J Grid Computing (2018) 16:647–662
https://doi.org/10.1007/s10723-017-9411-5

CrossMark

Parallel Tree Search in Volunteer Computing: a Case Study

Wenjie Fang · Uwe Beckert

Received: 20 April 2017 / Accepted: 3 October 2017 / Published online: 23 October 2017
© The Author(s) 2017. This article is an open access publication

Abstract While volunteer computing, as a restricted model of parallel computing, has proved itself to be a successful paradigm of scientific computing with excellent benefit on cost efficiency and public outreach, many problems it solves are intrinsically highly parallel. However, many efficient algorithms, including backtracking search, take the form of a tree search on an extremely uneven tree that cannot be easily parallelized efficiently in the volunteer computing paradigm. We explore in this article how to perform such searches efficiently on volunteer computing projects. We propose a parallel tree search scheme, and we describe two examples of its real-world implementation, Harmonious Tree and Odd Weird Search, both carried out at the volunteer computing project yoyo@home. To confirm the observed efficiency of our scheme, we perform a mathematical analysis, which proves that, under reasonable assumption that agrees with experimental observation, our scheme is only a constant multiplicative factor away from perfect parallelism. Details on improving the overall performance are also discussed.

W. Fang (✉)
Institute of Discrete Mathematics, Technical University of Graz, Steyrergasse 30, Graz 8010, Austria
e-mail: fang@math.tugraz.at

U. Beckert
Rechenkraft.net e.V., Marburg, Germany
e-mail: yoyo@rechenkraft.net

Keywords Volunteer computing · Parallel tree search · Power law · Performance · Implementation

1 Introduction

Since the founding of one of the first volunteer computing projects, Great Internet Mersenne Prime Search (GIMPS), in 1996 [28], the paradigm of volunteer computing has been gradually established as a cost-efficient means of scientific computing with good public outreach. This trend has been sped up by the appearance of the middle-ware BOINC [3], with various successful projects, including SETI@home [4] and Einstein@home [1]. Umbrella projects have also appeared, such as yoyo@home [31] held by a volunteer community in Germany. Today, researchers from various fields, including astrophysics, structural biology, high energy physic and mathematics, set up volunteer computing projects to solve problems in their own fields.

Despite the ever-growing number of projects, most of the problems they solve are intrinsically parallel, for example checking a segment of radio wave for specific signals (Einstein@home, SETI@home). However, there are many large problems, especially those in discrete mathematics aiming at verifying conjectures or finding solutions, that have no efficient parallel algorithm. For these problems, variants of backtracking search are used, which often consist of searching through an extremely uneven tree, and are

thus difficult to parallelize. It is thus interesting to try to parallelize efficiently massive tree search on uneven trees in the volunteer computing paradigm, in order to extend the applicability of volunteer computing to previously unsuitable domains.

Some previous effort has been devoted to such parallelization of tree search on specific problems, resulting in several public projects. For instance, Rectilinear Crossing Number is a project held by researchers at TU Graz, with the goal of finding the rectilinear crossing number for $n \leq 18$ points using a backtracking algorithm [2]. As another example, the OGR sub-project of Distributed.net searches for optimal Golomb rulers with $n \leq 28$ markings [15], and it also uses a backtracking algorithm for its purpose. Despite the effort, from a volunteer's viewpoint, these projects have their inconvenience. We are therefore interested in catering the needs of volunteers while doing an efficient parallel tree search.

Tree search consists of traversing all the nodes in a given tree. It includes the special case of backtracking search, which is a general algorithm that can be applied to many problems, mainly in discrete optimization (see, e.g., Chapter 8 of [5]). In general, a backtracking search often has an extremely unbalanced search tree, which makes it difficult to parallelize. Upon the development of distributed systems, researchers tried to find ways to exploit parallelism in backtracking search, mainly in clusters. Studies on the parallelization of backtracking search were pioneered by Karp and Zhang [27], followed by others (e.g., [18, 24, 32, 33]). Real world implementations of such parallel search are performed by many (e.g., [13, 16]) for finding combinatorial objects or verifying conjectures. We also see development of software infrastructures (e.g., Stolee's *Treesearch* [35]) that allows automation of such searches. Readers can also see [22] for a survey of some work in this direction. The overall idea is to split the search tree into sub-trees and distribute them to workers while balancing workload.

However, the previously stated effort is partly superseded by advances in efficient scheduling, which is a well-studied topic in distributed systems. It is easy to see how parallelization of tree search can be done simply by scheduling search on sub-trees as tasks, which can be further subdivided if needed. For efficient scheduling, much has been done in environments such as multi-core processors and clusters, for example in [7, 9], where workers are highly available and can communicate with low cost and latency. There are also well-established APIs (such as OpenMP [12]), languages (such as Cilk [8]) and job schedulers (such as HTCondor [36]) in the industry for efficient scheduling in such environments.

In volunteer computing, the reduction of parallelizing tree search to efficient scheduling is not so straightforward. It is because the task of scheduling itself becomes tricky in this paradigm, in which workers can have low and variable availability [25, 29], and they only pull work from a central server, but never talk to peers. These constraints restrict the use of established techniques such as work stealing [9]. Recently, there has been some study on efficient scheduling in volunteer computing, such as [17, 23, 30]. However, these results may not apply directly to our goal, since they concentrate more on selecting good hosts for a given set of tasks with already a good estimation of workload, but in parallel backtracking search, we need to deal with tasks with high variance but no estimation on the workload.

Without a suitable scheduling method in volunteer computing, we may want to resort to previous studies on parallelizing tree search that do not rely on existing scheduling methods. However, many of these studies rely more or less on a cluster model, where inter-processor communication is easy (maybe with exception of [18, 32]), which do not directly apply to the volunteer computing paradigm. In [32], the parallelization was done by splitting the whole search space into a huge number of tasks, at least a large constant times the number of processors. These tasks are then put into a queue and dealt out to processors to achieve maximal parallelism until the queue is dried up. This approach can be easily adapted to volunteer computing, but a straight-forward subdivision may introduce extremely large tasks, unwanted for volunteer machines with varying availability. In [18], by introducing an overhead of splitting the search tree locally and deterministically upon each task, the work balancing is dealt with by selecting sets of non-overlapping sub-trees for each task. This approach can also be easily adapted to volunteer computing, but the huge size of problems we deal with in volunteer computing projects may lead to a large overhead, sabotaging the effectiveness of this approach. Moreover, all these studies deal with machines in a cluster, managed by technical people, but in volunteer computing, machines are managed by laypeople, and we have

to consider some “human factors” to attain maximal efficiency.

Another “human factor” in volunteer computing is malicious behavior, which may endanger the integrity of final result. This problem is more important for parallel backtracking search, since most of the results will be negative, which gives malicious users an incentive to cheat by returning negative results without performing any real computation. Anti-cheating mechanisms are not new in volunteer computing. For instance, in [20], an anti-cheating mechanism called “ringers” was introduced in the context of cryptological computations, and in [34], a version of “client reputation” was used to mitigate cheating behavior. However, these existing approaches are either too domain-specific, or unable to essentially eliminate cheating. We also need to deal with the cheating issue here.

The main contribution of this article is a scheme that we propose to parallelize massive tree search on extremely uneven trees in the volunteer computing paradigm, which includes the useful case of backtracking search in constraint satisfaction problems. Our scheme is similar to that in [32], consisting of simply dividing the search tree into many sub-trees, but we mitigate the problem of extremely large tasks by allowing workers to report partly finished results in the form of checkpoints as in [10]. In our scheme, we also propose some practices on the side of “human factors” of volunteer computing, mainly showing progression in time to reassure volunteers, and preventing cheating. We then present two case studies of implementation, which indicate the effectiveness of our scheme. Finally, in order to provide formal evidence of the efficiency of our scheme, we propose and analyze a simplified mathematical model of the behavior of our scheme, and we show that, under reasonable assumptions and good choice of parameters, the performance of our scheme is only a constant factor away from the perfect parallelism. This analysis indicates that our scheme can still achieve good parallelism despite all the complications due to unreliable workers in volunteer computing. The analysis also provides a guide on how to choose crucial parameters in the scheme.

For the organization of this paper, in the following, we first postulate some requirements we want for a parallel tree search in the volunteer computing paradigm (Section 2), then we propose a scheme accommodating our needs (Section 3). We then discuss our implementations on the volunteer computing umbrella project yoyo@home (Section 4), named Harmonious Tree and Odd Weird Search. In order to understand the behavior of our scheme and to explain the phenomena observed in our implementations, we analyze a simplified mathematical model for practical choices in our model (Section 5). We conclude by some discussions of future improvements on the implementation (Section 6).

Upon the writing of this article, we are aware of an effort by Bazinet and Cummings [6] of subdividing long computation into workunits with a fixed short length, which shares some ideas with our work. However, the motivation of their work is to reduce the variance of workunit running time to achieve higher throughput and faster turnaround time of batches, which is radically different from ours. Nevertheless, their work shows that our approach may have extra benefit that we have not considered.

2 Problem Statement

Tree search, or tree traversal, is an elementary algorithm in computer science. It consists of going through all the nodes of a tree in some order. The tree can be given explicitly or implicitly. One of its most important applications is the backtracking search, which is a general search method used in various contexts, especially in constraint satisfaction problems. Due to its importance, we will discuss tree search mostly in the case of backtracking search, but the discussion on parallelization will be applicable to the general case of tree search.

In the context of constraint satisfaction, a backtracking search is equivalent to a tree search on a tree whose nodes are valid partial assignments, which are those not violating any prescribed constraints. The root of the tree is the empty assignment, and the children of a node are all the valid one-step extensions of the corresponding partial assignment. The backtracking search thus consists of searching this partial assignment tree for a valid full assignment. It is a complete search method, since it always explores the whole search space and gives an exact answer on whether there is a solution to the given problem. To accelerate the search, by early detection of dead-end nodes (see Fig. 1), we can prune a larger part of the search tree, thus increasing efficiency. With judicious pruning method, backtracking search can be used to give efficiently a definite answer to many constraint

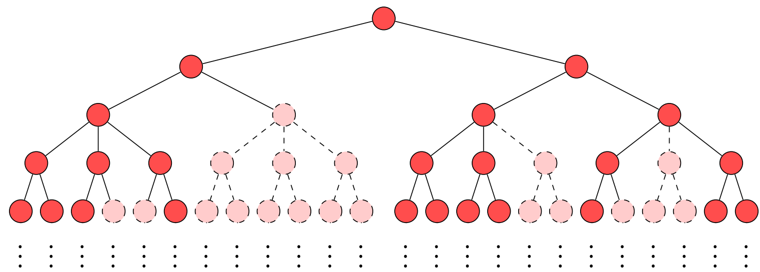

Fig. 1 A search tree, with some dead-end partial assignments pruned

satisfaction problems, especially those in combinatorial optimization and conjecture verification.

Ironically, when backtracking search is needed, it is almost always the case that no efficient (i.e. polynomial) algorithm exists for the problem that finds a solution or establishes the absence of solution, and we can only explore the exponentially large solution space using backtracking. Therefore, a parallelized backtracking search, which is a special case of parallel tree search, will be interesting due to the enormous speedup. Apart from the conventional supercomputer approach, we are more interested in the paradigm of volunteer computing, which is as massive in computational power but more cost-efficient. In volunteer computing, a computational problem is broken into small pieces called *workunits*, then distributed by a server to clients on delocalized volunteers' machines to be computed, without communication between clients. When workunits are completed, they are sent back to the server to be reconstructed into the solution of the original problem.

Another specialty of volunteer computing is that the "human factor" of volunteers is as important as the "machine factor", since any practice that damages volunteer experience will quickly lead to loss in computational power. A parallel tree search scheme with consideration of human factor is thus needed in the volunteer computing paradigm.

Therefore, in addition to being massive parallelization with low workunit generation overhead, more requirements concerning the human factor should be added in the quest of the targeted parallel tree search model. Here is a tentative list of such requirements in the general context of volunteer computing, drawn from inputs from volunteers and the need of scientific research.

1. **Checkpoint**: Reliability and availability of volunteers' machines are far from those of professionally managed clusters. Therefore, to avoid waste of resources and frustration of volunteers, checkpoints are mandatory. Intervals between checkpoints should be neither so long that they induce potentially large waste in accidents, nor so short that harm performance.
2. **Moderate running time**: Also due to reliability and availability issues, extremely long workunits are to be avoided, since they are more prone to error and lead to greater volunteer frustration when errors occur, although these inconveniences can be alleviated by some additional mechanisms, for instance the trickle mechanism in BOINC. Extremely short workunits can also damage user experience through frequent communication with the server, but less seriously if mixed with longer workunits. An estimated recommendation of running time is from 1 to 12 h.
3. **Progress bar**: Volunteers love to see the progress of their workunits, which gives them a sense of certainty that the workunit is being processed without error. An application without progress bar may lead to frustration of volunteers due to uncertainty. However, the progress bar does not need to be precise, since it is used only as a reference.
4. **Validation**: Due to the reliability issue and possible malicious participants, a way to verify the integrity of each result is needed to guarantee that the search is completed without missing any possible solution.

These requirements show a real challenge for parallel backtracking search, since the efficiency of backtracking search lies in a good pruning strategy, which also makes run-time unpredictable, as the pruning of useless large sub-trees is known only at run-time. Hence, there is no easy and good estimate of workunit running time and progress, which prevents running time control and progress bar. Existing practices often sacrifice some of these requirements. For example, the Rectilinear Crossing Number project is known for highly unbalanced workunit run-time that ranges from

a few seconds to a few hundred hours, a phenomenon that already appeared for smaller cases [2]. The same also happens to the previous OGR sub-project of Distributed.net, which is alleviated in the new OGR-NG framework by grouping potentially small workunits into a larger one, which can also lead to relatively large grouped workunits. The two projects also lack a satisfying progress bar. We thus want a parallel tree search scheme on extremely uneven trees that is capable of satisfying all the stated requirements.

3 Our Parallel Tree Search Scheme

We now propose a parallel tree search scheme satisfying all the stated requirements. Our scheme is based on a simple idea: smoothing out the variation of sub-tree size by dividing the whole search tree into many small sub-trees (called *sections* hereinafter). Then a fairly large number of adjacent sub-trees can be packed into each workunit, reducing the variance of workunit size. We now try to devise a model based on this idea to meet the four requirements (checkpoint, running time, progress bar, validation) mentioned in the previous section. The checkpoint requirement is easily met by writing the current branch of search into the checkpoint file.

We now focus on the running time requirement. There is a compromise to take between dividing the whole search tree into as many small sections as possible, and the computational power consumed during workunit generation. One solution (taken by distributed.net on OGR-NG projects, see [14]) is to off-load the workunit generation to clients to harness greater power for finer-grain decomposition, in exchange of possible complication of client code and project structure. To avoid such complication, we choose to take the compromise head on, by restricting ourselves to generating workunits locally with moderate computational power.

Not going deep enough in the search tree when generating workunits has its problem, that is, section sizes can vary greatly, introducing giant workunits. To meet the running time requirement, instead of trying to search deeper in workunit generation for a finer-grain decomposition, we instead use the following strategy: when a workunit has been run for a given amount of time on a volunteer's machine, it should stop prematurely, write a checkpoint and send it back to the server as a result. To minimize network traffic, we suppose that the checkpoint file is small, for instance less than 10KB. This limit suffices in general for most constraint satisfaction problems, as we only need to record a partial assignment, which is typically small. Since the checkpoint contains all the necessary information to continue the search, the server can use it to produce a new workunit to send out. We call it the *recycle mechanism*. We should note that the idea of transferring workunits in distributed environments similar to volunteer computing is not new. For instance, in [10] there is an approach of scheduling in a desktop grid that migrates tasks in the form of checkpoints.

Now it suffices to find a good measure of the workload already accomplished by the volunteer's machines during run-time. However, it complicates if we also consider the validation requirement. We would like to verify volunteers' result, which has no apparent pattern due to the complex structure of the search tree, and we are left to the only option to compare verbatim the results generated by two different machines from two different volunteers. In the case where the computation involves float-point arithmetic, verbatim comparison may fail for legitimate results from different computing architectures. However, in the case of tree search, in general float-point arithmetic is not involved or can be avoided, so we can use verbatim comparison here. If we use running time as the workload measure, the heterogeneity of volunteer machines will guarantee no possible verbatim match results. Therefore, we have to find a measure of workload that does not change between machines, something intrinsic to the computation itself. We do not need the measure to be precise, it can be off by a few percents, or even up to 20% or 30%.

The best way to implement such a measure is to count the number of execution of a certain routine that is representative for the whole computation, preferably one that uses a considerable but roughly constant amount of computation, to minimize the extra cost of updating measure counter. In tree search, a natural choice of such a measure is the number of nodes we have traversed. However, in some cases, the amount of computation needed for each node may differ greatly, rendering this natural choice less realistic. In these cases, we need to choose a measure that is adapted to the algorithm we used. We will see the choice of such routine in the case studies in the following section. In this way, if the chosen routine is

representative enough, we will have a workload measure that is intrinsic to the computation code and identical across all machines. We thus set the stopping requirement as that this workload measure reaches a certain fixed amount, empirically set beforehand using a reference machine. Always due to the difference of float-point arithmetic on various architectures, we demand the workload measure to be an integer. In this way, we reach a design that practically meets the running time requirement combined with the validation requirement using verbatim comparison.

We notice that the solution above also meets the progress bar requirement automatically. We only need to announce the progress using our workload measure. Since we have a good approximation of the real workload, the progress bar will also only be slightly off. We therefore have a scheme that meets all four requirements.

It was pointed out by an anonymous referee that checkpoint files may create security holes that allow malicious participants to inject code into the server or other volunteers' machine. This issue is also shared by other efforts in volunteer computing, as participants are allowed to send information to the server. However, by properly sanitizing the input during coding, we can close the security hole. This is easy to implement for tree search in particular, as the checkpoint file is small (less than 10 KB), with a predefined structure and a predefined limit for each element of the checkpoint. To further protect volunteers, when recycling workunits, the server can check the sanity of returned checkpoint files (for example by checking their sizes and structural integrity) and throw away suspicious ones. Verbatim comparison also serves as a barrier for this type of attack.

Below we recapitulate the details of our model. We suppose that the following conditions can be met in our tree search.

- **Checkpoint**: It is possible to construct a fairly small (e.g., not exceeding an explicit limit, preferably less than 10KB) checkpoint file with which the same application can pick up the search. For simplicity, we impose the extra requirement that checkpoint files have the same structure as initial input files.
- **Workload measure**: There is a representative routine that can be used as an approximate workload measure in run-time.

Such assumptions are satisfied in many backtracking search algorithms for constraint satisfaction problems, and more general tree searches. Under these assumptions, our scheme of massive parallel tree search can be described as follows.

- **Workunit generation**: On the server side, with a moderate amount of computational power, we break the search tree into sections using a depth-limited tree search. If we have no prior knowledge on the search tree, a possible solution is to use iterative deepening. We then pack many adjacent sections into one *initial workunit*. We thus generate the batch of initial workunits.
- **Workunit execution**:
  - **Server**: The server sends out the initial workunits and collects results. Upon receiving a result that is an unfinished checkpoint file, the server checks its sanity and repackages it as a new workunit, called a *recycled workunit*, and send it out again. Each workunit, initial or recycled, generates at least two replicas. Validation of the result is done by verbatim comparison of the result files of these replicas.
  - **Client**: The client computes all workunits in the same way, regardless whether they are initial or recycled ones. There is an explicit workload limit set in the client, and the client maintains a workload counter using the approximate workload measure. If the workunit terminates before reaching the workload limit, the client simply sends back the end result. Otherwise, when the limit is reached during computation, the client halts, writes a checkpoint and sends it back as the result.
- **Convergence phase**: When the number of available workunits in the system has lower to the point that it is no longer possible to keep most of the clients busy, we enter the convergence phase. The server then sets shorter deadline, sends out more replica than necessary or distributes workunits only to ”fast clients” to get faster returns. The server may eventually stop distributing workunits, and computes them locally instead.

Figure 2 illustrates the workunit execution workflow. In this model, each initial workunit leads to a succession of recycled workunits, which is called its *lineage*. The parallelism of this model is only limited by the number of lineages, since workunits in the same lineage must be computed sequentially. As long as there are enough unfinished lineages for all clients, we reach full parallelism potential of the distributed system. However, when we enter the convergence phase, there will not be enough workunits for every client, which hurts the throughput. We thus need extra effort for mitigation. We should note that the notion of convergence phase is not mathematically defined, but can be seen in the practice as a shortage of workunits.

There are some empirical parameters to be tuned in our scheme, for example the choice of the number of initial workunits. These parameters may influence the efficiency of the parallel search.

4 Case Study

We now access the performance of our parallel tree search scheme in detail with two case studies of implementation. Our scheme has been implemented as two applications for different problems in discrete mathematics, run under the volunteer computing project yoyo@home. The first application is called Harmonious Tree, which aims to verify a conjecture in graph theory. The second one is called Odd Weird Search, which aims to find an odd weird number, a class of numbers defined in number theory whose existence is still unsettled. The two problems can be modeled as search problems with uneven search trees, and they both require a large amount of computational power to solve, making them good candidates for experimental evaluation of our scheme. The first application was a partial success, while the second application fully demonstrated the viability of our scheme.

4.1 Case 1: Harmonious Tree

A *tree graph* (or simply a *tree*) is a simple acyclic graph. Let $G = (V, E)$ be a tree graph with $n = |V|$ vertices. A *labeling* of G is a bijection $f : V \to \{1, 2, \ldots, n\}$ from vertices to integers from 1 to n. A labeling f is called *harmonious* if the induced labeling on edges, which is the absolute difference of the labels of vertices adjacent to an edge, is also unique for each edge. It was conjectured that every tree graph is harmonious. The effort of proving this conjecture has so far yielded few results, with no known counterexample. A more detailed summary of the current status of this conjecture can be found in [19].

In the application Harmonious Tree, we perform the verification of the aforementioned conjecture on tree graphs whose number of vertices varies from 32 to 37, using essentially the same procedure in the arXiv preprint 1106.3490 of one of the authors, in which the verification for trees with at most 31 vertices was done. The whole computation was done during the years 2011–2013. In our algorithm, trees are represented as the “level sequence” of its canonically rooted version. We use the algorithm in [37] to generate all trees with a given number of vertices under the level sequence representation with constant amortized time. Given a tree, this algorithm generate the next tree in lexicographical order. For each generated tree, we try to find a harmonious labeling using the algorithm in the arXiv preprint 1106.3490, composed by a randomized backtracking search and a stochastic search. All randomness used here is provided by a customized random number generator and a random seed given in the workunit, which makes the computation

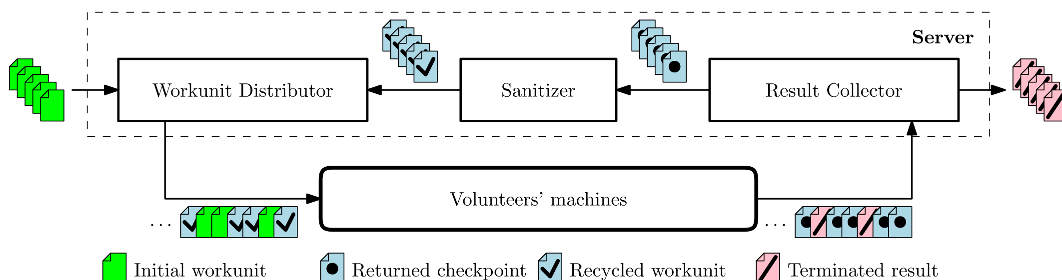

Fig. 2 An illustration of our parallel tree search scheme

deterministic. The numbers of backtracks in the backtracking search part and iterations taken in the stochastic search part are jointly used as workload indicator. We should note that the time taken to find a harmonious labeling for each tree varies greatly, which is why we did not use the number of trees processed as workload indicator. The random seed after the run also serves as a defense against malicious users, since its value is highly correlated to the whole computation.

To parallelize the verification, we pack trees into workunits. It is possible to evenly distribute trees by generating all trees on the server side. However, since the number of trees grows exponentially with the number of vertices, the computational power needed quickly explodes beyond the affordable limit on the server side. We thus need another way to generate workunits with less resources. Our solution is to package trees sharing the same first sub-tree in the level sequence representation into a section, then several sections into an initial workunit. Since the number of possible first sub-trees is much smaller (but still large) than the total number of trees, the computational cost of their generation is manageable. However, the number of trees contained in each section can vary greatly.

The Harmonious Tree project had 6 batches, each deals with trees with a given number of vertices. Since our method was gradually developed during this subproject, only in the last batch for $n = 37$ was our method fully applied. Therefore, only data from the last batch contributes to the analysis of our method. Unfortunately, the importance of time series statistics was underestimated at that time, and no useful statistics can be drawn from the available data. We here only report some qualitative aspects of the batches.

During the batches, volunteer's computing power stayed fully utilized except for the convergence phase. Average running time of a workunit stabilizes at around 3 h, with occasional large fluctuations. This is due to some extremely large workunits. In fact, in the last batch ($n = 37$), there were around 40 workunits that took too long for volunteers to finish, and are thus computed locally, which has taken a considerable amount of time (some reached several months). These workunits comes from the incorrect assumption that the number of trees with the same first sub-tree is independent of the first sub-tree itself. In fact it is largely dependent, and sections with similar first sub-trees tend to be of similar size. Furthermore, similar first sub-trees tend to be grouped together in the same initial workunit. The clustering of sections of similar sizes thus created extremely large workunits, much larger than what we expect when large sections are distributed randomly.

On the human factor side, there was also serious design flaw on the progress bar, which ignored the recycling mechanism, reporting the progress in the lineage instead of the workunit, thus made progressing workunits look like stagnant. It generated instability in the volunteer community. In conclusion, our first implementation of our parallel tree search scheme on Harmonious Tree was not entirely satisfactory, and the lack of data makes it hard to access the performance of our model through this example. Nevertheless, this first implementation still satisfies some of the prescribed requirements, such as moderate running time and integrity, and it successfully reached its computational goal. Therefore, we can still conclude that our scheme applies. Furthermore, we should note that our scheme was still in development while Harmonious Trees is in progress, and we have learned a good lesson about how delicacies in our scheme should be dealt with to get good performance, allowing us to extract full potential of our scheme in the second case study.

4.2 Case 2: Odd Weird Search

In number theory, an *abundant number* is a number with the sum of its proper factors larger than itself. A *weird number* is an abundant number that has no subset of its proper factors that sums to itself. An example of weird number is 70. An *odd weird number* is a weird number that is odd. The existence of an odd weird number is an open problem in number theory.

In the application Odd Weird Search, we search for odd weird numbers smaller than a certain upper bound. To check if a number is weird, we first compute all its proper factors to see if it is abundant, then use standard techniques to solve a subset sum problem of the set of its proper factors with the number itself as target. To get all the proper factors of a number, we need its full factorization, which is expensive to directly compute. Hence, instead of going through numbers in increasing order and factorizing each of them, we explore integers according to their factorization.

For a natural number n, its unique factorization can be written as $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where $k \geq 0$ is the number of distinct prime factors, $p_1 < p_2 < \cdots < p_k$

primes and $e_i \in \mathbb{N}^*$. Via this representation, we can thus endow $\mathbb{N}^*$ with a tree structure by appointing that the parent of $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ is np_k^{-1}. By traversing this tree structure, we can visit every number with free access to its factorization. Furthermore, this tree structure is also suitable for the search of weird number. A semi-perfect number is an abundant number that is not weird. We know that any multiple of a semi-perfect number is also semi-perfect. Therefore, in the traversal of the search tree, once we meet a semi-perfect number, we can backtrack.

The search strategy above has only one drawback, that is, the search tree for numbers in the interval $[1, n]$ has a highly unpredictable and uneven tree structure due to the involved multiplicative structure of integers. It is therefore a problem particularly suitable for our parallel tree search scheme. Using our scheme, we have checked all numbers less than 10^{21} and numbers less than 10^{28} with an excess (i.e., sum of proper divisors minus itself) less than 10^{14}, but no odd weird number was found. The two batches were done during the years 2013–2015.

Having learned from the experience of Harmonious Tree, we have a better design for workunit generation and for progress bar this time. The workunit generation is solved with the change of problem, since in this search problem, proximate search sub-trees do not correlate so much in size, due to the intrinsic quasi-randomness of integer factorizations. Therefore, the correlation between sizes of nearby search sub-trees is weaker, which gives in turn more evenly distributed workunit sizes. There were still some large workunits that needed local computation, but they only took a few days instead of months. For workload measure, we used the sum of sizes of subset sum problems we solve, which also serves well as an integrity check, since it is deeply correlated to the computation. Progress bar was computed using both the internal workload measure versus the threshold and the percentage of finished sections among all sections in the workunit. In this way, both types of progress are represented, which satisfies the volunteers.

We now present a statistical analysis of the batches. Table 1 lists some data about these two batches, while Fig. 3 illustrates the distribution of lineage length in each batch. Workunits in the first batch typically take 1.5 h on a modern machine, while those in the second one take roughly 6 h. Therefore, the total workload of the second batch is 1.5 times that of the first one.

Table 1 Characteristics of the two batches of Odd Weird Search

	First batch	Second batch
Number of lineages	41776	47780
Number of workunits	527191	232800
Average lineage length	12.62	4.87
Median lineage length	6	3
Maximum lineage length	333	496

We first look at some general statistics presented in Table 1. In the first batch, the longest lineage was recycled 333 times, while in the second one, the longest lineage was recycled 496 times. In contrast, the average length is 12.62 for the first batch and 4.87 for the second. We conclude that most of the lineages are relatively short, but there are also a few lineages significantly longer than others, suggesting a heavy tail distribution of lineage lengths.

We now look at Fig. 3 for the detailed distribution of lineage lengths. We observe that they seem to follow a power law distribution, which means their distribution comes with a heavy tail. More precisely, let $p(k)$ be the proportion of workunits of length k, then we have

$$p(k) \propto k^{-\alpha}, \tag{1}$$

where $\alpha > 0$ is the *exponent* of the power law. However, the power law assumption works less well on the number of lineages of length 1 or 2. This deviation is typical for naturally arisen distributions, and is also reasonable in our case, since when we pack the lineages, we make sure that few lineages take trivial time to be computed. We also observe that the second batch has a smaller average length of lineages but a larger maximum lineage length than the first batch, which means that the second batch has a longer tail, thus a smaller exponent. By regression, we have $\alpha \approx 4.09$ ($R^2 = 0.86$) for the first batch, and $\alpha \approx 2.69$ ($R^2 = 0.98$) for the second batch (taking only lineages of medium-to-long length). These values agree with the observation that the second batch has a longer tail. Furthermore, the exponents fall in the regime $\alpha > 2$.

We suspect that the power law behavior is a general phenomenon for many search problems treated with backtracking search. Further supporting arguments are laid out in Section 5. Since if we ignore the convergence phase, the time taken for the batch will roughly

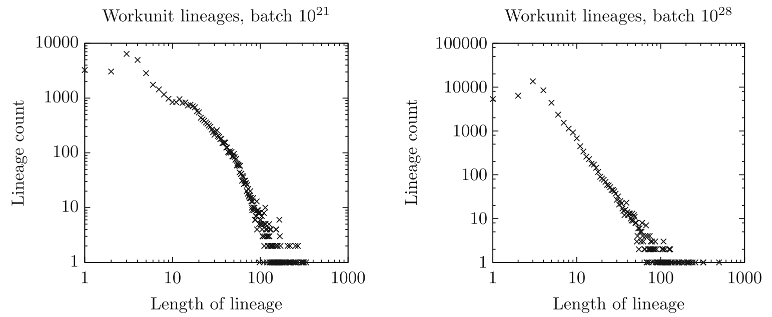

Fig. 3 Distribution of lineage lengths for the two batches 10^{21} and 10^{28}

depend on the longest lineage, to reduce the total computation time, we should try to mitigate the influence of the heavy tail.

In Odd Weird Search, we were also able to record some interesting time series statistics during each batch run. Figure 4 shows the daily workunit statistics for the two batches, with the number of returned workunits, the number and the proportion of completed workunits. The surge of the percentage of completed workunits at the end of the second batch was due to a re-issue of missing workunits. We note that, since the computational power donated by volunteers is nowhere constant, the number of returned workunits cannot be used to determine the convergence phase, where the consumed computational power ought to be lower due to limited parallelism. This variation of computational power is especially pronounced at the end of the second batch, as a curious result of the badge system of yoyo@home. In some volunteer computing projects, when the contribution (accumulated computation time, normalized credit or financial donation) of a volunteer reaches certain thresholds, a badge (sometimes taking the shape of a medal or a trophy) will be displayed on pages related to the volunteer as a recognition. The data of these badges are often exported and made into signatures in online forums, as a proof of contribution. Therefore, badges are generally considered desirable by volunteers. In the case of yoyo@home, each application has its set of badges,

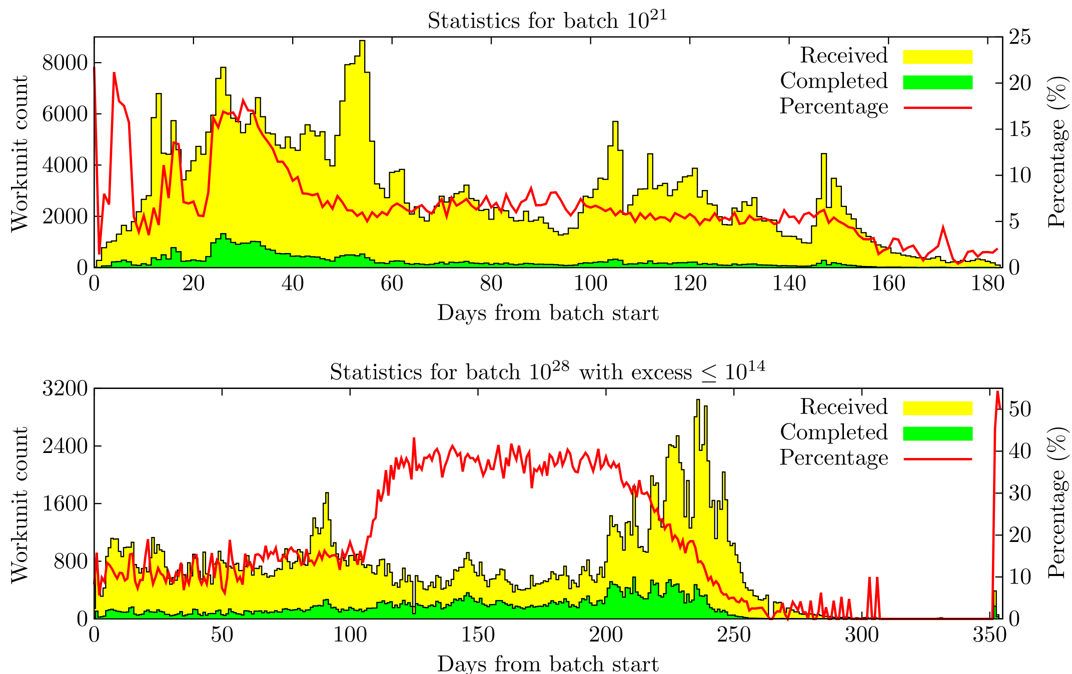

Fig. 4 Daily workunit statistics of the two batches 10^{21} and 10^{28}

using credits (a normalized measure of donated computational power) as evaluation. When entering the convergence phase, some volunteers observed the low workunit supply and the shortened deadline, but they still need to process some more workunits to get the next badge. As a consequence, they put more computers (thus more computational power) to our sub-project, which induced a peak at the end of the batch. This badge-scooping behavior helps shorten the convergence phase, and is also an interesting phenomenon coming from the unique human factor of volunteer computing.

To identify the convergence phase, the percentage of finished workunit among returned workunits may be a better indicator. In the main processing phase, this percentage should remain roughly constant, since we are dealing with the bulk of the batch with full parallelism. However, in the convergence phase, we should expect the same percentage to slowly decrease, since we will be dealing with a few long lineages with limited parallelism. This partially confirms what we observe from Fig. 4, which is also relatively close to the real convergence phase. In the first batch, the convergence phase was roughly the last 40 days, corresponding to 22% of the whole computation time. In the second batch, the convergence phase was from day 200 to day 300, taking roughly 33% of the whole time. We can thus tentatively conclude that the convergence phase takes roughly a constant fraction of total time.

5 A Simplified Mathematical Analysis of Our Scheme

In this section, we access the performance of our scheme using formal method. We will prove that, under some reasonable assumptions, in a simplified model, our scheme is asymptotically only a constant factor away from perfect parallelism.

In the following, we assume that the distribution of lineage lengths is a power law with exponent $\alpha > 2$ (i.e., with finite expectation). This assumption agrees with the experimental results discussed in the case study. One may doubt whether this is the correct assumption, as it comes from only two samples. It would also be nice to analyze experimentally the structure of more problems in discrete mathematics and constraint satisfaction to see if they also obeys the power law. However, such large scale experiments take a lot of time, even on volunteer computing projects. It took at least half a year for each of our batches in Odd Weird Search, and there are few volunteer computing projects of our kind whose related statistics are available. The lack of sample can be made up from similar observations from related fields. For instance, power law distributions appear in run-time of a homogeneous ensemble of backtracking computations for constraint satisfaction problems (see [21, 26], mind the different definition of exponent). These observations are clearly related to our situation. Breaking down a constraint satisfaction problem into smaller ones by fixing a set of its variables gives a homogeneous set of problems. But since backtracking is also a tree search, this breakdown is equivalent to breaking down the search tree by considering all subtrees obtained from cutting the whole tree at a certain depth, and this is exactly what we do when generating initial workunits. Nevertheless, exponents appears in constraint satisfaction problems are typically with $1 < \alpha < 2$ (i.e., with infinite expectation), which is different from our situation ($\alpha > 2$, i.e., with finite expectation) here in tree search parallelization. The reason is that we can change how we break down the tree if a probe test shows an exponent $\alpha < 2$, thus avoiding the problematic region of infinite expectation. Another reason that we do not treat the infinite expectation case is due to its intrinsic hardness to parallelize, which will be discussed later at the end of this section.

As discussed in Section 3, there are two phases in the model, one is the normal phase where there are enough workunits for all clients, and the other is the convergence phase, where there is a deficit in workunits. We now perform a simplified mathematical analysis to determine the length of the convergence phase.

Let m be the threshold of the number of lineages to enter convergence phase, i.e., when there are less than m unfinished lineages in the system, we enter the convergence phase. The value of m is correlated to the number of active clients, and is given as a fixed input. We now consider a simplified model of workunit processing. We suppose that all lineages form a circular list, and we process lineages in parallel without changing their order. We can imagine that the circular list is form by N boxes, the ith box containing a random integer X_i, which is the length of the corresponding lineage. All X_i's are independent and identically distributed with the probability law $\mathbb{P}[X_i = k] =$

$\zeta(\alpha)^{-1}k^{-\alpha}$. Here, $\zeta(s) = \sum_{n=1}^{\infty} n^{-s}$ is the Riemann ζ function, and α is the exponent of power law. We suppose that $\alpha > 2$, where X has a finite mean (but not necessarily a finite variance). This probability setup can be seen as a randomization of our observation in (1). When there are at least m boxes, we take m boxes starting from the current pointer at once, subtract their numbers by 1, retire all boxes with number 0, and move the pointer to the box next to boxes we just processed. Where there are no more than m boxes (lineages), we enter the convergence phase by definition. In the convergence phase, at each step we subtract the number of every box by 1 and retire boxes with 0, until no box is left. This is our simplified model.

We now analyze the simplified model. Since $\alpha > 2$, the distribution X has a finite mean $\mathbb{E}[X] = \zeta(\alpha)^{-1}\zeta(\alpha-1)$. The expectation of total workunit number, that is, the sum of lengths of lineages, is given by

$$W = \mathbb{E}\left[\sum_{i=1}^{N} X_i\right] = N\mathbb{E}[X] = N\zeta(\alpha-1)\zeta(\alpha)^{-1}.$$

We now study the length of the two phases. Let $X_1' > X_2' > \cdots > X_N'$ the sorted list of $X_1, X_2, \ldots, X_N$. The probability distribution of X_1' is given by

$$\mathbb{P}[X_1' = k] = N\zeta(\alpha)^{-1}k^{-\alpha}\left(1 - \zeta(\alpha)^{-1}\sum_{i\geq k} i^{-\alpha}\right)^{N-1}.$$

We start by the probabilistic aspects of these random variables. Since for $k \geq 1$, we have

$$\int_k^{k+1} x^{-\alpha}dx \geq k^{-\alpha} \geq \int_k^{k+1} (x+1)^{-\alpha}dx = \int_k^{k+1} x^{-\alpha}(1 + O(x^{-1}))dx.$$

We deduce the following estimate:

$$\sum_{i\geq M} k^{-\alpha} = \int_M^{+\infty} x^{-\alpha}(1 + O(x^{-1}))dx = (\alpha-1)^{-1}M^{-\alpha+1} + O(M^{-\alpha}).$$

We thus have the following expression of X_1':

$$\mathbb{P}[X_1' = k] = N\zeta(\alpha)^{-1}k^{-\alpha}\left(1 - \frac{k^{-\alpha+1}}{(\alpha-1)\zeta(\alpha)} - O(k^{-\alpha})\right)^{N-1}.$$

The mean of X_1' is the sum

$$\mathbb{E}[X_1'] = N\zeta(\alpha)^{-1}\sum_{k\geq 1} A_k, \quad \text{where}$$

$$A_k = k^{-\alpha+1}\left(1 - \frac{k^{-\alpha+1}}{(\alpha-1)\zeta(\alpha)} - O(k^{-\alpha})\right)^{N-1}.$$

We define the function $B(k) = \log A_k$, which is

$$\begin{aligned} B(k) := \log A_k &= (1-\alpha)\log(k) + (N-1) \log\left(1 - \frac{k^{-\alpha+1}}{(\alpha-1)\zeta(\alpha)} - O(k^{-\alpha})\right) \\ &= (1-\alpha)\log(k) + (N-1)\frac{k^{-\alpha+1}}{(\alpha-1)\zeta(\alpha)}(1 + O(k^{-1})) \end{aligned}$$

The expression of $B(k)$ extends to $\mathbb{R}^+$. The maximal of $B(k)$ is asymptotically at $k_* = \left(\frac{N-1}{(1-\alpha)\zeta(\alpha)}\right)^{\frac{1}{\alpha-1}} + O(1)$. We notice that $k_* = O(N^{\frac{1}{\alpha-1}})$ and $\exp(B(k_*)) = O(N^{-1})$. We verify that $\log A_k$ is increasing before k_* and decreasing after k_*. By approximating the sum as an integral, we have

$$\mathbb{E}[X_1'] = N\zeta(\alpha)^{-1}\int_1^{+\infty} \exp(B(x))dx + O(1).$$

Given $\epsilon > 0$, we consider the contribution of three intervals $[1, k_*^{1-\epsilon}]$, $[k_*^{1-\epsilon}, k_*^{1+\epsilon}]$ and $[k_*^{1+\epsilon}, +\infty)$. When $N \to +\infty$, we have

$$\int_1^{k_*^{1-\epsilon}} \exp(B(x))dx \leq k_*^{1-\epsilon}\exp(B(k_*^{1-\epsilon})) = k_*^{(1-\epsilon)(2-\alpha)}O(\exp(-cN^{\epsilon})) = O\left(N^{\frac{2-\alpha}{1-\alpha}(1-\epsilon)}\right),$$

$$\int_{k_*^{1+\epsilon}}^{+\infty} \exp(B(x))dx = \int_{k_*^{1+\epsilon}}^{+\infty} O(x^{1-\alpha})dx = O\left(k_*^{(1-\alpha)(1+\epsilon)}\right) = O\left(N^{\frac{2-\alpha}{1-\alpha}(1+\epsilon)}\right),$$

$$\int_{k_*^{1-\epsilon}}^{k_*^{1+\epsilon}} \exp(B(x)) \leq \int_{k_*^{1-\epsilon}}^{k_*^{1+\epsilon}} \exp(B(k_*))dx = O\left(N^{\frac{1+\epsilon}{\alpha-1}-1}\right).$$

The first estimate uses the monotonicity of $B(k)$ to bound all function values by $B(k_*^{1-\epsilon})$, while the second uses $B(x) = O(x^{1-\alpha})$ for large x, and the last uses the fact that k_* maximizes $B(k)$. The estimates are valid for all $\epsilon > 0$. By letting $\epsilon \to 0$, we see that all three estimates contribute $O\left(N^{\frac{2-\alpha}{\alpha-1}}\right)$. Furthermore, a simple computation shows that the interval $[k_*/2, 2k_*]$ contributes already $cN^{\frac{2-\alpha}{\alpha-1}}$ for some constant c. Therefore, for some constant $c_{\mathbb{E}}$, we have

$$\mathbb{E}[X'_1] = c_{\mathbb{E}} N^{\frac{1}{\alpha-1}}(1 + o(1)).$$

Using the same method, by supposing that $m \gg 1$ and $m \ll N$, we have $\mathbb{E}[X'_m] = c'_{\mathbb{E}} N^{\frac{1}{\alpha-1}}(1 + o(1))$ for another constant $c'_{\mathbb{E}}$.

We now determine the range of parameter N, which is the number of initial workunits, that minimizes the total computation time, given the fixed number of active clients m in the system. We see that the processing enters the convergence phase after $Y = m^{-1} \sum_{i=m}^{N} X'_i$ steps, and the convergence phase takes $Z = X'_1 - X'_m$ steps. We have

$$\mathbb{E}[Y] \le \mathbb{E}[Wm^{-1}] = Nm^{-1}\zeta(\alpha - 1)\zeta(\alpha)^{-1},$$
$$\mathbb{E}[Z] = \mathbb{E}[X'_1] - \mathbb{E}[X'_m] = c_m N^{\frac{1}{\alpha-1}}(1 + o(1)).$$

Here, c_m is a constant that depends only on m and α, but not on N. In practice, we have no control on the number of active clients on which m depends. We now have the following bound on the ratio between our model and perfect parallelism:

$$\frac{\mathbb{E}[Y + Z]}{\mathbb{E}[Wm^{-1}]} \le 1 + \frac{mc_m N^{\frac{2-\alpha}{\alpha-1}}(1 + o(1))}{N\zeta(\alpha - 1)\zeta(\alpha)^{-1}}$$

We observe that, as long as $N = \Omega(m^{\frac{\alpha-1}{\alpha-2}})$, the efficiency of our model is only a constant factor away from perfect parallelism when m tends to infinity. Any further increase in N beyond $N = \Theta(m^{\frac{\alpha-1}{\alpha-2}})$ will still decrease the total length of the two phases, but only marginally. This bound on N gives us an indication on the choice of the number of lineages. We note that we also want N to be small to reduce the computation needed to split the whole workload into sections and to regroup them into initial workunits. Therefore, we should take N of the same order as $m^{\frac{\alpha-1}{\alpha-2}}$.

We should remind readers that our analysis here, although mathematically rigorous, is oversimplified in the modeling. Nevertheless, we expect our mathematical model to be a close approximation of the real situation.

We now check the theoretical results above against simulation results. We use parameters m and α from our second case study. In general, it is difficult to estimate m in a volunteer computing project, since availability of volunteer machines varies greatly, with a power-law distribution [29]. We estimate that, in both batches, the average computational power is around 150–200 single cores. We thus pick $m = 175$ in the following. The exponent α depends on the problem structure. We thus pick $\alpha = 4.09$ and $\alpha = 2.69$ as in the first and the second batch of our second case study.

Here are the simulation results in Fig. 5. The optimal lineage number N from the analysis above is in the order of magnitude of $N \approx 2100$ for $\alpha = 4.09$ and $N \approx 310000$ for $\alpha = 2.69$. We observe that efficiency improves when N grows, but the improvement slows down significantly around the order of magnitude of respective optimal values, which corresponds well to our theoretical prediction. Therefore, the optimal choice of N we computed is sound. Although we only tested for a fixed value of m, but since the accuracy of the model gets better when m grows, we can say that the theoretical prediction holds for larger m.

By simulation, we can also see how good an approximation is our simplified model comparing to real-world situations. We have done some simulations with the exact histogram data of the two batches of Harmonious trees. In simulations, the convergence phase of the first batch takes up on average 7% of the whole time, and that of the second batch takes up 26% on average, both shorter than real-world situations. In general, we should expect the convergence phase in real world to take longer than theoretical prediction, because we suppose full availability in our simplified model, which never occurs in real-world volunteer computing. The difference for the second batch is less significant, always thanks to the surge of badge-scooping volunteers at the end of the run. We thus conclude that our simplified model is a relatively good approximation to the real world.

In practice, when we try to determine the number N of initial workunits into which we partition the whole search tree, we do not know the value of α in advance. An approximation of α can be computed either by sampling or by using the value for a smaller search in the same search space. Since the distribution to be

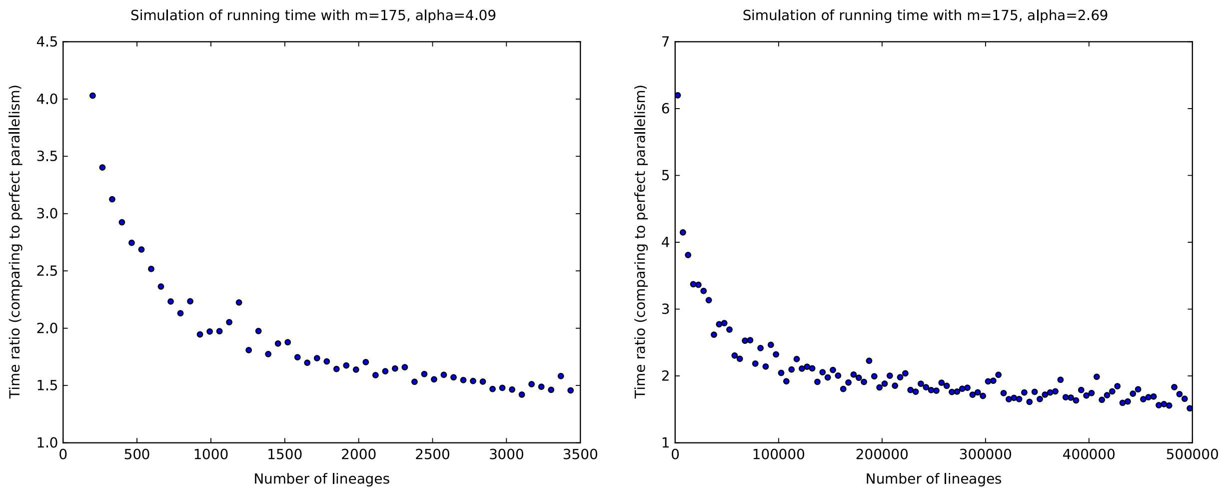

Fig. 5 Simulation results of the efficiency of our simplified mathematical model, showed in the form of the ratio from maximal efficiency, for $m = 175$, $\alpha = 4.09$ (left, averaged over 100 instances for each point) and $\alpha = 2.69$ (right, averaged over 300 instances for each point)

determined is a power law at the tail, in a sampling, it would be difficult to see the tail whose exponent we want to find out. Special care should thus be taken in the sampling, for instance by increasing the sampling size and by using a proper estimator, such as explained in [11].

After the exponent α of a certain partition of the search tree is estimated, we can use it to determine the quality of the partition, that is, how well suited it is to our scheme. The bigger α is, the better the partition works under our scheme. It is because, when α is too close to 2, that is, when the tail is too heavy, the value of $N = m^{\frac{\alpha-1}{\alpha-2}}$ may be too large even for reasonable m, therefore not realistic. In this case, we may want to change the way to partition the search tree to get a better α. Possible ways of achieving this goal include changing the algorithm and reordering choices related to the branching of the search tree.

When we get a good enough α, we then determine the number of initial workunits to generate, which is the number of lineages N. Since the efficiency always improves when N grows, we should partition the whole search tree into as many parts as possible within reasonable pre-computation time, while avoiding extremely short workunits. The optimal value $N = m^{\frac{\alpha-1}{\alpha-2}}$ serves as an indication of how many is enough and where the trade-off should be between workunit splitting and overall efficiency.

As a final remark, we should note that our scheme still works for any $\alpha > 2$, only that when α is too close to 2 it may give unreasonable value of optimal number of lineages N. If all attempts to get a better α fail, one can still proceed by taking the maximal N within reasonable pre-computation time. Our scheme will still work, but the performance will not be a constant factor away from perfect parallelism. But in this case, since the tail is too heavy, there will be a few workunits taking enormous time, rendering any parallelization of these tasks in volunteer computing difficult. This is also the reason that we do not consider the case $\alpha < 2$, although it occurs often in constraint satisfaction problems. It is because, in this case, the expectation of the run-time of a workunit is infinite, which means there will be a constant number of workunits that takes on a constant proportion of the whole computation task, thus nearly impossible to parallelize.

6 Conclusion

Volunteer computing is a special parallel computing paradigm, with only a minimal communication ability while having an extra human factor. Few attempts have been done in volunteer computing to parallelize massive tree searches with an uneven search tree. We gave a solution of this problem by first listing some requirements of such a parallelization, then proposing a working scheme with implementation examples. The first example (Harmonious Tree) was only a half success, while the second (Odd Weird Search) was fully

successful. We gave some details of our implementations, especially for the second one, and we also gave a formal analysis on the efficiency of our scheme. We thus conclude that our model is practical, and may be used on other problems.

One direction of future work is the verification of the power law assumption on more problems in discrete mathematics and constraint satisfactions, preferably in the context of volunteer computing, where problems treated are enormous. It takes time to deal with this kind of huge problems, but also interesting and the data we get may provide further insight on how to better parallelizing the solution of these problems in volunteer computing.

We now discuss some possible ways to improve our framework. The most problematic part of our model is the convergence phase, where active clients are not fully occupied due to a lack of workunits because clients are unable to return workunits in a timely fashion. To mitigate the situation, we can issue workunits only to clients with a short turnaround time, shorten the deadline, or issue extra replica. The first two ideas are straight-forward, but the third is more interesting. In [25], it was showed that the power law distribution of client availability in volunteer computing projects might induces a biased distribution on turnaround time with a somehow heavy tail (see Fig. 4 therein). Issuing extra replica is then interesting, as it can be used to "avoid" the heavy tail. Other more advanced scheduling in volunteer computing, such as the one in [23], can also be used in combination to accelerate the convergence phase.

If we have multiple batches to be computed, and we do not care as much the completion time of a single batch than the whole throughput of the system, we can mitigate the loss of efficiency in the convergence phase by issuing a new batch in the system to provide enough workunits for all the clients.

Lastly, we can also rely on the human factor that is unique for volunteer computing. We have seen that the badge-scooping at the end of the second batch of Odd Weird Search greatly accelerated the computation in the convergence phase. This is because workunits become a scarcity in the convergence phase, and as they are regenerated only when results are returned, volunteers craving for badges have the incentive to return the result of each workunit as fast as possible, in order to get workunits faster for the badges. Therefore, a badge system based on computational contribution is highly recommended for volunteer computing projects adopting our scheme. It may be also a good idea to set up special "rush" badges that compensate volunteers for finishing workunits in the convergence phase.

Acknowledgements Open access funding provided by Graz University of Technology. The first author was partially supported by IRIF (previously LIAFA) of Université Paris Diderot. Both authors thank Prof. David Anderson, Prof. Anne Benoit, Linxiao Chen and the anonymous referees for their useful advises. We also thank the volunteers in yoyo@home that contributed to Harmonious Trees and Odd Weird Search.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Abbott, B., Abbott, R., Adhikari, R., Ajith, P., Allen, B., Allen, G., Amin, R.S., Anderson, S.B., Anderson, W.G., Arain, M.A., et al.: Einstein@Home search for periodic gravitational waves in early S5 LIGO data. Phys. Rev. D **80**(4), 042003 (2009)
2. Aichholzer, O., Krasser, H.: Abstract order type extension and new results on the rectilinear crossing number. In: Proceedings of the Twenty-First Annual Symposium on Computational Geometry, pp. 91–98. ACM (2005)
3. Anderson, D.P.: BOINC: a system for public-resource computing and storage. In: Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10. IEEE (2004)
4. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002)
5. Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
6. Bazinet, A.L., Cummings, M.P.: Subdividing long-running, variable-length analyses into short, fixed-length BOINC workunits. J. Grid Comput. **14**(3), 429–441 (2016)
7. Blelloch, G.E., Gibbons, P.B., Matias, Y.: Provably efficient scheduling for languages with fine-grained parallelism. J. ACM **46**(2), 281–321 (1999)
8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
9. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
10. Bouguerra, M.S., Kondo, D., Trystram, D.: On the scheduling of checkpoints in desktop grids. In: 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2011, pp. 305–313. IEEE (2011)

11. Clauset, A., Shalizi, C.R., Newman, M.E.J.: Power-law distributions in empirical data. SIAM Rev. **51**(4), 661–703 (2009)
12. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
13. Distler, A., Jefferson, C., Kelsey, T., Kotthoff, L.: The semigroups of order 10. In: Principles and Practice of Constraint Programming, pp. 883–899. Springer, Berlin (2012)
14. Distributed.net: Distributed.net Faq-O-Matic: why is progress of OGR-28 stubspaces 1 and 2 so slow? http://faq.distributed.net/cache/299.html, Cited: 21 Jan 2015
15. Distributed.net: Distributed.net: Project OGR. http://www.distributed.net/OGR, Cited: 21 Jan 2015
16. Drakakis, K., Iorio, F., Rickard, S., Walsh, J.: Results of the enumeration of Costas arrays of order 29. Adv. Math. Commun. **5**(3), 547–553 (2011)
17. Estrada, T., Flores, D.A., Taufer, M., Teller, P.J., Kerstens, A., Anderson, D.P.: The effectiveness of threshold-based scheduling policies in BOINC projects. In: Second IEEE International Conference on e-Science and Grid Computing, 2006 (e-Science'06), p. 88. IEEE (2006)
18. Fischetti, M., Monaci, M., Salvagnin, D.: Self-splitting of workload in parallel computation. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pp. 394–404. Springer, Berlin (2014)
19. Gallian, J.A.: A dynamic survey of graph labeling. Electron. J. Comb. **Dynamic Surveys**, DS6 (2014)
20. Golle, P., Mironov, I.: Uncheatable distributed computations. In: Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA, pp. 425–440. Springer, Berlin (2001)
21. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1), 67–100 (2000)
22. Grama, A., Kumar, V.: State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. Knowl. Data Eng. **11**(1), 28–35 (1999)
23. Heien, E.M., Anderson, D.P., Hagihara, K.: Computing low latency batches with unreliable workers in volunteer computing environments. J. Grid Comput. **7**(4), 501 (2009)
24. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable distributed depth-first search with greedy work stealing. In: 16th IEEE International Conference on Tools with Artificial Intelligence, 2004, pp. 98–103. IEEE (2004)
25. Javadi, B., Kondo, D., Vincent, J.M., Anderson, D.P.: Discovering statistical models of availability in large distributed systems: An empirical study of SETI@home. IEEE Trans. Parallel Distrib. Syst. **22**(11), 1896–1903 (2011)
26. Jia, H., Moore, C.: How much backtracking does it take to color random graphs? Rigorous results on heavy tails. In: Proceedings of Principles and Practice of Constraint Programming - CP 2004, pp. 472–476 (2004)
27. Karp, R.M., Zhang, Y.: Randomized parallel algorithms for backtrack search and branch-and-bound computation. J. ACM **40**(3), 765–789 (1993)
28. Mersenne Research Inc.: GIMPS history. http://www.mersenne.org/various/history.php, Cited: 21 Jan 2015
29. Nurmi, D., Brevik, J., Wolski, R.: Modeling machine availability in enterprise and wide-area distributed computing environments. In: Euro-Par 2005 Parallel Processing, pp. 612–612. Springer, Berlin (2005)
30. Pataki, M., Marosi, A.C.: Searching for translated plagiarism with the help of desktop grids. J. Grid Comput. **11**(1), 149–166 (2013)
31. Rechenkraft.net e.V.: yoyo@home. http://www.rechenkraft.net/wiki/index.php?title=Yoyo%40home_en, Cited: 21 Jan 2015
32. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Principles and Practice of Constraint Programming, pp. 596–610. Springer, Berlin (2013)
33. Reinefeld, A., Schnecke, V.: Work-load balancing in highly parallel depth-first search. In: Proceedings of the Scalable High-Performance Computing Conference 1994, pp. 773–780. IEEE (1994)
34. Sonnek, J., Chandra, A., Weissman, J.: Adaptive reputation-based scheduling on unreliable distributed infrastructures. IEEE Trans. Parallel Distrib. Syst. 18(11) (2007)
35. Stolee, D.: TreeSearch user guide. http://www.math.uiuc.edu/stolee/SearchLib/TreeSearch.pdf (2011)
36. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurr. Comput. - Pract. E **17**(2–4), 323–356 (2005)
37. Wright, R.A., Richmond, B., Odlyzko, A., McKay, B.D.: Constant time generation of free trees. SIAM J. Comput. **15**(2), 540–548 (1986)