Theoretical Computer Science 411 (2010) 3823–3833

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Concurrent counting is harder than queuing☆

Costas Busch [a,1], Srikanta Tirthapura [b,*]

[a] Department of Computer Science, Louisiana State University, 286 Coates Hall, Baton Rouge, LA 70803, USA

[b] Department of Electrical and Computer Engineering, Iowa State University, 2215, Coover Hall, Ames, IA 50011, USA

ARTICLE INFO

Article history:
Received 14 March 2007
Accepted 3 July 2010
Communicated by P. Spirakis

Keywords:
Distributed algorithms
Distributed data structures
Distributed counting
Distributed 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.

1. Introduction

We compare the complexities of two fundamental distributed coordination problems, distributed queuing and distributed counting. In distributed counting, processors in a distributed system increment (perhaps concurrently) a globally unique shared counter. Each processor in return receives the value of the counter after its increment operation took effect (see Fig. 1). Equivalently, operations issued by the 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 Fig. 1). Distributed queuing has also been studied under many guises, for example in distributed directories [4], or in token based mutual exclusion [9]. One of the most efficient solutions for queuing is the arrow protocol, due to Raymond [9].

In many situations, the required 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

☆ A preliminary version of this work has appeared in the Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006.

* Corresponding author. Tel.: +1 515 294 3546; fax: +1 515 294 3637.
E-mail addresses: busch@csc.lsu.edu (C. Busch), snt@iastate.edu (S. Tirthapura).
1 Tel.: +1 225 578 7510.

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.07.002

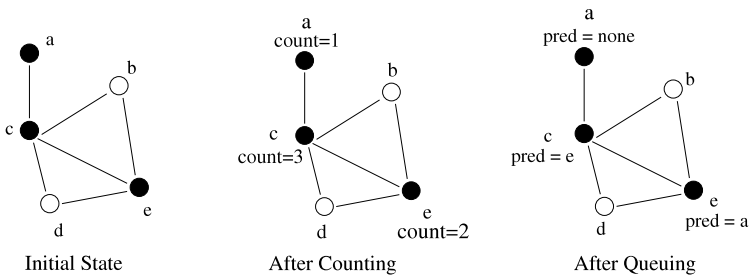

Fig. 1. Counting and Queuing. The graph represents the processor interconnection network. Solid nodes issue counting (or queuing) operations, while the white nodes do not. The total order is a, e, c and "pred" means predecessor.

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. [7] pointed out that totally ordered multicast can be solved using queuing too, as follows. The sender of a multicast message obtains the identifier of the predecessor message using distributed queuing, and attaches it to the multicast message. Different receivers 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. [7] 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 to ordered multicast is definitely better than the counting based one. It would also suggest that it is worth reexamining if counting based solutions to other distributed coordination problems can be replaced with queuing based solutions.

At the heart of both distributed queuing and distributed counting lies the task of forming 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. Further, it suggests that given a choice between queuing and counting to solve a distributed coordination problem, queuing is often the better solution.

We analyze the delay of concurrent queuing and counting in the *concurrent one-shot* scenario, where all 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 length of time elapsed between the instant the operation was issued and the instant when 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 network contention. In order to model practical networks, we mandate that a node cannot process more than one message in a single time step. More formal definitions of the above terms are presented in Section 2.

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 independent 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 on graphs with a high diameter. This ignores contention in the network, and is solely based on the long latencies in receiving information from far away nodes. Hence, this is much simpler to apply than the first, though substantially less general, since it cannot be applied to low-diameter graphs.

Upper bound on queuing. The upper bound uses an analysis of the upper bound on the arrow queuing protocol, due to Herlihy et al. [6], 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 list and the perfect m-ary tree.

1.2. Related work

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

Previous works have studied the distributed counting and queuing problems separately, but have not attempted to compare them. The one-shot concurrent distributed queuing scenario was studied in [6], and the connection to their analysis has been outlined above. Recently, Herlihy et al. [8] have presented an analysis of the long-lived case, when not all queuing requests are issued concurrently. Wattenhofer and Widmayer [11] give a lower bound on the maximum contention experienced by any counting algorithm on a completely connected graph. However, this does not translate into a lower bound on the average delay experienced by the processors, which is what we study here. Further, [11] consider solely the completely connected graph while we consider a larger class of graphs including the complete graph, and high diameter graphs such as the list.

Busch et al. [2] show that for a class of mathematical operations, which includes addition and multiplication (but not counting), any distributed implementation must be linearizable. This condition imposes a fundamental limit on the efficiency of a distributed implementation of any operation in this class.

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 problems 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 (in order).

- Send up to one message to a neighbor in G
- Receive up to one message from a neighbor in G
- Local computation

Our lower bounds for the synchronous model also carry over to the general asynchronous model, where there is no upper bound on the delay of a communication link. In deriving upper bounds in the asynchronous model, the worst case behavior can be that all operations, in spite of starting concurrently, may get executed sequentially, one after the other. In such a case metrics such as the worst total completion time turn out to be equal for both counting and queuing, and unrealistic in either case.

We use the terms "time step" and "round" interchangeably. 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. For example, if there was no restriction on the number of messages sent/received in a time step, and G is the complete graph, then there is a trivial one-round algorithm for distributed queuing and counting, where all nodes obtain a global view

of the system through an all-to-all communication step, and then assign counts (or positions in the queue) to themselves locally. Our results can also be easily generalized to a model where each node can send/receive a constant number of messages in a time step, through simulating each time step of the more powerful distributed system (where a node can process a constant c number of messages in a time step) by c time steps of the distributed system modeled above.

2.2. Concurrent queuing and counting

We consider the one-shot concurrent setting, where 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 *total delay* of all operations.

For queuing algorithm *alg*, if v has a queuing request, 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 $R \subset V$.

$$C_Q(alg, G) = \max_{R \subset V} \left\{ \sum_{v \in R} \ell_Q(v, R, alg) \right\} \tag{1}$$

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

$$C_Q(G) = \min_{alg} \{C_Q(alg, G)\} \tag{2}$$

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:

$$C_C(alg, G) = \max_{R \subset V} \left\{ \sum_{v \in R} \ell_C(v, R, alg) \right\} \tag{3}$$

The counting complexity of graph G is:

$$C_C(G) = \min_{alg} \{C_C(alg, G)\} \tag{4}$$

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.

Discussion on the metric: Our metric of concurrent delay complexity (or equivalently, total delay) captures the aggregate response time of the counting or queuing data structure. We can compare this metric with the metric of the *maximum delay*. The total delay is not significantly affected by a single operation having a large delay, whereas the maximum delay is easily affected by a single operation having a large delay. Hence, the total delay is a more robust metric than the maximum delay for comparing the complexities of queuing and counting.

3. Lower bound on concurrent counting

We are interested in a lower bound for the cost of any protocol that accomplishes the concurrent counting task. At a high level, our approach to a lower bound for the counting delay is as follows. A node that is counting and which outputs a count of i must know of the presence of at least $i - 1$ nodes that are counting. A lower bound on the time to obtain this information (that at least $i - 1$ other nodes are counting) is also a lower bound on the counting delay of that node. Adding up the delays of all nodes that are counting, we obtain a lower bound for the concurrent delay complexity of counting.

A technical difficulty in the synchronous model that we consider is that information may be communicated between nodes without any messages being sent or received. 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, otherwise it will send a message in the fifth round. 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 is K_n, the complete graph on n vertices, 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 $\ell_C(i, R, alg) \leq t$. 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 state is the same in the view of processor i. 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 should 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 time t, define $a(t) = \max_i |A(alg, i, t)|$ and $b(t) = \max_i |B(alg, i, t)|$.

Since changing the inputs of the rest of the processors will have no effect on the state of i at time 0, we have:

Fact 1. *For every processor i, $A(alg, i, 0) = \{i\}$ and $B(alg, i, 0) = \{i\}$.*

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

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*, sends a message to processor i in round τ. Each processor k which belongs in $A(alg, i, t + 1)$ that does not belong 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 $t + 1$. To see this, suppose there was no 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 we switched the state of processor k's input p_k (from 0 to 1 or vice versa). By the definition of $A(\cdot, \cdot, \cdot)$, for every processor that is a candidate for sending a message to i in round $t + 1$, the state of the processor at the end of time step t is not affected by this switch. For processor i, the interaction in round $t + 1$ is restricted to those processors that are candidates for sending a message to i in round $t + 1$. Thus changing the initial state of processor k will have no effect on the state of processor i at the end of $t + 1$ steps, and thus k cannot belong to $A(alg, i, t + 1)$.

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 time step $t + 1$, 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 $A(alg, j, t) \cap A(alg, k, t) \neq \phi$ is no more than $a(t) \cdot b(t)$. The reason is as follows. For each processor $m \in A(alg, j, t)$, $|B(alg, m, t)| \leq b(t)$. The number of sets $A(alg, k, t)$, $k \neq j$ 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. $b(t + 1) \leq b(t) \cdot (1 + 2^{a(t)})$.

Proof. Consider processor i at the beginning of time step t. We know $|A(alg, i, t)| \leq a(t)$. 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 $|R(alg, i, t)| \leq 2^{|A(alg,i,t)|}$. 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, $|R(alg, i, t)| \leq 2^{|A(alg,i,t)|} \leq 2^{a(t)}$.

Each processor $j \in B(alg, i, t)$ similarly has $|R(alg, j, t)| \leq 2^{a(t)}$. By the definition of $b(t)$, $|B(alg, i, t)| \leq b(t)$. 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 $b(t) \cdot 2^{a(t)}$. □

Definition 3.4. For positive integer j, $tow(j)$ is defined as:

$$tow(j) = 2^{2^{2^{\cdots}}} \text{ } j \text{ times}$$

For positive k, $\log^*(k)$ is defined as $\min\{i \geq 0 | \log^{(i)} k \leq 1\}$.

Lemma 3.4. *For positive integer τ,*

$$a(\tau) \leq tow(2\tau)$$
$$b(\tau) \leq tow(2\tau)$$

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 $\tau = t$. It remains to prove the lemma for $\tau = t+1$. From Lemma 3.2 we have

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

If $t \geq 1$, then $tow(2t) \geq 4$, and for $x \geq 4$, it can be verified that $x + x^3 < 2^{2^x}$. Using this in the above equation, we get:

$$a(t+1) \leq 2^{2^{tow(2t)}} = tow(2t+2)$$

From Lemma 3.3 we have

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

If $t \geq 1$, then $tow(2t) \geq 4$, and for $x \geq 4$, it can be verified that $x + x2^x \leq 2^{2^x}$. Using this in the above equation, we get:

$$b(t+1) \leq 2^{2^{tow(2t)}} = tow(2t+2)$$

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 processors start counting at time 0. From Lemma 3.4 and 3.1, it follows that any processor which 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 obtain 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 $k = (n - \alpha/2 + 1), \ldots, n-1, n$, the lower bound on v_k's latency ranges from $1, \ldots, \alpha/2 - 1, \alpha/2$. Hence, $C_C(G) \geq \alpha/2 + \alpha/2 - 1 + \cdots + 1 = \Omega(\alpha^2)$. □

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 concurrent 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 [9], and is based on path reversal on a spanning tree of the network. We give a brief description here and refer the reader to [9,4] for greater detail.

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. A node can handle a constant number of messages by synchronously computing through 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 et al. [6], which shows a connection to the cost of the nearest neighbor traveling salesperson tour (referred to as "nearest neighbor TSP" from here onward) 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, the nearest neighbor TSP starts from an initial node (the "root") and visits all nodes 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 [6]*)**.** *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 et al. [10] 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. In particular, we now show that the cost of a nearest neighbor TSP on a list of n nodes is $O(n)$.

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 T that starts from any node and visits request set R is no more than $3n$.*

Proof. We use the term "root" to denote the starting node of the nearest neighbor TSP. Let the set of vertices in $V = \{1, 2, 3, \ldots, n\}$, and let $\pi = \pi(1), \pi(2), \ldots, \pi(|R|)$ denote the permutation in which the nearest neighbor TSP on T visits the vertices in R, starting from the root. We can write π as the concatenation of "runs" $\pi^1\pi^2 \ldots \pi^m$, defined as follows. A run is defined as a maximal subsequence of $\pi, \pi(i), \pi(i+1), \ldots, \pi(i+k)$ such that either (1) For all $j = 1, \ldots, k, \pi(i+j)$ is to the right of $\pi(i+j-1)$ or (2) For all $j = 1, \ldots, k, \pi(i+j)$ is to the left of $\pi(i+j-1)$.

Let c denote the cost of the nearest neighbor TSP visiting R starting from an arbitrary root node $root \in V$. Let $d(u, v)$ denote the distance (on the tree) between vertices u and v. Let u_j and v_j respectively denote the first and last vertices in $\pi(j)$. For $i > 1$, let x_i denote $d(v_{i-1}, v_i)$, and let $x_1 = d(root, v_1)$. We have:

$$c = x_1 + x_2 + \cdots + x_m \tag{5}$$

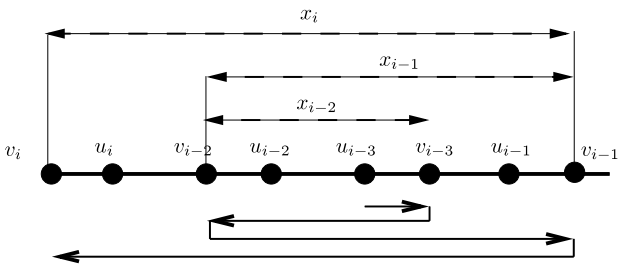

Fig. 2. Proof of Lemma 4.4.

Using Lemma 4.4 we obtain

$$\begin{aligned} x_m &\geq x_{m-2} + x_{m-1} \\ x_{m-1} &\geq x_{m-2} + x_{m-3} \\ &\cdots \\ x_3 &\geq x_1 + x_2 \\ x_2 &\geq x_1 \end{aligned}$$

Adding up the above equations and canceling common terms,

$$x_m \geq x_1 + x_2 + \cdots + x_{m-2}$$

Since the diameter of the list is n, it must be true that $x_m \leq n$ and $x_{m-1} \leq n$. Thus:

$$c = x_1 + \cdots + x_m \leq x_{m-1} + 2x_m \leq 3n \quad \square$$

Lemma 4.4. *For $i = 3 \ldots m$, $x_i \geq x_{i-1} + x_{i-2}$.*

Proof. See Fig. 2. Since u_{i-1} is visited by the nearest neighbor TSP immediately after v_{i-2}, and u_i is visited later, the following must be true.

$$d(v_{i-2}, u_i) \geq d(v_{i-2}, u_{i-1}) \tag{6}$$

Since both u_i and v_i lie on the same side of v_{i-2} along the list, the following must be true (see Fig. 2).

$$d(v_{i-2}, v_i) \geq d(v_{i-2}, u_i) \tag{7}$$

$$\begin{aligned} x_i &= x_{i-1} + d(v_{i-2}, v_i) \\ &\geq x_{i-1} + d(v_{i-2}, u_i) \quad \text{using Eq. (7)} \\ &\geq x_{i-1} + d(v_{i-2}, u_{i-1}) \quad \text{using Eq. (6)} \\ &= x_{i-1} + x_{i-2} \quad \square \end{aligned}$$

4.1. Complete graph, mesh, hypercube

Theorem 4.5. *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 $C_Q(G) = O(n)$. From Theorem 3.5, we have $C_C(G) = \Omega(n \log^* n)$. The lemma follows. $\square$

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 4.6. *For all the following graphs, concurrent 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 can prove using induction that a Hamilton path exists. It is easy to verify that a 2-dimensional mesh has a Hamilton path. 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 can be similarly shown through induction. $\square$

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 $d = \lfloor \log_2 n \rfloor$, every internal node has exactly two children, and all leaves are at a distance of either $d-1$ or d from the root.

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 $\Omega(n \log^* n)$. We look for a tighter upper bound on 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 $O(n)$. This analysis can be extended to any perfect m-ary tree in a straightforward manner.

Theorem 4.7. *If $T = (V, E)$ is a perfect binary tree on n vertices, then the cost of the nearest neighbor TSP visiting any subset of vertices $R \subset V$ starting from the root is $O(n)$.*

Proof. Let π denote the order in which the nearest neighbor TSP on T visits the vertices in R, starting from the root. For each node $v \in R$, let $cost(v)$ denote the distance from v to its successor in π (for the last vertex in π, $cost(\cdot)$ is defined to be 0). 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, which is its distance from the root of T. Let $d = \lfloor \log_2 n \rfloor$ denote the depth of T. For each integer $\ell = 0, 1, \ldots, d$, $cost(\ell)$ is defined as follows.

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

Lemma 4.9 shows that for any integer ℓ, $0 \le \ell \le d$, $cost(\ell) \le \frac{4n2^\ell}{2^d} + 2d$. Summing this up over $\ell = 0, 1, \ldots, d$, we get:

$$\begin{aligned} cost(T) &= \sum_{\ell=0}^{d} cost(\ell) \\ &\le 2d(d+1) + 4n \sum_{\ell=0}^{d} \frac{2^\ell}{2^d} \\ &\le 2d(d+1) + 8n = \Theta(n) \quad \square \end{aligned}$$

For positive integer k, function $f(k)$ is defined inductively as follows:

$f(0) = 0$

For $k > 0$, $\quad f(k) = 2f(k-1) + 2k$

Lemma 4.8. $f(k) < 2^{k+2}$

Proof. For $k > 0$, expanding out the recursive expression for $f(k)$, we get:

$$\begin{aligned} f(k) &= 2k + 2^2(k-1) + 2^3(k-2) + \cdots + 2^k(1) + 2^{k+1} f(0) \\ &= 2[k + 2(k-1) + 2^2(k-2) + 2^3(k-3) + \cdots + 2^{k-1}] \\ &= 2 \sum_{i=0}^{k-1} \sum_{j=0}^{i} 2^j = 2 \sum_{i=0}^{k-1} (2^{i+1} - 1) \\ &\le 2 \sum_{i=0}^{k} 2^i \le 2^{k+2} \quad \square \end{aligned}$$

Lemma 4.9. *For any integer ℓ, where $0 \le \ell \le d$,*

$$cost(\ell) \le \frac{4n2^\ell}{2^d} + 2d$$

Proof. We first add dummy nodes and edges to T (these nodes do not belong to R), so that there are 2^d nodes at a depth of d. The dummy nodes do not change the cost of the nearest neighbor TSP on R, but this will simplify the presentation of the proof. Let n' denote the number of nodes in T after the addition of these nodes. Note that $n' < 2n$, since T is a perfectly balanced tree. Let T_ℓ denote the induced subtree of T restricted to vertices v such that $depth(v) \le \ell$.

For any vertex u in T_ℓ, let T_u denote the subtree of T_ℓ that is rooted at u, n_u the number of vertices in T_u, and d_u the depth of T_u (the maximum length of a root-leaf path in T_u). Define $c(u)$ as:

$$c(u) = \sum_{(v \in \text{ leaves of } T_u) \wedge (v \in R)} cost(v) \tag{8}$$

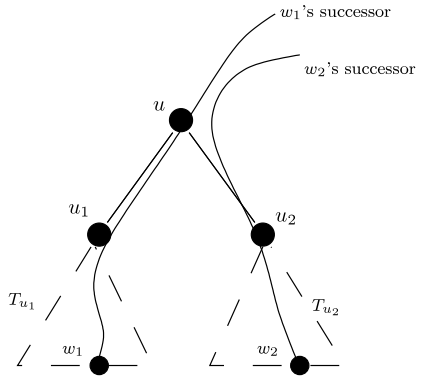

Fig. 3. Proof of Lemma 4.10.

We wish to bound $cost(\ell) = c(root)$. Using Lemma 4.10, we have: $c(root) \leq f(\ell) + cost(w)$ where w is some node at a depth of ℓ in T, and f is defined above. Since for any node v in T, $cost(v) \leq 2d$, we have $c(root) \leq f(\ell) + 2d \leq 2^{\ell+2} \leq \frac{4n2^{\ell}}{2^d} + 2d$, using Lemma 4.8. □

The following lemma shows that the cost of the nearest neighbor TSP due to leaves in T_u depends only on the size of T_u, except for one leaf node w.

Lemma 4.10. *For every vertex $u \in T_\ell$, it is possible to find w, a leaf of T_u, such that*

$$c(u) \leq f(d_u) + cost(w)$$

Proof. We prove Lemma 4.10 through induction on d_u. The base case $d_u = 0$ is trivial: T_u consists of only node u, and $c(u) = cost(u) = f(d_u) + cost(u)$.

Assuming that Lemma 4.10 is true for $d_u < k$, we consider the case $d_u = k$. The subtree T_u is composed of two smaller subtrees of equal size, rooted at u_1 and u_2, as shown in Fig. 3. By the inductive hypothesis, since $d_{u_1} = d_{u_2} = k - 1$, we have:

$$\begin{aligned} c(u_1) &\leq f(d_{u_1}) + cost(w_1) \\ c(u_2) &\leq f(d_{u_2}) + cost(w_2) \end{aligned}$$

where w_1 is a leaf in T_{u_1} and w_2 is a leaf in T_{u_2}.

$$c(u) = c(u_1) + c(u_2) \leq f(d_{v_1}) + f(d_{v_2}) + cost(w_1) + cost(w_2) \tag{9}$$

Without loss of generality, among w_1 and w_2, suppose w_2 appeared later in the order π. Then, $cost(w_1)$ must be less than the distance between w_1 and w_2 on the tree, which is $2d_u = 2k$. Using this in Eq. (9), we get:

$$\begin{aligned} c(u) &\leq f(d_{v_1}) + f(d_{v_2}) + 2k + cost(w_2) \\ &= 2f(k-1) + 2k + cost(w_2) \\ &= f(k) + cost(w_2) \end{aligned}$$

This proves the inductive case $d_u = k$, and thus Lemma 4.10 is true for all $u \in T_\ell$. □

Lemma 4.11. *If G has a perfect binary tree as a spanning tree, then $C_Q(G) = o(C_C(G))$.*

Proof. Theorem 4.7 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 the desired result. □

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 4.12. *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 4.13. *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 the counting delay complexity is greater than the queuing delay complexity 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.

Acknowledgements

We thank Eric Ruppert and Maurice Herlihy for helpful discussions. The first author is supported in part through NSF grant CNS 0520009. The second author is supported in part through NSF grant CNS 0520102.

References

[1] J. Aspnes, M. Herlihy, N. Shavit, Counting networks, Journal of the ACM 41 (5) (1994) 1020–1048.
[2] C. Busch, M. Mavronicolas, P. Spirakis, The cost of concurrent, low-contention read modify write, Theoretical Computer Science 333 (2005) 373–400.
[3] S. Cook, C. Dwork, R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, SIAM Journal on Computing 15 (1) (1986) 87–97.
[4] M.J. Demmer, M. Herlihy, The arrow distributed directory protocol, in: Proc. 12th International Symposium on Distributed Computing, DISC, 1998, pp. 119–133.
[5] P. Fatourou, M. Herlihy, Adding networks, in: Proc. 15th International Symposium on Distributed Computing, DISC, 2001, pp. 330–342.
[6] M. Herlihy, S. Tirthapura, R. Wattenhofer, Competitive concurrent distributed queuing, in: Proc. 20th ACM Symposium on Principles of Distributed Computing, PODC, 2001, pp. 127–133.
[7] M. Herlihy, S. Tirthapura, R. Wattenhofer, Ordered multicast and distributed swap, Operating Systems Review 35 (1) (2001) 85–96.
[8] F. Kuhn, R. Wattenhofer, Dynamic analysis of the arrow distributed protocol, in: Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, 2004, pp. 294–301.
[9] K. Raymond, A tree-based algorithm for distributed mutual exclusion, ACM Transactions on Computer Systems 7 (1) (1989) 61–77.
[10] D. Rosenkrantz, R. Stearns, P. Lewis, An analysis of several heuristics for the traveling salesman problem, SIAM Journal on Computing 6 (3) (1977) 563–581.
[11] R. Wattenhofer, P. Widmayer, An inherent bottleneck in distributed counting, Journal of Parallel and Distributed Computing 49 (1) (1998) 135–145.