$$32 \rightarrow 31$$
$$33 \rightarrow 32$$

In this case all switches in the right column are set to the local down-end-around shift, i.e., $0 \rightarrow 3, 1 \rightarrow 0, 2 \rightarrow 1, 3 \rightarrow 2$. The top switch in the left column is also set this way.

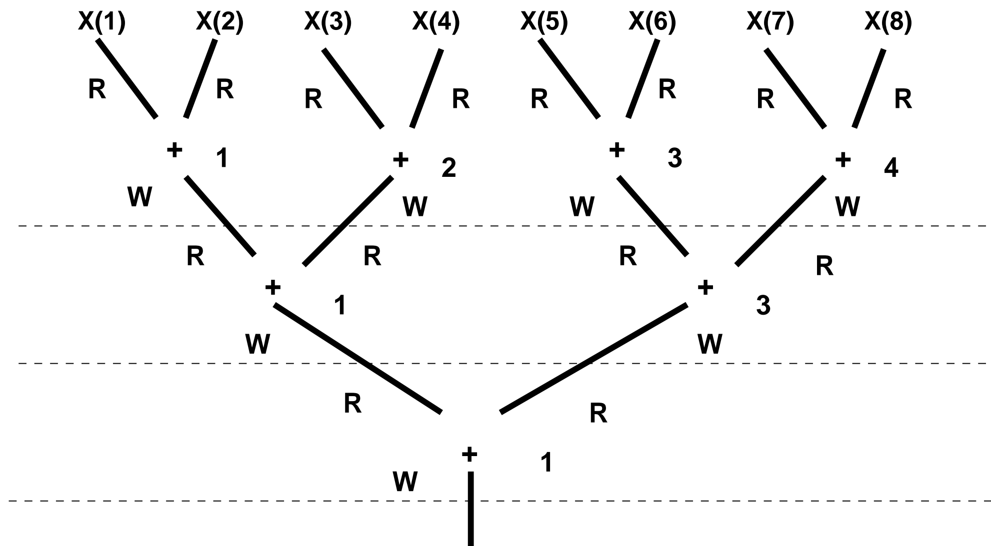
The second, third, and fourth switches in the left column are set to the local identity permutation, i.e., $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3$.

Problem 2

Suppose you have p processors in a shared memory programming / physically shared memory / physically centralized memory environment that does not use caches. Suppose the only synchronization mechanism available is a barrier. Describe how you would implement the summation

$$s = \Sigma_{i=1}^{p} x(i)$$

If we assume that p is a power of two for simplicity we can exploit the fact that summation is associative to execute the summation in $O(logp)$ major parallel steps. The dependence graph is as follows with synchronization enforced by barriers. The last barrier is used to indicate the completion of the summation to all processors.



We have also labelled each edge to show the memory traffic (a simple assumption is made that nothing is kept in registers across the barriers). Also the index of the processor on which each operation is performed is given as well as Read/Write behavior. Note that each set of writes is followed by a barrier to ensure completion.

In general, if $p = 2^k$ we have k barriers. Before each barrier each processor that is still active performs two reads, an add and a write followed by the barrier implying a time function of

$$T = k(2T_R + T_W + T_{add} + T_{barrier})$$

Note we have made the simplifying assumption that the barrier and write do not overlap (which is often not true).

A single processor without a cache would take

$$T_1 = pT_R + (p-1)T_{add} + T_W$$

So to fill these times in we can use the assumption about the number of outstanding requests to memory. We make the simplifying assumption that both reads and writes take time proportional to the latency in the network, i.e., $T_W = T_R = Clogp = Ck$.

$$\begin{aligned} T &= k(2T_R + T_W + T_{add} + T_{barrier}) \\ &= k(3Ck + T_{add} + T_{barrier}) \\ T_1 &= pT_R + (p-1)T_{add} + T_W \\ &= (p+1)Ck + (p-1)T_{add} \end{aligned}$$

The speedup can then be estimated as

$$S = \frac{Cplogp + pT_{add}}{3Clog^2p + T_{add} + T_{barrier}}$$

If we assume that an add can be done in time less than or equal to one memory transfer (which it almost always can) then we have

$$S \approx \frac{Cplogp}{3Clog^2p + T_{barrier}}$$

A barrier tends to be proportional to $logp$ in time so we have

$$S \approx \frac{Cplogp}{3Clog^2p + Dlogp} \approx \frac{p}{3logp}$$

This implies that the trend is to degrade perfect speedup by a factor of $O(logp)$ due to memory latency.

If the single processor has a cache that can use its cache line to further improve the single processor time, this speedup degrades even further. Assuming the cache line is at least p data long then $x(1), \ldots, x(p)$ can be brought in paying latency for the first element but bringing the others in at the bandwidth of the network. So rather than a $Cplogp$ cost for data reads we would only pay $Clogp + E(p-1)$ where E would be 1 if the network bandwidth is the same as the cache bandwidth.

Problem 3

Since α_i is the amount of work that can be done using exactly i processors and there are only p processors, we have $\Sigma_{i=1}^{p} \alpha_i = 1$. In Amdahl's law we would expect that if we used all p processors we would require a factor of p less time, i.e., our speedup would be p. So Amdahl's law with only two modes of processing sequential and parallel would have a time function of:

$$t = T_{sequential}(\alpha_1 + \frac{\alpha_p}{p})$$

that is the parallel fraction reduces by a factor of p and the sequential fraction stays the same. This is simply a special case with $\alpha_i = 0$ for $i = 2, \ldots, p-1$ of a more general formula where it is assumed that the portion of work done on i processors experiences a speedup of i.

This general notion can be expressed as the following time function:

$$t = T_{sequential}(\alpha_1 + \frac{\alpha_2}{2} + \cdots + \frac{\alpha_{p-1}}{p-1} + \frac{\alpha_p}{p})$$

Problem 4

Suppose each processor is making a series of true false tests and we want to record for each processor the number of true results. We can create a data structure that is a one dimensional array $count(1) \ldots count(p)$ with one entry for each processor. We can assume that these elements are contiguous in the address space (as is the case for C and Fortran).

We also assume that $count(1) \ldots count(p)$ are all in the same line.

So every time a processor wants to update its count it will cause a false sharing of the line, i.e., there is no data dependence between the processors since all are accessing different words in memory, but they all want exclusive write control of the same line. This causes the coherence mechanism to invalidate all copies of the line in the caches on every write! This type of situation is not as contrived as it sounds. It is fairly common to see situations like this in certain scientific computations.

The remedy is to lay out the data in memory so as to split data that tends to be accessed by distinct processors across cache lines. One way is to pad the array count (e.g., make it a two dimensional array with a leading dimension greater than the cache line size). Another is to organize the data using a more record oriented approach rather than functional approach, i.e., do not group variables that perform the same function for different processors together as above, rather group count(i) with the other data that is consumed by processor i.

Problem 5

Given that each processor has the loop bounds and the code that constitutes a single iteration, the only information missing is the particular iteration that the processor should execute before looking for more work. One can implement this by having a counter in shared memory (in fact when hardware support is provided this count is often in a special shared register/fast memory). When each processor needs a new iteration number it executes a

critical section that reads the current value of the counter (which is the next unconsumed iteration number) and increments it by one. Note more than one processor cannot do this lest more than one think that they are to execute the same iteration. If the processor reads a counter value larger than the loop bound it knows that there is no further work for it to do and waits for a signal that the other processors have completed all of their work.

The critical section can be implmented as discussed in the notes, i.e., via a lock variable.

Contention for this lock is the main performance bottleneck especially at start up when all processors need work to do. There are several ways to mitigate lock contention, a hierarchy of locks may be used so subgroups of processors compete before in a tournament-like structure in order to get access to the counter. It is also possible to have the processor that succeeds in getting to the lock to make scheduling decisions for other processors who lost in the competition for the lock but currently have no work to do. The winner could assign iterations to them and itself incrementing the counter by the appropriate number before releasing the lock. The other processors would then not have to compete for the lock until the completed this newly assigned iteration.