Electrical and Computer Engineering Publications

Electrical and Computer Engineering

10-9-2010

Concurrent counting is harder than queuing

Costas Busch
Louisiana State University

Srikanta Tirthapura
Iowa State University, snt@iastate.edu

Concurrent counting is harder than queuing

Abstract

We compare the complexities of two fundamental distributed coordination problems, distributed counting and distributed queuing, in a concurrent setting. In both distributed counting and queuing, processors in a distributed system issue operations which are organized into a total order. In counting, each participating processor receives the rank of its operation in the total order, where as in queuing, a processor receives the identity of its predecessor in the total order. Many coordination applications can be solved using either distributed counting or queuing, and it is useful to know which of counting or queuing is the easier problem.

Our results show that concurrent counting is harder than concurrent queuing on a variety of processor interconnection topologies, including high and low diameter graphs. For all these topologies, we show that the *concurrent delay complexity* of a particular solution to queuing, the arrow protocol, is asymptotically smaller than a lower bound on the complexity of any solution to counting.

Keywords

Distributed algorithms, Distributed data structures, Distributed counting, Distributed queuing

Disciplines

Computer Sciences | Electrical and Computer Engineering

Comments

Creative Commons License

Concurrent Counting is Harder than Queuing

Srikanta Tirthapura[1], Costas Busch[2]

[1] Iowa State University
Dept. of Elec. and Computer Engg.
Ames, IA 50010
snt@iastate.edu

[2] Rensselaer Polytechnic Inst.
Computer Science Department
Troy, NY 12180
buschc@cs.rpi.edu

Abstract

In both distributed counting and queuing, processors in a distributed system issue operations which are organized into a total order. In counting, each processor receives the rank of its operation in the total order, where as in queuing, a processor gets back the identity of its predecessor in the total order. Coordination applications such as totally ordered multicast can be solved using either distributed counting or queuing, and it would be very useful to definitively know which of counting or queuing is a harder problem.

We conduct the first systematic study of the relative complexities of distributed counting and queuing in a concurrent *setting. Our results show that concurrent counting is harder than concurrent queuing on a variety of processor interconnection topologies, including high diameter graphs such as the list and the mesh, and low diameter graphs such as the complete graph, perfect m-ary tree, and the hypercube. For all these topologies, we show that the* concurrent delay complexity *of a particular solution to queuing, the arrow protocol, is asymptotically smaller than a lower bound on the complexity of* any *solution to counting. As a consequence, we are able to definitively say that given a choice between applying counting or queuing to solve a distributed coordination problem, queuing is the better solution.*

1. Introduction

This paper compares the complexities of two fundamental distributed coordination problems, distributed queuing and distributed counting. In distributed counting, processors in a network increment (perhaps concurrently) a globally unique shared counter. Each processor in return receives the value of the counter after its increment operation took effect. Equivalently, the operations issued by processors are arranged into a total order, and each processor in return receives the *rank* of its operation in the total order. Distributed counting is a very well studied problem, and many solutions have been proposed, perhaps the most prominent being Counting Networks [1].

In distributed queuing, similar to counting, processors issue operations which are organized into a total order (or a "distributed queue"). However, in contrast with counting, each processor receives the *identity of its predecessor operation* in the total order (see Figure 1). Distributed queuing has also been studied under many guises, for example in distributed directories [4], or in token based mutual exclusion [10]. One of the most efficient solutions for queuing is the arrow protocol, due to Raymond [10].

In many situations, distributed coordination can be achieved using either queuing or counting. For example, *totally ordered multicast* requires that all messages multicast to a group be delivered in the same order at all receivers. The conventional solution to totally ordered multicast uses distributed counting: the sender of a multicast message obtains a sequence number from a distributed counter, and attaches it to the message being multicast. Different receivers may receive the same set of messages in different orders, but they deliver them to the application in the order of the sequence numbers attached to the messages. The coordination part in this application is essentially distributed counting. Herlihy *et al.* [8] pointed out that totally ordered multicast can be solved using queuing too: the sender of a multicast message obtains the id of the predecessor message using distributed queuing, and attaches it to the multicast message. Different re-
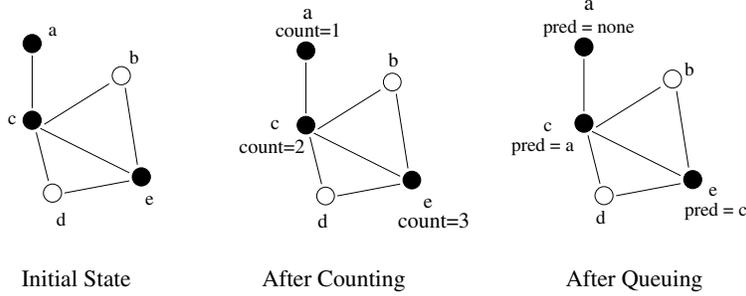
Figure 1. Counting and Queuing. The graph represents the processor interconnection network. Solid nodes issue counting (or queuing) operations, while the white nodes do not. The total order is a,c,e and "pred" means predecessor.

ceivers may again receive messages in different orders, but they deliver them to the application in a consistent order that is reconstructed by using the predecessor information that is piggybacked on the messages. Herlihy *et al.* [8] mention that the queuing based solution to ordered multicast is potentially more efficient than the counting based solution, but were unable to prove such a fact. Knowing that counting is inherently harder than queuing would imply that the queuing based solution is better than the counting based one. It would also suggest that it is worth reexamining if counting based solutions to other problems can be replaced with queuing based ones.

At the heart of both queuing and counting lies the formation of a total order among operations issued by processors in a distributed system. But these problems differ in the information that the nodes learn about the total order. In counting, each processor receives some global information about the total order, in the form of the rank of the operation. In contrast, in queuing, each processor receives the identity of its operation's predecessor in the total order, which gives the processor only a *local* view of the total order. This suggests that counting might be an inherently "harder" coordination problem than queuing. However, there are no known efficient reductions between the two problems, which will help us prove such a result (by an "efficient" reduction we mean a reduction that will not introduce much additional delay). Knowing the identity of an operation's neighbors in the total order tells us little about its rank, and vice versa.

1.1. Our Results

We show that for a large class of interconnection topologies, concurrent counting has inherently greater delay than concurrent queuing. Our result shows that

the queuing based solution for ordered multicast is better than the counting based one for many interconnection networks, and suggests that given a choice between queuing and counting to solve any coordination problem, queuing is often the better solution.

We study the *delay* of concurrent queuing and counting, and analyze the *concurrent one-shot* scenario, where all the operations are issued at the same time, and no more operations are issued thereafter. We use a synchronous model, where the link delays are predictable. The delay of an operation is the time till it receives its response. The *concurrent delay complexity* is the worst case value of the sum of the delays of all the operations. Our model takes into account the contention in the network. In order to model practical networks, we mandate that a node cannot process more than a constant number of messages in a single time step.

For the following classes of graphs, we show that the concurrent delay complexity of any counting algorithm is asymptotically greater than the concurrent delay complexity of a specific queuing algorithm, the arrow protocol.

- Any graph which has a Hamilton path. This class includes the d-dimensional mesh and low diameter graphs such as the complete graph and the hypercube.

- High diameter graphs: Any graph G which satisfies the following conditions.

  - G's diameter is $\Omega(n^{1/2+\delta})$ (where n is the number of vertices and $\delta > 0$ is a constant)
  - G has a spanning tree of a constant degree

- Any graph which contains the perfect m-ary tree as a spanning tree (where $m > 1$ is a constant in-

dependent of n). In a perfect m-ary tree, each internal node has exactly m children and the depths of different leaves differ by at most 1.

Our proofs consist of two parts, a lower bound on the complexity of counting and an upper bound on the complexity of queuing.

Lower Bound on Counting. In the synchronous model, lower bounds are technically challenging, since information can be obtained by a processor without actually receiving a message [3] (this point is further elaborated in Section 3). Our proofs of the lower bound on counting use two techniques.

The first technique is general, works on any graph, and relies on a careful analysis of the contention in the network. The proof uses an information-theoretic argument: a node that outputs a high value of count must have received a substantial amount of information from other nodes, and this takes a minimum amount of time, even on a completely connected graph.

The second technique works for graphs with a high diameter. This ignores contention in the network, and is solely based on the long latencies in receiving any information from far away nodes. Hence, this is much simpler to apply than the first, though substantially less general.

Upper Bound on Queuing. The upper bound uses an analysis of the upper bound on the arrow queuing protocol, due to Herlihy, Tirthapura and Wattenhofer [7], who show that the complexity of the arrow protocol can be upper bounded by the cost of an appropriately defined nearest neighbor traveling salesperson tour (TSP). A crucial technical ingredient in our proofs is the analysis of the nearest neighbor TSP on different graphs, including the perfect m-ary tree.

1.2. Related Work

Previous works have studied the distributed counting, queuing and adding problems separately, but have not attempted to compare them. From the point of view of solvability under faults, these problems are equivalent, but from the point of view of latency and delay, they are not. In this work, we are not concerned with the fault-tolerance aspects of these coordination problems.

Wattenhofer and Widmayer [14] give a lower bound on the contention of any counting algorithm. Their lower bound is on the *maximum delay* experienced by any processor, where as our lower bound is on the *sum of the delays* experienced by the processors.

Cook, Dwork and Reischuk [3] study a Parallel Random Access Machine (PRAM) model where simultaneous writes are disallowed. Our message passing model, where a processor can send no more than one message and receive no more than one message in a time step, bears resemblance to theirs. Cook, Dwork and Reischuk [3] study lower bounds for simple functions, such as finding the OR of many input bits.

The problem of one-shot concurrent distributed queuing was studied in [7], and the connection to their analysis has been outlined above. Recently, Kuhn and Wattenhofer [9] have presented an analysis of the long-lived case, when all queuing requests are not issued concurrently.

The rest of the paper is organized as follows. In Section 2, we define our model of computation, and the concurrent counting and queuing problems precisely. Section 3 contains the lower bound on concurrent counting. Section 4 contains an upper bound on concurrent queuing as well as a comparison between the complexities of the counting and queuing complexities on various graphs.

2. Model and Definitions

2.1. System Assumptions

The distributed system is modeled as a point to point communication network described by a connected undirected graph $G = (V, E)$, where V is the set of processors, and E is the set of reliable FIFO communication links between processors. Each process $v \in V$ can send a message to any of its neighbors in G. Let $n = |V|$ and $V = \{1, 2, 3 \ldots n\}$. We assume a *synchronous* system, where all communication links have a delay of one time unit. In each time step, a processor can do all of the following:

- Receive up to one message from a neighbor in G

- Send up to one message to a neighbor in G

- Local computation

Note that the model does not allow a processor to receive more than one message in a time step. The restriction of sending/receiving at most one message per time step is required to model practical networks, and rules out trivial algorithms for queuing and counting that are based on all to all communication. Our results can be generalized to a synchronous system where different communication links have different delays. They can also be easily generalized to a model where each node can send/receive a constant number of messages in a time step.

2.2. Concurrent Queuing and Counting

We consider the one-shot concurrent problem. In the one-shot concurrent queuing (counting) setting, a subset of the processors $R \subset V$ issue queuing (counting) operations at time zero. Each processor $i \in V$ has a binary input p_i which is private to i. If $p_i = 1$, then i has a queuing (counting) request, otherwise it does not. Thus $R = \{i | p_i = 1\}$.

A computation is a sequence of many rounds, where processors send messages to each other according to a protocol. An operation is complete when the processor receives the return value. We are mainly concerned with the *average delay* till the operations complete.

For queuing algorithm alg, the *queuing delay* of node v's operation, denoted by $\ell_Q(v, R, alg)$ is defined as the time at which v receives the identity of its predecessor in the total order, when all nodes in R issue queuing operations at time 0. The *queuing complexity of algorithm alg on graph G*, denoted by $C_Q(alg, G)$ is defined as the maximum value of the sum of the queuing delays of all the operations, the maximum being taken over all possible subsets of nodes $R \subset V$.

$$C_Q(alg, G) = \max_{R \subset V} \left\{ \sum_{v \in R} \ell_Q(v, R, alg) \right\}$$

Finally, the *queuing complexity of graph G* is defined as the queuing complexity of the best queuing algorithm for G.

$$C_Q(G) = \min_{alg} \{C_Q(alg, G)\}$$

The corresponding definitions for counting are similar. The *counting delay* of node v's operation for counting algorithm alg, denoted by $\ell_C(v, R, alg)$ is the time till v receives its count, when all the nodes in R start counting at time 0. The counts received by all the processors in R must be exactly $\{1, 2, 3, \ldots, |R|\}$, and processors which do not belong to R do not receive a count.

The counting complexity of algorithm alg on graph G is:

$$C_C(alg, G) = \max_{R \subset V} \left\{ \sum_{v \in R} \ell_C(v, R, alg) \right\}$$

The counting complexity of graph G is:

$$C_C(G) = \min_{alg} \{C_C(alg, G)\}$$

Initialization: We allow the counting or queuing algorithm to perform initialization steps, which are not counted towards the delay complexity. However, the set of nodes performing operations (R) is not known to the algorithm during this step. For example, the algorithm can build a spanning tree of the graph, or embed a counting network on the graph during the initialization step.

3. Lower Bound on Concurrent Counting

We are interested in finding a lower bound for the cost of any protocol that accomplishes the counting task. A technical difficulty is that in the synchronous model, a node may receive information by not receiving a message. For example, processors p_1 and p_2 may agree that if p_1 has a counting request, then it will not send a message to p_2 in the fifth round, and otherwise it will. Thus, p_2 may learn that p_1 has a counting request without actually receiving a message.

3.1. General Lower Bound for Arbitrary Graphs

We first consider the case when $G = K_n$ is the complete graph, since that is the most powerful communication graph possible. A lower bound for the complete graph will apply to any graph, for the following reason. Consider any counting algorithm for a graph on n vertices $G' \neq K_n$. Since graph G' can be embedded on K_n, the same algorithm can run on the complete graph with exactly the same complexity. Thus, if α is a lower bound on the counting complexity of K_n, then α must also be a lower bound on the counting complexity of any graph.

Consider a counting algorithm alg on K_n. When all processors are executing algorithm alg, for processor i and time step t, let $A(alg, i, t)$ denote the "processors affecting i at time t", which is informally the set of all processors which can influence the state of processor i at the end of round t. More formally, we have the following definitions.

Definition 3.1 *The* state *of processor i at any time step is the contents of its local memory.*

Definition 3.2 *For a counting algorithm alg, processor i and time t, $A(alg, i, t)$ is the smallest set A such that changing the inputs of some or all of the processors $\{1, 2, \ldots, n\} - A$ will not change the state of processor i at the end of time step t in algorithm alg.*

Lemma 3.1 *Suppose processor i outputs a count of k. If time t is such that $|A(alg, i, t)| < k$, then $\ell_C(i, R, alg) > t$.*

Proof: Suppose there exists t such that $|A(alg, i, t)| < k$ and $\ell_C(i, R, alg) \le t$. Thus processor i has output a count of k at or before time t.

We change the inputs to the processors without affecting the state of processor i as follows. For every processor $j \notin A(alg, i, t)$, we set $p_j = 0$. By the definition of $A(alg, i, t)$, the state of i at the beginning of time t is not affected by this change. Even if the inputs are changed as above, processor i would still output a count of k, since the system is the same in its view. This is clearly incorrect, since the number of processors which are counting is less than k (none of the processors outside $A(alg, i, t)$ are counting), and no processor can output a count of k. ∎

Definition 3.3 *For processor i and time t, define* $B(alg, i, t) = \{j | i \in A(alg, j, t)\}$.

For every processor i, $A(alg, i, 0) = \{i\}$. Clearly, changing the inputs of the rest of the processors will have no effect on the state of i at time 0. Thus, it is also true for every processor i, $B(alg, i, 0) = \{i\}$. Let $a(t) = \max_i |A(alg, i, t)|$ and $b(t) = \max_i |B(alg, i, t)|$.

Lemma 3.2 $a(t + 1) \le a(t) + \{a(t)\}^2 \cdot b(t)$

Proof: For algorithm alg, a *candidate for sending a message to processor i in round τ* is defined to be a processor that in some execution of protocol alg, can send a message to processor i in round τ. Each processor k which can get added to $A(alg, i, t+1)$ that was not already in $A(alg, i, t)$ must satisfy the following condition: there exists j such that $k \in A(alg, j, t)$ and j is a candidate for sending a message to i in round $t + 1$.

Suppose there were two processors j_1 and j_2 such that $A(alg, j_1, t) \cap A(alg, j_2, t) = \phi$. Then, both j_1 and j_2 cannot be candidates for sending a message to i in time $t+1$. The reason is that the states of processors j_1 and j_2 at the end of time t are completely independent of each other (there is no processor which could have influenced both j_1 and j_2), so there exists an input where both processors send a message to processor i in timestep $t + 1$, and this is not allowed by the model.

Consider processor j that is a candidate for sending a message to i in time $t+1$. The number of processors k such that $A(alg, j, t) \cap A(alg, k, t) \ne \phi$ is no more than $a(t) \cdot b(t)$. The reason is as follows. For each processor $m \in A(alg, j, t)$, $|B(alg, m, t)| \le b(t)$. The number of sets $A(alg, k, t), k \ne j$ that intersect $A(alg, j, t)$ is no more than the number of elements times the number of intersecting sets per element, which is bounded by $a(t) \cdot b(t)$.

Thus, the number of candidate processors which can send a message to i in time step $t + 1$ is no more than $a(t) \cdot b(t)$. Each such candidate processor j has no more than $a(t)$ elements in $A(alg, j, t)$. Thus the total number of elements added to $A(alg, i, t + 1)$ that were not already present in $A(alg, i, t)$ is no more than $\{a(t)\}^2 \cdot b(t)$. ∎

Lemma 3.3 $b(t + 1) \le b(t) \cdot (1 + 2^{a(t)})$

Proof: Consider processor i at the beginning of time step t. We know $|A(alg, i, t)| \le a(t)$. Depending on its state at the start of step t, processor i may send a message to one of many different processors. Let $R(alg, i, t)$ denote the set of all possible destination processors for a message from processor i at time t.

We now argue that $|R(alg, i, t)| \le 2^{|A(alg,i,t)|}$. Suppose this was not true, and $|R(alg, i, t)| > 2^{|A(alg,i,t)|}$. Any processor outside $A(alg, i, t)$ has no influence on the state of i at the beginning of time t. The number of different inputs for all processors in $A(alg, i, t)$ is no more than $2^{|A(alg,i,t)|}$. By the pigeonhole principle, there must be some input to processors in $A(alg, i, t)$ such that there are two different executions for processor i (the two executions differ since i sends messages to different processors in time step t in the executions). Since we are concerned with deterministic algorithms, this is impossible. Thus, $|R(alg, i, t)| \le 2^{|A(alg,i,t)|} \le 2^{a(t)}$.

Each processor $j \in B(alg, i, t)$ similarly has $|R(alg, j, t)| \le 2^{a(t)}$. By the definition of $b(t)$, $|B(alg, i, t)| \le b(t)$. Thus the total number of potential additions to $B(alg, i, t + 1)$ that were not already in $B(alg, i, t)$ is no more than $b(t) \cdot 2^{a(t)}$. ∎

Let $tow(j) = 2^{2^{2^{\cdots}}}\;$ j times $\quad$.

Lemma 3.4

$$a(t) \le tow(2t)$$
$$b(t) \le tow(2t)$$

Proof: The proof is by induction. The base case is easily checked since $a(0) = b(0) = 1$. For the inductive case, suppose that the lemma is true for some value of t. It remains to prove the lemma for time $t + 1$.

From Lemma 3.2 we have

$$
\begin{aligned}
a(t + 1) &\le a(t) \cdot (1 + a(t) \cdot b(t)) \\
&\le tow(2t) \cdot (1 + tow(2t) \cdot tow(2t)) \\
&\le 2^{tow(2t)} \\
&= tow(2t + 1)
\end{aligned}
$$

From Lemma 3.3 we have

$$
\begin{aligned}
b(t+1) &\leq b(t) \cdot (1 + 2^{a(t)}) \\
&\leq tow(2t) \cdot (1 + 2^{tow(2t)}) \\
&= tow(2t) \cdot (1 + tow(2t+1)) \\
&\leq tow(2t+2)
\end{aligned}
$$

Thus the inductive cases: $a(t+1) \leq tow(2t+2)$ and $b(t+1) \leq tow(2t+2)$ are proved. ∎

Theorem 3.5 *The cost of concurrent counting is at least $\Omega(n \log^* n)$ for any counting protocol on any graph G on n vertices.*

Proof: For any protocol, consider the case when all the processors start counting at time 0. From Lemmas 3.4 and 3.1, it follows that any processor that outputs a count of k must have latency at least t where $tow(2t) \geq k$. Thus, the latency of a processor that outputs a count of k must be at least $\frac{\log^* k}{2}$. Summing this over all processors which output a count of at least $n/2$ (there are $\lfloor n/2 + 1 \rfloor$ such processors), we get a lower bound of $\Omega(n \log^* n)$. ∎

3.2. Better Bounds for High Diameter Graphs

Thus far, our lower bound for counting applies to any graph, and the bound relies on a delicate analysis of the *contention* in the network. We now use arguments based on latency to obtain better lower bounds for graphs with a high diameter.

The proof of the lower bound for high-diameter graphs relies on the following argument. A node u which receives a high count must know the existence of at least one far away node which wants to count, so that its counting latency must be high.

Theorem 3.6 *If graph G has diameter α then $C_C(G) = \Omega(\alpha^2)$.*

Proof: Consider the case when all the nodes in V decide to count, i.e. $R = \{1, 2, \ldots, n\}$. Each node receives a different value in the range $1, 2, \ldots, n$. Let node v_i receive count i, for $i = 1 \ldots n$.

Consider node v_k, where $k > n - \alpha/2$. It must be that v_k's latency is at least $\alpha/2 + k - n$. We will prove this statement by contradiction. Suppose v_k's counting latency was $x < \alpha/2 + k - n$. Node v_k must know that there are at least $k - 1$ other nodes that are counting, otherwise it cannot output a count of k. Since v_k's latency is x, the farthest of these nodes cannot be at a

distance of greater than x. There are totally n nodes, and k of them (including v_k itself) are at a distance of no more than x from v_k. This implies that there is is no node at a distance greater than $x + n - k$ from v_k. Thus, G's diameter can be no more than $2(x + n - k) < \alpha$, which is a contradiction.

Thus v_k's latency is at least $\alpha/2 + k - n$. For $k = (n - \alpha/2 + 1) \ldots n$, the lower bound on v_k's latency ranges from $1 \ldots \alpha/2$. Hence, $C_C(G) \geq \alpha/2 + \alpha/2 - 1 + \ldots + 1 = \Omega(\alpha^2)$ ∎

Theorem 3.6 shows that the counting complexity of the list on n nodes is $\Omega(n^2)$, and on the two-dimensional mesh is $\Omega(n\sqrt{n})$.

4. Upper Bound on Queuing

We now focus on deriving an upper bound on the concurrent cost of a queuing algorithm, which also yields an upper bound on the queuing complexity. We use a specific queuing algorithm, the *arrow protocol* to derive the upper bound. The arrow protocol was invented by Raymond [10], and is based on path reversal on a spanning tree of the network. We give a brief description here and refer to [10, 4] for detailed descriptions.

During the initialization step, the protocol chooses T, a spanning tree of the network $G = (V, E)$. The tail of the queue initially resides at some node, say t. Each node $v \in V$ has a "pointer" (or an arrow), denoted by $link(v)$, which always points to a neighbor in the spanning tree, or to v itself. The arrows are initialized so that following the arrows from any node leads to the tail, t. Informally, every node except for the tail itself only knows the "direction" in which the tail lies, and not the exact location. Each node v also has a variable $id(v)$ which is the identifier of the previous operation originating from v. The protocol is based on the idea of *path reversal*, and is described in the following steps.

1. If node v issues a queuing operation whose identifier is a, it sets $id(v)$ to a, and sends out a $queue(a)$ message to $u_1 = link(v)$, and "flips" $link(v)$ to point back to v.

2. Suppose a node u_i receives a $queue(a)$ message from u_{i-1}, a neighbor in the spanning tree, and say $u_{i+1} = link(u_i)$ currently. If $u_{i+1} \neq u_i$ then u_i flips $link(u_i)$ back to u_{i-1}, and forwards the $queue(a)$ message to u_{i+1}. If $u_{i+1} = u_i$, then operation a has been queued behind $id(u_i)$.

Concurrent $queue()$ messages may arrive in the same time step from neighbors in the tree. A node may receive up to deg $queue()$ messages in a time step where

deg is its degree in the tree. As long as the maximum degree of the tree T is a constant, this algorithm can be executed in the model where each node can send and receive only one message per time step. The node can handle a constant number of messages by synchronously proceeding by an "expanded" time step during which a constant number of messages are sent or received during each such step. Note that this will not change the asymptotics. For our analysis, we assume that concurrent *queue*() messages are processed in the same "expanded" time step, thus we will use spanning trees which have a constant degree.

The one-shot concurrent complexity of the arrow protocol has been studied by Herlihy, Tirthapura and Wattenhofer [7], which shows a connection to the cost of the nearest neighbor traveling salesperson tour (referred to as "nearest neighbor TSP" from here onwards) on an appropriately defined graph.

Consider a one-shot execution where a set R of nodes have issued queuing requests at time zero. For spanning tree T and a set of requesting nodes R, define the nearest neighbor TSP visiting all nodes in R in the following order. Start from t as the origin (t is the location of the first element in the queue), and visit all vertices in R in the following order: next visit a previously unvisited vertex in R that is closest to the current position, distances being measured along the tree T. The cost of the nearest neighbor TSP is the total distance traveled on T in visiting all nodes of R.

Theorem 4.1 From [7]. *If the maximum degree of T is bounded by a constant, then the concurrent queuing complexity of the arrow protocol over the request set R is no more than twice the cost of a nearest neighbor TSP on T visiting all nodes in R.*

Rosenkrantz, Stearns and Lewis [11] have shown that the nearest neighbor algorithm is a $\log k$ approximation algorithm for the TSP on any graph on k vertices whose edge weights satisfy the triangle inequality. Since the metric of the shortest distance on a tree satisfies the triangle inequality, this yields that the cost of a nearest neighbor TSP on tree T visiting request set R is $O(n \log n)$ where n is the number of nodes in the tree. When used in conjunction with Theorem 4.1, we obtain the following corollary.

Corollary 4.2 *If graph G has a spanning tree whose maximum degree is bounded by a constant, then $C_Q(G) = O(n \log n)$, where n is the number of vertices in G.*

For specific graphs, it might be possible to find better bounds than $O(n \log n)$ on the cost of the nearest

neighbor TSP. An example is the list; a nearest neighbor TSP on a list of n nodes costs only $O(n)$. We omit the proof here, and only state the result; the proof can be found in [12].

Lemma 4.3 *If tree T is a list on n vertices, then for any vertex set $R \subset V$, the cost of a nearest neighbor TSP on tree T visiting request set R is $O(n)$.*

4.1. Complete Graph, Mesh, Hypercube

Theorem 4.4 *If G has a Hamilton path, then $C_Q(G) = o(C_C(G))$.*

Proof: Choose the Hamilton path of G as the spanning tree T, and execute the arrow protocol on this tree. From Lemma 4.3 and Theorem 4.1, it follows that $C_Q = O(n)$. From Theorem 3.5, we have $C_C(G) = \Omega(n \log^* n)$. The lemma follows. ∎

The above theorem states that counting is an inherently harder problem than queuing on any graph which has a Hamilton path. This theorem has useful implications for the following popular interconnection networks.

Lemma 4.5 *For all the following graphs, counting is inherently harder than queuing, i.e. $C_Q(G) = o(C_C(G))$.*

- *K_n: the complete graph on n vertices*

- *The d-dimensional mesh for any positive integer d*

- *The hypercube of dimension d*

Proof: The lemma follows since each of the above graphs has a Hamilton path. For K_n, the proof is obvious. For the d-dimensional mesh, we sketch a proof by induction. Assume that a $(d-1)$-dimensional mesh has a Hamilton path. A d-dimensional mesh can be viewed as many $(d-1)$-dimensional meshes stacked one on top of the other. A Hamilton path for the d-dimensional mesh can be constructed by visiting the vertices in the individual $(d-1)$-dimensional meshes in order. The proof that a hypercube of dimension d has a Hamilton path is similar, and can be shown through induction. ∎

4.2. Perfect Binary Trees

Thus far, our upper bound on the cost of the arrow protocol has used the list as a spanning tree. We now turn to another type of spanning tree, the *perfect binary tree*. The perfect binary tree T on n vertices has

depth $d = \lfloor \log_2 n \rfloor$, and all the leaves are at a distance of either $d-1$ or d from the root.

Using Corollary 4.2 yields an upper bound of $O(n \log n)$ on the cost of the nearest neighbor TSP on the perfect binary tree. However, this bound is not tight enough for our purposes, since the lower bound on the counting complexity is only $\Omega(n \log^* n)$. Hence we look for a tighter upper bound for the queuing complexity on the perfect binary tree. We now show that the cost of the nearest neighbor TSP on the perfect binary tree with n vertices is $O(n)$. Our analysis can be extended to any perfect m-ary tree in a straightforward manner.

Theorem 4.6 *If T is a perfect binary tree on n vertices, then the cost of the nearest neighbor TSP visiting any subset of vertices R starting from the root is $O(n)$.*

Proof: For each node $v \in R$, let $cost(v)$ denote the distance from v to its successor in the nearest neighbor TSP on T. The cost of the nearest neighbor TSP is $cost(T) = \sum_{v \in R} cost(v)$. For a node u, let $depth(u)$ denote its depth in T. Let $d = \lfloor \log_2 n \rfloor$ denote the depth of T. For each level of the tree $l = 0 \ldots d$, define

$$cost(l) = \sum_{(v \in R) \wedge (depth(v)=l)} cost(v)$$

In Lemma 4.7 we show that for any level l, $cost(l) = O(n/2^{d-l} + d)$. Taking this sum over all levels $l = 0 \ldots d$, we get:

$$
\begin{aligned}
cost(T) &= \sum_{l=0}^{d} cost(l) \\
&= O(d^2) + \sum_{l=0}^{d} O(n/2^{d-l}) \\
&= O(d^2) + O(n) = O(n)
\end{aligned}
$$

■

Lemma 4.7 *For any level $l \in \{0 \ldots d\}$,*

$$cost(l) = O\left(\frac{n}{2^{d-l}}\right) + 2d$$

Proof: We first add dummy nodes (which do not belong to R) to fill level d of T. This does not change the cost of the TSP, but will simplify the presentation of the proof. Let T_l denote the subtree of T restricted to all vertices at level l or lesser. For any vertex v in T_l, let T_v denote the subtree of T_l that is rooted at v, n_v the number of vertices in T_v, and d_v the depth of T_v. Define

$c'(v)$ as: $c'(v) = \sum_{(u \in \text{ leaves of } T_v) \wedge (u \in R)} cost(u)$. We want to compute $cost(l) = c'(root)$.

We will show that we can write $c'(v)$ as the sum of two terms: $c'(v) = f(n_v) + cost(w)$, where f is a function which we will describe below, which depends only on the size of T_v, and w is some leaf in T_v which belongs to R. Informally, the cost due to the leaves in T_v depends only on the size of T_v, except for one node w. We now inductively show that $c'()$ can be written as shown above, and determine the function $f()$. There are two cases.
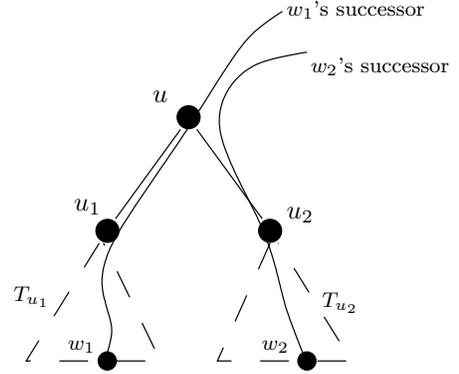

Figure 2. Proof of Lemma 4.7

(1) Suppose u is a leaf of T_v. If $u \in R$ then $c'(u) = cost(u)$, else if $u \notin R$ then $c'(u) = 0$. Either way, we can write this as $c'(u) \leq f(n_u) + cost(u)$, where $f(n_u) = f(1) = 0$.

(2) Suppose u is not a leaf of T_v. Then the subtree T_u is composed of two smaller subtrees of equal size, rooted at u_1 and u_2, as shown in Figure 4.2. Inductively, $c'(u_1)$ can be written as $c'(u_1) = f(n_{u_1}) + cost(w_1)$ where w_1 is a leaf in T_{u_1}, and similarly $c'(u_2) = f(n_{u_2}) + cost(w_2)$ where w_2 is a leaf in T_{u_2}. In this case, we have: $c'(u) = c'(u_1) + c'(u_2) = f(n_{v_1}) + f(n_{v_2}) + cost(w_1) + cost(w_2)$

Here is the key point. Among w_1 and w_2, (without loss of generality) suppose w_2 was visited later by the nearest neighbor TSP. Then, $cost(w_1)$ must be lesser than the distance between w_1 and w_2 on the tree, which is $2d_u$.

Thus, $c'(u) \leq f(n_{u_1}) + f(n_{u_2}) + 2d_v + cost(w_2)$, which can be rewritten as $c'(u) = f(n_u) + cost(w_2)$ where w_2 is a leaf in T_u, and $f()$ satisfies the following equation: $f(n_u) \leq f(n_{u_1}) + f(n_{u_2}) + 2d_u$. Since T_{u_1} and T_{u_2} are of equal size, $n_{u_1} = n_{u_2} = (n_u - 1)/2$, and $d_u = \log n_u$, and the recurrence relation for f becomes: $f(x) \leq 2f(\frac{x-1}{2}) + 2\log x$ and $f(1) = 0$, which yields the solution $f(x) = O(x)$. Putting this back in $c'(u)$, we get: $c'(u) = O(n_u) + cost(w) \leq O(n_u) + 2d$. Thus,

$cost(l) = c'(root) = O(n/2^{d-l}) + 2d$, and the proof is complete. ∎

Theorem 4.6 shows that the cost of a nearest neighbor TSP on the perfect binary tree is $O(n)$. Because of Theorem 4.1, this yields the same upper bound on the concurrent queuing complexity of the arrow protocol. Thus, we have $C_Q(G) = O(n)$ if G has a perfect binary tree as a spanning tree. Combining this with Theorem 3.5, we get:

Lemma 4.8 *If G has a perfect binary tree as a spanning tree, then $C_Q(G) = o(C_C(G))$.*

The analysis of the perfect binary tree can easily be extended to any perfect m-ary tree, where m is a constant (we omit this proof since it is similar to the proof of the binary case). Thus, we have the following more general result.

Theorem 4.9 *If G has a perfect m-ary tree as a spanning tree, where m is a constant, then $C_Q(G) = o(C_C(G))$.*

4.3. High Diameter Graphs

Theorem 4.10 *If graph G satisfies the following:*

- *G's diameter is $\Omega(n^{1/2+\delta})$ where $\delta > 0$ is a constant independent of n*

- *G has a spanning tree whose maximum degree is bounded by a constant*

then $C_Q(G) = o(C_C(G))$.

Proof: From Theorem 3.6, we know $C_C(G) = \Omega(n^{1+2\delta})$, and from Corollary 4.2, we have $C_Q(G) = O(n \log n)$, hence $C_C(G)$ is asymptotically greater. ∎

5. Conclusions

Queuing and counting are both important coordination problems, and there are occasions where one could use either of them in solving the task on hand. Given such an option, it is often better to use queuing. We have shown that for a variety of graphs, including the complete graph, perfect m-ary tree, list, hypercube and the mesh, the counting delay complexity is asymptotically greater than the queuing delay complexity.

A natural question is whether this is true for all topologies. The answer is negative. Consider S, the star on n vertices. Since all messages will get serialized at the central vertex, $C_C(S) = \Theta(n^2)$, and $C_Q(S) = \Theta(n^2)$, so that counting is (asymptotically)

no harder than queuing. On such graphs, the delay due to contention dominates, and overshadows other factors.

An open question is as follows. There are other coordination problems that require the formation of a total order, such as distributed addition [5]. It would be interesting to compare the inherent delays imposed by different coordination problems. In a related work, Busch *et al.* [2] show that for a class of mathematical operations, any distributed implementation must be linearizable. This condition imposes a fundamental limit on the efficiency of such a distributed implementation.

References

[1] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[2] C. Busch, M. Mavronicolas, and P. Spirakis. The cost of concurrent, low-contention read modify write. *Theoretical Computer Science*, 333:373–400, Mar 2005.

[3] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[4] M. Demmer and M. Herlihy. The Arrow Distributed Directory Protocol. In *Proc. 12th International Symposium on Distributed Computing (DISC)*, pages 119–133, 1998.

[5] P. Fatourou and M. Herlihy. Adding networks. In *Proc. 15th International Symposium on Distributed Computing (DISC)*, pages 330–342, 2001.

[6] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Prog. Lang. Syst.*, 5(2):164–189, Apr. 1983.

[7] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive Concurrent Distributed Queuing. In *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 127–133, 2001.

[8] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, 2001.

[9] F. Kuhn and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 294–301, 2004.

[10] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[11] D. Rosenkrantz, R. Stearns, and P. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

[12] S. Tirthapura. *Distributed Queuing and Applications.* PhD thesis, Brown University, 2002.

[13] S. Tirthapura. Brief announcement: Concurrent counting is harder than queuing. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing*, page 112, 2003.

[14] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal of Parallel and Distributed Computing*, 49(1):135–145, 1998.