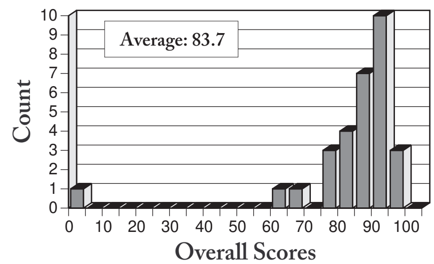
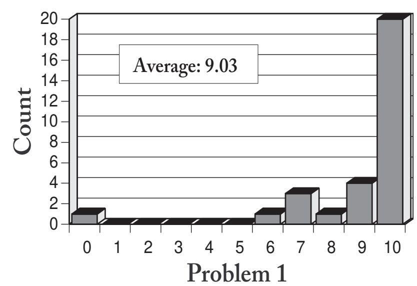
Multiprocessor Networks, CS294-5 — **Handout 4**
Prof. Eric A. Brewer — **November 14, 1994**

Solutions to Problem Set 1



The average score was 83.7 out of 100. The distribution of the scores is shown at right. Scores on the boundaries, such as 75, are counted in the bar to the right. The total point value for each problem is given at the beginning of the problem in square brackets. The end of each sub-question is similarly marked with that question's value.



1. [10] **The CM-5 allows time sharing as well as space sharing, and uses gang scheduling to ensure that all nodes within a partition execute the same job. During a context switch, what happens to the messages in the network?** [5]

As each node enters all-fall-down mode (AFD), messages fall from the routers to the nearest processor, which is essentially the node (leaf) directly below the router in the typical fat-tree diagram. Packets are marked as AFD packets to ensure proper handling even in the case in which the subsequent routers have not yet entered AFD mode.

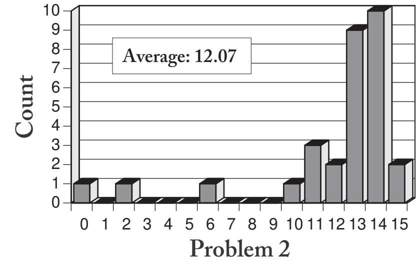
The advantage of "falling" over normal routing is even distribution of the packets among the nodes, which maximizes the bandwidth during draining and thus reduces context-switching time. For a deadlocked user, this mechanism drains the network for the next user and then reinjects the packets when the original user continues, which reinstates the deadlock.

Do they affect the next user? [2]

No, the network is cleared between users.

Are users penalized for having messages in the network when their time slice expires? [3]

Yes. As shown in Brewer & Kuszmaul '94, user timings reflect an increase of up to 21% for full networks over empty networks. This inflation is due to the cost of AFD processing, which is charged to the user (to an unknown extent). In effect, AFD shortens the user's time slice.



2. [15] **What are the primitive operators for the CM-5 control network?** [2]

Bit-wise OR, XOR, Signed max, Signed add, Unsigned add

How do you implement unsigned minimum (the minimum of unsigned 32-bit words) on top of this set? [2]

The key idea is to map the range of the unsigned integers one-to-one onto the signed integers and to convert the "<"

ordering into a ">" ordering. This allows us to use the signed maximum control network operation. The following code is from Strata; you can check the mapping by looking at the endpoints: 0 and 2^{32}-1.

```
#define INT_MAX 0x7fffffffu /* this must be unsigned */

GLOBAL unsigned GlobalMin(unsigned value)
{
    unsigned inverse = INT_MAX - value;
    assert(value <= INT_MAX);
    return INT_MAX - CombineWord(inverse, ReduceMax);
}
```

Give an algorithm for the median of n values (one from each node) that uses only the control network for communication (hard). [4]

Strata provides a median operation to assist with data collection in the presence of faulty timers. The algorithm, shown in Figure 1, guesses the median based on the average and then adjusts its guess up or down until the number of elements less than or equal to the guess is $n/2$. One key observation is that we can count the number of elements below the guess in one operation. This is an $O(n)$ algorithm in the worst case, but takes only a few iterations in practice. In the presence of duplicates,

```
GLOBAL unsigned GlobalMedian(unsigned x)
{
    unsigned avg, splitter, leq_count;
    int mid = PartitionSize/2;
    Bool was_less = False;

    /* find the average */
    avg = CombineWord(x, ReduceUadd) / PartitionSize;

    /* splitter starts at largest element <= average. Uses unsigned max operation. */
    if (x <= avg) splitter = GlobalMax(x);
    else splitter = GlobalMax(0);

    for (;;) {
        if (x <= splitter) leq_count = CombineWord(1, ReduceAdd);
        else leq_count = CombineWord(0, ReduceAdd);

        if (leq_count == mid) {
            return(splitter);
        } else if (leq_count < mid) {
            was_less = True;
            /* move splitter up: splitter = the smallest element of the remaining set */
            if (x > splitter) splitter = GlobalMin(x);
            else splitter = GlobalMin(UINT_MAX);
        } else { /* leq_count > mid */
            /* handle duplicates by detecting jump from leq_count < mid to
                leq_count > mid (without intervening leq_count==mid) */
            if (was_less) return(splitter);

            /* move splitter down: splitter = largest element of the remaining set */
            if (x >= splitter) splitter = CombineWord(0, ReduceMax);
            else splitter = CombineWord(x, ReduceMax);
        }
    }
}
```

Figure 1: Strata code for Median on the Control Network

the count many <u>never</u> be $n/2$. Instead, we detect the crossing from $count < n/2$ to $count > n/2$. Note that we can ignore the crossing in the other direction since it will immediately cross back over (and be detected).

Almost everyone turned in a version of quickselect, which is the median algorithm that follows from quicksort. It is quite tricky and I don't think anyone got it right. I took off only if it ignored duplicates (-1) or had an obvious infinite loop (-1). Another prominent error was forgetting to count the eliminated elements: the median of the reduced set is <u>not</u> the overall median (-1). Although quickselect is $O(\lg n)$ on average, the algorithm given below is simpler and just as fast in practice.

Note that we can't really have a processor opt out of a control-network operation. Instead, we have excluded processors submit dummy values that won't change the result. For example, excluded nodes contribute a 0 to the (unsigned) max operation, which can't possibly affect the result.

The CM-5 libraries build 32-bit floating-point addition on top of the control network; how is this done (roughly)? Hint and follow-up question: why doesn't this work for double-precision floating point? How could you implement 64-bit floating-point scan operations? [7]

The 32-bit floating point format is:

$$1.f \times 2^{e-127}(-1)^{s} \tag{1}$$

where f is the 23-bit mantissa, e is the 8-bit exponent, and s is the sign bit. The <u>rough</u> algorithm is:

1. Find the maximum exponent using the maximum operation.
2. Convert all elements to the common exponent, which means shifting f and updating e.
3. Sign extend and add the shifted mantissas using the signed addition operation.
4. Converted the sum back into IEEE format.

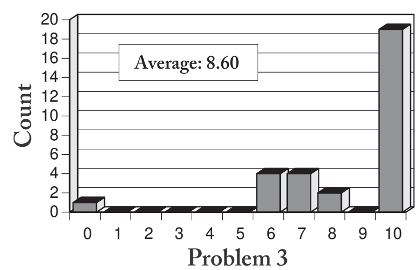
There are several problems with this algorithm. The basic one is that with n processors, each bit of the summed mantissa is affected by the $\lg n$ bits to its right. Equivalently, each bit in the input values can affect the bit $\lg n$ bits to the left. For example, by normalizing to the highest exponent, we throw out the $\lg n$ relevant bits to the right of the shifted LSB, which means we lose roughly $\lg n$ bits of precision. Similarly, with 24-bit numbers and $n > 2^8$, we may lose bits due to overflow. For simplicity, we will assume $n \le 256$. One way to improve the accuracy is to perform the sum iteratively: we start with the maximum exponent and we repeatedly <u>increment</u> the exponent until we get an overflow bit in the control network. The top 23 bits of the sum form our new f, with the implicit one now in the carry. This process ensures that we lose no bits to overflow and that the LSB correctly reflects all of the $\lg n$ bits below it. I marked off only if you <u>completely</u> ignored precision/overflow; mentioning the problem was sufficient.

Although we can squeeze out 23-bit mantissas with 32-bit arithmetic, it is hopeless with the 53-bit mantissas used in double-precision floating point. Furthermore, we would really like to handle n=16384, which is the advertised maximum size of the machine. The simplest way to do this is to use a sequence of 16-bit additions with 16 bits of overflow. To get full precision without iterating, we need to emulate a mantissa with 16+53+16=85 bits. We compute the maximum exponent, say E, and then normalize to E+16 to allow for overflow. We then perform a 96-bit addition via five 16-bit operations to compute the 96-bit result, which has no overflow and contains at least 64 bits of precision. Note that we only need 5 operations, not 6, since the for sixth one all of the 16-bit values are zero except for the 16-bit carry from the fifth operation. Thus, the fifth add really gives us the top 32 bits. Finally, we convert this result back to double-precision floating point.

Other issues we have ignored include sticky bits, guard bits, rounding, and the various forms of infinity and NaN's (not a number).

We could also use the data network to do 64-bit floating-point operations. This is nice in that we can use handlers with normal floating-point instructions to provide a complete IEEE implementation. This is actually worse for accuracy! A fundamental requirement for scan/reduction operations is that the binary operator must be associative, so that the operations can occur in any order. Floating point addition is not associative! For example, suppose $y = x >> 53$, i.e., y is the same as x except that the exponent is smaller by 53. This implies that $x + y = x$, since only the top 53 bits matter. Similarly, $(...((x + y) + y) ... +y) = x$. However, if we add up the y's first and then add x, we get a much more accurate answer. With the control network we avoided this problem by only using integer addition, which is associative. Presumably, we could build a data-network version that uses two 64-bit additions to end up with one accurate 64-bit result, even with 16384 nodes. This would be nearly as fast, since we can fit two 64-bit numbers in one active message.

3. [10] **The normal-case bidirectional transfer rate for Strata on the CM-5 is one active message in each direction per 99 cycles, or 10.66 MB/sec. (16 bytes of payload $\Rightarrow$ $2\frac{16\,(33\text{ Mhz})}{99}$ = 10.66 MB/sec.) Assume that floating-point conversion due to heterogeneous nodes requires 50 cycles per 32-bit floating-point value. What is the net bandwidth counting the conversion?** [5]



One packet requires 4 conversions for a total of 200 cycles. This gives us a total period of 299 cycles, which corresponds to 3.53 MB/sec.

What if the conversion is done on the sender side? [2]

In the normal case for bidirectional transfer, it does not matter which side performs the conversion, since the period is the total of the sender and receiver cycles. Since the processor performs both message passing and conversion, there is no opportunity for overlap.

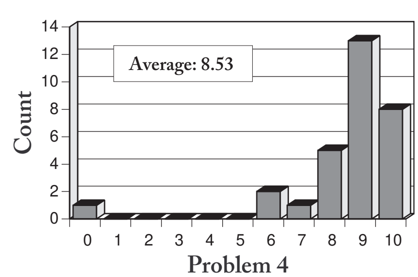
Where should the conversion be done and why? [3]

The conversion should be done on the sender side; there are many situations in which conversion at the sender is an advantage. For example, for unidirectional transfers, sender-side conversion leads to 237 cycles per packet, versus 262 cycles for receiver-side conversion. In general, sender-side conversion keeps the network empty.

An interesting optimization is to split the conversion between the sender and the receiver. If we perform 2 conversions at the sender and 2 at the receiver, we get times of 137 and 162 cycles respectively, which gives us (with bandwidth matching) the best unidirectional throughput (6.52 MB/sec).

In general, the optimal location of the conversion depends on the bottleneck. For example, for 1-to-n sending the receivers can operate in parallel and thus completely hide the conversion cost.

4. [10] **Give some specific evidence that reducing the average packet size leads to lower latency and higher network utilization (ignoring software overhead).** [3]



Figures 2, 3 and 5 of Agarwal '91 all show that smaller messages lead to better network utilization. The use of cells in ATM also reflects the performance of small messages. Along the same lines, Brewer & Kuszmaul '94 shows that interleaving small packets makes better use of the network resources.

Next, give some specific evidence that smaller messages lead to lower throughput due to software overhead. [3]

Figures 2a and 2b from Kay & Pasquale '93 show that the relatively fixed non-data-touching overheads dominate the cost of small messages, but can be amortized for large messages. Similarly, Karamcheti & Chien '94 show that large messages are required to amortize the substantial fixed costs of message passing. Brewer & Kuszmaul '94 indeed shows that large block transfers can achieve twice the throughput of small transfers.

Given these two conflicting items, what is required to achieve maximum network performance? [4]

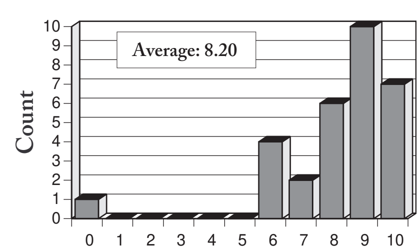
The best case occurs when long messages are segmented into many small packets automatically by the hardware and the reassembled in hardware at the receiver. This process, called segmentation and reassembly (SAR), is fundamental to the design of most ATM network interfaces. The software can amortize its costs over large messages, while the network only sees small cells.

Agarwal pointed out that a sequence of small packets to the same receiver may not see the benefits of small packets over one long packet, since the blocking behavior is nearly the same. Virtual circuits mitigate this substantially, since only the virtual circuit is blocked, while there are "passing lanes" for other circuits. Virtual channels do not mitigate the problem nearly as much, since a series of small packets can block all of the virtual channels for one link (other than those required for deadlock prevention). For virtual channels bigger packets probably are better than a series of small packets to the same receiver, since a single larger packet is restricted to one virtual channel. If the bottleneck is expected, bandwidth matching can exploit the benefits of small packets, since there will be space between the packets to allow for interleaving at the routers.

A better answer, of course, is to interleave the packets at the sender, which will clearly get all of the small-packet benefits. Thus, we really want an interface that handles multiple large blocks simultaneously, performs SAR in hardware, and interleaves (in hardware) the resulting small packets. The Magic chip in FLASH actually meets these requirements: it segments blocks into cache-line sized packets and then interleaves the packets.

For small messages, the best we can do is to reduce the software overhead. This can be done by raising the services provided by the network, as suggested in Karamcheti & Chien '94. User-level message passing, reliable delivery, and in-order delivery are the three most important steps in this direction. Using shared-memory hardware also helps by reducing the overhead to a single (load or store) instruction. The active-message model also helps by allowing the application to pay only for the extra services required, such as packet reordering or buffer allocation.

5. [10] **Clark and Tennenhouse argue that only the data-manipulation phases of TCP really affect performance. Kay and Pasquale argue that most of the overhead comes from non-data operations. When is data manipulation the dominant source of overhead? Answers should be complete and reasonably precise, which should take about half of a page.**



Problem 5

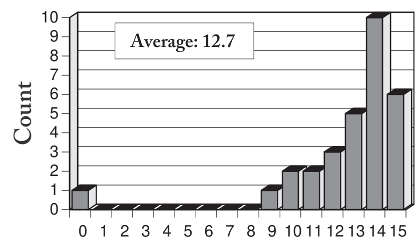
Many factors determine when data manipulation is the dominant source of overhead. The biggest factor is the size of the message: the cost of large messages is dominated by data manipulation. The key difference between the papers is the assumption regarding packet sizes. Clark & Tennenhouse assume 1000-

2000 byte packets, while Kay & Pasquale show that some workloads, such as NFS traffic, are dominated by short messages (<200 bytes). Both papers agree that small packets are dominated by non-data-touching overheads.

Another major factor is the overhead for format conversion, which can be 97% of the total overhead. Systems that require data conversion, such as those with heterogeneous nodes, are far more likely to be dominated by data manipulation.

Finally, for short packets, data manipulation dominates only when the fixed (non-data-touching) overheads are small. This occurs when the network hardware provides relatively high-level semantics. For example, reliable networks with in-order delivery substantially reduce the software overhead and thus decrease the crossover point at which data manipulation begins to dominate the overhead. Similarly, systems that require dynamic memory allocation upon message arrivals exhibit higher crossover points. User-level message passing decreases the crossover point substantially.

6. [15] **What happens in METRO if the receiver is not expecting a block transfer?** [4]



Problem 6

Since there is no buffering in the network, the arriving message is simply dropped. The receiver appears as a blocked router, which causes either a detailed report of the failure after the message is completely sent, or a short report that aborts the transfer immediately. In both cases, the sender retries the connection.

What should the network interface do to reduce this problem? [4]

The most important step is to provide highly available buffering, so that incoming messages can be buffered until the receiver is ready. For short messages, this should work quite well, but for long messages, the NI will eventually run out of buffer space and the entire message should be dropped. Thus, for long transfers it is probably worth the round trip to ensure either sufficient buffer space or an attentive receiver.

Some people suggested that the NI should indicate that the transfer failed at the receiver rather than the router, so that the sender can use a better retry strategy. This requires no change to the NI, since the aborted transfer already indicates the stage in which the transfer was aborted. Since all receivers are in the last stage, the sender can use the stage information to distinguish receivers from routers.

What happens when there are multiple senders for one receiver? [2]

It is unlikely that the receiver can actually handle multiple active senders, but we could allow multiple ports into the receiver with buffering to handle short-lived congestion at the receiver. More likely, all but one of the senders will fail to connect.

Classify the techniques from Brewer and Kuszmaul in terms of their usefulness for METRO. [5]

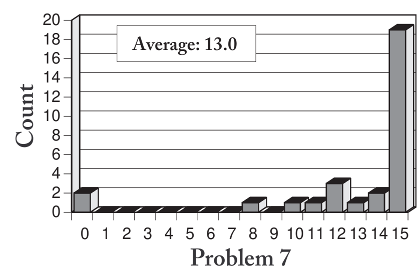
Permutations are definitely useful, since the make even use of the receivers. The use of dedicated circuits means that congestion is not an issue. Note that n-on-1 sending won't affect the (one) successful connection. Thus, barriers may not be that useful, since the receiver can already serialize the transfers simply by declining connections.

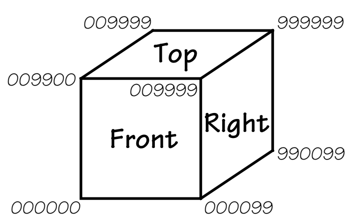
The lack of buffering implies that we probably need DMA hardware at the receiver to ensure that the incoming data can be handled. There are two interesting possibilities for DMA. In the first scenario, a node can do DMA for sending and receiving simultaneously, although the existence of a single bus couples their behavior. In particular, the reception rate depends on whether the receiver is also sending, since sending requires memory-bus cycles. This argues that the sender wants to use bandwidth matching to match the rate of a receiver that is also sending. Without matching, the sender could outstrip the receiver. Once again the normal case should be that both ends are both senders and receivers and there is no loss due to bandwidth matching.

In the second scenario, there is only one DMA engine, which implies that a node can send or receive (large blocks) but not both. Small blocks could be buffered in the NI. In this case, we don't need bandwidth matching, but barriers become more useful. The problem is that a long transfer to a node that is using its DMA engine will be buffered for a while and then dropped, which ties up network resources for a relatively long time without making any progress. By adding barriers, we avoid these failed transfers. In this scenario, the right solution is to use permutations based on pair-wise exchange, since we can exploit the "turn around" option to build a bidirectional transfer. This would cut the number of barriers required in half, since each round is now bidirectional rather than unidirectional.

The third technique is packet interleaving, which is not useful in Metro since 1) small packets have much higher overhead, and 2) once we get a connection, we know we have the undivided attention of the receiver. What is useful, however, is making sure the sender has several outstanding block transfers. This avoids head-of-line blocking, since the sender can try each of the connections in round-robin order until one succeeds.

7. [15] **Assume that we have a strided block transfer that can send a sequence of evenly spaced, identically sized blocks with a single coprocessor call, so that the transfer occurs concurrently with the main computation. Assume we are solving a 3D PDE problem iteratively, which means that after each iteration the six *faces* of our local 3D volume must be transferred to the corresponding six nearest neighbors. The local volume is a 100x100x100 array of doubles. How many calls are required to send all of the data from one node?** [5]





The short answer is six. To simplify discussion, lets label the faces and corners as shown at left. The corners are labelled with the corresponding index into a one-dimensional array of doubles, which has 100x100x100 = 1,000,000 elements. The top two digits are the z coordinate, the middle two are y, and the last two are x. The front and back faces are contiguous and thus require one message each. For example, the front face contains elements 000000 through 009999. The top and bottom faces can each be sent with one strided transfer with blocks of size 100 and a stride of 10,000 elements. For example, the bottom face contains elements: 000000 → 000099, 010000 → 010099, ..., 990000 → 990099. Finally, the left and right faces can also be sent as strided transfers, this time with a block size of 1 and a stride of 100. The trick is to view the face as a sequence of vertical columns. For example, for the right face, the front-most column contains elements: 000099 (front bottom right corner), 000199, ..., 009999 (front top right corner). The next column in runs from 010099 to 019999. Note that the first element of the second column is 100 units from the last element of the first column (010099 - 009999 = 100). Thus, all of the columns can be combined into one giant transfer.

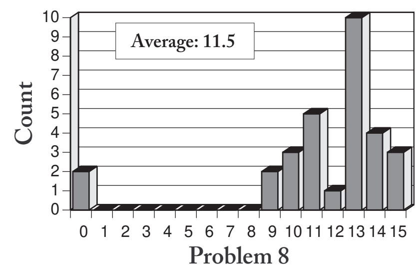
How many messages would be required using only contiguous block transfers? (You're not allowed to send any extra data; for example, sending the entire volume to all neighbors would require only six messages.) [2]

The front and back faces still only require one message each. The top and bottom faces each require 100 messages of 100 doubles. Finally, the left and right faces require 10,000 messages of only one double. The total message count is 20,202.

If we assume that it takes 100 + 0.25*k* cycles for any *k*-byte transfer, how much faster is the strided transfer per iteration? [8]

In each case, the total <u>data</u> sent is 60,000 doubles, which corresponds to (0.25)(8)(60,000) = 120,000 cycles. For the strided transfers, the <u>overhead</u> is (100)(6) = 600 cycles, for a total of 120,600 cycles. For the contiguous transfers, the overhead is (100)(20,202) = 2,020,200 cycles, for a total of 2,140,200 cycles. The strided transfer is thus $\frac{2{,}140{,}200}{120{,}600} = 17.7$ times faster.

8. [15] **List several reasons why adaptive routing may be a bad idea.** [8]



Here are the primary reasons:

- Out-of-order delivery typically implies higher software overhead.
- Adaptive routing is complicated and may increase the cycle time of the network.
- In the presence of link-level flow control (backpressure), adaptive routing can lead to extensive congestion, since a busy receiver backs up <u>all</u> paths to that receiver.
- Constrained routing to avoid deadlock leads to uneven use of the links, effectively reducing the maximum bandwidth.

There are additional reasons as well:

- A fixed path is much simpler in terms of trying to provide end-to-end quality of service, particularly control of the interarrival times.
- The complexity of adaptive routers implies longer design time and more chip area.

When does it make sense to use adaptive routing? There is no perfect answer here; just give a reasonable argument that covers several variables including (at least) topology, traffic pattern, fault tolerance, and end-to-end software overhead. [7]

There are only two justifications for adaptive routing. First, if we expect far more link/router failures than node failures, we can use adaptive routing to provide substantial fault tolerance. (Although it is by no means simple to do so.) Second, if the bisection bandwidth and traffic patterns are such that we expect more internal congestion than endpoint congestion, then we can use adaptive routing to reduce the internal congestion. Note that adaptive routing <u>hurts</u> performance in the presence of endpoint congestion. Thus, on topologies such as a mesh that have limited bisection bandwidth, we can use adaptive routing to make the most of that scarce resource.

Finally, adaptive routing at the time of connection setup, as in Metro, provides fault tolerance and avoids internal congestion without <u>any</u> of the above problems of adaptive routing. Thus, for short-haul networks with $O(n)$ bisection bandwidth, such as the metabutterfly, the Metro architecture makes a lot of sense.