

# THE SANDPILE LOAD BALANCER AND LATENCY MINIMIZATION PROBLEM

BERJ CHILINGIRIAN  
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ABSTRACT. I present the sandpile load balancer, an application of the abelian sandpile model to dynamic load balancing. The sandpile load balancer exploits the self-organized criticality of the abelian sandpile model to distribute web traffic across a network of servers. I also propose the sandpile load balancer latency minimization problem in which the goal is to minimize the latency of packets across the network of servers. I show the number of possible strategies for solving this problem and explore heuristic-based approaches via simulation.

## 1. INTRODUCTION

Bak, Tang, and Wiesenfeld introduced the abelian sandpile model in 1987. Their seminal paper was concerned with modeling the avalanches that occur in growing piles of sand and how the frequency of such avalanches relates to pink noise [1]. Dhar extended their work in 1990 by proving several mathematical properties of the abelian sandpile model [2]. One such property is self-organized criticality: the tendency of the system to converge on a critical state in which it is simultaneously stable and on the edge of chaos. Coincidentally, self-organized criticality has also been observed in a variety of dynamic systems in nature [1].

The notion of self-organized criticality conveniently applies to dynamic load balancing in which a network of servers distribute incoming traffic in order to “balance” the processing burden. This paper is concerned with the application of the abelian sand pile model to dynamic load balancing in order to exploit self-organized criticality for a real-world problem. The sandpile load balancer that results from this approach displays many of the properties of the abelian sandpile model.

To evaluate the sandpile load balancer as a dynamic load balancing system, I propose the latency minimization problem in which the goal is to minimize the latency of all packets throughout the network over the active period of the network. I show that discovering the best strategy for solving this problem is exceptionally difficult when considering the number of orderings of events in the sandpile load balancer model. Thus, heuristic-based strategies are considered and evaluated via simulation.

The rest of this paper is outlined as follows. Section 2 reviews the abelian sandpile model and its standard properties. Section 3 describes self-organized criticality and dynamic load balancing. Section 4 introduces the sandpile load balancer and proves several of its most notable characteristics. Section 5 motivates the latency minimization problem. Section 6 discusses several heuristic-based strategies for solving the latency minimization problem and presents results from sandpile load

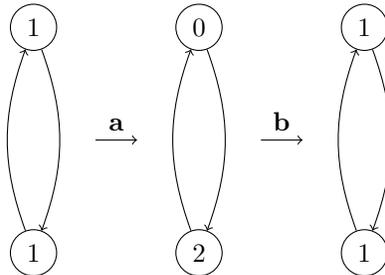


FIGURE 1. Each vertex in the graph above has  $d_i = 1$ . Thus, both vertices wish to topple in order to reach a stable configuration. **a** The upper vertex topples and sends a grain of sand to the lower vertex. **b** The lower vertex topples and sends a grain of sand back to the upper vertex. Note that the graph has not reached a stable configuration. In fact, it has returned to its original configuration.

balancer simulations. Section 7 presents related work and section 8 concludes with final remarks.

## 2. THE ABELIAN SANDPILE MODEL

The abelian sandpile model (ASM) is defined as follows.

**Definition 2.1.** The *abelian sandpile model* is a graph  $G$  with vertices  $V$  and directed edges  $E$  where  $|V| = N$  and vertices are labeled as  $v_i$  for  $i \in [1, N]$ . Each vertex  $v_i$  has associated with it a height  $h_i \in \mathbb{N}$  and an outdegree  $d_i \in \mathbb{N}^+$  where  $d_i = \text{Outdegree}(v_i)$ .

The height of each vertex is representative of the number of grains of sand at that vertex. Thus,  $G$  represents piles of sand of varying heights, explaining the name the “sandpile model.”

**Definition 2.2.** A *configuration* of  $G$  is a set of heights  $h \in \mathbb{N}^N$  for all vertices in  $G$ .

Any configuration of  $G$  has a notion of stability.

**Definition 2.3.** A configuration of  $G$  is *stable* if  $h_i < d_i$  for all  $i \in [1, N]$ .

There are two possible events on the graph  $G$ .

- (1) If a vertex  $v_i$  has  $h_i \geq d_i$  it may *topple*, updating  $h_{i,new} = h_{i,old} - d_i$  and  $h_{j,new} = h_{j,old} + 1$  for each of its neighboring vertices  $v_j$ . Note that  $|\text{Neighbors}(v_i)| = d_i$  by definition 2.1. In addition, note that a toppling event does not lead to additional grains of sand being created. It simply moves grains of sand from a vertex to its neighbors.
- (2) A grain of sand may be *added* to a vertex at random where vertex  $v_i$  has probability  $p_i$  of receiving the grain of sand. If vertex  $v_i$  receives the grain of sand it will update its  $h_{i,new} = h_{i,old} + 1$ . Note that  $\sum_{i=1}^N p_i = 1$ .

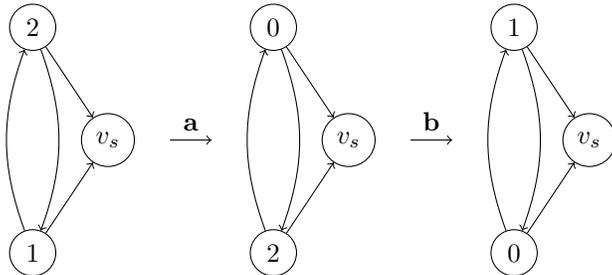


FIGURE 2. Each non-sink vertex in the graph above has  $d_i = 2$ . **a** The upper vertex has  $h_i = d_i$  and thus topples, passing one grain to the lower vertex and one to the global sink. **b** The lower vertex has  $h_i = d_i$  and thus topples, passing one grain to the upper vertex and one to the global sink. After this toppling the graph enters a stable state where  $\forall i, h_i < d_i$ .

In the construction above, grains of sand cannot leave the graph  $G$ . Then, as shown in Figure 1, there may exist configurations of  $G$  that never reach a stable state. To fix this, a vertex  $v_s$  referred to as the *global sink* of  $G$  may be added such that  $G' = G \cup \{v_s\}$ .

**Definition 2.4.** The *global sink* of  $G$  is a vertex  $v_s$  added to  $G$  such that every vertex  $v_i$  in  $G' = G \cup \{v_s\}$  has a directed path to  $v_s$ .

If a vertex topples a grain of sand to the global sink, the grain will be removed from the graph. This is demonstrated in Figure 2.

Definition 2.4 implies that the global sink  $v_s$  of a graph  $G$  must be unique. Note that if there exists two global sinks  $v_s$  and  $v_{s'}$ , then  $v_s$  must have a directed path to  $v_{s'}$ , but this contradicts the assertion that  $v_s$  is a global sink.

**2.1. Properties of the Abelian Sandpile Model.** There are three standard properties of the abelian sandpile model.

**Theorem 2.5** (Uniqueness of Stabilization [3]). *Let  $G$  be a directed graph and  $\sigma_0, \sigma_1, \dots, \sigma_n$  be a sequence of configurations produced by successive vertex topplings in  $G$ . Let  $\sigma'_0, \sigma'_1, \dots, \sigma'_m$  be another sequence with  $\sigma_0 = \sigma'_0$ .*

- (1) *If  $\sigma_n$  is stable then  $m \leq n$  and no vertex in  $\sigma'_1, \dots, \sigma'_m$  topples more times than in  $\sigma_1, \dots, \sigma_n$ .*
- (2) *If  $\sigma_n$  and  $\sigma'_m$  are both stable then  $m = n$ ,  $\sigma'_m = \sigma_n$ , and each vertex topples the same number of times in each sequence.*

*Proof.* Since (2) is a corollary of (1), it suffices to show (1). Suppose (1) is false by a counterexample with  $m + n$  minimal. Let  $v_i$  be the vertex that topples between  $\sigma_{i-1}$  and  $\sigma_i$ . Similarly, let  $v'_i$  be the vertex that topples between  $\sigma'_{i-1}$  and  $\sigma'_i$ . Vertex  $v'_1$  must topple in  $\sigma_1, \dots, \sigma_n$  because  $\sigma_n$  is stable. Suppose  $v_i = v'_1$ . Then,  $v_i, v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n$  produces the stable configuration  $\sigma_n$ . Note, however, that since the sequences  $v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n$  and  $v'_2, \dots, v'_m$  are a smaller counterexample for the theorem, minimality is contradicted.  $\square$

**Theorem 2.6** (Existence of Stabilization [3]). *If  $G$  is a directed graph with a global sink, then any configuration of  $G$  stabilizes.*

*Proof.* Let  $v_s$  be the global sink of  $G$ . Given a vertex  $v$ , let  $v_0, v_1, \dots, v_k, v_s$  be a directed path where  $v_0 = v$ . Every time  $v_k$  topples it passes a grain of sand to the sink. Thus,  $v_k$  can topple at most  $N$  times. Every time  $v_{k-1}$  topples it passes a grain of sand to  $v_k$ . It takes  $d_k$  grains of sand to topple  $v_k$ . Thus,  $v_{k-1}$  topples at most  $d_k N$  to pass all grains of sand to  $v_s$  through  $v_k$ . Then,  $v$  topples at most  $d_1 \cdots d_k N$ . Therefore, each vertex topples a finite number of times and by definition 2.3,  $G$  must eventually reach a stable configuration.  $\square$

Define the **sand addition operator**  $S_v$  to be the mapping of the sand pile configurations that occur from adding a grain of sand to vertex  $v$  and then stabilizing the system. Symbolically, we may define  $S_v$  on a configuration  $\sigma$  as  $S_v\sigma = (\sigma + 1_v)^*$  where  $1_v$  refers to adding one grain of sand to vertex  $v$  and  $*$  refers to stabilizing the given configuration [3].

**Theorem 2.7** (Abelian Property [3]). *If  $G$  is a directed graph with a global sink, then the sand addition operator commutes.*

*Proof.* Suppose  $G$  has configuration  $\sigma$  and two vertices:  $v$  and  $w$ . A new configuration,  $\sigma'$  is found by adding a grain of sand to each vertex, i.e.  $\sigma' = \sigma + 1_v + 1_w$ . After stabilizing  $v$  the new configuration is  $S_v\sigma + 1_w$ . After stabilizing  $w$  the stable configuration is  $S_w S_v\sigma$ . Note  $\sigma'^* = S_w S_v\sigma$ . In addition, swapping the order of sand addition also produces a stable state  $S_v S_w\sigma$ . By theorem 2.5, these stable configurations must be identical, i.e.  $S_w S_v\sigma = S_v S_w\sigma$ .  $\square$

### 3. SELF-ORGANIZED CRITICALITY AND DYNAMIC LOAD BALANCING

The ASM exemplifies a natural phenomenon known as self-organized criticality (SOC). SOC is demonstrated by dynamic systems in nature that tend towards a critical state. This critical state is realized by the application of small rules (e.g. vertex toppling) across a system that leads to global patterns (e.g. stability) [4].

The notion of SOC conveniently applies to dynamic load balancing in which a network of servers distribute incoming traffic in order to “balance” the processing burden. There are three layers to dynamic load balancing.

- **Packet:** A packet is a processing request made by a client. Each packet has associated with it a processing time,  $p_{time} \in \mathbb{N}^+$ , representing the number of time steps required to completely process the packet.
- **Server:** A server accepts packets for processing. A server may process at most one packet at a time and is deemed *busy* if it is currently processing a packet. If a busy server receives a packet, it will place the packet on its queue (“first-in-first-out” data structure) to process in the future. If a server is *free* at the start of a time step, it will remove a packet from the queue for processing or do nothing if the queue is empty.
- **Network:** A network is a topology of servers connected via bidirectional communication channels. Servers may pass packets along these edges for other servers to process. The activity of a network is discretized into time steps.

In light of the connection between dynamic load balancing and SOC, it is only natural to apply ASM to dynamically load balance packet traffic across a network of servers.

#### 4. THE SANDPILE LOAD BALANCER

The **sandpile load balancer** (SLB) is a graph  $G$  of  $N$  vertices where each vertex  $v_i$  represents a server  $s_i$  and each edge is a bidirectional communication channel between the two participating servers. Each server  $s_i$  has a queue  $q_i$  that may store incoming packets for processing. In addition, each server has a capacity  $c_i$  equal to the number of servers it is connected to. There are two events that may occur in the SLB's network of servers.

- (1) A packet may be added to one server's queue  $q_i$  at random with probability  $p_i$ . Then,  $|q_{i,new}| = |q_{i,old}| + 1$ . Note that  $\sum_{i=1}^N p_i = 1$ .
- (2) If the size of a server's queue is greater than or equal to the server's capacity, i.e.  $|q_i| \geq c_i$ , the server may topple and pass one packet to each of the servers it is connected to. Then,  $|q_{i,new}| = |q_{i,old}| - c_i$  and  $|q_{j,new}| = |q_{j,old}| + 1$  for each server  $s_j$  it is connected to.

Every time step a packet is added to the network. In addition, every time step each server  $s_i$  may topple if it is *overloaded* (i.e.  $|q_i| \geq c_i$ ). Each server is only permitted to topple at most once per time step. Note that not all servers will topple each time step (e.g. a server may remain stable throughout the activity of the timestep). An example of the SLB is shown in Figure 3.

The notion of a stable configuration of an SLB graph  $G$  is slightly different than that of the ASM.

**Definition 4.1.** A configuration of an SLB graph  $G$  is *stable* if there are no packets in  $G$ , i.e.  $\forall i, |q_i| = 0$  and  $State(s_i) = free$ . A stable configuration is denoted as  $\sigma^*$ .

The analogue of a stable configuration of an ASM graph  $G$  is a *calm* configuration of an SLB graph  $G$ .

**Definition 4.2.** A configuration of an SLB graph  $G$  is *calm* if  $\forall i, |q_i| < c_i$ .

**4.1. Properties of the Sandpile Load Balancer.** The standard properties proved in §2.1 for the ASM may also be proved for the SLB. These properties are proved for a *static* configuration of the SLB graph  $G$  in which no packets are being added.

**Theorem 4.3** (Existence of Stabilization). *Given any static configuration of  $G$  with a finite number of packets, if all packets in  $G$  have finite processing times then  $G$  stabilizes after a finite number of time steps.*

*Proof.* Without loss of generality, suppose each packet in  $G$  has processing time  $p_{time}$ . Because servers remain busy if there are packets in their queue, every  $p_{time}$  time steps at least one packet will be processed and disappear from the network. Then, the number of packets in  $G$  is strictly decreasing every  $p_{time}$  time steps and thus after a finite number of time steps all packets must be processed and  $G$  must reach a stable configuration.  $\square$

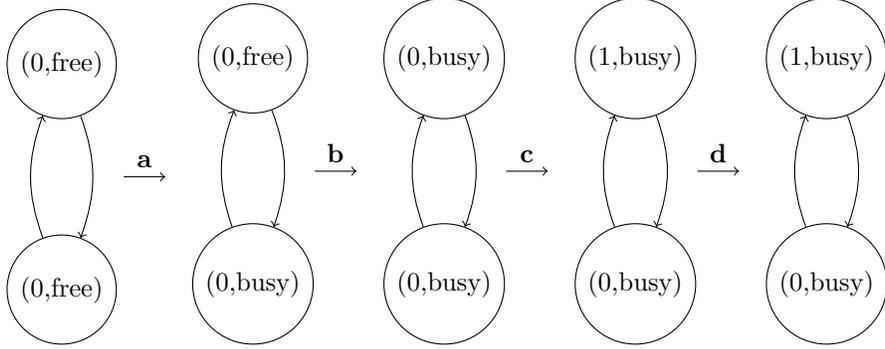


FIGURE 3. Each vertex in the graph above has  $c_i = 1$ . The tuple represents the number of packets in the server's queue and whether the server is busy or not, respectively. To start, each server has a queue size of zero and is free. In addition, all packets received by the network have  $p_{time} = 3$  and each transition represents the passing of one time step. **a** The lower server receives a packet and begins to process it, entering the busy state. **b** The upper server receives a packet and begins to process it, entering the busy state. **c** The upper server receives a packet and topples it to the lower server. The lower server, however, reaches capacity with the addition of that packet and topples it back to the upper server. **d** The lower server completes the processing of its initial packet and accepts a packet from the network. The upper server attempts to topple the packet in its queue to the lower server, but this packet is toppled back to the upper server where it is stored in its queue for later processing.

An immediate corollary of this (also implied by definition 4.1) is the uniqueness of an SLB graph  $G$ 's stabilization.

**Corollary 4.4** (Uniqueness of Stabilization). *Given a graph  $G$  with starting configuration  $\sigma$ , the stabilization of  $G$  is independent of the toppling order of servers.*

In addition, note that the **packet addition operator**  $P_s$  must commute because for any configuration  $\sigma$  and server  $s$ ,  $\sigma + 1_s = P_s \sigma = \sigma^*$  where  $\sigma^*$  is a configuration with zero packets.

## 5. THE SANDPILE LOAD BALANCER LATENCY MINIMIZATION PROBLEM

The underlying goal of dynamic load balancing is to minimize the time a client waits for its processing request to be completed. To measure this value and prioritize traffic, the network assigns a *latency* to each incoming packet and updates it throughout the packet's lifespan in the network.

**Definition 5.1.** The *latency* of a packet is the difference between the current time step and the time step at which it was added to the network. We use  $L(pkt_i)$  to denote the latency of a packet  $pkt_i$ .

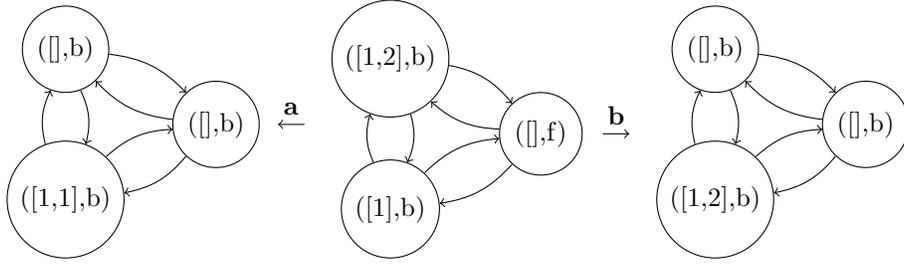


FIGURE 4. Refer to the middle configuration as  $\sigma_0$ , the configuration to its left as  $\sigma_a$ , and the configuration to its right as  $\sigma_b$ . The tuple represents the queue where elements are retrieved from the left and pushed on from the right as well as whether the server is busy (b) or free (f), respectively. The numbers in the queue represent the latencies of the packets in the queue. Note that the top-left vertex of  $\sigma_0$  is overloaded and must topple the packets in its queue. **a**  $\sigma_a$  is achieved if the packet with latency 1 is sent to the busy server and the packet with latency 2 is sent to the free server. **b** In contrast,  $\sigma_b$  is achieved if the packet with latency 2 is sent to the busy server and the packet with latency 1 is sent to the free server. If the goal is to minimize the latency of the packet with latency 2, it is clear that  $\sigma_a$  is a better option than  $\sigma_b$ . Thus, the manner in which a server distributes packets may affect the latencies of those packets.

Suppose a server wishes to minimize the latency of a given packet  $pkt_i$  in a given network configuration. Two natural questions arise.

- (1) Does the method by which a server distributes packets to its neighboring servers, or “distribution strategy,” affect the final latency of  $pkt_i$ ?
- (2) Does the order in which the servers topple, or “toppling strategy,” affect the final latency of  $pkt_i$ ?

As seen in Figures 4 and 5, the distribution method and toppling order affects the latency of a packet. Thus, the **SLB latency minimization problem** asks what strategy (i.e. the distribution and toppling strategies) minimizes the sum of the latency of all packets in  $G$  over all time steps where each packet has processing time  $p_{time}$ . In other words, the function to be minimized is

$$\sum_{t=t_0}^{t_f} \sum_{pkt_i \in G} L(pkt_i)$$

where  $t_0$  is the first time step in the network and  $t_f$  is the last.

Note that finding the best strategy is no trivial task. The set of  $N$  servers may be toppled in  $N!$  ways. In addition, each server  $s_i$  may distribute packets to its neighboring servers in  $c_i!$  ways. Thus, there are  $N! \cdot (c_i!)^N$  possible strategies which is exponential in the number of servers.

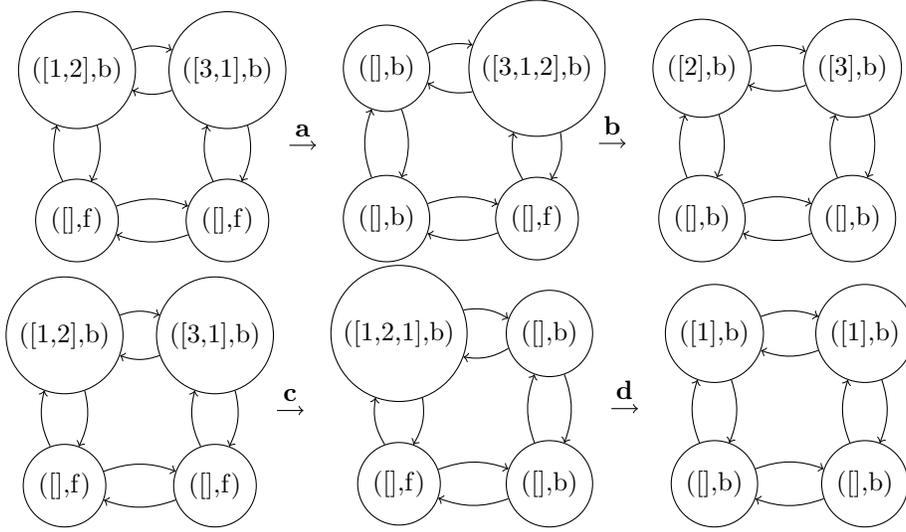


FIGURE 5. Refer to the left-most configurations (which are equivalent) as  $\sigma_0$ . The packets in this network have  $p_{time} = 10$ . The tuple represents the queue where elements are retrieved from the left and pushed on from the right as well as whether the server is busy (b) or free (f), respectively. The numbers in the queue represent the latencies of the packets in the queue. Note that the top-left and top-right vertices of  $\sigma_0$  are overloaded and must topple the packets in their respective queues. The goal is to select the ordering of server topplings that minimizes the latency of the packet with an accumulated latency of 3. There are two possible orderings. **a** The top-left server topples. **b** The top-right server topples and the packet with latency 3 remains where it was, waiting for the current packet being processed to complete. **c** The top-right server topples and the packet with latency 3 is sent to the bottom-right server where its processing starts. **d** The top-left server topples. Note that the latency of the packet with latency 3 is minimized by the lower sequence of orderings as opposed to the top. Thus, the order of topplings may affect the latencies of packets.

## 6. EMPIRICAL RESULTS

In light of the number of possible strategies to consider for the SLB latency minimization problem, a heuristic-based approach is likely to be more practical. Such heuristic-based strategies may be broken into distribution and toppling strategies.

To test the performance of such strategies, an SLB simulator was implemented in the Python programming language. The simulator tests strategies on a torus-mesh topology (see appendix A for details), varying both the circumference of the torus and the processing time of incoming packets. The simulation code may be found at <https://github.com/berjc/sandpile-load-balancer>.

**6.1. Packet Distribution Strategies.** Three packet distribution strategies were considered.

- (1) **Blind:** A server distributes its packets at random to its neighbors.
- (2) **Maximum:** A server distributes the packets with the highest latency to its least busy neighbors.
- (3) **Minimum:** A server distributes the packets with the lowest latency to its least busy neighbors.

**6.2. Toppling Order Strategies.** Two toppling order strategies were considered.

- (1) **Blind:** Servers are toppled at random.
- (2) **Maximum:** Servers are toppled in decreasing order of their respective queue sizes.

Each strategy was tested over 1000 trials with the same schedule of packet additions for varying process times and a torus circumference of 2 and 5. Then, the number of times each strategy performed the best as compared to the other strategies was computed. These “best-strategy” frequencies are plotted in Figure 6 against varying process times. For a torus-mesh topology with a circumference of 2, the combination of the maximum packet distribution strategy and the blind toppling strategy outperforms all other strategies for processing times past  $p_{time} = 3$ . For small  $p_{time}$  values a combination of the maximum packet distribution strategy and maximum toppling strategy outperforms all other strategies. In contrast, for a torus-mesh topology with a circumference of 5, the combination of the maximum packet distribution strategy and the blind toppling strategy does not dominate for all process times. It does appear, however, that it is the best strategy for higher processing times.

These results can be understood by revisiting Figure 4 where it is clear that the maximum packet distribution strategy produces the best strategy for minimizing the latency. It also appears that the best strategy is a function of both the topology and the processing time of packets.

## 7. RELATED WORKS

There are few works in the research community that apply the ASM to dynamic load balancing. Laredo *et al.* demonstrate a sandpile scheduler that uses the ASM to schedule incoming Bag-of-Tasks across a network of agents [4]. Their design is similar to that proposed in this paper, but differs in three distinct ways. First, they wish to maximize the performance of the computing resources of the given agents whereas this paper is concerned with latency minimization. In addition, their scheduler assumes the ability to gossip with neighboring agents to determine toppling strategies. Finally, their toppling rule is a function of an agent’s neighbors’ queue sizes, a variation from the rule proposed in this paper. Nevertheless, Laredo *et al.*’s sandpile scheduler has demonstrated impressive results compared to modern

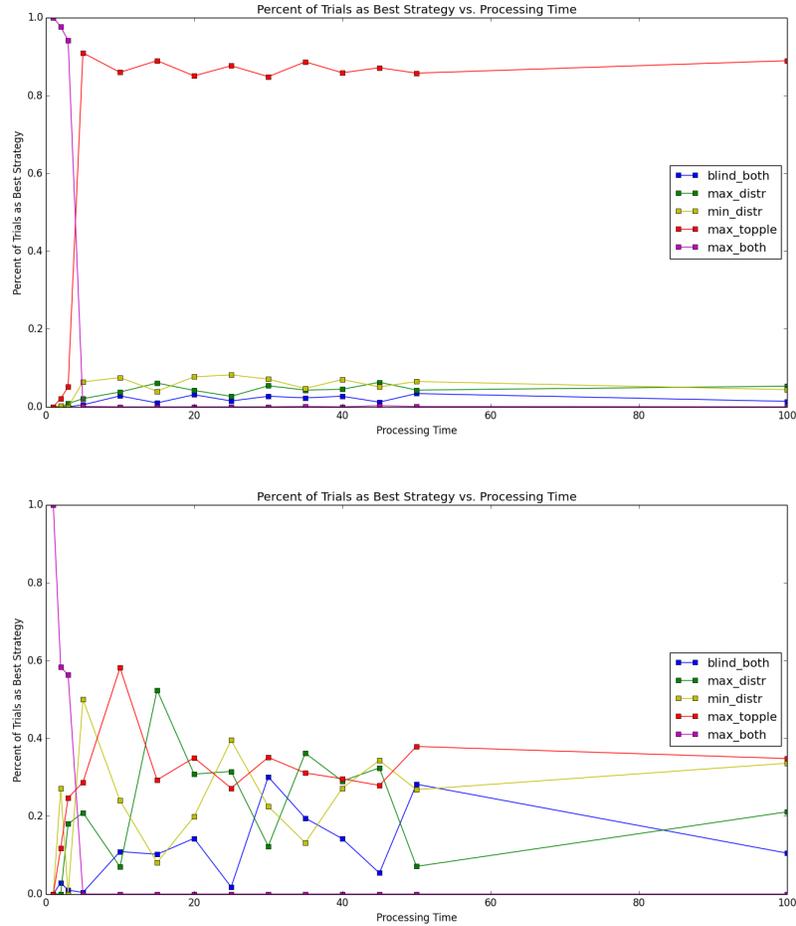


FIGURE 6. Five strategies were considered. `blind_both` uses the blind distribution and blind topple strategies. `max_distr` uses the maximum distribution and blind topple strategies. `min_distr` uses the minimum distribution and blind topple strategies. `max_topple` uses the blind distribution and maximum topple strategies. Finally, `max_both` uses the maximum distribution and maximum topple strategies. Each strategy was run over 1000 trials where each trial considered the same packet addition sequences. Then, strategies were compared for each trial and a winner was determined based on whether it minimized the latency as described in §5. The percentage of trials “won” by each strategy was then plotted for various processing times. The top graph displays the results for a torus-mesh topology with a circumference of 2 and the bottom graph displays the results for a torus-mesh topology with a circumference of 5.

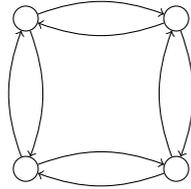


FIGURE 7. A torus-mesh topology with a circumference of 2.

dynamic load balancing techniques and underscores the importance of considering the application of SOC to such problems.

#### 8. FINAL REMARKS

It is clear from §6 that simple heuristic-based strategies may outperform blind strategies in which both packets and servers are chosen at random. Empowered with the notion of self-organized criticality, the sandpile load balancer's next logical challenge would be to pair it against existing dynamic load balancing algorithms and gauge its performance.

#### ACKNOWLEDGMENTS

I would like to thank Professor Alexander Postnikov of the Massachusetts Institute of Technology Department of Mathematics for introducing me to the abelian sandpile model and stimulating my interest in the field.

#### APPENDIX

**Appendix A: Torus-Mesh Topology.** The torus-mesh topology is a grid of servers in which the vertices at the edges of the grid are connected to the vertices at the opposing, parallel edge. Figure 7 demonstrates a torus-mesh topology with a circumference of two.

#### REFERENCES

- [1] Per Bak, T. C., and Kurt Wiesenfeld. "Self-Organized Criticality: An Explanation of  $1/f$  Noise." *Physical Review Letters* 59 (1987): 381-384.
- [2] Dhar, Deepak. "Self-Organized Critical State of Sandpile Automaton Models." *Physical Review Letters* 64.14 (1990): 1613.
- [3] Holroyd, Alexander E., *et al.* "Chip-firing and rotor-routing on directed graphs." *In and Out of Equilibrium 2*. Birkhäuser Basel, 2008. 331-364.
- [4] Laredo, Juan Luis Jiménez, *et al.* "The Sandpile Scheduler." *Cluster Computing* 17.2 (2014): 191-204.