The sandpile scheduler

Juan Luis Jiménez Laredo, Pascal Bouvry, Frédéric Guinand, Bernabé Dorronsoro, Fernandes Carlos

Noname manuscript No.
(will be inserted by the editor)

The Sandpile Scheduler

How self-organized criticality may lead to dynamic load-balancing

J.L.J Laredo · P. Bouvry · F. Guinand · B. Dorronsoro · C. Fernandes

Received: date / Accepted: date

Abstract This paper studies a self-organized criticality (SOC) model called sandpile for dynamically load-balancing tasks arriving in the form of Bag-of-Tasks (BoTs) in large-scale decentralized system. The sandpile is designed as a decentralized agent system characterizing a cellular automaton, which works in a critical state at the edge of chaos. Depending on the state of the cellular automaton, rather different responses may occur when a new task is assigned to a resource: it may change nothing or generate avalanches that reconfigure the state of the system. The abundance of such avalanches is in power-law relation with their sizes, a scale-invariant behavior that emerges without requiring tuning or control parameters. That means that large – catastrophic – avalanches are very rare but small ones occur very often. Such an emergent pattern can be efficiently adapted for non-clairvoyant scheduling, where tasks are load balanced into computing resources trying to maximize the performance but without assuming any knowledge on the tasks features. In order to validate the algorithm design, we conduct an empirical experimentation showing that the sandpile is able to find near-optimal schedules by reacting differently to different conditions of workloads and architectures.

Keywords optimization · self-organization · scheduling · distributed systems

J.L.J. Laredo and P. Bouvry
FSTC-CSC/SnT, University of Luxembourg, Luxembourg.
E-mail: {juan.jimenez,pascal.bouvry}@uni.lu

F. Guinand
LITIS lab, Normandy University, France.
E-mail: Frederic.Guinand@univ-lehavre.fr

B. Dorronsoro
Laboratoire dInformatique Fondamentale de Lille, University of Lille, France.
E-mail: bernabe.dorronsoro_diaz@inria.fr

C. Fernandes
Laseeb, Technical University of Lisbon, Portugal.
E-mail: cfernandes@laseeb.org

1 Introduction

Complexity sciences have not yet devised a mathematical language that describes the origins and dynamics of self-organization, which we may define here as global patterns emerging from local rules. Nevertheless, some researchers have embarked in this task. Self-Organized Criticality (SOC), proposed by Bak, Tang and Wiesenfeld (BTW) [3] is one of those attempts to present a general theory of self-organization. SOC describes a property of complex systems that consists of a critical state formed by self-organization at the border of order and chaos. One of the characteristics of SOC is that small disturbances can lead to the so-called avalanches, i.e., events that are spread spatially or temporally through the system. Such events occur independently of the initial state; moreover, the same perturbation may lead to small or large avalanches, which show a power-law proportion between their size and quantity. This means that large (catastrophic) events may hit and reconfigure the system from time to time.

With these issues in mind, we presented in [17] a non-clairvoyant scheduler based on a SOC model called sandpile [2]. The sandpile is a cellular automaton which models the process of dropping grains of sand on a surface and the sliding of grains due to the increasing gradient of the slope. Therefore, the system has a global state that can be altered when new grains of sand are dropped in. The number of grains in a given cell or "site" are represented by a height function. That way, sand accumulates in a lattice of cells until the height difference of adjacent neighbors exceeds a given threshold. Such an event starts an avalanche leading to a new state of equilibrium in which the sand is balanced throughout the system.

Our hypothesis is that such a smart and emergent behavior fits well with the idea of dynamic load-balance in large-scale infrastructures, where tasks are assumed to be "grains" and the load-balance process results in a new allocation of tasks in computing resources. We hypothesize that a distributed agent system behaving as a sandpile automaton will display organic properties such as self-organization of tasks in resources. Given that the SOC phenomenon is invariant to the scale and robust to varying dropping rates, our approach was shown in [17] to behave consistently in architectures of different sizes and speeds. This paper extends from previous work and presents a detailed description of the algorithm. Furthermore, it explores different design settings for optimal operation, such as the influence of the topology on the performance or how a gossiping mechanism can minimize the overheads caused by the migration of tasks.

Given the novelty of our proposal, we will constrain our initial objective to demonstrate the viability of the approach in terms of reducing the makespan for workloads arriving in the form of Bags-of-Tasks (BoTs) [4] where groups of sequential jobs are submitted in single batches. Iosup et al. [14] have identified this type of workloads to involve a large proportion of the tasks submitted to large-scale computing environments. Therefore, BoTs represent an adequate

framework for assessing the performance of the sandpile scheduler in these preliminary steps of design.

The rest of the paper is organized as follows. Section 2 provides a summary on some of the most relevant works in self-organized dynamic load-balancing. Section 3 characterizes the BoTs problem that will be used to generate workloads for benchmarking our proposal. Section 4 describes our extended version of the BTW sandpile model, which consists in a decentralized agent system. For the sake of an efficient design, section 5.1 explores different topologies for interconnecting the agent system in addition to a gossip-based version of the cellular automaton. Section 6 analyzes the main properties of the sandpile scheduler such as performance independence, scale-invariance or flexibility to heterogeneous conditions. Finally, some conclusions and future lines of work are presented in section 7.

2 Related Work

In scheduling, dynamic load-balancing tackles the problem of assigning tasks to resources when workloads are unpredictable and change at runtime [7]. Classic methods, such as gradient or diffusion based models [21], are being extended and reinforced in order to tackle the particularities of new large-scale computing systems, such as the loosely-coupled heterogeneous architectures in Grid systems or the scale-on-demand requirements of Cloud computing. This section summarizes some of the current approaches in dynamic load-balancing research.

Jelasity et al. [16] propose a skeleton for dynamic load-balancing through gossiping; rather than a fully-operative scheduling system, the authors aim at illustrating the application potentials of gossiping protocols. They describe the problem as an equivalent to the problem of averaging a set of distributed numbers using decentralized *aggregation*. Despite simple, this work constitutes one of the basis for many of the approaches in current research.

Franceschelli, Giua and Seatzu [9] describe a distributed load-balancing algorithm based on the consensus between nodes. To reach the consensus –which is described as a heuristic assuming knowledge on the weights of tasks–, nodes communicate within a homogeneous architecture via gossiping. This work was lately extended to heterogeneous architectures in [10], where the authors analyze the convergence time. The study assumes that all tasks are initially in the system. Therefore, the problem of the convergence is simplified into a static optimization problem instead of a dynamic (a.k.a. *non-stationary*) problem.

Hu and Klefstad [13] propose a dynamic load-balancing approach which is inspired by the dynamics of liquids. The approach has a strong connection with the one presented here. Both are based on self-organization and both try to mimic the way gravity has to flatten different elements. However, there is a key difference concerning the critical state (i.e. the "C" in SOC). While our approach behaves at the border of chaos, the liquid-based approach only

recognizes two states: either the system is in equilibrium, which means that the workload is perfectly-balanced, or the system is in a non-equilibrium state. No matter how small a perturbation is, it will always lead to a non-equilibrium state and the consequent reallocation of tasks.

Fouad Semaan [18] proposes to study the behavior of parallel applications to the light of SOC. In his work, processors are organized as a torus and each processor, according to its own characteristics, can process, in a concurrent way, a limited number of processes. Once the limit is exceeded, the processor becomes overloaded and this triggers a load-balancing mechanism that consists in transferring tasks to other processors, which behave in the same way, leading to the well-known phenomenon of avalanches. He shows that there exist three different kinds of behavior for the whole system: underloaded, overloaded and critical, and these behaviors are policy-independent. The best efficiency being reached for critical values. Some real experiments reported in the thesis confirm the power law distribution of avalanches. The study of this critical point has been more deeply investigated in a recent research [12]. In this work, from an energy-saving perspective, the authors propose to dynamically adjust the number of computing devices of a HPC environment thanks to a mechanism able to predict the coming of an overload. The considered environment is a 2-dimensional grid, and the mechanism for predicting the overload relies on the frequency and the size of avalanches during a load-balancing phase.

Our current proposal is inspired by the emergent SOC behavior displayed by the BTW sandpile model [3, 2], that we have extended for a best fit with the specific features of the dynamic load-balancing problem. We have firstly modified the so-called transition rule, which is the policy in the sandpile for triggering avalanches any time the height difference between two piles surpasses a given threshold. Instead, the transition rule in our proposal is only triggered if the height of a given pile is larger than the accumulated heights of two neighbor resources. We find that this modification provides two main advantages: on the one hand, there is no parameter to tune since there is no threshold. On the other hand, the transition rule automatically adjusts to different levels of the workload. For instance, being all resources overloaded, it is not so likely that a pile surpasses the addition of two neighbor piles. A second extension to the canonical model is to consider a small-world connection topology instead of the commonly employed regular lattices. We show that the algorithm can yield near-optimal performances using this type of topology. Finally, the third main extension is inherited from peer-to-peer gossiping protocols [15]. Instead of real avalanches where tasks suffer multiple hops, the gossip-based version implements avalanches *virtually*, i.e., nodes negotiate the state of equilibrium before the real transfer of tasks takes place.

3 Bag-of-tasks Scheduling Problem

In Computer Architecture, scheduling is the problem of assigning tasks to resources as to minimize the overall execution time. Despite simple in its formu-

lation, scheduling has been proven a high-dimensional NP-hard problem with time constraints [11], which may involve few hundreds of tasks and resources even for medium size instances. Since the process of optimization itself sums up to the final schedule, an efficient scheduler must minimize its own computational efforts while searching for best solutions. The scheduler that we propose in this paper is a reactive scheduling system, meaning that it does not interfere with the actual assignation of tasks unless the workload is detected to be in a non-equilibrium state. Such a strong dependence scheduler/workload makes of a good characterization of the workload a key to underpin the analysis and design of an efficient scheduling strategy. In order to conduct our in silico experimentation, we have selected a workload arriving in the form of BoTs.

BoTs are a particularly challenging type of workload and have a great impact on modern large-scale distributed systems: Iosup et al. [14] present some evidences on the predominance of this form of workload in grid and cloud computing systems. Additionally, Casanova et al. [4] warn on the difficulties for doing estimates on the execution times of such workloads and encourage the use of non-clairvoyant scheduling, i.e., not assuming any precondition on tasks duration or arrival patterns, which may lead to scheduling errors in real systems.

In BoTs, tasks arrive in bursts called bags, which are tipically a set of multiple instances of the same sequential program. A bag is, therefore, composed by a group of independent tasks that can be executed in parallel. This section aims at providing a general definition of the BoTs scheduling problem, taking into account all the basic features but allowing an easy parametrization at the same time. The idea is to be able to create different workloads and architectures in order to explore different aspects of the sandpile scheduler. The main components for the formulation of the BoTs scheduling problem are:

- A set Q of q heterogeneous processors.
- A set B of b BoTs of size k-tasks.
- W is a $v \times q$ computation cost matrix, with v being the total number of tasks.
- C is a $q \times q$ communication cost matrix.

3.1 Computing Architecture

A computing architecture is a set of interconnected processors able to process and transfer tasks with a given speed. A computing architecture can be divided into the computation and communication subsystems:

- *Computation subsystem:* Let's denote p_i as the speed of the i-th processor in an architecture of q processors ($i \in [1 \dots q]$). Given a processor of reference $p_{ref} = 1 \frac{instruction}{cycle}$, the speed of p_i can be calculated by:

$$p_i = p_{ref} \times \rho_i \tag{1}$$

  where $\rho_i \in \mathbb{N}$ is a speeding factor.

– *Communication subsystem:* We denote C as a transfer rate matrix of size $q \times q$, where $C_{i,j}$ is defined with respect to a reference network-link $C_{ref} = 1\frac{data_{transfer}}{cycle}$ as:

$$C_{i,j} = C_{ref} \times \tau_{i,j} \tag{2}$$

where $\tau_{i,j} \in \mathbb{N}$ is the network-link speeding factor.

3.2 BoTs Workload

A BoTs-based workload is composed of a set of b BoTs. Every BoTs, $b_j \in B$, is a set of tasks such that $|b_j| = k$. The total number of tasks in the workload is therefore $v = \sum_{b_j \in B} |b_j|$, where $\forall b_j \in B$:

– $n_{j,h}$ is the length in terms of computing instructions of the h^{th} task in b_j.
– $d_{j,h}$ is the code size of the program representing this task.
– $a_{j,h}$ is the arrival time; we assume that all tasks within the same BoTs arrive in a single batch: $\forall a_{j,h}, a_{j,r} \in b_j : a_{j,h} \equiv a_{j,r}$.

Alternatively, every value in n, d or a can be accessed by the absolute index $i \in [1 \dots v]$ where e.g. n_i is equivalent to $n_{j,h}$ by substituting $i = j \times k + h$.

3.3 General problem definition

Being W a $v \times q$ computation time matrix where $w_{i,j} = \frac{n_i}{p_j}$ is the time it takes to the n_i task to be evaluated in processor p_j and, $c_{i,h,k} = \frac{d_i}{C_{h,k}}$ the communication time for transferring a task source-code d_i from processor p_h to processor p_k. The objective function is to find the schedule that minimizes the processing time of the tasks in a given architecture, i.e. minimize the *makespan* which is given by:

$$makespan = \max_{\forall p_j \in Q} T_t \tag{3}$$

where T_t is the time that processor p_j takes to process all its assigned t tasks:

$$T_t = \sum_{\forall w_{i,j} \in FIFO(p_j)} \max(T_{t-1} + w_{i,j}, TR(n_i, p_j) + w_{i,j}) \tag{4}$$

$FIFO(p_j)$ is a *First-in First-out* function which provides a list with all tasks assigned to processor p_j in order of arrival and $TR(n_i, p_j)$ stands for the *Tasks Ready* function which takes into account the migrations that a given task has suffered in the system. Therefore, if a task n_i remains in the initially assigned processor p_j, the task ready time is the arrival time a_i:

$$TR(n_i, p_j) = a_i \tag{5}$$

otherwise TR can be defined recursively as:

$$TR(n_i, p_j) = c_{i,prev(j),j} + TR(n_i, p_{prev(j)}) \tag{6}$$

where $prev(j)$ refers to the previous processor where the task n_i was scheduled, and $c_{i,prev(j),j}$ to the cost of transferring the source-code d_i of the task from processor $p_{prev(j)}$ to p_j.

3.4 Additional metrics

In addition to minimize the makespan, we also consider the throughput and flowtime metrics. The throughput is related with the utilization of computing resources and its average can be expresed by:

$$\overline{throughput} = \frac{v}{makespan} \tag{7}$$

On the other hand, the flowtime provides the time a task remains queued in the system and, therefore, is related to users satisfaction.

$$\overline{flowtime} = \frac{\sum_{\forall w_{i,j} \in FIFO(p_j)} T_t}{v} \tag{8}$$

4 Model description

The SOC phenomenon was firstly identified by Bak et al. [3] in a model called sandpile, a cellular automaton in which grains of "sand" are dropped randomly on a lattice and accumulate with a height function. In its original form, when the height difference between two adjacent sites exceeds a threshold value, the grains in the site with higher height topple to adjacent sites. Avalanches of all sizes may occur, from a single tumble to events that reconfigure the entire pile. Without any fine-tuning of parameters, the system evolves to a non-equilibrium critical state in which the frequency of an avalanche is in power-law proportion to its size. This paper adapts such a behavior to the problem of dynamically load-balancing workloads in parallel computing systems.

To define a scheduler based on the sandpile model, we build an agent system in which the agents render the cells of a cellular automaton. The main components of the architecture are, therefore, the agents and the topology for interconnecting them as depicted in figure 1. Every agent denotes itself as α and handles the workloads of a computing resource, deciding whether to migrate tasks to adjacent/neighboring β resources. Either α or β sites represent the same class of an agent and the difference only relies on the nomenclature: α when referring to a particular resource and β to refer to the neighbors of such a resource. Therefore, the agent system is symmetric and decentralized. For the system to acquire the sandpile dynamics, we assume that "grains" and "tasks" are interchangeable terms. The tasks accumulate in the different

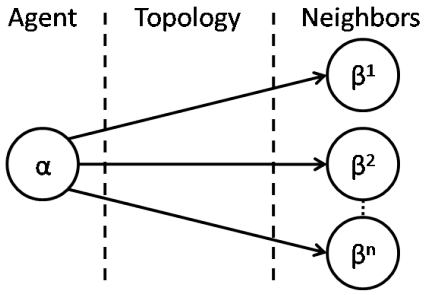


Fig. 1 Sketch of the agent system in the sandpile scheduler.

computing resources with a height function $h(\alpha)$, which represents the number of tasks queued in the processor. The system is in equilibrium as long as the aggregated workloads of two adjacent sites[1] to α do not exceed its own workload, i.e., $h(\alpha) \leq h(\beta^1) + h(\beta^2)$. If it happens that $h(\alpha) > h(\beta^1) + h(\beta^2)$, grains topple from the highest pile $h(\alpha)$ to the lower ones, $h(\beta^1)$ and $h(\beta^2)$, updating the sandpile and reaching a new state. When the process involves more resources, avalanches propagate until all resources meet the condition: $\forall \alpha : h(\alpha) \leq h(\beta^1) + h(\beta^2)$, i.e. the sandpile has converged to a new state of equilibrium. Figure 2 depicts the process involving three resources α, β^1 and β^2.

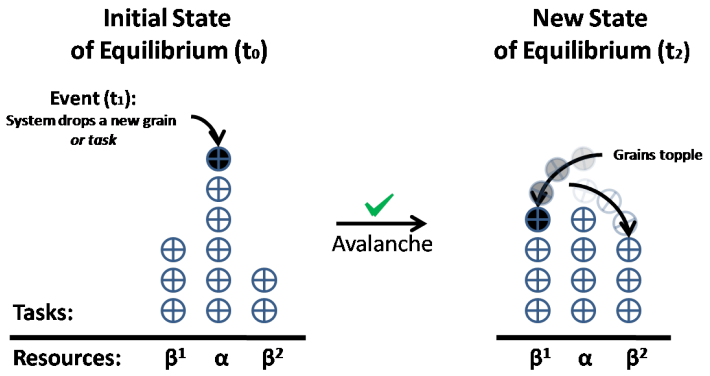


Fig. 2 Example of the sandpile running in resources α, β^1 and β^2. The update policy acts as follows: at t_0 the sandpile is in a state of equilibrium since $h(\beta^1) + h(\beta^2) \geq h(\alpha)$. At t_1 a new grain/task arrives in α breaking the inequality and leading to an avalanche. Resource α computes the overall number of tasks in the neighborhood, calculates the average $\frac{h(\alpha)+h(\beta^1)+h(\beta^2)}{3}$ and balance the workload among the neighbors.

[1] These two sites, β^1 and β^2, should be selected according to some criterion out of all neighbors of α. For the sake of simplicity, this study assumes that the neighbors are selected uniformly at random.

4.1 Set of functions

In order to construct the cellular automaton[2], every computing node must implement a double-ended queue with the functions in table 1.

Function	Description	Pre-/Postconditions
PUSH(l)	Insert l tasks at the back	l is a list of tasks
POP(n)	Removes n tasks at the back	Returns a list with the n removed tasks
H	Counts the no. of tasks in the queue	Returns the size/height of the queue
SELECT	Select two adjacent queues	Returns a list with the two queues
SHIFT	Removes a task at the front	Returns the removed element

Table 1 Main functions of the double-ended sandpile queue

This set is sufficient to implement the basic functionality of the sandpile. The *push* and *pop* functions act at the back of the queue so that tasks migrations can be implemented using both. The h function monitors the state of the queue and *select* provides access to adjacent resources. Finally, the *shift* function acts at the front of the queue and is the only way for the processor to retrieve tasks; this condition assumes a non-preemptive operational mode for the sake of simplicity.

4.2 Algorithmic description of a sandpile agent

Using previous set of functions, a sandpile agent can be easily built up according to algorithm 1, that we will refer to from now on as *basic case*. By every computing node running an agent, the emergent behavior of the system is expected to display SOC properties and act as a decentralized sandpile scheduler.

Algorithm 1 Pseudo-code of a sandpile agent in resource α

loop
  WAIT
  $[\beta^1, \beta^2] \leftarrow \alpha.SELECT$
  $x = \frac{\alpha.H + \beta^1.H + \beta^2.H}{3}$
  if $\alpha.H > \beta^1.H + \beta^2.H$ **then**
    $\alpha_{\beta^1} = \lfloor x \rfloor - \beta^1.H$
    $\alpha_{\beta^2} = (\alpha.H - \lceil x \rceil) - \alpha_{\beta^1}$
    $\beta^1.PUSH(\ \alpha.POP(\alpha_{\beta^1})\ )$
    $\beta^2.PUSH(\ \alpha.POP(\alpha_{\beta^2})\ )$
  end if
end loop

[2] Source-code with the simulator is available at `https://sandpile-scheduler.googlecode.com`, published under GPL v3 public license.

Every agent runs endlessly and monitors a resource denoted as α, in contrast to its adjacent resources β^1 and β^2. Within the loop, the algorithm performs the following actions:

- (*Line 2)*: the agent waits for a new event to alter the status of the queue/pile, which can be either the processor retrieving a task or new tasks being pushed into the queue.
- (*Lines 3-4)*: as soon as an event changes the status of the queue, the algorithm selects two adjacent resources and computes x, the average workload.
- *(Line 5)*: the following step is the transition rule. If $h(\alpha) \leq h(\beta^1) + h(\beta^2)$, the resource is considered to be in an state of equilibrium and the agent waits back in *line 2*. Otherwise, the transition rule is triggered and the resource α initiates an avalanche.
- (*Lines 6-7)*: α_{β^1} and α_{β^2} are respectively the number of tasks to be transferred from α to β^1 and β^2. In order to estimate both numbers, formulas apply the ceiling and floor functions to the average workload $x \in \mathbb{R}$, as x is a real number but tasks are indivisible. $\lceil x \rceil$ refers to the number of grains to remain in α after the avalanche and $\lfloor x \rfloor$ to the final status in β^1. If the total workload is a multiple of three this would result in an evenly-distributed scenario in which $\lceil x \rceil = \lfloor x \rfloor$, otherwise, $\lceil x \rceil - \lfloor x \rfloor = 1$.
- (*Lines 8-9)*: in the last step, α_{β^1} and α_{β^2} grains are transferred to β^1 and β^2 respectively. As this very process triggers new events in resources β^1 and β^2, the avalanche will iteratively continue throughout the entire system until a global state of equilibrium is met.

This algorithm establishes the basic skeleton of the sandpile scheduler. However, a number of decisions need to be made for the system to behave efficiently, including the design of the interconnection topology or the mechanisms for balancing the workloads throughout the system. The next section tackles, therefore, the design of the overall system for an optimal operation.

5 Designing an Efficient Scheduler

The sandpile scheduler is expected to dynamically self-organize workloads in computing resources as to maximize the system performance. However, SOC dynamics also generate a certain overhead in the system when migrating tasks, which may undermine the limited capacity of physical resources such as the bandwidth. This section tries to explore different design strategies in order to find best trade-offs between the quality of the final schedule and the overhead due to SOC dynamics; the overhead computed as the number of migrating tasks. Specifically, we pursue a clear design objective: how to maximize the performance while minimizing the number of migrations?

First, we explore different virtual topologies for interconnecting the physical resources with the aim of favoring a more efficient dissemination of tasks. Besides, we present a gossiping version of the sandpile which minimizes the number of required migrations to yield states of equilibrium. Finally, the runtime dynamics of the sandpile scheduler are analyzed.

5.1 Designing the sandpile topology

In its simplest form, the BTW sandpile [3] can be defined over a linear lattice where grains accumulate at different "sites" forming piles and toppling either right or left. The model, however, has been mostly considered in its 2-dimensional version, typically using grid lattices with, e.g., von Neumann or Moore neighborhoods [2]. More recently, de Arcangelis and Herrmann [1] devised the idea of interconnecting sites using small-world networks. The advantage of using such topologies is that "*the system releases the potential of building up catastrophic avalanches more easily and produces fewer catastrophic avalanches*" [5].

Given that reducing the size and frequency of large avalanches benefits our scheduling proposal, we define the default lattice of the sandpile scheduler as a small-world overlay network (see Figure 3), in which every agent controls the state of a computing resource. Small-world networks have the advantage of connecting resources which are physically close but establishing at the same time few long-distance links. While closer links favor the locality of the load-balancing process, long-distance links allow an efficient dissemination of the workload throughout the entire system.

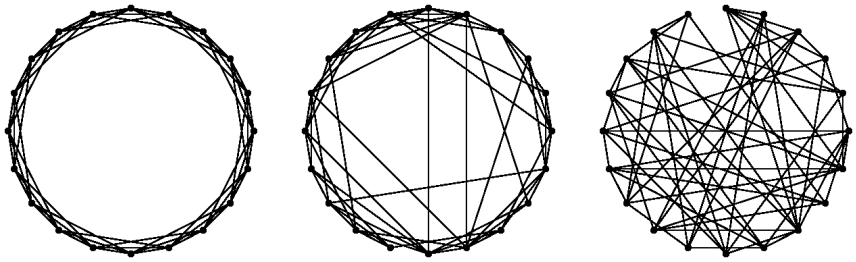


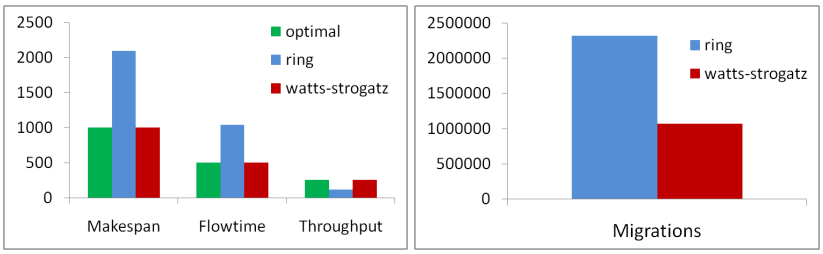
Fig. 3 The procedure for building a small-world graph can start from a ring lattice with n vertices and k edges per vertex. With a given probability p, each edge is rewired at random. Since the procedure does not allow duplicate edges, no edge is generated whenever it matches an existing one. This way for a rewiring factor of $p = 0$ the ring lattice is kept while for $p = 1$ a random graph is generated. In the figure, example of Watts-Strogatz graphs [20] with $n = 20$ and $k = 6$. From left to right, the original ring lattice for $p = 0$, a small-world graph for $p = 0.2$ and a random graph for $p = 1$. We will use $p = 0.2$ and $k = 8$ in all experiments.

In order to assess the aforementioned benefits, this section assumes a scenario in which 256 sandpile agents take control over a homogeneous architecture of $q = 256$ nodes where $\forall i, j : p_i = 1$ and $C_{i,j} = \infty$, i.e. every node computes $1 \frac{instruction}{cycle}$ and migrations are instantaneous. The workload is composed of 256000 homogeneous tasks with a number of $n_i = 1$ instructions. That means that a perfect load-balance in 256 nodes would lead to an optimal makespan of 1000 cycles. The 256000 tasks arrive in a single batch (all tasks

arrive at t_0, or alternatively, $\forall i : a_i = 0$) which is initially allocated in a node acting as a front-end.

With these experimental settings, two versions of the algorithm have been considered: one implementing a ring structure and the other a small-world structure based on the Watts-Strogatz model. Apart of the topology, both versions of the algorithm are identical.

Figure 4(a) shows how the performance of the sandpile improves when rewiring a small proportion of the edges, i.e. turning the topology from a ring to small-world. In fact, near-optimal results –in terms of makespan, flowtime and throughput– are only reached when combining the sandpile dynamics with such a topology. Furthermore, figure 4(b) shows that an additional effect of rewiring is that the overhead caused by migration diminishes. Therefore, it can be concluded that a small-world topology maximizes the performance while minimizing communications.



(a) Optimization results (b) No. of migrations (overhead)

Fig. 4 Performances of the sandpile scheduler running on a ring and a Watts-Strogatz small-world topology.

To gain further insights on the nature of these results, figure 5 depicts the heat-maps and frequencies of the workloads over a run of 1000 cycles. The huge impact that the choice of a topology has on the performance can be better explained in terms of locality: while avalanches in regular lattices spread locally, avalanches in small-world structures flood throughout the whole system. That is, a ring neighborhood, in the critical state, promotes a pyramid-like shape of the workload, while small-world neighborhoods promote fractal-like structures, where many pyramidal structures repeat at different scales.

5.2 Gossip-based forwarding

Gossiping (*or epidemic*) protocols have been widely studied as mechanisms for efficiently disseminating information in large-scale systems [6,8,19], in which peer-sampling services [15] provide the basic functionality required to interact with decentralized resources. Our proposal aims to acquire through gossiping the ability of forwarding *virtual* avalanches and therefore, to establish a *virtual* state of equilibrium before proceeding with the real migration of tasks. As

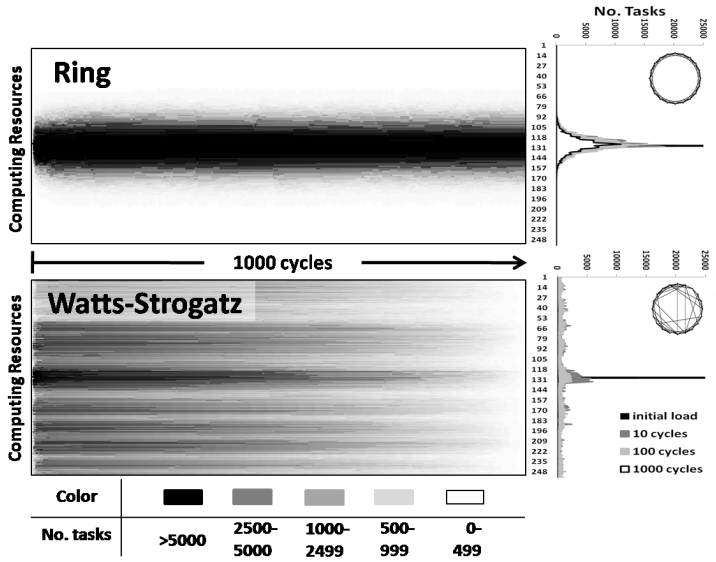


Fig. 5 Heat-maps and workloads frequencies of the sandpile scheduler using the ring (*top*) and the Watts-Strogatz topology (*bottom*).

the *servents* in peer-to-peer systems, the design of a gossip-based forwarding requires of two components: the client-side, described in algorithm 2, and the server-side implementing the forwarding procedure in algorithm 3.

The algorithm 2, or $\alpha - client$, only modifies slightly the basic design of the sandpile scheduler (algorithm 1): both algorithms are equal except for lines 8 and 9, where the latter uses the $PUSH$ function while the former forwards a request (see FWD function in Table 2).

Algorithm 2 Altering algorithm 1 to allow gossiping

```
1: loop
2:     WAIT
3:     [β¹, β²] ← α.SELECT
4:     x = (α.H + β¹.H + β².H) / 3
5:     if α.H > β¹.H + β².H then
6:         α_β¹ = ⌊x⌋ − β¹.H
7:         α_β² = (α.H − ⌈x⌉) − α_β¹
8:         β¹.FWD( α_β¹, α )
9:         β².FWD( α_β², α )
10:    end if
11: end loop
```

Unlike $PUSH(l)$, the $FWD(|l|, source)$ function does not transfer real tasks but a request with the cardinality of the tasks to be migrated from the *source* resource.

Function	Description	Pre-/Postconditions
PUSH(l)	Insert l tasks at the back	l is a list of tasks
FWD($\|l\|, source$)	Request to transfer $\|l\|$ tasks from $source$	$\|l\|$ is a natural number $source$ is a resource

Table 2 Forward function for gossiping

Algorithm 3, or $\alpha - server$, describes the FWD procedure. The procedure is also analogous to algorithm 1, in which an avalanche may start at a given node and then flood the system until finding an state of equilibrium by iteratively migrating tasks from node to node. However, the gossiping-based forwarding procedure tries to avoid multi-hops of tasks: they are not tasks but requests that hop in the system. Since such requests carry information about the number of tasks to transfer, they can be treated as *virtual* tasks and, therefore, added to the *real* tasks in a node to compute its *virtual* workload. In other words, the flooding process can be seen as an avalanche where grains are not physically transferred but virtually. As soon as the *virtual* workload is in equilibrium, every node will request to the *source* of the avalanche to transfer the tasks in an end-to-end fashion. Therefore, the goal of a gossip-based forwarding is to skip intermediate tasks migrations and directly jump from non-equilibrium to equilibrium states, i.e., criticality is *virtualized*.

Algorithm 3 Procedure for a gossip-based forwarding of the avalanche.

1: **procedure** FWD($z, source$)
2: $[\beta^1, \beta^2] \leftarrow \alpha.SELECT$
3: $x = \frac{\alpha.H+z+\beta^1.H+\beta^2.H}{3}$
4: **if** ($(\alpha.H + z > \beta^1.H + \beta^2.H)$ AND $(x > \alpha.H)$) **then**
5: $z_\alpha = \lceil x \rceil - \alpha.H$
6: $z_{\beta^1} = min(\lfloor x \rfloor - \beta^1.H, z - z_\alpha)$
7: $z_{\beta^2} = min(\lfloor x \rfloor - \beta^2.H, z - z_\alpha - z_{\beta^1})$
8: $\beta^1.FWD(\ z_{\beta^1}, source\ )$
9: $\beta^2.FWD(\ z_{\beta^2}, source\ )$
10: $\alpha.PUSH(source.POP(z_\alpha))$
11: **else**
12: $\alpha.PUSH(source.POP(z))$
13: **end if**
14: **end procedure**

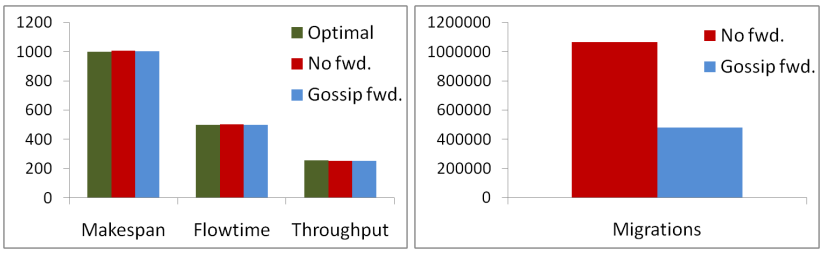
A *forward* request requires of two parameters: the number of tasks to be transferred $z = |l|$, and the *source* of the avalanche from which tasks will be finally retrieved. Since the process only involves *virtual* tasks, z represents an upper bound to forward in iterative calls. Therefore, the z tasks are spread out in the following way:

– *(Lines 2 - 3)*: as in algorithm 1, every node selects two neighbors. To avoid cycles, the $SELECT$ function in the FWD procedure marks every node with an identifier of the avalanche, as to impose the restriction that a node can only participate in forwarding once per avalanche. The averaging

process also differs from the basic case and is calculated by adding z tasks to the dividend.

- *(Line 4)*: the transition rule imposes two new conditions: first, the *virtual* workload $(h(\alpha) + z)$ has to be larger than the accumulated workload of neighbor resources. Second, the averaged workload x should be larger than the *real* workload $(h(\alpha))$. Otherwise, the current resource would end up forwarding more *virtual* tasks than those assigned. If the transition rule is not triggered, all z tasks are transferred from *source* to the current node (line 12).
- *(Lines 5 - 10)*: these lines establish the number of *virtual* tasks to be forwarded to β^1 (z_{β^1}) and to β^2 (z_{β^2}) as well as the share that corresponds to current resource α (z_α). Both, z_{β^1} and z_{β^2} tasks, are forwarded to respective resources and continue with the *virtual* avalanche, while z_α tasks are directly *pushed* into the current resource.

Virtual avalanches are expected to minimize the number of tasks migrations while preserving the good results in terms of makespan. In order to evaluate the performance of the gossiping version of the algorithm, we reproduce here experiments of Section 5.1, in which 256000 homogeneous tasks arrive at t_0 to a homogeneous architecture of $q = 256$ nodes. Experiments are conducted for the two versions of the algorithm, which respectively switch off and on virtual forwarding. Note here that the optimal schedule can be known because of the over-simplified assumptions on the workload and architecture, but from the perspective of the complexity the problem is NP-hard.



(a) Optimization results (b) No. of migrations (overhead)

Fig. 6 Performance of a gossip-based forwarding run (*gossip fwd.*) vs. the basic case (*no fwd.*)

Figure 6 shows the performance of both approaches averaged over 30 independent runs. In sub-figure 6(a) results are compared against the optimal values in terms of makespan, throughput and flowtime. It can be seen that both approaches are able to yield near-optimal results for the three metrics. Furthermore, the gossip-based approach performs slightly better than its counterpart and minimizes makespan and flowtime while increasing the throughput. However, the main outcome of the experiment is provided in sub-figure 6(b), which shows the cumulative migrations that lead to the respective perfor-

mances. The gossip-based approach clearly outperforms the basic case, which requires to double the number of migrations in order to optimize the workload.

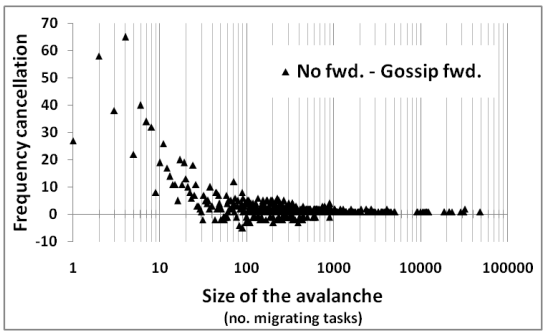
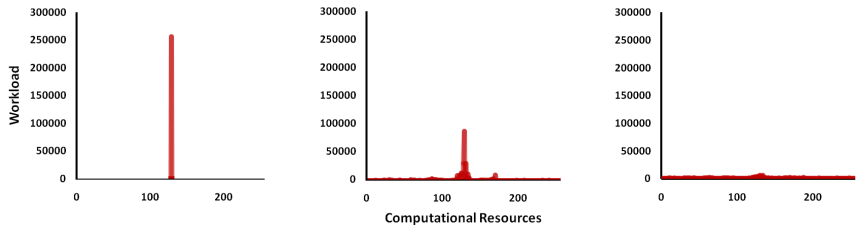


Fig. 7 Cancellation of frequencies between the basic case (*No fwd.*) and gossiping approach (*Gossip fwd.*).

The cancellation of frequencies in figure 7 explains these results. The cancellation works by subtracting, for each size of an avalanche, the frequency of the case without *forwarding* to the corresponding frequency of the gossip-based approach. A positive cancellation means that the basic case causes more avalanches of such size, while a negative cancellation refers to more avalanches due to the gossip-based approach. Gossiping clearly diminishes the number of avalanches in almost every frequency and, therefore, requires of less migrations to reach equivalent performances.

5.3 Sandpile Basic Dynamics

Taking into account the results of previous sections, the sandpile scheduler yields its optimal operation when combining a small-world topology and a gossip-based forwarding of the avalanches. This section analyzes the runtime dynamics of the scheduler for these settings.



(a) Initial State. (b) after 1 cycle. (c) after 5 cycles.

Fig. 8 Status of the workload during the first cycles of evolution.

Figure 8 shows the on-line balance of the sandpile during the first 5 cycles of evolution for an architecture of $q = 256$ homogeneous nodes and a workload of 256000 homogeneous tasks. It can be seen how the load balancing process resemblance the natural process of an avalanche with grains/tasks toppling to lower potential areas in such a way that after 5 cycles every node has some tasks assigned.

In a second round of experiments, we have modified the arrival pattern of the previous experiment to create two different types of workloads:

- *The first workload* is defined by $b = 1000$, $k = 256$ and $a_i = \lfloor i/256 \rfloor$, meaning that the 256000 tasks arrive as 1000 BoTs of size 256, in a rate of 1 BoTs per cycle.
- *The second workload* is defined by $b = 100$, $k = 2560$ and $a_i = \lfloor i/256 \rfloor \times 10$, i.e. every 10 cycles a BoTs of 2560 tasks arrive to the system.

It is straightforward to appreciate that both workloads lead to an optimal makespan of 1000 cycles in the given architecture. However, the different sizes and arrival patterns of the BoTs challenge different responses of the sandpile scheduler. Figure 9 depicts the dynamics of the sandpile under both workloads where avalanches of different sizes take place. The resulting makespans (respectively 1007 and 1009 cycles) show the capacity of the sandpile to react to different workloads and find near-optimal solutions. To that end, the sandpile lead to an emergent behavior that is related to the arrival patterns.

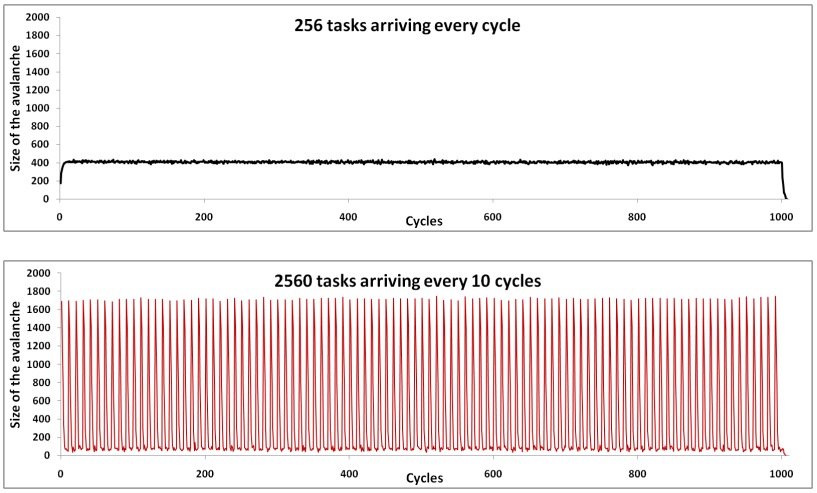


Fig. 9 Avalanches (or alternatively number of tasks migrating) in every cycle during the entire run. Upper graph represents the $b = 1000$, $k = 256$ scenario while bottom graph stands for the $b = 100$, $k = 2560$ one. Resulting makespans values are respectively 1007 and 1009 cycles. Note that, in absence of a scheduler, a run would result in 256000 cycles while an optimal balance, i.e., 1000 tasks per processor, in 1000 cycles.

Figure 10 describes the phenomenon from the perspective of the size (x axis) and frequency (y axis) in which avalanches take place. The distribution depicted in a log-scale shows a typical power-law relation between the quantity and frequency of grains/tasks toppling: small avalanches/migrations are common events while big avalanches only happen in rare occasions. The plot also depicts the different responses of the sandpile to both workloads. The workload arriving in BoTs of 256 tasks has some frequency attractors in avalanches of ~ 170 and ~ 56 grains while the one arriving in BoTs of 2560 tasks has attractors located in $\sim 1700, \sim 570, \sim 190$ and ~ 64. Such attractors correspond to respective responses for balancing the tasks arriving.

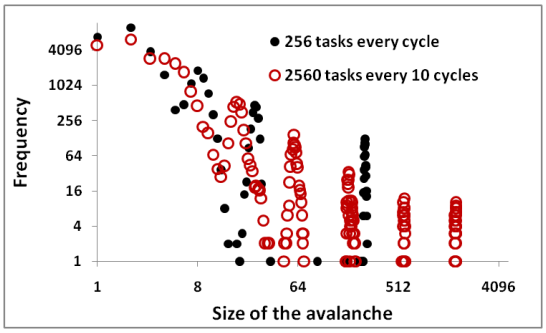


Fig. 10 Distribution of avalanches depicting the effects of the workloads in the frequencies.

In order to explain these dynamics, let us assume a perfectly balanced workload in resources α, β^1, β^2 at t_n. If that is the case, right before the arrival of a new BoTs the workload of the resources should be:

$$h(\alpha_{t_n}) = h(\beta^1_{t_n}) = h(\beta^2_{t_n}) = 1 \tag{9}$$

one task per processor, which will be retrieved at t_{n+1}:

$$h(\alpha_{t_{n+1}}) = h(\beta^1_{t_{n+1}}) = h(\beta^2_{t_{n+1}}) = 0 \tag{10}$$

If at t_{n+1} a new BoTs of, for example, 2560 tasks arrives to α, the load of the resource increases to $h(\alpha_{t_{n+1}}) = 2560$. The response of the sandpile to that event will be balancing the tasks among the three resources by estimating the average first:

$$\frac{\overbrace{h(\alpha_{t_{n+1}})}^{2560} + \overbrace{h(\beta^1_{t_{n+1}})}^{0} + \overbrace{h(\beta^2_{t_{n+1}})}^{0}}{3} \simeq 853 \tag{11}$$

and then toppling 853 grains from α to β^1 and other 853 from α to β^2. This justifies having an attractor located in avalanches of size $2 \times 853 \simeq 1700$ grains. The new settings at t_{n+2} would be:

$$h(\alpha_{t_{n+2}}) \simeq h(\beta^1_{t_{n+2}}) \simeq h(\beta^2_{t_{n+2}}) \simeq 853 \tag{12}$$

It is likely that any of these resources (e.g. β^2) has respective neighbor resources (e.g. β^3, β^4) where:

$$h(\beta^3_{t_{n+2}}) = h(\beta^4_{t_{n+2}}) = 0 \tag{13}$$

causing then a second avalanche of size:

$$\frac{\overbrace{h(\beta^2_{t_{n+2}})}^{853} + \overbrace{h(\beta^3_{t_{n+2}})}^{0} + \overbrace{h(\beta^4_{t_{n+2}})}^{0}}{3} \simeq 284 \tag{14}$$

where 284 are the grains to be migrated to β^3 and β^4, which justifies the second attractor in avalanches of size $2 \times 284 \simeq 570$ grains. The process would continue, always minimizing the respective sizes of the avalanches, until reaching an state of equilibrium.

6 Properties of the Sandpile Scheduler

In reactive systems such as the sandpile, the main properties are related to the capacity of performing well in non-clairvoyant conditions, i.e., when no assumptions are made on the size and initial allocation of the workload or the speed and size of the architecture. This section aims to assess such properties using two different test-cases. The first one tries to show the independence of the sandpile to the initialization criterion. In other words, how the system is able to react to different initial states of the workload and finally yield equivalent performances. The purpose of the second test-case is twofold: to show that the system behaves consistently in architectures of different sizes (scale-invariance) and in architectures of heterogeneous processors. Unlike in homogeneous conditions, where many algorithms are able to yield optimal or near-optimal schedules, heterogeneous conditions challenge the scheduling strategy.

6.1 Initialization criterion and performance independence

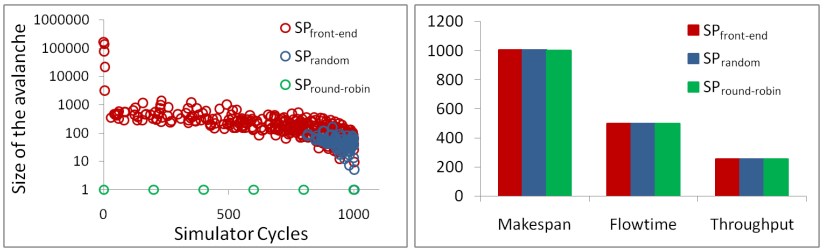
The decentralized nature of the sandpile scheduler allows multiple forms of static initialization. In previous sections, we have assumed a typical star architecture in which tasks are initially assigned to a master node or "front-end". However, many distributed systems implement some basic scheduling heuristics, such as round-robin, to initially allocate arriving tasks. In these cases, the workload is distributed among the resources and the sandpile acts as a dynamic mechanism for load-balancing.

Investigating the responses of the sandpile to different initial states of the workload can provide some insights about the correlation between the initial

allocation of tasks and the sandpile performance, i.e., whether the sandpile performance is independent to initial conditions. To that end, this section analyzes the dynamics of the sandpile for three static mechanisms assigning tasks to resources:

- $SP_{front-end}$ **:** All tasks are initially assigned to a single node acting as front-end.
- SP_{random} **:** As tasks arrive, they are distributed uniformly at random among all resources.
- $SP_{round-robin}$ **:** Tasks are assigned to resources according to the outcome of a static round-robin scheduler, which assigns tasks to processors in correlative order.

The characteristics of the workload for this study are the following ones: $b = 256$, $k = 1000$ and $\forall i, j : n_i = 1, a_i = 0$ and $C_{i,j} = \infty$. The architecture of reference is a $q = 256$ homogeneous processors with $p_i = 1$. For these settings, an optimal schedule has a *makespan* of 1000 cycles.



(a) Avalanches at runtime (b) Optimization results

Fig. 11 Performance of the sandpile for different initialization criteria.

Figure 11(a) shows the number of grains toppling during a run for the three different initialization criteria. Given that *round-robin* is optimal under previous homogeneous conditions, $SP_{round-robin}$ reflects the sandpile causing no interference to the initial schedule, where not a single grain topples along the run. However, the sandpile causes avalanches in both, $SP_{front-end}$ and SP_{random}. Despite responses being different, both approaches lead to a *makespan* of 1004 cycles, which is a near-optimal value. $SP_{front-end}$ leads to larger avalanches at the first stages of the run since the workload is primarily assigned to a single resource. Meanwhile, avalanches in SP_{random} take place in later stages and they are also smaller in size. These three different responses show the adaptive capacity of the sandpile. In fact, figure 11(b) shows that, independently of the initialization criterion, the sandpile tends to track down near-optimal schedules in all cases.

6.2 Scale-invariance and heterogeneity

An efficient scheduler has to be scalable and capable of dealing with heterogeneity; good performance should not be only restricted to pre-established architectures but also being invariant to the scale and type of architecture. To show that the sandpile is both, scalable and flexible, we have considered an scenario of scaling heterogeneous architectures. The aim is analyzing the behavior of the sandpile in scenarios with increasing complexities where static schedulers such as *round-robin* are unable to approach optimal solutions. We try to demonstrate that the sandpile can outperform this static counterpart as well as finding near-optimal schedules in more realistic scenarios.

For a fair comparison, experiments are conducted for a $SP_{round-robin}$ version of the sandpile. The scaling architectures are defined by different sets of $q = \{32, 64, 128, 256, 512, 1024, 2048\}$ processors. For each of these architectures, processors speeds scale linearly from $p_0 = 1$ to $p_q = \rho$, where ρ stands for a pre-set speeding factor of 9. That is, processor q is always 9 times faster than processor 0. Formally: $p_0 = 1, p_i = p_0 + i * (\rho/q), \rho = 9$. To perform the scalability analysis, we impose the stringent condition of the number of BoTs being equal to the number of processors ($b = q$). The rest of the settings for characterizing the workload are defined by $k = 1000$, $\forall i, j : n_i = 5, a_i = 0$ and $C_{i,j} = \infty$, meaning that different workloads represent instances of a single-scalable problem in which the optimal *makespan* is always 1000 cycles independently of the number of processors.

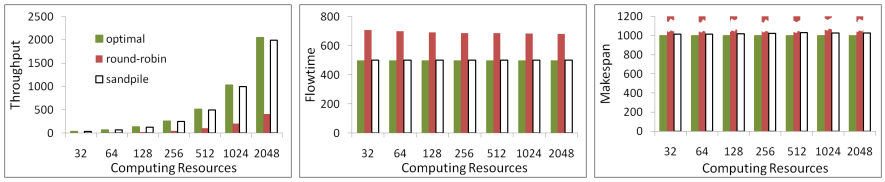


Fig. 12 Results for the *round-robin* and *sandpile* schedulers in scaling scenarios of heterogeneous architectures. Optimal values are depicted as a baseline for comparison. From *left* to *right* the throughput, flowtime and makespan of the respective schedules. The makespan bars for the *round-robin* approach are cut off as they go up to 5000 cycles.

Figure 12 shows the performance of the *round-robin* and *sandpile* approaches for scaling instances. The *round-robin* approach, in contrast to the high-performance in homogeneous architectures, shows to perform poorly under heterogeneous conditions. On the other hand, the sandpile is able to converge to near-optimal schedules with independence of the problem instance and heterogeneity. Since the sandpile starts on the basis of a *round-robin* workbalance, differences between both approaches can only be due to SOC.

To gain further insights in the scalability of the sandpile scheduler and how it reacts to problems of different sizes, figure 13 compiles the distribution of probabilities for migrating tasks in previous scaling instances. It can be seen

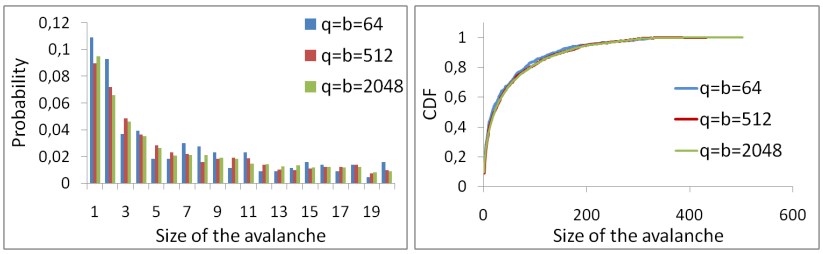


Fig. 13 Distribution of probabilities for the 20 more common sizes of avalanche (*left*) and the respective cumulative distribution functions (*right*). Independently of the instance, up to 80% of all avalanches involve less than 90 tasks while the long tail represents that there is a not-null probability for larger avalanches to take place.

how the distribution follows a scale-invariant power law distribution: the rate with which tasks migrate do not depend on the size of the problem but on its nature. It is important to note that the sandpile is non-clairvoyant and that such a behavior can be achieved, without any fine-tuning of parameters, as an emergent property of SOC systems.

7 Conclusions and Future Works

In this paper we have presented an on-line and decentralized scheduler based on a Self-organized Criticality model called sandpile, a cellular automaton working in a critical state between order and chaos. In our version of the sandpile, computing resources are under the control of agents which can interact locally within a pre-established neighborhood. In such system, tasks arrive to resources and accumulate as grains of sand. When a given agent detects that a pile exceeds the accumulated workload of two of its neighbors, the pile topples starting an avalanche, which may be propagated throughout the entire system until a new state of equilibrium is met.

In order to find an efficient design, we have analyzed two different interconnection topologies, which are respectively based on a ring and a small-world graph. The latter has been proven more efficient as tasks are easily disseminated in the system. As a consequence the throughput increases and the system yields a near-optimal performance. Besides, we have also tried to minimize the sizes an quantities of the avalanches using a gossiping-based version of the agent system. Instead of propagating a real avalanche, the gossiping protocol forwards the avalanche *virtually* until a new state of equilibrium is found. This reduces the overhead of intermediate migrations and tasks can be directly moved from the source of the avalanche to the final destinies.

All in all, the small-world gossiping-based design constitutes the best trade-off. For such a version of the algorithm, we have conducted experiments for analyzing the main properties of the sandpile. These experiments show that the approach adapts to different characteristics of workloads and architectures in an unsupervised way. Additionally, the decentralized nature of the approach

allows multiple ways of initialization, for example, static schedulers can be used to primarily allocate tasks to resources and use then the sandpile for dynamic load-balancing. Other important features tested are the invariance of the model to the scale and the ability to deal with heterogeneity: despite the model being non-clairvoyant, the different capacity of the resources leads to avalanches from low speed processors towards high speed ones as piles sink faster in the latters. That increases the overall throughput of the system and therefore the performance.

As future lines of work, we plan to extend the sandpile model for tackling energy efficiency in Cloud Computing systems. On the one hand, energy efficiency can be achieved in such systems by a smart consolidation of virtual machines. On the other hand, it is our belief that the sandpile model can be easily modified not only for spreading but also for consolidating workloads and therefore, trade-off performance and energy consumption.

Acknowledgements This work was supported by the Luxembourg FNR Green@Cloud project (INTER/CNRS/11/03).

References

1. de Arcangelis, L., Herrmann, H.: Self-organized criticality on small world networks. Physica A: Statistical Mechanics and its Applications **308**(1-4), 545–549 (2002). DOI 10.1016/S0378-4371(02)00549-6
2. Bak, P., Sneppen, K.: Punctuated equilibrium and criticality in a simple model of evolution. Phys. Rev. Lett. **71**, 4083–4086 (1993). DOI 10.1103/PhysRevLett.71.4083. URL `http://link.aps.org/doi/10.1103/PhysRevLett.71.4083`
3. Bak, P., Tang, C., Wiesenfeld, K.: Self-organized criticality: An explanation of the $1/f$ noise. Phys. Rev. Lett. **59**, 381–384 (1987). DOI 10.1103/PhysRevLett.59.381. URL `http://link.aps.org/doi/10.1103/PhysRevLett.59.381`
4. Casanova, H., Gallet, M., Vivien, F.: Non-clairvoyant scheduling of multiple bag-of-tasks applications. In: Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I, EuroPar'10, pp. 168–179. Springer-Verlag, Berlin, Heidelberg (2010). URL `http://dl.acm.org/citation.cfm?id=1887695.1887715`
5. Chen, C.C., Chiao, L.Y., Lee, Y.T., wen Cheng, H., Wu, Y.M.: Long-range connective sandpile models and its implication to seismicity evolution. Tectonophysics **454**(4), 104–107 (2008). DOI 10.1016/j.tecto.2008.04.004
6. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, PODC '87, pp. 1–12. ACM, New York, NY, USA (1987). DOI 10.1145/41840.41841. URL `http://doi.acm.org/10.1145/41840.41841`
7. Devine, K.D., Boman, E.G., Heaphy, R.T., Hendrickson, B.A., Teresco, J.D., Faik, J., Flaherty, J.E., Gervasio, L.G.: New challenges in dynamic load balancing. Appl. Numer. Math. **52**(2-3), 133–152 (2005). DOI 10.1016/j.apnum.2004.08.028. URL `http://dx.doi.org/10.1016/j.apnum.2004.08.028`
8. Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulieacute;, L.: Epidemic information dissemination in distributed systems. Computer **37**(5), 60–67 (2004). DOI 10.1109/MC.2004.1297243. URL `http://dx.doi.org/10.1109/MC.2004.1297243`
9. Franceschelli, M., Giua, A., Seatzu, C.: Load balancing on networks with gossip-based distributed]algorithms. In: Decision and Control, 2007 46th IEEE Conference on, pp. 500 –505 (2007). DOI 10.1109/CDC.2007.4434904

10. Franceschelli, M., Giua, A., Seatzu, C.: Load balancing over heterogeneous networks with gossip-based algorithms. In: American Control Conference, 2009. ACC '09., pp. 1987 –1993 (2009). DOI 10.1109/ACC.2009.5160452
11. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
12. Guinand, F., Semaan, F.: High performance computing environments and sand piles. In: International Conference on Metaheuristics and Nature Inspired Computing (META 2012). Port El Kantaoui, Tunisia (2012)
13. Hu, J., Klefstad, R.: Decentralized load balancing on unstructured peer-2-peer computing grids. In: Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on, pp. 247 –250 (2006). DOI 10.1109/NCA.2006.21
14. Iosup, A., Sonmez, O., Anoep, S., Epema, D.: The performance of bags-of-tasks in large-scale distributed systems. In: Proceedings of the 17th international symposium on High performance distributed computing, HPDC '08, pp. 97–108. ACM, New York, NY, USA (2008). DOI 10.1145/1383422.1383435. URL `http://doi.acm.org/10.1145/1383422.1383435`
15. Jelasity, M., Guerraoui, R., Kermarrec, A.M., van Steen, M.: The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Middleware '04, pp. 79–98. Springer-Verlag New York, Inc., New York, NY, USA (2004). URL `http://dl.acm.org/citation.cfm?id=1045658.1045666`
16. Jelasity, M., Montresor, A., Babaoglu, O.: A modular paradigm for building self-organizing peer-to-peer applications. In: In Engineering Self-Organising Systems, G. Di Marzo Serugendo, pp. 265–282. Springer (2003)
17. Laredo, J., Dorronsoro, B., Pecero, J., Bouvry, P., Durillo, J., Fernandes, C.: Designing a self-organized approach for scheduling bag-of-tasks. In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on, pp. 315–320 (2012). DOI 10.1109/3PGCIC.2012.28
18. Semaan, F.: Répartition dynamique de charge et phénomènes d'avalanche (2006)
19. Subramaniyan, R., Raman, P., George, A., Radlinski, M.: Gems: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. Cluster Computing **9**, 101–120 (2006). DOI 10.1007/s10586-006-4900-5. URL `http://dx.doi.org/10.1007/s10586-006-4900-5`
20. Watts, D., Strogatz, S.: Collective dynamics of "small-world" networks. Nature **393**, 440–442 (1998). DOI http://dx.doi.org/10.1038/30918
21. Willebeek-LeMair, M., Reeves, A.: Strategies for dynamic load balancing on highly parallel computers. Parallel and Distributed Systems, IEEE Transactions on **4**(9), 979 –993 (1993). DOI 10.1109/71.243526