Work-Competitive Scheduling for Cooperative Computing with Dynamic Groups*

Chryssis Georgiou[1]
cg2@cse.uconn.edu

Alexander Russell[1]
acr@cse.uconn.edu

Alex A. Shvartsman[1,2]
aas@cse.uconn.edu

[1]Dept. of Computer Science & Engineering
University of Connecticut
Storrs, CT 06269

[2]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

The problem of cooperatively performing a set of t tasks in a decentralized setting where the computing medium is subject to failures is one of the fundamental problems in distributed computing. The setting with partitionable networks is especially challenging, as algorithmic solutions must accommodate the possibility that groups of processors become disconnected (and, perhaps, reconnected) during the computation. The efficiency of task-performing algorithms is often assessed in terms of their *work*: the total number of tasks, counting multiplicities, performed by all of the processors during the computation. In general, an adversary that is able to partition the network into g components can cause any task-performing algorithm to have work $\Omega(t \cdot g)$ even if each group of processors performs no more than the optimal number of $\Theta(t)$ tasks.

Given such pessimistic lower bounds, and in order to understand better the *practical* implications of performing work in partitionable settings, we study distributed work-scheduling and pursue a *competitive* analysis. Specifically, we study a simple randomized scheduling algorithm for p asynchronous processors, connected by a dynamically changing communication medium, to complete t known tasks. We compare the performance of the algorithm against that of an "off-line" algorithm with full knowledge of the future changes in the communication medium. We describe a notion of *computation width*, which associates a natural number with a history of changes in the communication medium, and show both upper and lower bounds on competitiveness in terms of this quantity. Specifically, we show that a simple randomized algorithm obtains the competitive ratio $(1 + \mathbf{cw}/e)$, where $\mathbf{cw}$ is computation width; we then show that this ratio is tight.

Categories and Subject Descriptors

F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency, probabilistic computation*

General Terms

Algorithms, Theory

Keywords

On-line algorithms, competitive analysis, partitionable networks, distributed computation, independent tasks, randomized algorithms, work complexity

1. INTRODUCTION

The problem of cooperatively performing a known set of tasks in a decentralized setting where the computing medium is subject to failures is one of the fundamental problems in distributed computing. Variations on this problem have been studied in a variety of settings, for example, in message-passing models [6, 7, 10] and in partitionable network models [9, 16]. In the settings where network partitions may interfere with the progress of computation, the challenge is to maintain efficiency despite the dynamically changing processor connectivity.

This problem is normally abstracted in terms of a set of t tasks that must be performed in a distributed environment consisting of p processors, subject to processor failures and communication disruptions. Algorithmic solutions for this problem are typically evaluated by determining their worst-case *work*, the total number of computation steps performed by all processors during the computation. We consider the situation where the tasks are *similar*, that is, completion of each each task requires the same number of computation steps, and where task-oriented work dominates local bookkeeping. In this case the work incurred by an algorithm is simply the total number of tasks, counting multiplicities, completed by the processors.

The details of the computation model naturally have a dramatic impact on the existence of efficient (or even interesting) algorithms for the problem. In this paper, we con-

*This research is supported in part by the NSF ITR Grant 0121277. The work of the second author is supported in part by the NSF CAREER Award 0093065 and NSF Grants 0220264 and 0218443. The work of the third author is supported in part by the NSF CAREER Award 9984778 and the NSF Grant 9988304.

STOC'03, June 9–11, 2003, San Diego, California, USA.

sider the *partitionable network* scenario consisting of p asynchronous processors with a communication medium that is subject to arbitrary *partitions* during the life of the computation. This model is motivated by the abstraction provided by a typical *group communication scheme*; see, for example, the surveys in [18]. Specifically, at each point of the computation, we assume that the communication medium effectively partitions the processors into non-overlapping *groups*: communication within a group is instantaneous and reliable, communication across groups is impossible. Naturally, processors in the same group can share their knowledge of completed tasks and, while they remain connected, avoid doing redundant work. For the remainder of the paper we refer to a transition from one partition to another as a *reconfiguration.*

We do not charge for coordination within a group, simply treating grouped processors as a single (virtual) asynchronous processor. In particular, if a group of processors performs a set of t tasks during the lifetime of the group, we charge this group t units of work, ignoring, for example, partially completed tasks which may remain at the group's demise or the cost of synchronizing processors' knowledge during the group's inception. Each processor may cease executing tasks *only* when it knows the results of all tasks. While processors are asynchronous, they do not crash.

An algorithm in this model is a rule which, given a group of processors and a set of tasks known by this group to be complete, determines a task for the group to complete next. In the case where all processors are disconnected during the entire computation, any algorithm must incur $\Omega(tp)$ work. On the other hand, any reasonable algorithm should attain $O(t)$ work in the case where all processors remain connected during the computation. Considering that *every* algorithm performs poorly in the totally disconnected case, it seems reasonable to treat the problem as an on-line problem and pursue competitive analysis.

Fix, for the moment, an algorithm A. For expository purposes, let us treat both the processors' asynchrony and the dynamics of the network as if they were determined by an adversary $\mathcal{A}$. The adversary determines an initial partition $\mathcal{P}_1$ of the processors into groups and determines how many tasks each group of $\mathcal{P}_1$ completes before the next reconfiguration; while the *number* of tasks completed by each group is determined by the adversary, the actual subset of tasks (that is, the *identity* of the tasks) so completed by each group is determined by the algorithm A. The adversary then determines a reconfiguration of the processors, giving rise to a new partition $\mathcal{P}_2$, and, as before, determines how many tasks each group of $\mathcal{P}_2$ completes before the next reconfiguration. Any group created during such a reconfiguration is assumed to have the combined knowledge of all its members: any task known to be complete by a processor of the group G is known to be complete by all processors of G. This process of reconfiguration and computation continues until every processor is aware of the outcome of every task. Groups with knowledge of the outcome of all tasks cause no work: in effect, they may "idle" until the next reconfiguration. Note that for this algorithm A, the work caused by the adversary $\mathcal{A}$ is completely determined by *(i)* the collection of groups that existed during the computation, *(ii)* the number of tasks $\mathcal{A}$ allots to each group, and *(iii)* for each group G, the identities of all those groups in which processors of G have previously been members. (Note that the initial knowledge of the group G is determined in part by *(iii)*.) These characteristics can be captured by a certain directed acyclic graph, to which we refer as a *computation pattern.* This is formally defined in the next section. Note that different sequences of reconfigurations can in fact give rise to the same computation pattern.

As an example, consider the scenario with 3 processors which, starting from isolation, are permitted to proceed synchronously until each has completed $t/2$ tasks; at this point an adversary chooses a pair of processors to merge into a group. It is easy to show that if A_1, A_2, and A_3 are subsets of $[t]$ of size $t/2$, then there is a pair (A_i, A_j) (where $i \neq j$) so that $|A_i \cap A_j| \geq t/6$: in particular, for *any* scheduling algorithm, there is a pair of processors which, if merged at this point, will have $t/6$ duplicated tasks; this pair alone must then expend $t + t/6$ work to complete all t tasks. The optimal off-line algorithm that schedules tasks with full knowledge of future merges, of course, accrues only t work for the merged pair, as it can arrange for zero overlap. Furthermore, if the adversary partitions the two merged processors immediately after the merge (after allowing the processors to exchanged information about task executions), then the work performed by the merged and then partitioned pair is $t + t/3$; the work performed by the optimal algorithm remains unchanged, since it terminates at the merge.

Contributions. We analyze the natural randomized algorithm *Random Select* (RS) where each processor (or group) determines the next task to complete by randomly selecting the task from the set of tasks this group does not know to be completed. We compare the expected work of this algorithm to the work of an optimal off-line algorithm, which may schedule tasks with full knowledge of future partitions.

In order to precisely state the results of the paper, we pause to introduce some notation. In the literature, groups of processors are given structured names, such that a group G is a pair $\langle G.id, G.set\rangle$, where $G.id$ is the unique identifier of G and $G.set$ is the set of processor identifiers that constitute the membership of the group. To reduce notation clutter, given a group named G, we use G to stand for $G.set$ in this paper (e.g., if two, possibly distinct, groups G and G' have identical membership, we express this by $G = G'$).

As discussed previously, an adversary determines a *computation pattern* C in a natural way; this is a directed acyclic graph, each vertex corresponding to a group of processors that exists during some point of the computation; a directed edge is placed between group G and group G' if $G' \cap G \neq \emptyset$ and G' was formed by a reconfiguration involving processors in G (this is discussed in more detail in the next section). We say that two groups G and G' are *independent* if there is no directed path connecting one to the other. For such a pattern C, the *computation width* of C, denoted $\mathbf{cw}(C)$, is the maximum number of independent groups reachable (along directed paths) in this DAG from any vertex. We show the following:

- (Upper bound.) For any computation pattern C, the randomized algorithm RS discussed above is $(1 + \mathbf{cw}(C)/e)$-work competitive.
- (Lower bound.) For any scheduling algorithm $A(p,t)$, any $\epsilon > 0$, and any nonzero $k \in \mathbb{N}$, there exists p, t, and a computation pattern C so that $\mathbf{cw}(C) = k$ and the work performed by algorithm $A(p,t)$ is at least

$(1+k/e-\epsilon)$ times that of the off-line algorithm. Specifically, RS achieves the optimal competitive ratio over the set of all computation patterns with a given computation width.

Prior and related work; motivation. Though this problem has been the subject of active research, known solutions address narrow special cases, or provide substantially weaker bounds. Malewicz, Russell and Shvartsman [16] introduced the notion of *h-waste* that measures the worst-case redundant work performed by h groups (or processors) when started in isolation and merged into a single group at some later time. While these results are deterministic, they only adequately describe such computation to the point of the *first* reconfiguration, where the reconfiguration is further assumed to simply *merge* groups together. Dolev, Segala, and Shvartsman [9] model ultimately diverging reconfiguration patterns for which the termination time of any on-line task-performing algorithm is greater than the termination time of an off-line task-performing algorithm by a factor linear in p. Georgiou and Shvartsman [14] give upper bounds on work for an algorithm that performs work in the presence of network fragmentations and merges (i.e., limited patterns of reconfigurations) using a group communication service where processors initially start in a single group. They establish an upper bound of $O(\min(t \cdot p,\ t + t \cdot g(C)))$ where $g(C)$ is the total number of new groups formed during the computation pattern C. Note that there can be an arbitrary gap between $\mathbf{cw}(C)$ and $g(C)$.

Thus prior work established reasonably tight (in the length of the processor schedule) results for a *single* merge, illustrated the fact that on-line algorithms subject to diverging reconfiguration patterns incur linear (in p) overhead relative to an off-line algorithm, and showed an upper bound for an algorithm using group communication services for a limited pattern of reconfigurations starting with a *single* group.

The problem of cooperation on a common set of tasks in distributed settings has been studied in message-passing models [6, 7, 10], using group communications [9, 14], and in shared-memory models [15]. Our definition of work follows that of Dwork, Halpern and Waarts [10]. These studies present various load-balancing techniques for structuring the work for the computing devices that are able to communicate by some means. The studies of Georgiades, Mavronicolas, and Spirakis [12] and Papadimitriou and Yannakakis [17] investigate the impact of communication topology on the effectiveness of load-balancing. A related problem, called *collect*, has been studied in the shared-memory model of computation. This problem was originally abstracted by Saks, Shavit, and Woll [19].

The notion of competitiveness was introduced by Sleator and Tarjan [20] (see also Fiat and Rabani [4], Awerbuch, Kutten and Peleg [2], and Ajtai, Aspnes, Dwork and Waarts [1]).

Group communication services have become important as building blocks for fault-tolerant distributed systems. Such services enable processors located in a fault-prone network to operate collectively as a group, using the services to multicast messages to group members (see the special issue [18]). To evaluate the effectiveness of partitionable group communication services, Sussman and Marzullo [21] proposed a measure (*cushion*) precipitated by a simple partition-aware application. Babaoglu *et al.* [3] study systematic support for partition awareness based on group communication services in a wide range of application areas.

The structure of this paper is as follows. In Section 2 we define the problem and model of computation. In Section 3 we present and analyze the randomized algorithm RS. In Section 4 we prove a lower bound for the problem. We conclude in Section 5. A summary of preliminary results for merges appears in [13].

2. MODEL AND DEFINITIONS

We consider a distributed system consisting of p asynchronous processors connected by communication links; each processor has a unique identifier from the set $[p] = \{1, 2, \ldots, p\}$; the value p is known to all processors. The problem is then defined in terms of t tasks with unique identifiers, initially known to all processors. The tasks are independent and idempotent—multiple executions of the same task have the same effect as a single execution. Processors may only cease executing tasks when they know the results of all tasks. This general problem is often referred to as *Do-All*.

The model is complicated by subjecting the processors to dynamic changes in the communication medium. In particular, at each instant of time, the network is partitioned into a collection of *groups*. Communication between processors in the same group is instantaneous and reliable, so that grouped processors may perfectly cooperate to complete tasks; communication across groups, however, is not possible. We consider the dynamic case where communication can be arbitrarily lost and re-established. In particular, the computation of the processors is punctuated by a sequence of *reconfigurations*; each reconfiguration may induce an arbitrary change in the partition of the processors into groups. We shall assume that reconfigurations are atomic with respect to task executions. That is, a reconfiguration does not occur when some tasks are half-way through execution.

In order to focus on scheduling issues, we assume that processors in a single group work as a single virtual unit; indeed, we will treat them as a single asynchronous processor. In particular, upon the establishment of a new group by a reconfiguration, the processors in the group share their knowledge (of completed tasks) before they continue processing. A deterministic algorithm D in this model is a rule which, given a processor (or group of processors) and a collection of tasks known to be completed, determines the next task for this processor (or group) to complete. Specifically, an algorithm is a function $D : 2^{[p]} \times 2^{[t]} \rightarrow [t]$; we note that the lower bounds proved in this paper actually apply to a wider class of algorithms that may in fact take into account the entire history of the computation of the group in question. For simplicity, we assume that $\forall P \subset [p], \forall T \subsetneq [t], D(P,T) \notin T$, which is to say that the algorithm never chooses to complete a task it already knows to be completed. Our goal will be to design algorithms which schedule the execution of the tasks to minimize the total *work*, where work is defined to be *the number of tasks executed by all the processors during the entire computation (counting multiplicities)*. Ideally, the sets of tasks completed by two groups of processors when these groups are merged should be disjoint to avoid wasted effort. This is impossible in general, as processors must schedule their work in ignorance of future reconfigurations and, moreover, circumstances where two groups of processors merge who have collectively completed more than t tasks will necessitate wasted work. A processor may cease executing tasks only when it knows the results of all tasks. We refer to this version of the *Do-All* problem as *Omni-Do*.

We will consider the behavior of an algorithm in the face of an adversary (which is *oblivious* in the sense of [5]) that determines both the *sequence of reconfigurations* and the *number of tasks completed* by each group before it is involved in another reconfiguration. Taken together, this information determines a *computation pattern*: this is a directed acyclic graph (DAG), each vertex of which corresponds to a group G of processors that existed during the computation; a directed edge is placed from G_1 to G_2 if G_2 was created by a reconfiguration involving G_1. We label each vertex of the DAG with the group of processors associated with that vertex and the total number of tasks that the adversary allows the group of processors to perform before the next reconfiguration occurs. As mentioned before, different adversaries (causing different sequences of reconfigurations) may give rise to the same computation pattern; the *work* caused by an adversary, however, depends only on the computation pattern determined by that adversary.

Specifically, if t is the number of tasks and p the number of processors, then such a computation pattern is a labeled and weighted directed acyclic graph, that we call a (p, t)-DAG:

Definition 1. A (p, t)**-DAG** is a directed acyclic graph $C = (V, E)$ augmented with a weight function $h : V \to \mathbb{N}$ and a labeling $g : V \to 2^{[p]} \setminus \{\emptyset\}$ so that:

- $\forall v \in V$, $h(v) \leq t$ and for any maximal path $p = (v_1, \ldots, v_k)$ in C, $\sum h(v_i) \geq t$. (This guarantees that any algorithm terminates during the computation described by the DAG.)
- g possesses the following "initial conditions":
$$[p] = \dot{\bigcup_{v:\ in(v)=0}} g(v).$$
- g respects the following "conservation law": there is a function $\phi : E \to 2^{[p]} \setminus \{\emptyset\}$ so that for each $v \in V$ with $in(v) > 0$,
$$g(v) = \dot{\bigcup_{(u,v)\in E}} \phi\big((u, v)\big),$$
and for each $v \in V$ with $out(v) > 0$,
$$g(v) = \dot{\bigcup_{(v,u)\in E}} \phi\big((v, u)\big).$$

Here $\dot{\cup}$ denotes disjoint union and $in(v)$ and $out(v)$ denote the in-degree and out-degree of v, respectively. Finally, for two vertices $u, v \in V$, we write $u \leq v$ if there is a directed path from u to v; we then write $u < v$ if $u \leq v$ and u and v are distinct.

Example. As an example, consider the $(12, t)$-DAG shown on Figure 1. Here we have $g_1 = \{p_1\}$, $g_2 = \{p_2, p_3, p_4\}$, $g_3 = \{p_5, p_6\}$, $g_4 = \{p_7\}$, $g_5 = \{p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $g_6 = \{p_1, p_2, p_3, p_4, p_6\}$, $g_7 = \{p_8, p_{10}\}$, $g_8 = \{p_9, p_{11}, p_{12}\}$, $g_9 = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}\}$, $g_{10} = \{p_5, p_{11}\}$, and $g_{11} = \{p_9, p_{12}\}$.

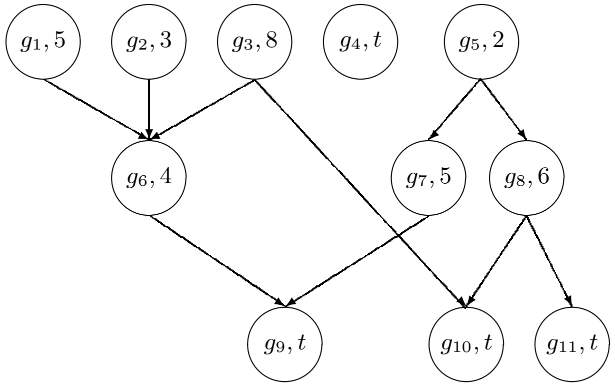

Figure 1: An example of a $(12, t)$-DAG

This computation pattern models all asynchronous computations (adversaries) with the following behavior: (*i*) The processors in groups g_1 and g_2 and processor p_6 of group g_3 are regrouped during some reconfiguration to form group g_6. Processor p_5 of group g_3 becomes a member of group g_{10} during the same reconfiguration (see below). Prior to this reconfiguration, processor p_1 (the singleton group g_1) has performed exactly 5 tasks, the processors in g_2 have cooperatively performed exactly 3 tasks and the processors in g_3 have cooperatively performed exactly 8 tasks (assuming that $t > 8$). (*ii*) Group g_5 is partitioned during some reconfiguration into two new groups, g_7 and g_8. Prior to this reconfiguration, the processors in g_5 have performed exactly 2 tasks. (*iii*) Groups g_6 and g_7 merge during some reconfiguration and form group g_9. Prior to this merge, the processors in g_6 have performed exactly 4 tasks (counting only the ones performed after the formation of g_6 and assuming that there are at least 4 tasks remaining to be done) and the processors in g_7 have performed exactly 5 tasks. (*iv*) The processors in group g_8 and processor p_5 of group g_3 are regrouped during some reconfiguration into groups g_{10} and g_{11}. Prior to this reconfiguration, the processors in group g_8 have performed exactly 6 tasks (assuming that there are at least 6 tasks remaining, otherwise they would have performed the remaining tasks). (*v*) The processors in g_9, g_{10}, and g_{11} run until completion with no further reconfigurations. (*vi*) Processor p_7 (the singleton group g_4) runs in isolation for the entire computation.

Let D be a deterministic algorithm for *Omni-Do* and C a computation pattern, we let $W_D(C)$ denote the total work expended by algorithm D, where reconfigurations are determined according to the computation pattern C. W_D is formally defined as follows:

Definition 2. Let C be a (p, t)-DAG and D a deterministic algorithm for *Omni-Do*. $W_D(C)$ is defined inductively as follows. For a vertex v of C with $in(v) = 0$, define L_v to be the set containing the first $h(v)$ tasks completed by group $g(v)$ according to D. Otherwise, $in(v) > 0$; in this case, let $\check{L}_v = \bigcup_{u<v} L_u$ denote the collection of all tasks known to be complete at the inception of the group $g(v)$. Then let L_v be the first $h(v)$ tasks completed by group $g(v)$ according to D starting with knowledge $\check{L}_v$. If $h(v) > t - |\check{L}_v|$, define $L_v = [t] \setminus \check{L}_v$. Then $W_D(C) = \sum_{v \in C} |L_v|$.

We treat randomized algorithms as distributions over deterministic algorithms; for a set Ω and a family of deterministic algorithms $\{D_r \mid r \in \Omega\}$ we let $R = \mathcal{R}(\{D_r \mid r \in \Omega\})$ denote the randomized algorithm where r is selected uniformly at random from Ω and scheduling is done according to D_r. For a real-valued random variable X, we let $\mathbb{E}[X]$ denote its expected value. We let OPT denote the optimal (off-line) algorithm. Specifically, for each C we define $W_{\mathrm{OPT}}(C) = \min_D W_D(C)$.

Definition 3. [20, 11, 5] Let α be a real valued function defined on the set of all (p,t)-DAGs (for all p and t). A randomized algorithm R is **α-competitive** if for all computation patterns C,

$$\mathbb{E}[W_{D_r}(C)] \leq \alpha(C) W_{\mathrm{OPT}}(C),$$

this expectation being taken over uniform choice of $r \in \Omega$.

Note that usually, α is fixed for all inputs; we shall see that this would be meaningless in our model. Presently, we will use a function α that depends on a certain parameter (see Definition 7) of the graph structure of C.

We now present some terminology that we will use in the rest of the paper.

Definition 4. A **partially ordered set** or **poset** is a pair $(P, \leq)$ where P is a set and $\leq$ is a binary relation on P for which *(i.)* for all $x \in P$, $x \leq x$, *(ii.)* if $x \leq y$ and $y \leq x$, then $x = y$, and *(iii.)* if $x \leq y$ and $y \leq z$, then $x \leq z$. For a poset $(P, \leq)$ we overload the symbol P, letting it denote both the set and the poset.

Definition 5. Let P be a poset. We say that two elements x and y of P are **comparable** if $x \leq y$ or $y \leq x$; otherwise x and y are **incomparable**. A **chain** is a subset H of P such that any two elements of H are comparable. An **antichain** is a subset A of P such that any two distinct elements of A are incomparable. The **width** of P, denoted $\mathbf{w}(P)$, is the size of the largest antichain of P.

Associated with any directed acyclic graph (DAG) $C = (V, E)$ is the natural **vertex poset** $(V, \leq)$ where $u \leq v$ if and only if there is a directed path from u to v. Then the **width of** C, denoted $\mathbf{w}(C)$, is the width of the poset $(V, \leq)$.

Definition 6. Given a DAG $C = (V, E)$ and a vertex $v \in V$, we define the **predecessor graph at** v, denoted $P_C(v)$ (or $P(v)$ when C is implied), to be the subgraph of C that is formed by the union of all paths in C terminating at v. Likewise, the **successor graph at** v, denoted $S_C(v)$ (or $S(v)$ when C is implied) is the subgraph of C that is formed by the union of all the paths in C originating at v.

Definition 7. The **computation width** of a DAG $C = (V, E)$, denoted $\mathbf{cw}(C)$, is defined as

$$\mathbf{cw}(C) = \max_{v \in V} \mathbf{w}(S(v)).$$

Note that the processors that comprise a group formed during a computation pattern C may be involved in many different groups at later stages of the computation, but no more than $\mathbf{cw}(C)$ of these groups can be computing in ignorance of each other's progress.

In the $(12,t)$-DAG of Figure 1, the maximum width among all successor graphs is 3: $\mathbf{w}(S((g_5, 2))) = 3$. Therefore, the computational width of this DAG is 3. Note that the width of the DAG is 6 (nodes $(g_1, 5), (g_2, 3), (g_3, 8), (g_4, t), (g_7, 5)$ and $(g_8, 6)$ form an antichain of maximum size).

3. ALGORITHM RS AND ITS ANALYSIS

In this section we present algorithm Random Select (RS) and its analysis.

3.1 Description of Algorithm RS

We consider the natural randomized algorithm RS where a processor (or group) with knowledge that the tasks τ in a set $K \subset [t]$ have been completed selects to next complete a task at random from the set $[t] \setminus K$. More formally, let $\Pi = (\pi_1, \ldots, \pi_p)$ be a p-tuple of permutations, where each π_i is a permutation of $[t]$. We describe a deterministic algorithm D_Π so that

$$\mathrm{RS} = \mathcal{R}\big(\{D_\Pi \mid \Pi \in (S_t)^p\}\big);$$

here S_t is the collection of permutations on $[t]$. Let G be a group of processors and $\gamma \in G$ the processor in G with the lowest processor identifier. Then the deterministic algorithm D_Π specifies that the group G, should it know that the tasks in $K \subset [t]$ have been completed, next completes the first task in the sequence $\pi_\gamma(1), \ldots, \pi_\gamma(t)$ which is not in K.

3.2 Analysis of Algorithm RS

We now analyze the competitive ratio (in terms of work) of algorithm RS. We write $W_{\mathrm{RS}}(C) = \mathbb{E}\left[W_{\mathrm{RS}}(C)\right]$, this expectation taken over the random choices of the algorithm. Where C can be inferred from context, we simply write W_{RS} and W_{OPT}.

We first recall Dilworth's Lemma [8], a duality theorem for posets:

LEMMA 1. *[8] The width of a poset P is equal to the minimum number of chains needed to cover P. (A family of nonempty subsets of a given set S is said to **cover** S if their union is S.)*

We will also use a generalized degree-counting argument:

LEMMA 2. *Let $G = (U, V, E)$ be an undirected bipartite graph with no isolated vertices and $h : V \to \mathbb{R}$ a non-negative weight function on G. For a vertex v, let $\Gamma(v)$ denote the vertices adjacent to v. Suppose that for some $A > 0$ and for each vertex $u \in U$ we have $\sum_{v \in \Gamma(u)} h(v) \leq A$ and that for some $B > 0$ and for each vertex $v \in V$ we have $\sum_{u \in \Gamma(v)} h(u) \geq B$, then*

$$\frac{\sum_{u \in U} h(u)}{\sum_{v \in V} h(v)} \geq \frac{B}{A}.$$

PROOF. We compute the quantity $\sum_{(u,v) \in E} h(u)h(v)$ by expanding according to each side of the bipartition:

$$\begin{aligned} A \sum_{u \in U} h(u) \quad &\geq \quad \sum_{u \in U} \Big(h(u) \cdot \sum_{v \in \Gamma(u)} h(v) \Big) = \sum_{(u,v) \in E} h(u)h(v) \\ &= \quad \sum_{v \in V} \Big(h(v) \cdot \sum_{u \in \Gamma(v)} h(u) \Big) \geq B \sum_{v \in V} h(v). \end{aligned}$$

As $A > 0$ and $\sum_v h(v) \geq B > 0$, we conclude that

$$\frac{\sum_{u \in U} h(u)}{\sum_{v \in V} h(v)} \geq \frac{B}{A},$$

as desired. □

We now establish an upper bound on the competitive ratio of the algorithm RS.

THEOREM 1. *Algorithm RS is $(1 + \mathbf{cw}(C)/e)$-competitive for any (p,t)-DAG $C = (V, E)$.*

PROOF. Let C be a (p,t)-DAG; recall that associated with C are the two functions $h : V \to \mathbb{N}$ and $g : V \to 2^{[p]} \setminus \{\emptyset\}$. For a subgraph $C' = (V', E')$ of C, we let $H(C') = \sum_{v \in V'} h(v)$. Recall that $P_C(v)$ and $S_C(v)$ denote the predecessor and successor graphs of C at v. Then, we say that a vertex $v \in V$ is *saturated* if $H(P_C(v)) \leq t$; otherwise, v is *unsaturated.* Note that if v is saturated, then the group $g(v)$ must complete $h(v)$ tasks *regardless of the scheduling algorithm used.* Along these same lines, if v is an unsaturated vertex for which $t > \sum_{u<v} h(u)$, the group $g(v)$ must complete at least $\max(h(v), t - \sum_{u<v} h(u))$ tasks under any scheduling algorithm. As these portions of C which correspond to computation which must be performed by any algorithm will play a special role in the analysis, it will be convenient for us to rearrange the DAG so that all such work appears on saturated vertices. To achieve this, note that if v is an unsaturated vertex for which $\sum_{u<v} h(u) < t$, we may replace v with a pair of vertices, v_s and v_u, where all edges directed into v are redirected to v_s, all edges directed out of v are changed to originate at v_u, the edge (v_s, v_u) is added to E, and h is redefined so that

$$h(v_s) = t - \sum_{u<v} h(u) \qquad \text{and} \qquad h(v_u) = h(v) - h(v_s).$$

Note that the graph C' obtained by altering C in this way corresponds to the same computation, in the sense that $W_D(C) = W_D(C')$ for any algorithm D. For the remainder of the proof we will assume that this alteration has been made at every relevant vertex, so that the graph C satisfies the condition

$$v \text{ unsaturated } \Rightarrow \sum_{u<v} h(u) \geq t. \tag{1}$$

Finally, for a vertex v, we let T_v be the random variable equal to the number of tasks that RS completes at vertex v. Note that if v is saturated, then $T_v = h(v)$. Let $\mathcal{S}$ and $\mathcal{U}$ denote the sets of saturated and unsaturated vertices, respectively. Given the above definitions, we immediately have

$$W_{\mathrm{OPT}} \geq \sum_{s \in \mathcal{S}} h(s)$$

and, by linearity of expectation,

$$\begin{aligned} W_{\mathrm{RS}} = \mathbb{E}\Big[\sum_v T_v\Big] &= \sum_{s \in \mathcal{S}} h(s) + \sum_{u \in \mathcal{U}} \mathbb{E}[T_u] \\ &\leq W_{\mathrm{OPT}} + \sum_{u \in \mathcal{U}} \mathbb{E}[T_u]. \end{aligned} \tag{2}$$

Our goal is to conclude that for some appropriate β,

$$\mathbb{E}\left[\sum_{u \in \mathcal{U}} T_u\right] \leq \beta \cdot \sum_{s \in \mathcal{S}} h(s) \leq \beta \cdot W_{\mathrm{OPT}}$$

and hence that RS is $1+\beta$ competitive. We will obtain such a bound by applying Lemma 2 to an appropriate bipartite graph, constructed next.

Given $C = (V, E)$ construct the (undirected) bipartite graph $G = (\mathcal{S}, \mathcal{U}, E_G)$ where $E_G = \{(s, u) \mid s < u\}$. As in Lemma 2, for a vertex v, we let $\Gamma(v)$ denote the set of vertices adjacent to v. Now assign weights to the vertices of G according to the rule $h^*(v) = \mathbb{E}[T_v]$. Note that for $s \in \mathcal{S}, h^*(s) = h(s)$ and hence by condition (1) above, we immediately have the bound

$$\forall u \in \mathcal{U}, \sum_{s \in \Gamma(u)} h^*(s) \geq t. \tag{3}$$

We now show that $\forall s \in \mathcal{S}$,

$$\sum_{u \in \Gamma(s)} h^*(u) \leq \mathbf{cw}(C) \cdot \frac{t}{e}. \tag{4}$$

Before proceeding to establish this bound, note that equations (3) and (4), together with Lemma 2 imply that

$$\begin{aligned} W_{\mathrm{RS}}(C) &\leq \sum_{s \in \mathcal{S}} h(s) + \sum_{u \in \mathcal{U}} h^*(u) \leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) \sum_{s \in \mathcal{S}} h(s) \\ &\leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) W_{\mathrm{OPT}}(C), \end{aligned}$$

as desired.

Returning now to equation (4), let $s \in \mathcal{S}$ be a saturated vertex and consider the successor graph (of C) at s, $S_C(s)$. By Lemma 1 (Dilworth's Lemma), there exist $w \triangleq \mathbf{w}(S_C(s)) \leq \mathbf{cw}(C)$ paths in $S_C(s)$, $P_1, P_2, \ldots P_w$ so that their union covers $S_C(s)$. Let X_i be the random variable whose value is the number of tasks performed by RS on the portion of the path P_i consisting of unsaturated vertices. Note that if $u \in V$ is unsaturated and $u \leq v$, then v is unsaturated and hence, for each path P_i, there is a first unsaturated vertex u_i^0 after which every vertex of P_i is unsaturated. Note now that for a fixed individual task τ, conditioned upon the event that τ is not yet complete, the probability that τ is *not* chosen by RS for completion at a given selection point in $P_C(u_i^0)$ is no more than $(1 - 1/t)$. Let L_i be the random variable whose value is the set of tasks left incomplete by RS at the formation of the group $g(u_i^0)$. As u_i^0 is unsaturated, $\sum_{v<u_i^0} h(v) \geq t$ by condition (1) and hence, for each i,

$$\Pr[\tau \in L_i] \leq (1 - 1/t)^t \leq 1/e.$$

As there are a total of t tasks,

$$\mathbb{E}[|L_i|] \leq t/e.$$

Of course, since RS completes a new task at each step, $X_i \leq |L_i|$ so that $\mathbb{E}[X_i] \leq t/e$ and by the linearity of expectation

$$\mathbb{E}\Big[\sum_i X_i\Big] \leq w \cdot t/e.$$

Now every unsaturated vertex in $S_C(s)$ appears in some P_i and hence

$$\sum_{u \in \Gamma(s)} h^*(u) \leq \mathbb{E}\Big[\sum_i X_i\Big] \leq wt/e \leq \mathbf{cw}(C) \cdot t/e,$$

as desired. □

Theorem 1 implies a constant upper bound for patterns that consist entirely of merges (that is, where all reconfigurations are given by taking unions of existing groups). This subsumes the results reported in [13].

COROLLARY 1. *Algorithm RS is $\left(1 + \frac{1}{e}\right)$-competitive for any (p,t)-DAG C with $\mathbf{cw}(C) = 1$.*

4. A LOWER BOUND

We begin with a lower bound for *deterministic* algorithms. This is then applied to give a lower bound for randomized algorithms in Corollary 2.

THEOREM 2. *Let $a : \mathbb{N} \to \mathbb{R}$ and D be a deterministic scheduling algorithm for Omni-Do so that D is $a(\mathbf{cw}(\cdot))$-competitive (that is D is α-competitive, for a function $\alpha = a \circ \mathbf{cw}$)). Then $a(c) \geq 1 + c/e$.*

PROOF. Fix $k \in \mathbb{N}$. Consider the case when $t = p = g \gg k$ and $t \bmod k = 0$, g being the number of initial groups. We consider a computation pattern $C_{\mathbf{G}}$ determined by a tuple $\mathbf{G} = (G_1, \ldots, G_{t/k})$ where each $G_i \subset [t]$ is a set of size k and $\bigcup_i G_i = [t]$. Initially, the computation pattern $C_{\mathbf{G}}$ has the processors synchronously proceed until each has completed t/k tasks; at this point, the processors in G_i are merged and allowed to exchange information about task executions. Each G_i is then immediately partitioned into c groups. Note that the off-line optimal algorithm accrues exactly t^2/k work for this computation history (it terminates prior to the partitions of the G_i).

We will show that for any D, there is a selection of the G_i so that

$$W_D(C_{\mathbf{G}}) \geq t^2/k \left[1 + c(1 - \frac{1}{k})^k - o(1)\right],$$

and hence that $a(c) \geq 1 + c/e$. Consider the behavior of D when the $\mathbf{G}$ is selected at random, uniformly among all such tuples. Let $P_i \subset [t]$ be the subset of t/k tasks completed by processor i before the merges take place; these sets are determined by the algorithm D. We begin by bounding

$$\underset{\mathbf{G}}{\mathbb{E}}\left[\Big|\bigcup_{i \in G_1} P_i\Big|\right].$$

To this end, consider an experiment where we select k sets $Q_1, \ldots, Q_k$, each Q_i selected independently and uniformly from the set $\{P_i\}$. Now, for a specific task τ, let $p_\tau = \Pr_{Q_1}[\tau \notin Q_1]$, so that $\Pr_{Q_i}\left[\tau \notin \bigcup_i Q_i\right] = p_\tau^k$. As the Q_i are selected independently,

$$\underset{Q_i}{\mathbb{E}}\left[\Big|[t] - \bigcup_i Q_i\Big|\right] = \sum_\tau p_\tau^k.$$

Observe now that

$$\sum_\tau (1 - p_\tau) = \sum_\tau \underset{Q_1}{\Pr}[\tau \in Q_1] = \underset{Q_1}{\mathbb{E}}\left[|Q_1|\right] = t/k$$

and hence $\sum_\tau p_\tau = t(1 - 1/k)$. As the function $x \mapsto x^k$ is convex on $[0, \infty)$, $\sum_\tau p_\tau^k$ is minimized when the p_τ are equal and we must have

$$\underset{Q_i}{\mathbb{E}}\left[\Big|[t] - \bigcup_i Q_i\Big|\right] \geq t \cdot \left(1 - \frac{1}{k}\right)^k.$$

Now observe that, conditioned on the Q_i being distinct, the distribution of $(Q_1, \ldots, Q_k)$ is identical to that of $(P_{g_1^1}, \ldots, P_{g_k^1})$ where the random variable $G_1 = \{g_1^1, \ldots, g_k^1\}$. Considering that $\Pr[\exists i \neq j, Q_i = Q_j] \leq k^2/t$, we have

$$\underset{Q_i}{\mathbb{E}}\left[\Big|[t] - \bigcup_i Q_i\Big|\right] \leq \left(1 - \frac{k^2}{t}\right) \underset{\mathbf{G}}{\mathbb{E}}\left[t - \Big|\bigcup_{i \in G_1} P_i\Big|\right] + 1 \cdot \frac{k^2}{t}$$

and hence as $t \to \infty$ we see that the expected number of tasks remaining for those processors in group G_1 is

$$\underset{\mathbf{G}}{\mathbb{E}}\left[t - \Big|\bigcup_{i \in G_1} P_i\Big|\right] \geq t(1 - 1/k)^k - o(1).$$

Of course, the distribution of each G_i is the same, so that

$$\underset{\mathbf{G}}{\mathbb{E}}\left[\sum_{i=1}^{t/k}\Big(t - \Big|\bigcup_{j \in G_i} P_j\Big|\Big)\right] = [1 - o(1)] \left(\frac{t}{k}\right) \cdot t \left(1 - \frac{1}{k}\right)^k.$$

In particular, there must exist a specific selection of $\mathbf{G} = (G_1, \ldots, G_{t/k})$ which achieves this bound. Recall that every G_i is partitioned into c groups. Therefore, for such $\mathbf{G}$, the total work is at least

$$\frac{t^2}{k} \cdot \left(1 + [1 - o(1)] \cdot c \cdot (1 - \frac{1}{k})^k\right).$$

As $\lim_{k \to \infty}(1 - \frac{1}{k})^k = \frac{1}{e}$, this completes the proof. □

As the above stochastic computation pattern $C_{\mathbf{G}}$ is independent of the deterministic algorithm D, this immediately gives rise to a lower bound for randomized algorithms:

COROLLARY 2. *Let $\mathcal{R}(\{D_r \mid r \in \Omega\})$ be a randomized scheduling algorithm for Omni-Do that is $(a \circ \mathbf{cw})$-competitive. Then $a(c) \geq 1 + c/e$.*

PROOF. Assume for contradiction that for some c, $a(c) < 1 + c/e$ and let k be large enough so that $(1 - \frac{1}{k})^k > a(c) - 1$. For this k we proceed as in the proof above, considering a random $\mathbf{G}$ and the computation pattern $C_{\mathbf{G}}$ with $t = g = p$ congruent to 0 mod k, g being the number of initial groups. Then, as above,

$$\begin{aligned}
\underset{\mathbf{G}}{\mathbb{E}}\left[\underset{r}{\mathbb{E}}\left[W_{D_r}(C_{\mathbf{G}})\right]\right] &= \underset{r}{\mathbb{E}}\left[\underset{\mathbf{G}}{\mathbb{E}}\left[W_{D_r}(C_{\mathbf{G}})\right]\right] \\
&\geq \min_r \left[\underset{\mathbf{G}}{\mathbb{E}}\left[W_{D_r}(C_{\mathbf{G}})\right]\right] \\
&\geq \frac{t^2}{k} \cdot \left(1 + [1 - o(1)] \cdot c \cdot (1 - \frac{1}{k})^k\right).
\end{aligned}$$

Hence there exists a $\mathbf{G}$ so that $\mathbb{E}_r\left[W_{D_r}(C_{\mathbf{G}})\right] \geq \frac{t^2}{k} \cdot \left(1 + [1 - o(1)]\frac{c}{e}\right)$, which completes the proof. □

5. CONCLUSIONS & OPEN PROBLEMS

We established bounds on the competitive ratio of a natural randomized algorithm for scheduling in partitionable networks and show, furthermore, that for the relevant gradation of computation patterns these bounds are tight. These results lead to a better understanding of the effectiveness of computation in *group communication schemes*, a widely used paradigm for computing in distributed environments.

One outstanding open question is to derandomize the schedules used by task-performing algorithms in this work. Another promising direction is to study the task-performing paradigm in the models of computation that combine network reconfigurations with processor failures. The goal is to establish complexity results that show how performance of task-performing algorithms depends both on the extent of the network reconfiguration and on the number of processor failures.

6. REFERENCES

[1] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the* 35^{th} *Symposium on Foundations of Computer Science (FOCS 1994)*, pages 401–411, 1994.

[2] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing (STOC 1992)*, pages 571–580, 1992.

[3] O. Babaoglu, R. Davoli, A. Montresor, and R. Segala. System support for partition-aware network applications. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS 1998)*, pages 184–191, 1998.

[4] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing (STOC 1992)*, pages 39–50, 1992.

[5] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

[6] B. Chlebus, R. De Prisco, and A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[7] R. De Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proceedings of the* 13^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1994)*, pages 161–172, 1994.

[8] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[9] S. Dolev, R. Segala, and A.A. Shvartsman. Dynamic load balancing with group communication. In *Proceedings of the* 6^{th} *International Colloquium on Structural Information and Communication Complexity (SIROCCO 1999)*, pages 111–125, 1999.

[10] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998. A preliminary version appears as "Accomplishing work in the presence of failures" in the *Proceedings of the* 11^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pages 91–102, 1992.

[11] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.D. Sleator, and N.E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[12] S. Georgiades, M. Mavronicolas, and P. Spirakis. Optimal, distributed decision-making: The case of no communication. In *Proceedings of the* 12^{th} *Int-l Symposium on Foundamentals of Computation Theory (FCT 1999)*, pages 293–303, 1999.

[13] Ch. Georgiou, A. Russell, and A.A. Shvartsman. Optimally work-competitive scheduling for cooperative computing with merging groups (brief announcement). In *Proceedings of the* 22^{nd} *ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[14] Ch. Georgiou and A.A. Shvartsman. Cooperative computing with fragmentable and mergeable groups. *Journal of Discrete Algorithms*, to appear. (Also in *Proc. of the* 7^{th} *Int-l Colloquium on Structural Information and Communication Complexity (SIROCCO 2000),* pages 141–156, 2000).

[15] P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.

[16] G.G. Malewicz, A. Russell, and A. A. Shvartsman. Distributed cooperation during the absence of communication. In *Proceedings of the* 14^{th} *International Symposium on Distributed Computing (DISC 2000)*, pages 119–133, 2000.

[17] C.H. Papadimitriou and M. Yannakakis. On the value of information in distributed decision-making. In *Proceedings of the* 10^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1991)*, pages 61–64, 1991.

[18] D. Powell, editor. *Special Issue on Group Communication Services*, volume 39(4) of *Communications of the ACM*. ACM Press, 1996.

[19] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the* 2^{nd} *ACM-SIAM Symposium on Discrete Algorithms (SODA 1991)*, pages 351–362, 1991.

[20] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[21] J.B. Sussman and K. Marzullo. The bancomat problem: An example of resource allocation in a partitionable asynchronous system. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC 1998)*, pages 363–377, 1998.