Randomized and Adversarial Load Balancing

(Extended Abstract)

Petra Berenbrink*
Universität Paderborn, Germany
Email: pebe@uni-paderborn.de

Tom Friedetzky and Angelika Steger
Technische Universität München, Germany
Email: (friedetz|steger)@in.tum.de

Abstract

In this paper we consider dynamic load balancing algorithms for randomized and adversarial load generation models.

Consider a system of n processors. In our randomized generation models every processor may generate a task with a certain probability at each time step, leading to an expected system load of $\mathcal{O}(n)$. We present a load balancing algorithm that assures that with high probability no processor has a load exceeding $\mathcal{O}(\log\log n)$ at an arbitrary point of time. This improves upon the $\mathcal{O}\left((\log\log n)^2\right)$ bound of [4].

In the case of the adversarial load generation model every processor can change its load by some constant at each time step. Thus, the system load may become arbitrarily large. We present a balancing algorithm and show that if at some point of time τ no processor has a load exceeding some constant times the average, with high probability this holds for the next polynomial number of steps. Furthermore, we show that if the system is unstable at some point of time (meaning that there *are* processors with load much more than the average), then our algorithm recovers the system within expected poly(n) steps.

1 Introduction

In general, load balancing is the process of distributing load units (tasks, jobs) among a set of processors, fulfilling certain criteria. In order to obtain a meaningful specification of a load balancing problem, one has to distinguish between different load generation and consumption schemes and balancing goals.

Load generation primarily describes how new tasks enter the system of processors. Here, one can follow an approach where new tasks are generated *in place*, that is, processors themselves produce new tasks, or a *server based* approach, where clients issue tasks that have to be distributed among a set of processors acting as servers. In the latter case, one can distinguish between *sequential* and *parallel* generation, that is, the new tasks either arrive one after the other, or more than one task is allowed to arrive at the same time. Another important parameter is the question of *when* new tasks get generated. Here, one can imagine probabilistic, deterministic, or even adversarial approaches. Furthermore, it is often distinguished between *static* and *dynamic* models. In the first model we have a fixed set of tasks to be distributed (cf. classic "balls into bins" games), whereas in the second one new tasks are generated as time goes on.

Load consumption describes how long it takes for the processors to work on their tasks. Again, this can be modelled by probabilistic, deterministic, or adversarial distributions.

The aim of load balancing heavily depend on the type of system, generally described by resource bottlenecks. If the transferring of tasks is expensive, as few as possible transfers are desirable. Since idle processor time is expensive, we want to minimize the number of idle processors, or since memory is the critical resource, minimizing the maximum load of any processor is the primary goal. Furthermore, the kind of application or a certain response behavior (i.e. real time requirements) has to be taken into account for the choice of the balancing algorithm.

As it has been done for the model, we can describe properties of the balancing algorithm and, thereby, distinguish several general approaches. Firstly and most importantly, there is the question of whether a balancing algorithm works *centralized* or *distributed*. In the first case we have only one instance making all balancing decisions. Note that this does not necessarily mean that the algorithm has knowledge of, say, the load of all processors. In the second case, we have algorithms where every processor locally makes its own decisions (but may be dependent on the decisions of other processors, in case of communication among the processors).

Furthermore, we have the question of *whether and how much* information is obtained in order to make balancing decisions. Generally, there are three approaches: no information (each new task is thrown away, for instance), local information (if the load was raised by a certain amount during the last steps), and (semi-) global information, where processors try to gather information from other ones (possibly all processors) before making decisions.

Another possibility of classification is *when* load actually is given away. Note that this is not a priori related to the question of how much information this decision is based on, as it has been described in the previous paragraph. One can imagine approaches with new tasks always being transferred as soon as they are generated (with or without information about the load situation of other processors), or approaches in which load is kept until a certain event takes place (excess

* Author supported in part the DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen" and by the EU ESPRIT Research Project 20244 (ALCOM-IT).

of some threshold, for instance). The classic "balls into bins" games are most noteworthy representatives of the first class.

In this paper we present two randomized and distributed load balancing algorithms for different load generation and consumption schemes. In both cases, the load is generated in place and our aim is to bound the maximum load of the processors. The first scheme is a randomized one where the load is generated and consumed in such a way that the expected overall system load and the load generated per time step is $\mathcal{O}(n)$. The second scheme is an adversarial one where each processor is allowed to generate and to consume an arbitrary constant number of tasks per step. Thus, it is possible that the overall system load significantly varies, and our load balancing algorithm has to gather information about the overall system load in order to be able to make its load balancing decisions.

Contrary to balls into bins games, our distributed load balancing algorithms initiate load balancing actions only if the load of some processors exceed a certain threshold T (depending on the overall system load), and then they try to find balancing partners in order to move some of their tasks to these partners. Another advantage of our approach is that it tries to keep together tasks which are generated on the same processor. This saves communications if these tasks are not independent from each other.

1.1 Known Results

Most of the work that has been done was in the area of static and dynamic balls into bins problems. It is well-known that, if $m = n$ balls are placed independently and uniformly at random (i.u.a.r.) into n bins, there will be one bin getting $\Theta(\log n/\log\log n)$ balls with a probability of $1 - o(1)$. Karp, Luby, and Meyer auf der Heide ([12]) were the first to present a process using several possible locations in order to lower the maximum load in the context of simulating parallel random access machines (PRAMs) on distributed memory machines (DMMs). Utilizing two randomly chosen hash functions they obtain a parallel allocation process that guarantees a maximum load of $\mathcal{O}(\log\log n)$ when distributing n memory accesses among n memory modules.

Azar et al. ([3]) examine a similar protocol in a sequential setting. Each ball is allowed to choose $d \geq 2$ bins and is placed into the bin with the lowest load among the chosen bins. They show that w.h.p. the maximum load decreases exponentially to $\Theta(\log\log n/\log d)$. Furthermore, they provide results for the dynamic version of their sequential process.

According to the approach of choosing several locations for each ball, much work has been done in analyzing balls into bins games in many different ways. For sequential games see [17], [18] [8], [7], [19], [6]. For parallel games see [2], [22], [1], and [5]. In the last work, the authors present the first analysis of a parallel balls into bins game for weighted balls.

Apparently somewhat less time has been spent on theoretically analyzing load balancing strategies differing from balls into bins games, i.e. strategies that do not work according to the server based approach.

In [20], Rudolph, Slivkin-Allalouf, and Upfal present a simple strategy that equalizes the load of two processors in one step. They assume a load generation model where at each time step the load variance of any processor due to local generation and consumption is bounded by some constant and they show that the expected load of any processor at any point of time is within a constant factor of the average load.

In [14], Lüling and Monien use a similar load generation model. A processor initiates a load balancing action if its load is doubled since its last balancing action. To balance load, the processors choose a constant number of processors at random which then equalize their load. Lüling et al. show that the expected load difference of any two processors is bounded by a constant factor. In [13], Lauer presents a load balancing algorithm assuming the average load av of the system is known. He shows that, with a probability that depends on av, no processor has load that exceeds $\mathcal{O}(av)$. He obtains a high probability result only for the case $av = \Omega(\log n)$. Additionally, he presents techniques to estimate the average load of the system and extends his results to this case.

Berenbrink, Friedetzky, and Mayr in [4] present and analyze a parallel randomized continuous load balancing algorithm. They consider several randomized load generation and consumption schemes which yield an expected load of $\mathcal{O}(n)$ in the whole system. Each of the n processors will initiate a load balancing action if its load exceeds a certain threshold t, and will try to find a balancing partner for moving some of its tasks to its partner. They show that the maximum load of any processor is upper bounded by $(\log\log n)^2$, with high probability, and the expected number of load requests required for the finding of a balancing partner is constant. Additionally, they consider a scenario where an upper bound T' on the current system load is known and every processor is allowed to generate up to $\mathcal{O}((\log\log n)^2)$ tasks in $(\log\log n)^2$ steps, and each processor consumes one task per step. The maximum number of tasks per processor is then bounded by $\mathcal{O}(T' + (\log\log n)^2)$, w.h.p.

Several algorithms and complete tools for continuous load balancing have been developed and properly examined with the help of experimental studies, for example see [11], [24], [9], and [15]. For an overview consult [21].

1.2 Generation Models and New Results

Before providing you with new results we briefly describe our balancing algorithm. We divide time into consecutive phases of length T. Each phase is split into sub-phases as described below. Note that they run as a sort of background process in parallel to the load generation and consumption, respectively.

Load estimation. Every processor calculates an estimation of the complete system load.

Classification. Classify processors as overloaded (heavy), underloaded (light), or neutral, corresponding to their load compared to the estimation from above.

Assignment. Determine an assignment from heavy processors to light ones, where each heavy processor exclusively is assigned to one light processor. Basically, heavy processors ask a constant number of randomly selected other ones, which, in case they cannot to accept additional load, in the next step forward the request to randomly selected processors. This is repeated until with high probability every heavy processor found a balancing partner.

Transfer. Transfer load from heavy processors to the assigned light ones. Basically, this can be done in two ways: either as a single "packet", or in smaller "portions" during the next phase.

The rest of this paper basically consists of two main parts. In Section 3 we assume a generation model where, roughly spoken, at each time step every processor generates and consumes one task with a certain probability. In order to maintain certain stability conditions we have to have a slightly larger probability for consuming than for generating a task. This leads us to the following generation model.

SINGLE: At each time step every processor generates a task with probability p and consumes one task with probability $q = (1+\epsilon)p$ (provided there is at least one task) for arbitrary constant $\epsilon > 0$.

On this assumption, with high probability[1] (w.h.p.) the complete system load is bounded by $\mathcal{O}(n)$ at any time step. Therefore, the first step of the algorithm sketch from above is not necessary and the classification of the processors as heavy, light, or neutral refers to this $\mathcal{O}(n)$ bound. The length T of any phase is of order $\log\log n$.

We present a balancing algorithm that improves upon the algorithm of [4] which yields a high probability bound of $\mathcal{O}(\log\log^2 n)$ for the maximum load of any processor at any time step is given, to $\mathcal{O}(\log\log n)$, also holding with high probability (throughout this paper with $\log\log^t n$ we always mean $(\log\log n)^t$). The constants are (almost) arbitrarily scalable and depend on ϵ.

Theorem 1 *Fix an arbitrary point of time. Given the* SINGLE *generation model, Balancing Algorithm I ensures that with high probability the maximum load of any processor is bounded by $\mathcal{O}(\log\log n)$.*

There are several models closely related to SINGLE. Our analysis has only to be modified slightly to give the very same result for any model with overall expected system load $\mathcal{O}(n)$, in which steady-state statements like in Lemma 2 of [4] can be made and a processor is allowed to generate up to $\mathcal{O}(\log\log n)$ tasks in an interval of the same length. For further examples see [4].

To our knowledge this is the first result providing the same high probability bound for dynamic load balancing as was previously known for dynamic balls into bins games under comparable conditions. The term "comparable" here refers to the analogy of situations where (a) for balls into bins games $\mathcal{O}(n)$ balls are thrown into n bins with $d = 2$ choices for every ball per round, obtaining for any bin a maximum load of $\mathcal{O}(\log\log n)$ (see[1]), and (b) we have $\mathcal{O}(n)$ tasks to be distributed among n processors in our scenario, obtaining – up to almost arbitrarily scalable constants – the same maximum load.

Compared to balls-into-bins games, our algorithm saves on communication as it moves tasks that have been generated on processor P only to another processor if the load of P exceeds a certain limit. During each phase of length T our algorithm w.h.p. generates $\mathcal{O}(n)$ messages only, whereas $\mathcal{O}(n\log\log n)$ tasks are generated during the whole phase. Balls-into-bins games generate at least one message per generated ball, thus our algorithm reduces the number of messages by a factor of $\log\log n$. Another advantage of our algorithm is the fact that it tries to keep the tasks generated on the same processor together, which will be important if the tasks are not independent.

In Section 4 we extend this scenario to a potentially unbounded total system load. We no longer assume a probabilistically determined load generation scheme but allow each processor to alter its load in any time step by at most some constant number by generation and consumption of tasks.

[1] With high probability means with probability at least $1 - n^{-\alpha}$ for some constant $\alpha > 1$.

UNBOUNDED: At each time step every processor may change its load due to generation and consumption by up to Δ tasks, with $\Delta \geq 1$ being an arbitrary integer constant.

Again, our algorithm can also be used in generation models where each processor is allowed to generate $\mathcal{O}(\log\log^2 n)$ tasks in an interval of the same length.

Of course, in order to provide a meaningful bound on the maximum load, now the complete system load has to be taken into account. Recall that our approach divides time into phases; here T, the length of any phase, is of order $\log\log^2 n$. We say that our system of n processors to be *balanced* for a given phase if at the first time step of this phase no processor has a load exceeding $\mathcal{O}\left(\max\left\{\frac{m}{n}, \log\log^2 n\right\}\right)$, with m being the complete system load at the current time step. We provide the following result.

Theorem 2 *If the system is balanced for a phase Π, then with high probability it will be balanced for the next phase $\Pi+1$, and, furthermore, with high probability it remains balanced for the next poly(n) phases.*

As in [4], we can also show that the expected number of requests needed to find a balancing partner is constant. Unfortunately, we need $n \cdot \log\log n$ messages per phase in order to estimate the system load for each processor.

Additionally, we note some sort of "recover property" of our algorithm: we show that, if the system is in an arbitrary state at some time step, our algorithm will bring the system back into stability (here with "arbitrary state" we mean some arbitrary distribution of the complete system load, where processors may have significantly more load than the average).

Theorem 3 *If, at some time step, the system is in an arbitrary (i.e. non-balanced) state, after expected* poly(n) *steps it will be balanced again.*

2 The Collision Protocol

In this section we describe the (n,δ,a,b,c)-collision protocol we are making use of in our balancing algorithms in order to estimate the overall system load and to determine an assignment of load balancing requests to processors. The collision protocol originates in shared memory simulations (see [16]).

Since we apply the collision protocol to perform different tasks we here provide you with a more or less abstract description which will be refined it requires. We talk of *requests* issued by processors (note: every processor issues at most one request). The number of processors issuing requests is at most δn for $0 < \delta \leq 1$. Informally, a request can be regarded as a wish to establish communication channels to a certain number of (randomly chosen) processors. For each request we now i.u.a.r. choose a different target processors to which so-called *queries* are sent. The collision protocol tries to find an assignment in which (1) no processor has to accept more than c queries directed to it, and (2) for every request at least b of the a queries are successful (i.e., are accepted).

The (n,δ,a,b,c)-collision protocol

Every of the at most δn processors initiating a request i.u.a.r. selects a pairwise distinct processors and sends queries to these processors. Set the status of each initiated request to *active*.

For $t = 1$ **to** r **do**

- Every processor with at most c queries directed to it accepts all queries and sends accept messages to the corresponding processors.
- If for some request at least b accept messages arrive (summed up over all previous rounds of the protocol), further queries will be canceled and the request becomes *inactive*.
- Every remaining active request re-sends queries to the processors not having accepted their queries up to now (note that no new random choices are made).

In the following, we state (without proof) two results estimating the runtime of different versions of the (n, δ, a, b, c)-collision protocol. The proofs are similar to [16] where the authors prove a more general version of a similar process for a wide range of parameters a, b and δ. Their results hold only with probability $1-(1/n)^{\beta}$ for a fixed constant $\beta < 1$ which is not sufficient for our needs.

Lemma 1 *For $\delta = 1$ and arbitrary $a \geq 2$, $b = a-1$, even $c \geq \max\{8ea^2, 4\alpha + 1\}$ it holds: the (n, δ, a, b, c)-collision protocol terminates with a valid assignment of queries after $r = \log\log n/\log c + 1$ rounds of the* **For***-loop. The total runtime of the process can be bounded by $\log\log n/\log c + 1$ steps.*

The last statement of the Lemma holds because every processor is allowed to communicate with a constant number of other processors during each round. Thus, one execution of the body of the **For**-loop can be performed in one single step.

In the following, we examine the collision protocol in another setting with $c = 1$, i.e. every processor is only allowed to accept one request and $\delta < 1$.

Lemma 2 *Let $\alpha \geq 1$. For $d = a - b = 2\alpha + 2$, $c = 1$, $\delta \leq \frac{1}{2(2a)^{d+1}}$ and $r \geq \frac{\log\log n}{\log d} + 2$ it holds: with probability at least $1 - \frac{1}{n^{\alpha}}$ the (n, δ, a, b, c)-collision protocol terminates with a valid assignment of queries after r rounds of the* **For***-loop. The total runtime of the process can be bounded by r steps.*

3 Bounded System Load

In this section we present a new balancing algorithm for the randomized load generation model which reduces, the maximum load by a factor of $\log\log n$ compared to the algorithm presented in [4]. It has to be noted that in this section the structure and in part the analysis are similar to [4]. Only the core part of the algorithm, the **assignment** sub-phase, has been changed significantly, so we concentrate on describing and analyzing this phase. We do not try to settle or even optimize constants, if this does not lead to further insights.

If not mentioned otherwise, in the following we assume the load generation model SINGLE. Recall that in this model in each time step every processor generates a task with a probability of p and consumes a task with a probability of $p(1+\epsilon)$, for an arbitrary but fixed $\epsilon > 0$.

By a *non-balancing system* we denote a system with no load balancing actions at all. If balancing actions take place (according to our algorithms) we speak of a *balancing system*. Furthermore, *system load* refers to the sum of all tasks in the system at a specific point of time.

In the following, we present a high-level description of our algorithm. In [4], $\Theta(\log\log n)$ collision games each with running time $\Theta(\log\log n)$ were played in order to exclusively assign a light processor to each heavy one. This leads us to the length of a phase of $\Theta(\log\log^2 n)$. As we will see, one collision game with an overall running time $\Theta(\log\log n)$ is sufficient. Note that the length of a phase is also a lower bound for the maximum load of any processor, since in a phase of length T a processor can generate up to T new tasks.

Balancing Algorithm I

According to the brief sketch of the algorithm from section 1.2, we take the following approach. Time is divided into consecutive phases of length $T = l \cdot \log\log n$, for some constant l. An arbitrary phase Π works as follows:

Load estimation. This stage is not necessary. As we will see later, due to our generation model we have a bound of $\mathcal{O}(n)$ on the system load.

Classification. Processors with a load of at least $8T$ at the beginning of phase Π are called *heavy* and processors with load at most T at the beginning of Π are called *light*.

Assignment. During this sub-phase, every heavy processor tries to find a light one in order to transfer $4T$ of its load to this light one (which we call *balancing partner* in the following) at the end of this phase. Every light processor must be assigned to at most one heavy processor.

We establish a global "access structure" in which heavy processors search for light ones. Therefore, we play a collision game. All processors are both initiators of a request and potential targets. We build a directed graph G with the processors as nodes; edges are determined by the collision game. To do this, each node initiates a request consisting of a queries (chooses a pairwise distinct processors at random), and the c-collision protocol (w.h.p.) ensures that b of them are successful, while no processor has to accept more than c queries. Every successful query is represented by an edge in G (directed from the initiator of the query to the corresponding randomly selected node, that accepted the query). Hence, every node has constant in- and out-degree of c and b, respectively.

We now utilize G to determine the assignment of heavy to light processors. For every processor p, let T_p be the subgraph of G, consisting of p as "root" and all nodes reachable from p by directed paths of a length of $t = \mathcal{O}(\log\log n)$. One can imagine T_p as a tree with out-degree b where some nodes can occur more than once. Assume we play a collision game with parameters $b = c = 2$, see figure 1 for an example outcome. Here, the node marked with a 1 "shares" one of its successors (the node marked with a 2) with another node. If we developed T_p starting with p along the directed edges, node 2 would occur twice, it would have two copies in T_p. Note that subtrees rooted at nodes of the tree representing the same processors are the same, too.

In our upcoming analysis, we need to have a certain bound on the number of *distinct* nodes in some T_p. We count at most one copy of every node as *relevant* (so the number of relevant nodes reflects the "true size" of T_p – no matter how often some node "logically" occurs in T_p, it counts just for one). In our example, T_p has 12 relevant nodes (if every processor would have in-degree 1, the best possible number would be 15).

Our assignment strategy works as follows. Every heavy processor p searches for a light balancing partner in its T_p. p sends search messages "down" the T_p (along the outgoing edges determined by the collision game). Every node

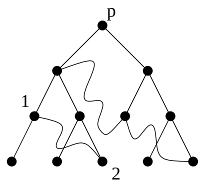

Figure 1: An example T_p

receiving such a message from its predecessor, forwards it down via its outgoing edges. Since every T_p has a depth of $\mathcal{O}(\log\log n)$, this lasts $\mathcal{O}(\log\log n)$ steps (a processor combines all messages received from above).

Every light processor, that received such a search message stores the identifier of the corresponding heavy processor in some list (we assume identifiers to be part of the messages). After these $\mathcal{O}(\log\log n)$ steps, every light processor randomly selects one of the heavy processors it received a message from.

In Section 3 we show that with high probability every heavy processor now is selected by at least one light processor. It remains to resolve the situation of some heavy processor being selected by more than one light processor. We assume the light processors in question to send a message to the chosen heavy one, which, in turn, selects an arbitrary one of them. Note that due to our communication model (based on the collision rule), at most a constant number of messages actually have to be evaluated, no matter of how many arrive. The heavy processor now returns an acknowledge to the selected light processor, and the assignment is done.

Transfer. After the assignment has been determined, every heavy processor now transfers $4T$ tasks to its balancing partner.

Analysis of Balancing Algorithm I

First, we estimate the probability that a processor has a certain load in the non-balancing system. The following lemma is stated without proof; it is a simple application of Markov chain theory, see e.g. [23] for details.

Lemma 3 *Fix an arbitrary point of time. In the non-balancing system, with a probability of at most $(1/\mu)^i$, a fixed processor has load i ($\mu > 1$ is some constant depending on ϵ).*

Now we provide some more technical details. First, we state a lemma estimating the system load. The proof is omitted due to space limitations.

Lemma 4 *Fix an arbitrary time step. With high probability, for some constant l_s, the system load is at most $l_s n$ both in the in the non-balancing and the balancing system.*

Proof (sketch) By modeling the state of a processor (with respect to its load) as a Markov chain, we derive bounds on the probability for a processor having a certain load, see Lemma 3 above. Then, we apply a standard Chernov-Hoeffding tail estimate to bound the system load (note that in the non-balancing system the random variables describing the load of the processors are independent).

The proof for the upper bound in the balancing system is very similar; we just have to modify the Markov chain in order to reflect the fact that some processor can receive tasks due to balancing actions (in addition to its normal load generation).

□

Now we present bounds on the numbers of light and heavy processors, respectively.

Lemma 5 *Fix an arbitrary time step. The probability for a fixed processor to be heavy is bounded by $\frac{\mu}{\mu-1}\cdot\left(\frac{1}{\mu}\right)^{2l\log\log n}$, for some constant $\mu > 1$ depending on ϵ. Furthermore, there are at least $^3/_4 n$ light processors.*

Proof Recall that in the balancing system we have phases of length $T = l\cdot\log\log n$, a processor is light if it has load at most T, and heavy if its load is at least $8T$. During a balancing action, a balancing partner gets $4T$ tasks. Since we only balance with processors that have load at most T at the beginning of one phase, such a processor has a load of at most $T + 4T + T = 6T$ ($4T$ tasks transferred from a heavy processor and at most another T tasks generated on its own during the phase) after a load balancing action. Thus, a light processor never exceeds load $6T$ directly after a balancing action, and, therefore, never has more than $6T$ tasks generated by other processors.

This implies that a processor must have raised its load by at least $8T - 6T = 2T$ tasks on its own in order to reach our heavy-threshold of $8T$, i.e., if a processor is heavy, at least $2T$ of its tasks are self-generated. Exploiting this observation, we now can bound the probability for a processor having a load of $8T$ (being heavy in the balancing system) by the probability that it has a load of $2T$ in the case of a non-balancing system.

Now, by Lemma 3 from above, we know that we can bound the probability for a processor having load $2T = 2l\cdot\log\log n$ in the non-balancing system by $(1/\mu)^{2l\cdot\log\log n}$. We can bound the probability for it having a load of *at least* $2l\cdot\log\log n$ by

$$\sum_{i=2l\cdot\log\log n}^{\infty}\left(\frac{1}{\mu}\right)^i = \frac{\mu}{\mu-1}\cdot\left(\frac{1}{\mu}\right)^{2l\cdot\log\log n}.$$

In order to provide a lower bound on the number of light processors, we apply a simple pigeonhole argument. Lemma 4 states that we have a system load of at most $l_s n$, with high probability. Clearly, there can be no more than $l_s n/T$ processors with load at least T, because otherwise we would have more than $l_s n$ tasks in total. Therefore, there are at least

$$n - \frac{l_s n}{T} = n\cdot\left(1 - \frac{l_s}{T}\right) = n\cdot\left(1 - \frac{l_s}{l\log\log n}\right) \geq \frac{3}{4}n$$

processors with a load of at most T (i.e., light processors), with high probability. □

In order to build our access graph G, we play a c-collision game with parameter $\delta = 1$, that is, *every* processor issues a request. In order to meet the requirements of the c-collision game (Lemma 1), we choose the parameters to $a = 2\alpha + 3$, $b = 2\alpha + 2$, and $c = 8ea^2$. Note that this implies that every processor has to be able to communicate with c processors during each round. We state the following result without proof; it follows directly from Lemma 1 (just plug in the parameters).

Observation 1 *With high probability, after* $\frac{\log\log n}{\log c}+1$ *steps the c-collision game terminates with a valid result, that is, a valid access graph G.*

As mentioned above, a heavy processor p searches for a balancing partner in its T_p. Although some nodes may occur more than once in T_p, we can provide bounds on the number of *relevant nodes.*

Lemma 6 *Let T_p be the sub-graph with degree $b = 2\alpha + 2$ induced by p as "root" and all the nodes reachable from p by paths of at most $t = d \log\log n / \log b + k'$ outgoing edges for some constants d and k. (T_p is a "multi-set" as some nodes may occur more than once.) Then it holds:*
With high probability, every T_p, consisting of $D = \frac{b^{t+1}-1}{b-1}$ nodes in total, has of at least $D/2$ relevant nodes.

We omit the proof due to space limitations.

Next, we show that once our access graph G is established, the assignment of heavy to light processors actually can be performed.

Lemma 7 *W.h.p., within an* **assignment** *sub-phase each heavy processor is exclusively assigned one light processor.*

Proof Recall that every node has out-degree b and in-degree c. We build the query trees up to depth $t = \frac{2\log\log(n)}{\log b} + \frac{\log(k)}{\log b}$ for $k = 16(\alpha+1)^2$. Thus, every T_p has $(b^{t+1}-1)/(b-1) \geq b^t$ nodes, with

$$\begin{aligned} b^t &= b^{\frac{2\log\log(n)+\log(k)}{\log b}} = b^{\frac{2\log\log(n)}{\log b}} \cdot b^{\frac{\log(k)}{\log b}} \\ &= 2^{2\log\log(n)} \cdot 2^{\log(k)} = \log^2(n) \cdot k \\ &= \log^2(n) \cdot 16(\alpha+1)^2 \end{aligned}$$

Some of these nodes may occur more than once, but we can bound the number of *relevant* nodes. Fix some arbitrary T_p. Lemma 6 shows that with a probability of $1 - n^{\alpha}$ at least half of the nodes of any T_p are *relevant.* Hence we have at least $\log^2(n) \cdot 8(\alpha+1)^2$ relevant nodes in our T_p.

Lemma 5 shows that we have at least $^3\!/_4 n$ light processors, so that if we select a processor at random, with a probability of at least 3/4 it will be light. Unfortunately, the collision game does not provide independent choices (i.e., the processors are not selected independently). To overcome this problem, we can simply subtract the size of T_p from the overall number of light processors (resulting in, say, at least $n/2$). This yields a lower bound for the probability that a processor determined by the collision game is light of 1/2.

Now, the expected number of light nodes of each T_p can be bounded by $\log^2(n) \cdot 4(\alpha+1)^2$ light nodes. We can apply Chernov bounds to show that the number of light processors in our T_p is sharply concentrated around the expectation; with a probability of at least $1 - (1/n)^{\alpha'}$, for any constant α', we have at least $\log^2(n) \cdot 3(\alpha+1)^2$ light nodes in T_p.

Now, we will bound the expected number of heavy processors that can *reach* a fixed light processor p. We call a processor q *reachable* by some other node p if $q \in T_p$ (the other way around, we then say that p can *reach* q). Clearly, some processor p is reachable by at most $\frac{c^{t+1}-1}{c-1} \leq 2c^t$ other processors, as c is the in-degree of any node, and any T_p has depth t.

$$\frac{\mu}{\mu-1} \cdot \left(\frac{1}{\mu}\right)^{2l \cdot \log\log n}.$$

Therefore, the expected number of heavy processors that can reach our light processor p can be bounded by

$$2c^t \cdot \left[\frac{\mu}{\mu-1} \cdot \left(\frac{1}{\mu}\right)^{2l \cdot \log\log n}\right]$$

If we now let

$$l \geq 3 + \frac{1}{\log \mu} \cdot \left(\frac{\log c}{\log b} - \frac{1}{2}\right)$$

a straightforward calculation shows that we have

$$2c^t \cdot \left[\frac{\mu}{\mu-1} \cdot \left(\frac{1}{\mu}\right)^{2l \cdot \log\log n}\right] \leq \log n.$$

(Recall the definition of t to be $t = \frac{2\log\log n + \log k}{\log b}$ with $k = 16(\alpha+1)^2$). Applying Chernov bounds, we can show that for any light processor p (or any other processor, for that matter), with a probability of at least $1 - (1/n)^{\alpha+1}$, there are at most $3(\alpha+1)\log(n)$ heavy nodes that can reach p.

Our algorithm now works in a manner that any light processor randomly selects one of the heavy processors that it has received a search message from. We now show that with high probability any heavy processor is selected by at least one light one. Fix some heavy processor p. As we have shown above, T_p w.h.p. contains at least $\log^2(n) \cdot 3(\alpha+1)^2$ light processors. Each of them is reachable by at most $3(\alpha+1)\log(n)$ heavy nodes, from which it selects one randomly. The probability that p is not selected by any of those $\log^2(n) \cdot 3(\alpha+1)^2$ light processors can be bounded by

$$\begin{aligned} &\left(1 - \frac{1}{3(\alpha+1)\log n}\right)^{3(\alpha+1)^2 \log^2 n} \\ =\; &\left[\left(1 - \frac{1}{3(\alpha+1)\log n}\right)^{3(\alpha+1)\log n}\right]^{(\alpha+1)\log n} \\ \leq\; &\left(\frac{1}{e}\right)^{(\alpha+1)\log n} \leq \left(\frac{1}{n}\right)^{\alpha+1} \end{aligned}$$

So we see that with a probability of at most $1/n^{\alpha+1}$, some fixed processor is not selected by at least one light processor, and, therefore, with a probability of at least $1 - 1/n^{\alpha}$, there is no heavy processor that is not selected by at least one light processor. This concludes the proof of the lemma. □

Lemma 8 *For some suitable constant l, a phase length of $l \cdot \log\log n$ is sufficient to run Balancing Algorithm I.*

Proof Both the **classification** and **transfer** sub-phase take one step. The **assignment** sub-phase consists of playing the collision game (building the access graph G), which, according to Observation 1, takes $\frac{\log\log n}{\log c} + 1$ steps.

The T_p's have a depth of $(2\log\log n + \log k)/\log b$, so the sending down of search messages takes as many steps. (A processor can receive and send a constant number of messages per step.) We need an additional step to have the light processors select a heavy one, send messages from the light processors to the chosen heavy ones, and receive acknowledgments. Therefore, the complete phase needs to have at least

$$2 + \left(\frac{\log\log(n)}{\log(c)} + 1\right) + \left(\frac{2\log\log(n) + \log(k)}{\log(b)} + 1\right)$$

steps. Clearly, this can be estimated as $l' \cdot \log\log n$, for some constant l', depending on b, c, and α. Finally, Lemma 7 has another restriction on the length of any phase in that we need at least $l \cdot \log\log n$ steps with

$$l \geq l'' := 3 + \frac{1}{\log \mu} \cdot \left(\frac{\log c}{\log b} - \frac{1}{2}\right).$$

Summing up, if we set the length of any phase to $l \cdot \log\log n$ with $l \geq \max\{l', l''\}$, all conditions and restrictions are fulfilled. □

Now we are ready to prove Theorem 1.

Proof (sketch) So far, we have showed the following (all holding with high probability). For any fixed phase:

1. The system load is bounded, both in the balancing and the non-balancing system (Lemma 4).
2. We have bounds on the numbers of heavy and light processors are bounded (Lemma 5).
3. The collision game builds the access graph G, and we have a lower bound on the number of relevant nodes in every T_p (Observation 1 and Lemma 6).
4. The assignment of light processors to heavy ones works (Lemma 7).
5. A phase length of $l \cdot \log\log n$ with constant l is sufficient.

Numbers 1)-5) above, together with the observation that processors that are either assigned load or that give away load due to a balancing action both are not heavy directly after the transfer (given that the heavy one became heavy recently) show the following: with high probability the first balancing action of any processor succeeds if we do not make further assumptions to the system, and afterwards it is below the heavy threshold again.

The remainder of this proof is done accordingly to[4]. Let $[\Pi_j, \ldots, \Pi_k]$, $k \geq j+2$, denote the sequence of phases where

- Π_k is the current phase,
- p is not heavy at the beginning of Π_{j-1}, and
- p is heavy during $\Pi_j, \ldots, \Pi_k$.

We relate the event of p being heavy at the beginning of the current phase Π_k to the event that its first balancing attempt (during phase Π_j) failed (a necessary condition). Of course, assuming that p is heavy at the beginning of some phase has certain consequences. We can show that the system load can still be bounded by $\mathcal{O}(n)$ (w.h.p.), even if we condition on p being heavy. Now Lemma 5, Lemma 7, and Lemma 8 hold as before (they relied on the bound on the system load). Now we can conclude that with high probability p's first balancing attempt is successful, what contradicts the assumption that p is heavy for at least two phases. □

4 Unbounded System Load

In this section we no longer assume generation models that lead to a bounded system load but model UNBOUNDED instead: at each time step every processor changes its load due to generation and consumption by up to Δ, where $\Delta \geq 1$ is an arbitrary integer constant.

What are the main problems? Since we no longer have an upper bound for the system load, the processors cannot be classified to be heavy or light based on their own load only. Therefore, every processor first estimates the system load. Again, the algorithm follows the outline provided in section 1.2. We divide time into consecutive phases of length $T = l \cdot \log\log^2 n$ now, for some constant l. Now fix some arbitrary phase Π.

Balancing Algorithm II

Load estimation. Like in the previous section, we utilize the collision game to establish an access structure. We again build an access graph G with the set of processors as nodes, where every node has some constant in-degree c and some constant out-degree b, where b and c are the parameters of the collision game. We use the same parameters as in the previous section, so Observation 1 (based on Lemma 1) still holds. Recall the definition of T_p as being the sub-graph induced by p as "root" and all the nodes reachable from p by directed paths a of length of at most

$$t = \frac{2\log\log n}{\log b} + \frac{\log k}{\log b}$$

for $k = 16(\alpha+1)^2$, with α being some constant, depending on the probability bound we want to obtain. Every processor now gathers load information from its T_p. To do this, every processor p initializes a register L_p with its own load. Then, every processor sends information about its load via its incoming edges up to its parents (if any). Then, for the next t steps, every processor that received information from its successors calculates the sum s of the received values, adds s to its register, and finally forwards s to its parents.

There are a couple of things to note here. The communication complexity now depends on the load of the processors and, therefore, may no longer be bounded. Second, this estimation phase is sort of a "background process" that runs parallel to the normal load generation and consumption. All data gathered in these t steps probably is no longer valid at the end of this sub-phase. The data exactly refers to the load situation at the very first step of this phase.

Obviously, every processor's register L_p contains the sum of the load of all processors in its T_p, including its own. Note that since some processors may occur more than once in some T_p, apparently the load of such processors is counted more than once. Now by dividing L_p by $|T_p|$, the number of (not necessarily distinct) processor in T_p, p calculates the average load in its T_p, denoted by $\bar{L}_p$. Finally, let $l_p = \max\{\bar{L}_p, \log\log^2 n\}$.

Classification. We call a processor p *heavy* if its load is at least $\gamma_h l_p$, and we call a processor p *light*, if its load is at most $\gamma_l l_p$, where γ_h and γ_l are suitably chosen constants to be defined later. Note that this classification takes place after $\mathcal{O}(\log\log n)$ steps of the current phase (after the access graph G has been "built" by the collision game and after the load estimation has been done).

Assignment. We now want to exclusively assign a light processor to every heavy one. Due to technical reasons (basically, we can only provide an upper bound on the number of heavy processors much larger than in the previous section), we no longer can simply utilize G to do so; we need to provide more "structure" to the search process.

We play a collision game with parameters $a = 4(\alpha+1)$, $b = 2(\alpha+1)$, and $c = 1$, where only heavy processors issue requests; the running time can be bounded by $\mathcal{O}(\log\log n)$. If any of those processors having accepted a query (b of the a queries are successful, that is, are accepted) is light, it sends an *offer message* to the initiator of the query. Note that (i) due to parameter $c = 1$ no processor accepts two queries at the same time (this implies that we have one unique heavy processor we have to send the message to), and (ii) we assume the collision mechanism not only for the collision game itself, but also for handling of said offer messages — no matter how many arrive at one time step, at most a constant number of them are evaluated.

Now every heavy processor that received at least one offer message arbitrarily selects one of them and sends an *acknowledgement message* to this selected one (this one is now assigned as its balancing partner). Further, it sends a *cancel message* to all the processors that have accepted one of the queries belonging to its request.

Now there can be heavy processors that so far have not been assigned a balancing partner because all of their queries have been accepted by non-light processors. We say that those non-light processors that have accepted a query *support* the heavy ones in their search for balancing partners in that they themselves issue requests (which again are resolved by a collision game with parameters a, b, and $c = 1$). This procedure continues for $t = \mathcal{O}(\log\log n)$ rounds: as long as no light processor has been assigned, all the processors that have accepted a query in some round issue new requests themselves in the next round (and only they do that), that is, they "spread" the search of the corresponding heavy processor.

Of course, it does not make sense to issue further requests if a heavy processor already is assigned a balancing partner. This is what the cancel messages are for: each time an assignment is done, the corresponding heavy processor sends the cancel message down to its direct successors, which in turn forward it down as far as the collision games have proceeded in developing the "search space" so far. Now the next collision game, where non-light processors that have accepted a query act as initiators of new requests, does not instantly begin, but is delayed until any processor can be sure to have received a cancel message if one was sent (clearly, this takes at most t steps).

We have to restrict the number of requests issued in any round in order to be sure that the collision game actually is able to resolve all of them. Since now, as we will see, we have an upper bound on the number of heavy processors that is a constant fraction of all processors, it would take only a constant number of rounds to have too many active requests. Therefore, we do not have all processors having accepted a query issue a new one (only in case the corresponding heavy processor has not found a balancing partner at all) but only those that either are heavy or are light and already have been selected by a heavy processor in some previous round. Clearly, this reduces the number of active requests to some constant fraction of n.

Transfer. Now the heavy processors can actually transfer load to their balancing partners. Similar to Section 3, we can assure that heavy processors are no longer heavy and that light processors do not become heavy due to the transfer, even in the worst case if they generate as much as they are allowed to. Note that the amount of transferred load can no longer be bounded independent of the system load.

Analysis of Balancing Algorithm II

Now we state some technical Lemmas that are needed in order to analyze the algorithm. First, we concentrate on the **load estimation** sub-phase. We count at most one occurrence of every processor as *relevant* with respect to some T_p (and, occasionally, speak of those as *relevant copies*). Lemma 6 states that with high probability at least half of all the $(b^{t+1}-1)/(b-1)$ nodes of any T_p are relevant, that is, we have at least $1/2 \cdot (b^{t+1}-1)/(b-1) \geq b^t/2$ relevant copies, with high probability.

The following Lemma now shows that the estimation of the average system load obtained by just looking at a rather small subset of all processors is within a constant factor of the real average.

Let M_τ denote the system load at time step τ, let $m_\tau = \max\{M_\tau/n, 3\Delta T\}$. We call the system *balanced* for some phase Π if no processor has load exceeding $\gamma_h m_\tau$, where τ is the first time step of phase Π.

Lemma 9 *Provided that every T_p contains at least a constant fraction of relevant nodes/processors and the system is balanced for some phase Π, then there is a constant $\beta > 1$ such that with high probability for each processor p we have*

$$\beta^{-1} m_\tau \leq l_p \leq \beta m_\tau.$$

Proof We first show $l_p \leq \beta m_\tau$. Recall that l_p was defined to be $l_p = \max\{\bar{L}_p, \log\log^2 n\}$ with $\bar{L}_p$ being the average load in T_p. By the definition of *balanced*, in time step τ no processor has load exceeding $\gamma_h m_\tau$, thus yielding $\bar{L}_p \leq \gamma_h m_\tau = \gamma_h \cdot \max\{M_\tau/n, 3\Delta T\}$ and, for any $\beta \geq \gamma_h$,

$$\begin{aligned} l_p &\leq \max\{\gamma_h \cdot \max\{M_\tau/n, 3\Delta T\}, \log\log^2 n\} \\ &= \gamma_h \cdot \max\{M_\tau/n, 3\Delta T\} \qquad (3\Delta T \geq \log\log^2 n) \\ &= \gamma_h m_\tau \leq \beta m_\tau \end{aligned}$$

The other part of the proof is somewhat more difficult. In the first part we exploited information about an upper load bound, but there is no *lower* bound for the load of a processor. We have to make use of a probabilistic argument, that, informally spoken, shows that with only small probability some T_p contains "too many" processors with "too little" load. First, we have to bound the *number* of processors with "too little" load, after which we estimate the worst possible impact of those processors on l_p.

We call a processor *dangerous*, if its load is at most $\gamma_l m_\tau$. Let P_l denote the set of dangerous processors, let $n_l = |P_l|$. Now we want to show that we do not have too many light processors. Under what condition is n_l as large as possible (recall this this would be the worst case for our situation)? In any case, the load that is not kept by processors in P_l somehow has to be distributed among the processors not in P_l. Since we assume an upper bound on the load of any processor, this means that the number of processors not in P_l would need to be large if much load has to be distributed among them (if little load is kept by processors in P_l). Therefore, the worst case (speaking of n_l as large as possible) arises if we assume all processors have maximum allowed load, with respect to their "class": all $p \in P_l$ have load $\gamma_l m_\tau$, and all $\notin P_l$ have load $\gamma_h m_\tau$. This leads to

$$\underbrace{n_l \cdot \gamma_l m_\tau}_{\text{dangerous}} + \underbrace{(n-n_l)\cdot\gamma_h m_\tau}_{\text{non-dangerous}} = n \cdot m_\tau \iff n_l = n \cdot \frac{\gamma_h - 1}{\gamma_h - \gamma_l}$$

Let $\delta = \frac{\gamma_h - 1}{\gamma_h - \gamma_l}$. We can see that, as long as $\gamma_l < 1$, we can have at most some constant fraction ($\delta < 1$) of all processors being dangerous. From Lemma 6 we know that w.h.p. any T_p consists of at least $b^t/2$ relevant nodes. Now fix some T_p, and let $\mathcal{R} \subset T_p$ be a subset of T_p consisting of pairwise distinct nodes, $\mathcal{R} = b^t/2$. We now estimate the probability that there are too many dangerous nodes in $\mathcal{R}$. Note that the load of some fixed processor is not described by a random variable, our experiment is the selection of processors: we can view $\mathcal{R}$ as a randomly selected set. Of course, due to the nature of the collision game, the choices are not truly random. But we can eliminate dependencies by modifying the probability for a fixed processor to be dangerous: $(\delta n)/n \to (\delta n)/(n - |T_p|) \leq \delta'$ for some constant $\delta' < 1$ (and n large enough).

Now define random variables X_i, $1 \leq i \leq b^t/2$, with $X_i = 1$ if the i-th processor in $\mathcal{R}$ is dangerous, further let $X = \sum X_i$. Obviously, $E[X] \leq \delta' \cdot |\mathcal{R}| = \delta' \cdot b^t/2$. Application of Chernov bounds (see, for example, [10]) yield

$$\begin{aligned} & \text{prob}\,(X \geq (1+\epsilon)E[X]) \leq e^{-\epsilon^2 E[X]/3} \\ \Leftrightarrow \quad & \text{prob}\left(X \geq (1+\epsilon)\delta' b^t/2\right) \leq e^{-\epsilon^2 \delta' \cdot b^t/6} \end{aligned}$$

for any $\epsilon > 0$. If we choose ϵ such that $(1+\epsilon)\delta' < 1$ and $\epsilon^2\delta' \geq 96(\alpha+1)^3/\log n$, we obtain

$$\text{prob}\left(X \geq (1+\epsilon)\delta' b^t/2\right) \leq \left(\frac{1}{n}\right)^{\alpha+1}$$

So, with high probability at most some constant fraction ($(1+\epsilon)\delta' < 1$) of all nodes in $\mathcal{R}$ are dangerous, or, vice versa, at least an $1 - (1+\epsilon)\delta'$ fraction is not dangerous (i.e., has load at least $\gamma_l m_\tau$). Even if we now assume all dangerous processor to be empty, we now have a lower bound for l_p of

$$\begin{aligned} l_p & \geq \frac{(1-(1+\epsilon)\delta') \cdot \frac{b^t}{2} \cdot \gamma_l m_\tau}{|T_p|} \\ & = \frac{(1-(1+\epsilon)\delta') \cdot \frac{b^t}{2} \cdot \gamma_l m_\tau}{(b^{t+1}-1)/(b-1)} \\ & \geq \frac{(1-(1+\epsilon)\delta') \cdot \frac{b^t}{2} \cdot \gamma_l m_\tau}{2b^t} \\ & = \frac{(1-(1+\epsilon)\delta')\gamma_l}{4} \cdot m_\tau . \end{aligned}$$

Clearly, as $(1-(1+\epsilon)\delta')\gamma_l/4$ is a constant, we have shown that l_p is at least some constant fraction of the real average load, m_τ. If we now set $\beta \geq 4/[(1-(1+\epsilon)\delta')\gamma_l]$, the lemma follows.

□

Now we know that w.h.p. every l_p is within a constant factor of the real system load average (as been valid at the first time step of the current phase). The next lemma shows that our estimations are still valid at the end of the complete phase. We omit the proof due to space limitations (basically, we exploit the fact that in a phase of length T the average load may change by up to $\mathcal{O}(T)$).

Lemma 10 *Fix some phase Π. By τ' denote Π's last time step. If for every processor p we have that $\beta^{-1}m_\tau \leq l_p \leq \beta m_\tau$, then there is some constant β' such that ${\beta'}^{-1}m_{\tau'} \leq l_p \leq \beta' m_{\tau'}$.*

Now we show that the collision games are able to resolve the requests (since for the **load estimation** sub-phase we applied the same parameters as in the previous section, we refer to Observation 1). We omit the proof; it follows from the fact that we can bound the number of heavy processors as a constant fraction of n, which implies that the number of active requests also can be bounded by a constant fraction of n (at most twice the first number), because requests are issued only by heavy processors and light processors that already are assigned as balancing partners. Then apply lemma 2.

Lemma 11 *Given that in no collision game we have more than $\frac{n}{2(2\alpha)^{d+1}}$ requests to be resolved, for $d = a - b = 2(\alpha+1)$, all $\mathcal{O}(\log\log n)$ collision games succeed with high probability (in $\mathcal{O}(\log\log n)$ steps each).*

It remains to show that the **assignment** sub-phase actually works, that is, every heavy processor is exclusively assigned a light one afterwards.

Lemma 12 *Fix an arbitrary phase Π, beginning at time step τ. Provided that the system is balanced at the beginning of a Π, it remains balanced at the beginning of the next phase $\Pi + 1$.*

Proof (sketch) The strategy is rather straightforward. As the system is balanced at the beginning of Π we know that no processor has load exceeding $\gamma_h m_\tau$.

Furthermore, we know that with high probability the first collision games which is used to estimate the load average is able to resolve the requests. This follows directly from Lemma 1. Then, Lemma 10 shows that our estimation of the average load is within a constant factor of the true average.

We classify a processor as heavy (light) if its load is at least $\gamma_h l_p$ (at most $\gamma_l l_p$), for constants γ_h and γ_l. Application of a simple pigeonhole argument shows that we can have no more than a constant fraction of all processors being heavy, where the constant depends on γ_h and β' (note that we can arbitrarily scale this constant, i.e., make it as small as necessary). A similar argument shows that we have at least have a constant fraction of light procesors, where again the constant is scaleable.

The successes of the remaining collision games used to find balancing partners for heavy processors follow from Lemma 11, as we always can bound the number of requests by some constant times n. We have estimated the number of light processors to be at least some constant fraction. So, it suffices to "ask" about $\log n$ processors to be able to assure that with high probability one is found. Since we play (about) $\log\log n$ collision games, we ask this number of processors for every heavy one. Note that, to reach this result, we are able to scale the constants from the estimation of the number of heavy and light processors such that we obtain a high probability bound of at least $1 - (1/n)^\alpha$ for the event that if the system is balanced for one phase, it remains so for the next poly(n) phases. □

From the above lemma it is easy to conclude Theorem 2, which says that with high probability the system remains balanced for the next poly(n) phases if it is balanced for a fixed phase Π.

Unfortunately it was not possible to say that the system will be stable at some arbitrary point of time, as been done in Theorem 1. The main difference – or problem – is that, due to the generation process of Section 3, Lemma 4 assured that at any point of time the system load is bounded, whereas there is no such statement possible for generation model UNBOUNDED.

So far, we always assumed a stable system at the beginning of a phase. Of course, if this is not the case, i.e. if the system is in an arbitrary state (meaning arbitrary load distribution), one cannot show that after just one phase of our algorithm the system is stable again. But we can show that if we have an arbitrary state, after expected poly(n) steps the system will be stable again. This works by defining a potentional function on the load distribution and by showing that our algorithm decreases the potentional such that it converges against a value that implies stability. We omit the proof due to space limitations.

5 Remarks

You may or may not believe it, but simulations show that both algorithms perform pretty well.

References

[1] M. Adler, P. Berenbrink, and K. Schröder. Analyzing an infinite parallel job allocation process. In *Proceedings of the 6th European Symposium on Algorithms*, 1998.

[2] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of the 27th Symposium on Theory of Computing*, pages 238–247, New York, NY, USA, May 29–June 1 1995. ACM Press.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the Twenty-Sixth Symposium on Theory of Computing*, pages 593–602, Montréal, Québec, Canada, 23–25 May 1994.

[4] P. Berenbrink, T. Friedetzky, and E. W. Mayr. Parallel continous randomized load balancing. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 192–201. ACM, 1998.

[5] P. Berenbrink, F. Meyer a.d. Heide, and K. Schröder. Allocating weighted balls in parallel. In *Proceedings of the 9th Symphosium on Parallel Algorithms and Architectures*. ACM, 1997.

[6] R. Cole, A. Frieze, B. M. Maggs, M. Mitzenmacher, A. W. Richa, R. K. Sitaraman, and E. Upfal. On balls and bins with delitions. In M. Luby, J. Rolim, and M. Serna, editors, *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 145–158, 1998.

[7] A. Czumaj. Recovery time of dynamic allocation processes. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*, 1998.

[8] A. Czumaj and V. Stemann. Randomized allocation processes. In *Proceedings of the 38th Symposium on Foundations of Computer Science*. IEEE, 1997.

[9] S. K. Das, D. J. Harvey, and R. Biswas. Design of novel load balancing algorithms with implementations on an ibm sp2. In *Proc. of the 3rd EURO-PAR Conference*, 1997.

[10] T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Information Processing Letters*, pages 305–308, 1990.

[11] C.-J. Hou and K. G. Shin. Implementation of decentralized load sharing in networked workstations using the Condor package. *Journal of Parallel and Distributed Computing*, 40:173–184, 1997.

[12] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulations on a distributed memory machine. In *Proceedings of the 24th Symposium on Theory of Computing*, pages 318–326. ACM, 1992.

[13] T. Lauer. *Adaptive Dynamische Lastbalancierung*. PhD thesis, Universität des Saarlandes, 1995.

[14] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*, pages 164–172. ACM Press, 1993.

[15] N. R. Mahapatra and S. Dutt. Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing. In *Proc. of the 10th International Parallel Processing Symposium (IPPS)*, pages pages 881–885, 1996.

[16] F. Meyer auf der Heide, C. Scheideler, and V. Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science*, pages 267–278. Springer, 1995.

[17] M. Mitzenmacher. Load balancing and density dependent jump markov processes. In *Proceedings of 37th Symposium on Foundations of Computer Science*, pages 213–223. IEEE, oct 1996.

[18] M. Mitzenmacher. On the analysis of randomized load balancing schemes. In *Proceedings of the 9th Symphosium on Parallel Algotithms and Architectures*. ACM, 1997.

[19] M. Raab and A. Steger. Balls into bins — a simple and tight analysis. In *2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM'98)*, pages 159–170, 1998.

[20] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures*, pages 237–245. ACM Press, 1991.

[21] T. Schnekenburger and G. Stellner. *Dynamic Load Distribution for Parallel Applications*. Teubner, 1997.

[22] V. Stemann. Parallel balanced allocations. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 261–269. ACM, 1996.

[23] K. S. Trivedi. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*, chapter 7, pages 309–360. Prentice-Hall, 1982.

[24] G. S. Wolffe, S. H. Hosseini, and K. Vairavan. Performance of an adaptive algorithm for dynamic load balancing. In *Proc. of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 613–618, 1995.