Handout 10

The following homework assignment is due in three weeks.

1 Third Homework Assignment

1. The performance of the *queue-lock* based implementation in Figure 1 of a *compare&swap* primitive on a the Symmetry 18 processor shared bus machine we described in class, degrades *rapidly* as the number of processors increases, giving the same curve as a regular *Test&Set* based one. What can explain this kind of bad performance (we saw in class that queue locks perform much better than *Test&Set* for the counting application).

2. A *sorting network* is a network built out of *comparators*, which are gates having two input lines and two output lines. Given two integer inputs x_1 and x_2, a comparator returns two outputs $y_1 = max(x_1, x_2)$ and $y_2 min(x_1, x_2)$. In other words, it gets two inputs, compares them, sends the larger to the lower output line and the smaller to the higher output line. A 0/1-*comparison network*, like a balancing network, is an acyclic network of comparators. It functions as follows: on each of its n input lines i, an integer $x_i \in \{0, 1\}$ arrives. Then in a synchronous manner, all values pass through the first layer of comparators (depth 1), then the second layer etc, all leaving at the same time on the network output wires. A 0/1-*sorting network* is a 0/1-*comparison network* such that for any two output lines y_i and y_j where $i < j$, it is the case that $y_i < y_j$. Prove that the layout of every counting network is isomorphic (has the same layout) as that of some 0/1-*sorting network*. Show that the opposite is not true, i.e., not every sorting network is isomorphic to some 0/1-*counting network*.

3. A *smoothing network* is a balancing network for which in any quiescent state, the output sequence of width n has the property that for all $y_0, .., y_{n-1}$,

   if $i < j$ then $|y_i - y_j| \leq 1$.

   Any counting network is thus a smoothing network. What do we get if we connect the outputs of an arbitrary smoothing network of width n to the inputs of a balancing network of width n which is isomorphic to some sorting network? Why?

4. A 3-*balancer* is a balancer with three input lines and three output lines, whose output sequences, just like its 2-line relative, have the step property in any quiescent state. Construct a depth-3 counting network with 6 input and output lines from 2-balancers and 3-balancers. Prove the correctness of your construction.

```
COMPARE&SWAP(r : register, old : value, new : value) returns(value);
    Aquire(queue-lock);
    previous := r;
    if previous = old then r := new;
      end if
    return (previous);
    Release(queue-lock);
    end COMPARE&SWAP
```

Figure 1: Queue-lock based Compare&Swap

5. (Following the last homework assignment) Prove that a single *three-valued* COMPARE&SWAP object (values $\perp, 0, 1$) can solve n process consensus if the inputs of processes are in $0, 1$.

6. (Following the last homework assignment) Prove that n instances of a *three-valued* COMPARE&SWAP object can solve n process consensus even if the inputs of processes are **not** in $0, 1$.

2 Review from last lecture

Last time we talked about how to implement a *shared queue* effectively. We coordinated so we would always have producers and consumers meeting – by using head and tail counters which have *locks*. The only sequential bottleneck is the counters, but we weren't getting best performance because of *contention* arising from the use of the *Test&Set* primitive.

We improved upon this be using the *Test&Test&Swap* primitve, testing locally. This still takes some time due to quiescence. Then we tried backoffs - delays when accessing the shared resources. We introduced four delay policies:

1. Dynamically-Sized
2. Statically-Sized
3. Delay after lock is released
4. Delay after each acceess to the lock

Exponential backoff worked best in distributed systems. The queue lock distributes the data structure so each process spins locally. Upon release of the critical section, it would tell the next process that it is its turn. This provides localization and immediate notification of the next guy in line. Distribution helps eliminate contention and extend control. The *queue-lock* is tightly synchronized.

The inherent problem with all these solutions is that we're getting very little parallelization. Optimal sequential performance is still a sequential bottleneck. On a bus based machine, maybe that's not too bad. However, on the butterfly or Alewife, this solution is very limiting.

```
function fetch_and_add(counter : tree, incr : int) returns int

  // Part One
  last_level, saved_result : int
  node : tree_node

  level : int := FIRST_LEVEL
  going_up : boolean := TRUE
  repeat
      node := get_node(counter, level, pid)
      lock(node)
      if node.status = RESULT then
          unlock(node)
          repeat while node.status = RESULT  // change: minimize locking
      else if node.status = FREE then
          node.status := COMBINE
          unlock(node)
          level := level+1
      else // COMBINE or ROOT node
          last_level := level
          going_up := FALSE
  while going_up

  // Part Two
  total : int := incr
  level := FIRST_LEVEL
  repeat
      visited : tree_node := get_node(counter, level, pid)
      lock(visited)
      visited.first_incr := total
      if visited.wait_flag then
         total := total + visited.second_incr
      level := level + 1
  while level < last_level
```

Figure 2: Combining Tree Code: Parts One and Two

```
// Part Three
if node.status = COMBINE then
    node.second_incr := total
    node.wait_flag := TRUE
    repeat
        unlock(node)
        repeat while node.status = COMBINE  // change: minimize locking
        lock (node)
    while node.status = COMBINE
    node.wait_flag := FALSE
    node.status := FREE
    saved_result := node.result
else
    saved_result := node.result
    node.result := node.result + total
unlock(node)

// Part Four
level := last_level - 1
repeat
    visited : tree_node := get_node(counter, level, pid)
    if visited.wait_flag then
        visited.status := RESULT
        visited.result := saved_result + visited.first_incr
    else
        visited.status := FREE
    unlock(visited)  // bug fix: need an unlock here
    level := level - 1
while level >= FIRST_LEVEL

return saved_result
```

Figure 3: Combining Tree Code: Parts Three and Four

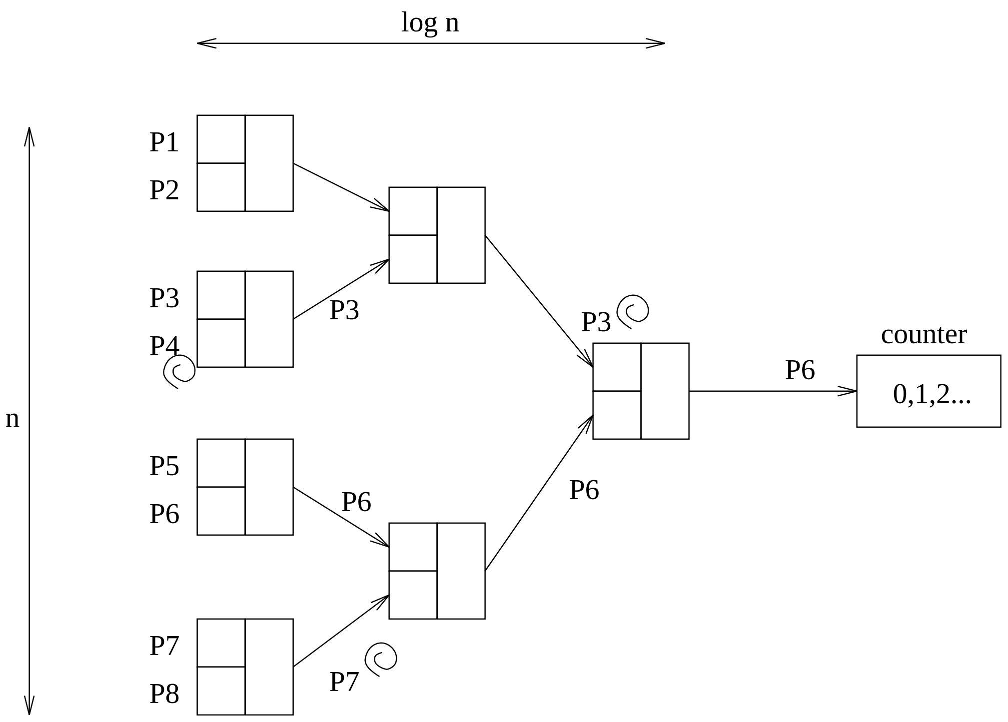


Figure 4: Combining Tree Example

3 Combining Trees

Figures 2 and 3 contain the code for a *combining tree* Goodman et. al. [Goodman]. The code is a revised version of the code of [Goodman] due to [beng], where comments mark changes to enhance performance of the algorithm on the Alewife machine, and a fix to a bug in the original code.

Consider the following example (illustrated in Figure 4): Of P_7 and P_8, P_7 arrives before P_8 and moves unchallenged up to the next level. The same thing happens with P_6. So P_6 and P_7 meet at the next level. P_6 wins and moves up with P_7's request while P_7 spins. Meanwhile, P_3 wins at the first level, brining P_4's request up to the next level while leaving P_4 to spin. As neither P_1 or P_2 arrived in time to compete, P_3 moves on to the next level uncontested to compete with P_6. P_3 wins, accumulating P_6's requests for itself and P_7. When P_3 reaches the counter, it increments it by four, the total number of requests it has accumulated. P_3, the grand winner, now has 0, 1, 2, and 3 and can hand these out any way he wants. P_3 gives P_6 two of these numbers as he returns along the tree. P_6 meets P_7 and gives him 2 while keeping 3. P_3 returns to P_4 and awards it 1 and keeps 0.

This is an example of *software combining* [Goodman]. *Combining trees* can also be implemented in hardware, for example the NYU Ultra Computer. Software combining works extremely well. Even though we're handing our numbers from only one location, the number of requests in at one time can be very high. Total speed up is: $Throughput = \frac{n}{2\ \log n}$ and $Latency = 2\ \log n$.

What kind of hardware is needed to implement a combining tree in software? It is possible

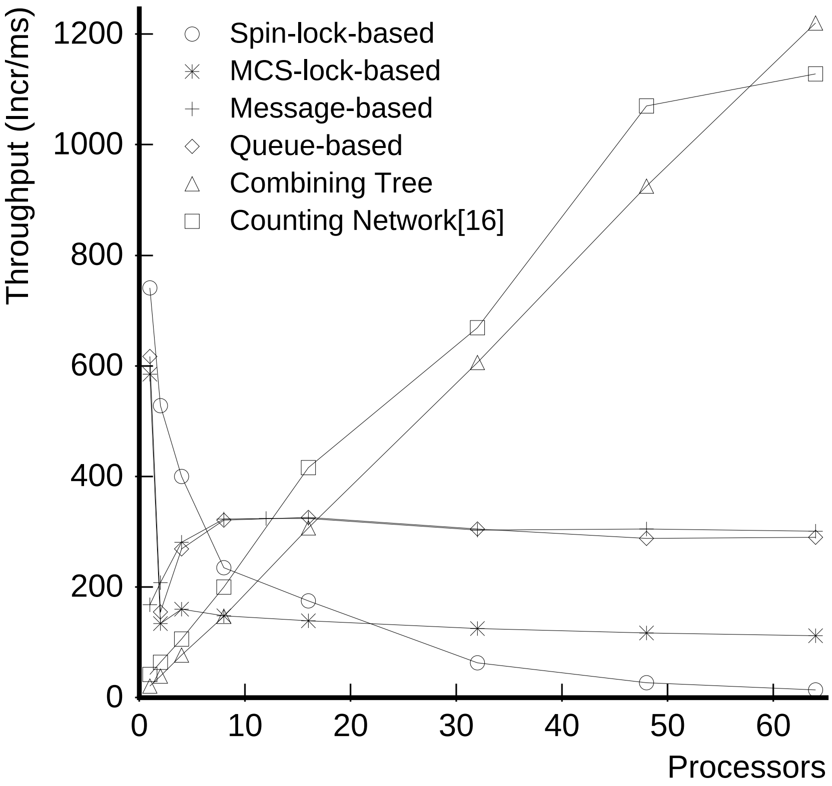


Figure 5: Throughput of the five Counting Techniques

to implement with a *queue-lock* using *Compare&Swap* or even just using a *Full/Empty Bit*.

You get optimal performance from a combining tree when all processes pound on the lock. Introducing delay drastically affects performance. Combining trees are great unless there's asynchrony in the system. Since you are depending on several processes to bring you your number, introducing even a small delay in one of those processes could greatly impact the performance of many other processes.

4 Balancing Networks

We need to find a way to parallelize without demanding synchronization. We'll do that by trying to distribute the locations passing out numbers (instead of trying to parallelize access to one place of number giving).

If we want the counters to all behave like one counter, what's the essential element? We want to avoid misses. Essentially, if you look at this implementation, what we want is a counter accessible by n asynchronous processors without duplication or omission. For example, if x gets 5, 0-4 must either have been taken, or are concurrent and will be taken. So when all the requests that have come in have left, the output needs to obey the step

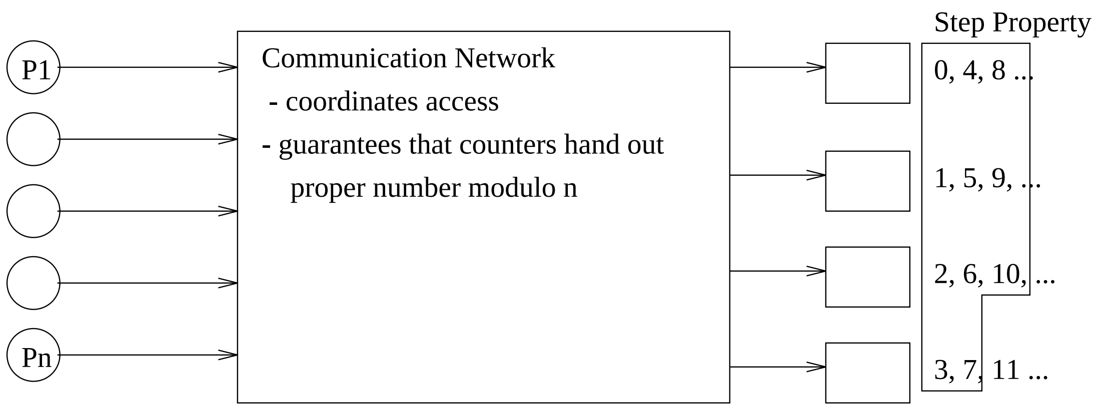


Figure 6: A Counting Network with the Step Property

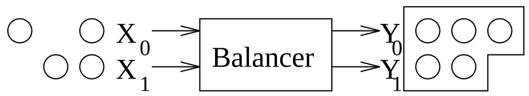


Figure 7: Balancer

property. In addition, for some number p of processes, we want to optimize.

How do we build the box for coordinating access – that's a Counting Network. A *Counting Network* is a network built out of very simple boxes called balancers. A balancer is an asynchronous device. Tokens arrive asynchronoussly on input wires and are output asychronously on output wires. Within a finite amount of time, all tokens that arrive leave. The system is said to be in a quiescent state when all the tokens that have arrived have left.

In any quiescent state, a balancer obeys the step property such that:

$$y_0 = \frac{x_0 + x_1}{2}$$

$$y_1 = \frac{x_0 + x_1}{2}$$

$$y_0 = \{y_1, y_1 + 1\}$$

$$x_0 + x_1 = y_0 + y_1$$

What if the tokens are arriving continuously? The balancer must have the step property only if you stop it. There are also no requirements about the order of arrivals in relation to the order of exit. Now we can use balancers to build balancing networks. A balancing network is an acyclic wiring of balancers. The depth of the network is:

$$depth(y) = max(depth(x_0), depth(x_1)) + 1$$

The depth of input wires (x_i) is zero.

In a MIMD shared-memory architecture, a balancer can be represented as a record with two fields: *toggle* is a boolean value that alternates between 0 and 1, and *next* is a 2-element array of pointers to successor balancers. A balancer is a *leaf* if it has no successors. A process shepherds a token through the network by executing the procedure shown in Figure 8. It toggles the balancer's state, and visits the next balancer, halting when it reaches a leaf. Advancing the *toggle* state can be accomplished either by a short critical section guarded by a spin-lock, or by a *read-modify-write* operation if the hardware supports it. Figure 8 shows the code for a network implementation.

```
balancer = [toggle: boolean, next: array [0..1] of ptr to balancer]
traverse(b: balancer)
  loop until leaf(b)
        i := rmw(b.toggle := ¬ b.toggle)
        b := b.next[i]
        end loop
  end traverse
```

Figure 8: Code for Traversing a Balancing Network

5 Counting Networks

A counting network of width w is a balancing network whose outputs $y_0 \ldots y_{w-1}$, satisfy the following step property: In any quiescent state,

$$0 \leq y_i - y_j \leq 1, \; for \; i < j$$

For an example, consider the BITONIC[4] Counting network (with depth 3) in Figure 9. Figure 9 shows tokens run through it asychronously to see how it performs.

The point about this data structure is that every concurrent execution can be mapped to a sequential execution. In addition, you can build a general construction for any width w that is a power of 2. To prove this we will first show a general construction algorithim and then prove that it meets the quiescent state condition. Concurrent executions, don't really concern us.

Construction of BITONIC[w] is shown in Figure 10. We define BITONIC[w] to be the network constructed by passing the outputs from two BITONIC[$w/2$] networks into a MERGER[w] network, where the induction is grounded in the BITONIC[1] network which contains no balancers and simply passes its input directly to its output.

$$Depth = \frac{\log k \cdot \log(k+1)}{2} = \frac{\log w \log w + 1}{2}$$
$$= O(\log^2 w)$$

So now how do you build a MERGER[w] network for w a power of 2? A merger gets as inputs 2 step properities, x and x', both with width w, (possibly different numbers of tokens)

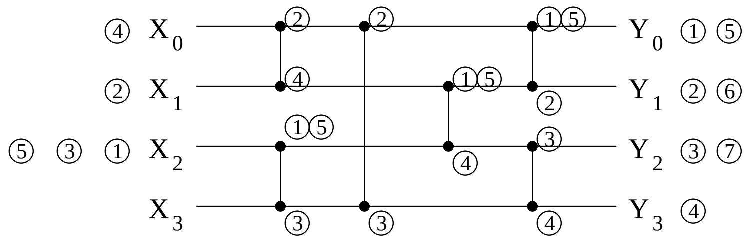


Figure 9: A Bitonic[4] Counting Network

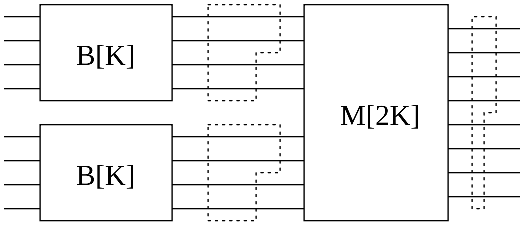


Figure 10: A General Bitonic[2K] Counting Network

and produces output with 2K step property. Again, since we're only making guarantees about the quiescent state, we only need to look at it after it ran. Define the width w balancing network MERGER$[w]$ as follows. It has two sequences of inputs of length $w/2$, x and x', and a single sequence of outputs y, of length w. MERGER$[w]$ will be constructed to guarantee that in a quiescent state where the sequences x and x' have the step property, y will also have the step property, a fact which will be proved in the next section.

We define the network MERGER$[w]$ inductively. Since w is a power of 2, we will repeatedly use the notation $2k$ in place of w. When k is equal to 1, the MERGER$[2k]$ network consists of a single balancer. For $k > 1$, we construct the MERGER$[2k]$ network from two MERGER$[k]$ networks and k balancers. Using a MERGER$[k]$ network we merge the even subsequence $x_0, x_2, \ldots, x_{k-2}$ of x with the odd subsequence $x'_1, x'_3, \ldots, x'_{k-1}$ (i.e., $x_0, \ldots, x_{k-2}, x'_1, \ldots, x'_{k-1}$ is the input to the MERGER$[k]$ network.) while with a second MERGER$[k]$ network we merge the odd subsequence of x with the even subsequence of x'. Call the outputs of these two MERGER$[k]$ networks z and z'. The final stage of the network combines z and z' by sending each pair of wires z_i and z'_i into a balancer whose outputs yield y_{2i} and y_{2i+1}.

The MERGER$[w]$ network consists of $\log w$ layers of $w/2$ balancers each. MERGER$[w]$ guarantees the step property on its outputs only when its inputs also have the step property—but we can ensure this property by filtering these inputs through smaller counting networks. Putting it all together we have a counting network construction consisting of $\binom{\log w+1}{2}$ layers each consisting of $w/2$ balancers.

Now to prove its correctness:

Lemma .1 *If you have a sequence that has the step property, then every subsequence has the step property.*

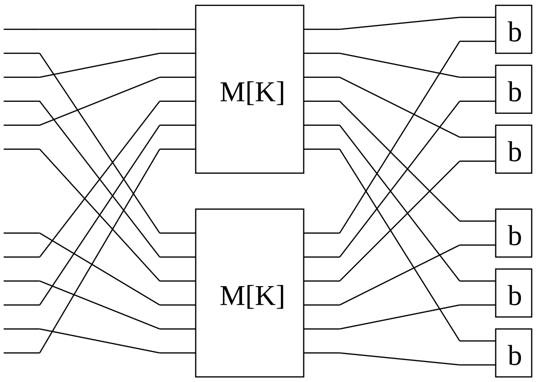


Figure 11: A General 2K Merger

Lemma .2

$$|x_{even}| = \lceil \sum_i \frac{x_i}{2} \rceil$$

$$|x_{odd}| = \lfloor \sum_i \frac{x_i}{2} \rfloor$$

$$|x_{even}| - |x_{odd}| = \{0, 1\}$$

Lemma .3 *For any two sequences, if $\sum_i x_i = \sum_i y_i$, then their step property looks the same (if they have the same number of tokens).*

Lemma .4 *If two subsequences differ by 1, then it is on a specific value.*

Claim .5

$$|z - z'| = \lceil \sum_i \frac{x_i}{2} \rceil + \lfloor \sum_i \frac{x_i}{2} \rfloor - (\lfloor \sum_i \frac{x_i}{2} \rfloor + \lceil \sum_i \frac{x_i}{2} \rceil)$$

Proof: Step z and z' differ at most by one. If they do differ, it will be at a specific j. z_j and z'_j feed into the same balancer, so it will come out ok in the end. ■

The resulting network will have depth of $O(\log^2 w)$. There are also other types of networks you can build. One example, periodic networks, also have depth $O(\log 2w)$ but are uniform in structure, every layer of depth $\log n$ has the same layout.

6 Implementation

We now look at the performance of counting networks [beng] relative to other known methods by running a collection of benchmarks on a simulated distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine [Alewife] developed by Agarwal, Chaiken, Johnson, Krantz, Kubiatowicz, Kurihara, Lim, Maa, and Nussbaum. In the performance graph in Figure 12, counting networks perform comparably to combining trees, but seem to saturate. (This is a shared memory implementation)

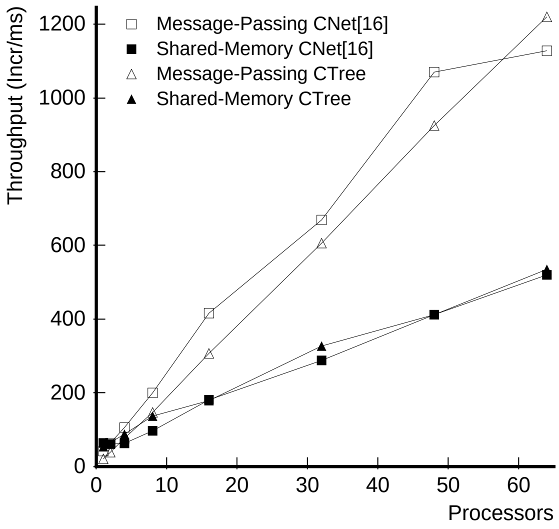


Figure 12: Throughput of Combining Trees and Counting Networks

For a message passing implementation, the balancers are mapped onto the processors. A message passing implementation also eliminates the need for cache coherence. In a shared memory implemenation, the only thing actually shared are the toggles, which are done with mutual exclusion and queue locks.

In general, the chances that a lot of processes will hit a balancer at the same time are small. The main advantage over combining tree is that a counting network is not as sensitive. (Although using mutual exclusion in the implementation can cause problems) The data structure is distributed and coordinated (Disco). The counting network and the combining tree both perform well. Whether the highly parallel implementation is worth it depends on the application.

Why does the performance of Counting Networks degrade in high numbers of processes in Figure 13?

1. If the number of processes is less than the number of balancers, then there are points in time during which fewer processers can come out in any time unit.
2. If the number of processes is greater than the number of balancers, then processes arrive at the balancers at the same time and have to compete, resulting in contention.
3. If the number of processes is equal to the number of balancers (saturation point), then we get good performance.

The best ratio of balancers to processors is: B[w]= $w \log^2 w = P$

7 More Performance Graphs

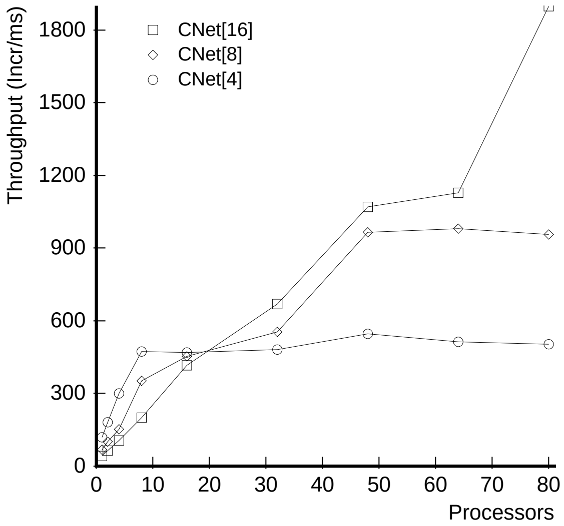
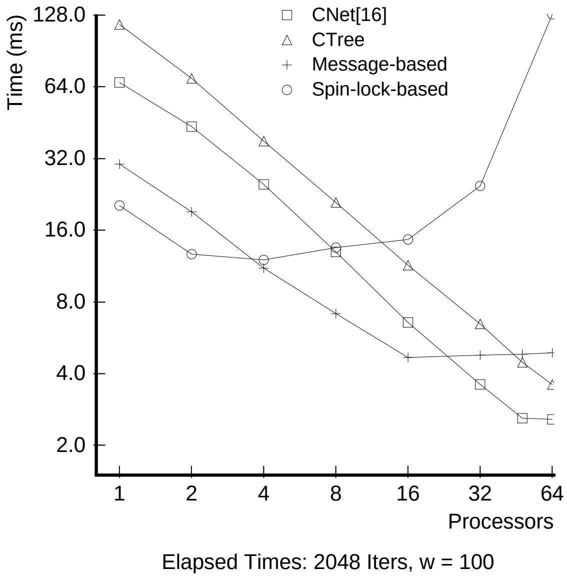
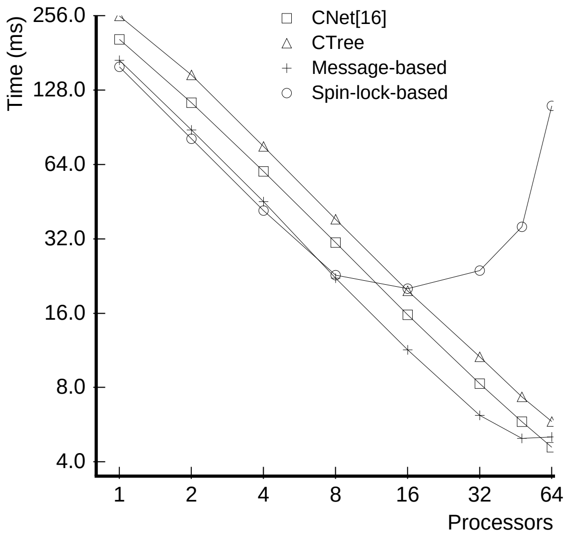


Figure 13: Performance of Counting Network Related to Number of Processors

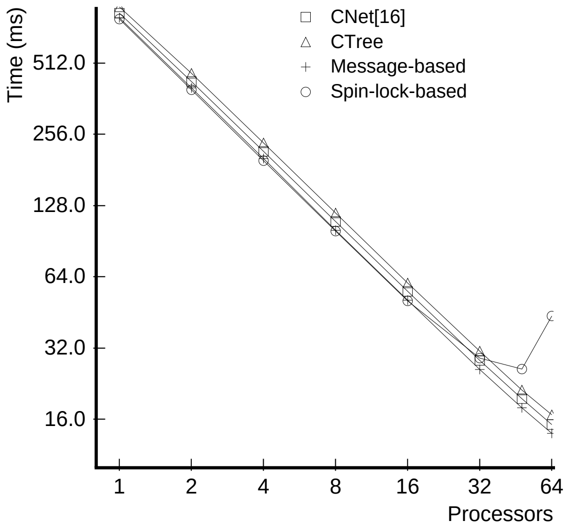


8 Preview

Next week – An Improvement! Depth affects latency, so we'll aim to keep width, while cutting down on depth. Theoretically, you could have depth $\log n$ based on the AKS sorting network [AKS,K], however the constants will be exponential...



Elapsed Times: 2048 Iters, w = 1000

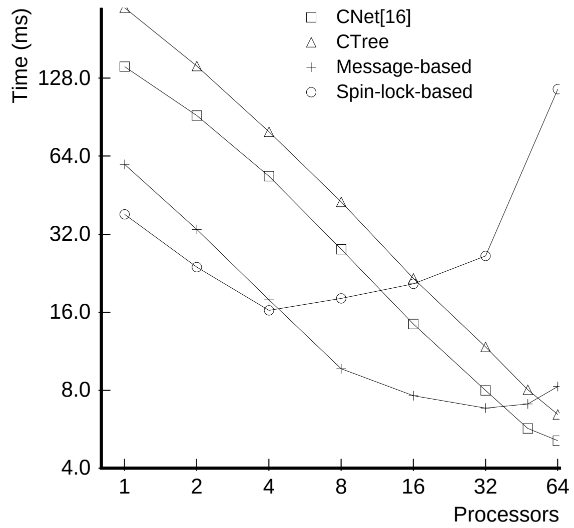


Elapsed Times: 2048 Iters, w = 5000

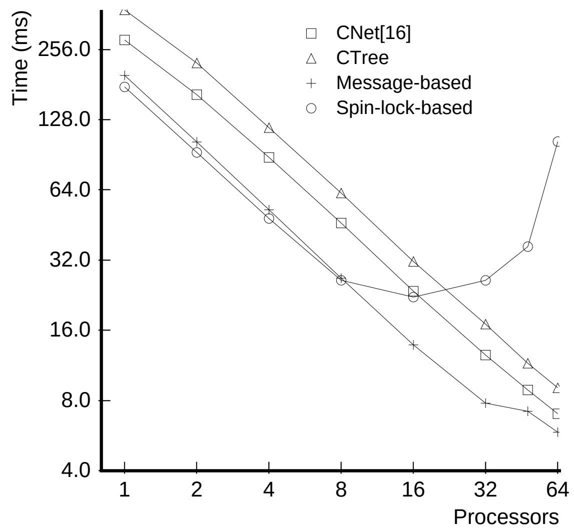
References

[1] M. Ajtai, J. Komlos and E. Szemeredi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, 1-9, 1983.

[2] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. To appear in *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also as MIT Technical Report MIT/LCS/TM-454, June 1991.



Elapsed Times: 2048 Tasks, w = 100



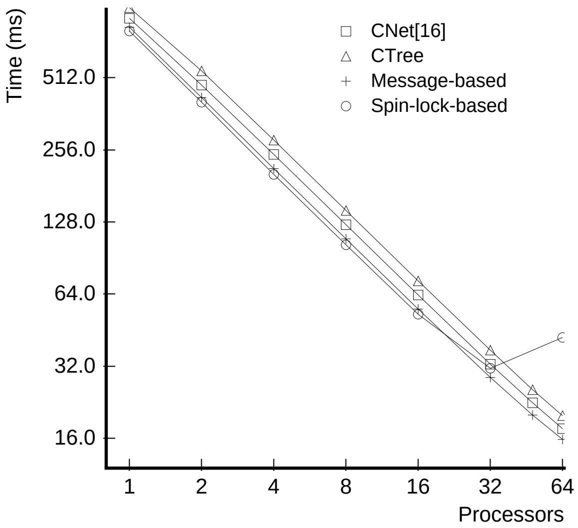
Elapsed Times: 2048 Tasks, w = 1000

[3] M. Klugerman, Small-Depth Counting Networks. Ph.D. Thesis, MIT, 1994.

[4] M. Herlihy, B.H. Lim and N. Shavit. Low Contention Load Balancing on Large Scale Multiprocessors. *Proceedings of the 3rd Annual ASM Symposium on Parallel Algorithms and Architectures*, July 1992, San Diego, CA. Full version available as a DEC TR.

[5] E.A. Brewer, C.N. Dellarocas, A. Colbrook and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT Technical Report /MIT/LCS/TR-561, September 1991.

[6] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, May



Elapsed Times: 2048 Tasks, w = 5000

1991.

[7] K.E. Batcher. Sorting Networks and their Applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[8] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the 3rd ASPLOS*, pages 64–75. ACM, April 1989.