BitWorker, a Decentralized Distributed Computing System based on BitTorrent

Arnaud Durand[1], Mikael Gasparian[1], Thomas Rouvinez[1], Imad Aad[2], Torsten Braun[2] and Tuan Anh Trinh[3]

[1] University of Fribourg, Switzerland
arnaud.durand@unifr.ch, mikael.gasparian@unifr.ch, thomas.rouvinez@unifr.ch
[2] University of Bern, Switzerland
aad@iam.unibe.ch, braun@iam.unibe.ch
[3] Budapest University of Technology and Economics, Hungary
trinh@tmit.bme.hu

Abstract. In this paper we present BitWorker, a platform for community distributed computing based on BitTorrent. Any splittable task can be easily specified by a user in a meta-information task file, such that it can be downloaded and performed by other volunteers. Peers find each other using Distributed Hash Tables, download existing results, and compute missing ones. Unlike existing distributed computing schemes relying on centralized coordination point(s), our scheme is totally distributed, therefore, highly robust. We evaluate the performance of BitWorker using mathematical models and real tests, showing processing and robustness gains. BitWorker is available for download [**?**] and use by the community.

Keywords: Distributed Computing, BitTorrent, Peer-to-Peer Networks

1 Introduction

The processing power requirements in almost every field of study have increased. Due to the ever increasing complexity of problems to solve, single processing units are often no longer able to yield results within an acceptable time. Despite the advances and progress achieved in the fields of photolithography and processor architecture, the processing power is often not enough. Many research works benefit from the ability to test and iterate on the results, which requires computations to be done multiple times. Breaking out of a single processing unit became essential. Distributed computing is characterized by multiple processing units connected together in a network and sharing computation. The goal of distributed computing is to allow multiple processing units (distinct machines) collaborating on the same task. This results in a computational speedup dependent on the number of collaborating machines.

Distributed computing faces challenges and limitations. The first issue is the speedup factor not being linear to the number of machines in the network. On a single machine operations are read and executed locally, in a procedural / multi-threaded fashion. The obvious advantage is the ease of distributed computing

as well as synchronization. Moreover, all the results are held in a single location, making data retrieval and exchange very easy. With multiple machines, operations are first distributed among all the machines in the network. Note that all the dependencies for the computation have to be shared, which adds to the computational overhead. Upon reception of a new task, a machine in a distributed computing network will have to process its share of computation and return the result. This involves a complex communication scheme that also adds computational overhead. Furthermore, reliability becomes a prerequisite. All the mechanisms required to guarantee data integrity between the machines in the distributed network, in turn, add to the overhead. All these reasons may affect the speedup.

Beyond the overhead issues, an organizational issue arises with distributed computing. The needs in terms of infrastructure regroup the machines, the complete setup of these machines, and a fully working network. The costs of such infrastructures do not follow a linear curve as more machines in the network increase the complexity of the requirements in terms of power, networking, and administration. Distributed computing allows faster computations but at greater financial cost. With this financial aspect in mind, new developments of distributed computing got oriented toward outsourcing computations to willing volunteers rather than in-house farming. Installation costs are moved to the volunteers but at the expense of security and data privacy. With potentially malicious machines in the network, further data verification is required. Nevertheless, the concept of tapping into the processing power of machines all around the world is attractive.

In this paper, we propose an implementation of a distributed computing system called BitWorker. BitWorker is based on a modification of BitTorrent and performs computations in a fully distributed way. In Section **??**, known distributed computing solutions are presented. Section **??** presents the fundamentals of our approach to distributed computing based on BitTorrent. In Section **??** the general architecture of BitWorker with its workflow is explained. Section **??** shows analytical and experimental performance evaluations. We discuss future work in Section **??** and conclude in Section **??**.

2 Related Work

2.1 Distributed Computing

Distributed Computing is an environment in which a group of independent and geographically dispersed computer systems solve a complex problem, each by solving a part of the solution and then combining the result from all computers [**?**]. Compared to centralized computing, distributed computing offers more flexibility, fault resilience, and scalability. Furthermore, it can be used to reduce the costs of problem solving in areas of public interest using computing resources of volunteers. The computing power could come from CPUs, GPUs, game consoles, or specialized hardware.

The idea of using computing resources of volunteers is a known concept. A wide variety of distributed computing projects are available, most of which

focus on scientific research. Folding@home [?] predicts protein structures (≈40 PetaFlops in Sep. 2014), SETI@home analyzes radio frequencies in search for extraterrestrial life and Milkyway@home [?] models our galaxy from available data. Some of these projects are based on the Berkeley Open Infrastructure for Network Computing (BOINC) [?], an open platform for distributed computing.

A recent approach [?] proposes novel techniques for management of large-scale distributed storage systems. However, there are two key areas which differentiate our approach from this approach. First, the proposed approach [?] focuses on the IP level of the network, whereas BitWorker focuses on the overlay networks. Second, the proposed approach focuses on storage systems, whereas BitWorker can be applied in different computing environments as well.

2.2 BitTorrent and Distributed Hash Tables

BitTorrent is a peer-to-peer file sharing protocol. Using BitTorrent, users can exchange large amounts of data without any data hosting servers. Instead of downloading a file from a server, users request small pieces of the data from other users and start exchanging these as they become available. The file transfer process is thus completely offloaded to peers in the network. The tracker is a dedicated server enabling peers' exchanges by providing a list of connected peers of its registered shared files. The tracker is regularly queried for new peers to connect to, thus enabling disconnected peers to be replaced by newer ones. In this regard the BitTorrent protocol fully handles node discovery and sharing of pieces in a centralized fashion. BitTorrent has been later on extended with PEX (Peer exchange) [?,?]. PEX allows peers to retrieve addresses from other peers rather than from the tracker. This is a semi-decentralized solution, as at least one connection to the tracker is required for bootstrapping. Distributed Hash Tables (DHTs) [?] extend BitTorrent file exchange, enabling BitTorrent to work in a totally decentralized fashion. It only requires an initial bootstrap to start exchanging files from any public torrent.

2.3 Distributed Computing and Peer-to-Peer

Available distributed computing projects such as BOINC use a centralized or semi-centralized approach where a main entity is responsible for coordinating work load distribution among the available resources. The result of the computational task is shared either using (1) a client-server architecture (Folding@home [?], BOINC-based projects, etc.) or (2) a P2P exchange system (BOINC with BitTorrent [?], CompTorrent [?]).

In the client-server case, the processing result of a volunteer is transmitted directly to a server. The clients are not informed about the state of the global computation or the available resources (other clients).

In the second case, task distribution is still coordinated by a centralized entity but the computation results are shared using P2P mechanisms. While having an entity coordinating the resources ensures efficient distribution of the tasks, this approach requires a permanently available infrastructure. Moreover,

it presents a single point of failure. Fully decentralized infrastructures involve a P2P data-distribution model: a decentralized mechanism to find and connect to other volunteers. Therefore, it replaces the centralized coordination system with peer signaling. Cryptocoins (Bitcoin [**?**] and derivative crypto currencies) mining is a good example of decentralized distributed computing systems.

SETI@home [**?**] is a distributed computing project allowing public clients to contribute. This project aims at searching for extraterrestrial intelligence in the universe through an analysis of radio signals. Based on a classic client / server architecture, packets of data and computation instructions are sent to each client and the computation results are gathered back to the unique server. Since then, new ways to perform distributed computations have appeared, involving different communication schemes and security improvements. A considerable improvement was made when the BitTorrent protocol first helped with data distribution. Efficient and fast data exchange was solved. BitTorrent has been known to work well for data management in distributed computing. BitTorrent can also offer node (peer) discovery, task splitting and distribution. There are implementations of distributed computing using a modified version of BitTorrent, e.g. CompTorrent. However, we did not find any implementation that fully harnesses the capabilities of BitTorrent or fully decentralizes the work distribution process.

2.4 BOINC

Berkeley Open Infrastructure for Network Computing (BOINC) is a project originating from the Space Sciences Laboratory at the University of California. The principle is to harness the processing power left on the host machine (full load - current CPU / GPU load) and to use it for research projects. Upon launching the BOINC client, a registration is performed and a list of available projects is proposed to the user. When a project is selected the client fetches instructions from the project's server. These instructions take into account the capabilities of the client and the project server returns a set of applications and input files for the task to be performed by the client. Upon completion, the client uploads the output files back to the server. Then, according to user preferences, the client restarts this process and asks for a new task. BOINC is, therefore, a centralized distributed computation system based on the classic client-server architecture.

BOINC was originally created for the SETI@home project. After opening BOINC to support other projects, optimizations such as multi-threaded CPU applications and the use of OpenCL appeared. Extensions have been developed to enhance BOINC functionalities and performance. Among these extensions a BitTorrent wrapper has been created to help any BOINC client to fetch large datasets required to make computations. A BOINC client can open a BitTorrent wrapper, which will contact a tracker, get a list of peers, who possess the required input datasets, and perform requests on these peers to obtain the data required. Therefore, the required data acquisition is performed via P2P networks rather than overloading one or multiple servers with direct downloads. Compared to

BitWorker, BOINC does not rely on BitTorrent to distribute the workload on the network. BOINC only uses BitTorrent as a data exchange / retrieval convenience mechanism.

2.5 Gnutella Processing Unit

Gnutella Processing Unit (GPU) [**?**] organizes clients in a Gnutella network. Each client allows the other users to request CPU resources. The concept of GPU involves trading network resources in exchange of CPU resources. A GPU network usually organises itself in a cluster of 5 to 15 peers. A GPU first determines a list of all the peers connected to the network and establishes a predefined number of connections to each peer. This list of IP addresses is stored in a Web cache and mimics the functionality of a tracker (without performance evaluations). Upon detection of a request, a GPU client will create a set of threads to handle the new task at hand. Each task is then processed thanks to the use of plugins. A GPU client can host multiple plugins that hold the algorithmic logic to handle a given request. For example, such plugins could contain the algorithms required to solve a brute force attack to the discrete logarithm [**?**], computation of a partial differential equation using the random walkers approach and the Feynman-Kac formula [**?**], or even computer rendering of images generated with Terragen [**?**]. Once a task has been processed, the GPU client transmits the result back to the peer that requested it. GPU also uses frontends, which are a complement to plugins as well as serve easing work distribution and result visualization.

GPU uses Gnutella as an underlying protocol and hence it sends requests by flooding the whole network. Since flooding is not an optimal solution to efficiently transmit messages between peers within the same network, Gnutella makes use of "Ultrapeers". An Ultrapeer is a specific node in the network whose role consists of maintaining connections between normal nodes while these simply keep one connection to the Ultrapeer. The result is the formation of a tree-shaped network. Though the concept is close to what we achieve with BitWorker, we do not use Gnutella to manage the network structure as it is inefficient and prone to both failures and flooding attacks. Moreover, no resource trading is done with BitWorker as contributing peers are also interested in the result of the computation. This results in a collaborative work rather than in requests for resources.

2.6 CompTorrent

CompTorrent's [**?**] approach to distributed computing makes use of BitTorrent. CompTorrent aims at providing an easy to use distributed computing environment by creating a CompTorrent file that holds the description of both an algorithm and a dataset. The CompTorrent file should then be made available for download. Potential peers have to download the CompTorrent file and run it. Each peer then connects to the "seed". The seed is a super node that handles workload distribution over the network. When a peer connects to the seed, it receives its tasks and starts to process them. When a task has been completed,

the seed is informed and the results are transferred to it. The seed also acts as a quality control agent by making sure that each sub-dataset is computed multiple times, hence allowing comparison of the results. When the seed determines that all computations have been carried out and that the data has been verified, it will simply let the other peers share the results among them. Data is verified by seeds that may ask two clients to compute the same work and compare results afterwards. Compared to BOINC with BitTorrent, CompTorrent is more decentralized as it lets BitTorrent distribute the workload by using super nodes. Unlike CompTorrent, BitWorker is completely decentralized and takes advantage of DHT instead of only relying on trackers. Furthermore, it can be downloaded and used by the community.

3 BitWorker Fundamentals

Throughout this paper, we focus on enabling distributed computations by modifying the BitTorrent protocol. BitTorrent has been previously used in distributed computing, but only for data management purposes such as data retrieval from all the machines taking part in the computation process. Our approach aims at fully transforming peers to become both server and client for distributed computations. To solve the issues above, the BitTorrent specification had to be modified to allow the specification of a distributed task in the metadata files. BitWorker supports any parallel computation achievable by using the UNIX shell environment. Distributed computations carried out by BitWorker are enabled thanks to three functionalities originally provided by the BitTorrent protocol, which we modified to fit new purposes:

1. File splitting into pieces
2. Piece sharing
3. Node discovery

File splitting reduces the size of files transmitted between peers. This is achieved by splitting the original file into multiple pieces of predefined equal sizes.

The reasonfor file splitting is explained by the second functionality: piece sharing. The BitTorrent protocol allows a peer to download a file by requesting and offering missing, respectively available, pieces of the torrent to other peers. Upon connecting to another peer, a "bitfield" message is exchanged directly after the handshake. The message payload represents pieces that have been successfully downloaded. When a piece (selected at random) is not available among all the peers, BitWorker will compute it and make it available.

"Have" messages play a major role in piece exchanges and, for BitWorker, in computation distribution. The exchange of bitfields is enabled by the third functionality of BitTorrent: node discovery. Unlike centralized distributed computing, BitWorker relies on a network of directly connected end users (peers). To start participating in a BitTorrent network, a peer must establish an initial

connection to a tracker or use the DHT. In terms of distributed computing, BitWorker takes care of distributing computations, respectively storing/retrieving the results. Participating in a task in BitWorker then fully harnesses the BitWorker protocol as it takes care of the following essential operations:

- Node discovery using DHT
- Task splitting
- Workload distribution
- Piece sharing

Therefore, the complete process of distributed computing with BitWorker is managed by the protocol itself. In the following section we present the modifications to the BitTorrent protocol to support distributed computations.

4 Architecture

4.1 Implementation Overview

Upon starting the client, the first process reads the content of the torrent file. The currently owned data is evaluated and a list of peers is retrieved from the announcement provided by the trackers specified within the torrent file. Performing these first tasks allows the client to create the list of pieces it possesses and the list of pieces already available in the network. Once the client knows which pieces are missing, it can request them from any peer it is connected to. With BitWorker, if a piece is present in the list of available pieces, it means that it has already been computed. Therefore, it is only downloaded to speed up sharing among peers and return the results. When a piece is not present in the list of available pieces, it means it is unavailable because no connected peer possesses it. Therefore, the client can choose to start computing the piece and upon completion send a BitTorrent "have" message. When all the pieces have been computed, the torrent enters into seed mode where it only uploads the data to peers requesting it. As explained previously, BitWorker is based on the BitTorrent protocol. We modified three parts of it:

1. Meta-information task file
2. Missing piece management
3. Support for variable piece sizes
4. Peer list acquirement process

4.2 Meta-information Task File

A distributed task is described in a bencoded [**?**] file. A bencoded file is a dictionary (key and value pairs) namely used by BitTorrent to provide meta data of shared files. The task and its parameters are specified in this file. The task file is very similar to torrent files; it contains metadata about the generated files and an announcement (list) (optional with DHT). Unlike a torrent file, it does not contain any hash list as the content is non-deterministic. In other words,

we cannot create hashes for the content we do not have yet. Ideally, piece sizes range from 256kB to 4MB, according to the recommendations for the BitTorrent network. The piece sizes are specified inside the task file and should be greater or equal to the biggest generated file. The communication protocol is able to deal with varying pieces size by exchanging piece size information among the peers.

4.3 Task Interpretation

Tasks should be described generic enough to fit most of the use-cases. We take advantage of the UNIX architecture and represent a distributed task as a unique shell command. A task is split into work items, the atomic subtasks of the distributed work. A work item is specified using the same command with different environment variables. Environment variables are defined according to piece index, file index, and part index. The pieces are then filled with the standard output of work item executed commands.

4.4 DHT Functionality

Using legacy BitTorrent, the tracker is periodically queried by the clients to obtain an updated list of peers. A peer connects to the tracker, receives a list of peers, and then communicate with these peers to obtain the locally missing pieces. The tracker is the only way to start downloading pieces from peers while using the original BitTorrent protocol. Although being convenient and beneficial in terms of performance, it is also the unique point of centralization, thus, the only critical point of BitTorrent. To solve this issue, we created a new independent DHT service to complement peer discovery. Therefore, we have the ability to run in a completely decentralized fashion.

4.5 Applications and Examples

The content of the pieces is evaluated using the standard output of bash shell commands. Any task that can be divided into a set of bash shell commands can be computed in a distributed fashion using BitWorker. BitWorker uses the results returned in the standard output of the bash shell. An example of such a task could be the computation of squares for integers. The following paragraphs presents two examples and how to process these in a distributed and decentralized fashion with BitWorker.

The first example is a video transcoding task that allows to encode a given video using FFmpeg [**?**]. FFmpeg is a library for video encoding. With BitWorker each client will encode a part of the video. The parts (pieces) are generated and shared with the peers. The generated piece sizes are non-deterministic. When a peer possesses all the pieces it will merge them to create the complete encoded video and continue to share the parts with connected peers.

The second example uses RainbowCrack [**?**] to generate rainbow tables in a distributed and decentralized fashion. Rainbow tables are precomputed tables

using time-memory tradeoff [**?**]. The tables allow to reverse cryptographic hash functions. As in the first example, a rainbow table is divided into multiple parts. The parts are generated and shared among the peers.

BitWorker allows to generate tasks both featuring deterministic or nondeterministic piece sizes. The client reads the task run-time command from the bencoded task file to run a task.

4.6 Workload Distribution

In approaches like BOINC or CompTorrent, work distribution is unified and centralized. BOINC uses the client-server architecture, which allows the server to fully control what each client has to compute. GPU is doing the same but the server is a peer that requests help on a specific computation. In CompTorrent special peers called "seeds" act as centralization points that distribute the workload. All these distributed computation systems feature a point of centralization that can optimally handle with minimum costs both the computation distribution and the verification of the work done. However, having a single point of centralization also means having a single point of failure. CompTorrent reduces the risk of the entire network going down by increasing the number of seed instances for redundancy. Considering the case of an attack, all instances of seeds could be targeted and the whole network would collapse. In GPU, since flooding is the only way to transmit computation requests, Ultrapeers are used to reduce the message overhead. Ultrapeers become hubs, which makes them interesting targets, since they are critical to make the network run efficiently. Finally, in BOINC if the main server fails, no more work requests can be issued.

BitWorker focuses on eliminating all critical single points of failure. Indeed there is no distinction between a regular BitTorrent peer and a CompTorrent seed. Therefore, in BitWorker, every peer is both a client and a server. Unlike all the discussed approaches, BitWorker has no point of centralization, which makes it naturally resilient to attacks: with every peer being able to work as a server, there is no need for a centralised entity anymore; the use of DHT removes all centralization during the establishment of the network.

5 Performance Evaluation

In this section we evaluate the performance of BitWorker in terms of processing speedup for a varying number of volunteer nodes. We first start evaluating the completion time of a specific task theoretically, without considering communication overhead. Evaluation is done analytically and using real testbeds. Additional experiments are also performed to evaluate the impact of the number of pieces on the overall performance of the system.

5.1 Expected Completion Time

We want to evaluate the completion time of a task. Assume that we do not have any communication overhead and that peers are perfectly synchronized for task completion, then the average completion time is $E_{n,m} \times t$ where:

$$E_{n,m} = \left(\sum_{j=1}^{n-1} \left| \sum_{k=1}^{j} (-1)^{k+1} \binom{j}{k} k^m \right| \binom{n}{j} \frac{E_{n-j,m}}{n^m} \right) + 1$$

$$= \left(\sum_{j=1}^{n-1} \left\{ \begin{matrix} m \\ j \end{matrix} \right\} \frac{n!}{(n-j)!} \frac{E_{n-j,m}}{n^m} \right) + 1 \ , \ E_{1,m} = 1$$

with $\left\{ \begin{matrix} n \\ m \end{matrix} \right\}$ being stirling numbers of the second kind using Knuth's [**?**] notation. n is the number of pieces, m is the number of connected peers and t is the average time to generate a piece.

This model assumes that the churn rate is 0. The churn rate measures the number of nodes moving out of the network over a given time period. We evaluate the average number of computing iterations (rounds) according to the expected number of collisions per iteration. We can assess this by evaluating the completion time of the scenarios and then compute the arithmetic mean. To do so, we start by evaluating $S(m, j)$, the number of possibilities to partition a set of m connected peers into j subsets. Here, j represents the number of remaining pieces. For each scenario, we have $(n - j)!$ possible outcomes for the current round. We should also evaluate the outcomes for the next rounds using the recursive function. We should also add 1 to the summation as there is always at least one round for any $n \geq 1$ and $m \geq 1$, so we only require to compute j from 1 to $n - 1$. We always need exactly one round to complete one piece for any $m \geq 1$ so we know $E_{1,m} = 1$. Because of communication overhead, churn rate and task synchronization are not taken into account. This model is optimistic and serves as a reference to experimental results.

5.2 Hypotheses

We expect that the time to complete the task will decrease when increasing the number of peers. A speedup significantly superior to 1 proves that the sharing mechanism is beneficial. The first hypothesis is that the task completion time will closely follow the expected completion time model previously described, with slightly lower performance due to communication overhead. The second hypothesis is that the speedup is higher with more pieces (the same task is split into more pieces).

5.3 Setup of Test Environment

The aim of the experiments is to calculate the completion time of a task given different numbers of volunteers and pieces. We have tested with Amazon small EC2 instances. To make sure that the machine performance variation does not

have high impact on the results, we created a task with a deterministic completion time of exactly 64 minutes when computed locally. The global output size is 64 MB, which is split into pieces. For an experiment with 64 pieces, this simulates a computation that takes 1 minute per piece and each piece has a size of 1MB. This allows to minimize machine performance variation and to consider essentially the time it takes to finish the computation in terms of piece selection and sharing among peers. The computation is finished when at least one peer possesses all the pieces.

5.4 Computation Speedup

The test is performed using 4, 8, 12 and 16 peers and for each number of peers we test it with 3 different number of pieces : 64, 128, 192. All tests were executed 5 times. The average computation times for the three different numbers of pieces are shown in the Fig. **??**.

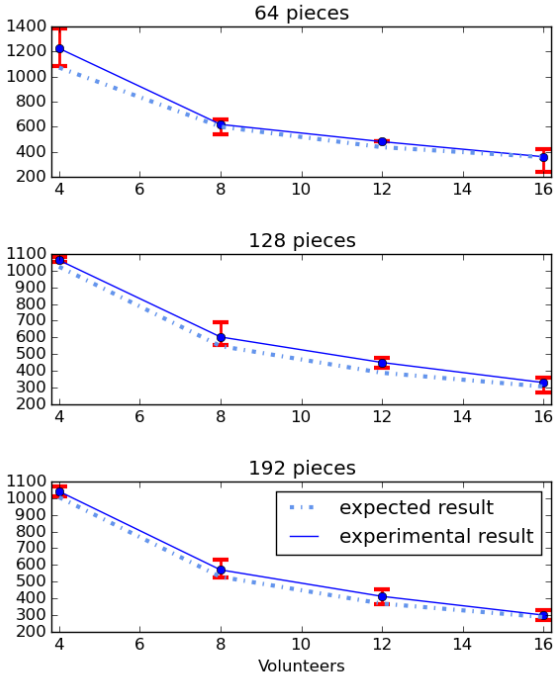

Fig. 1: Completion time (in seconds) using BitWorker, with 4, 8, 12 and 16 peers, computing 64, 128 and 192 pieces.

Figure **??** shows the expected and experienced computation time in seconds. The computation time is always slightly higher than the expected time due to

the communication delays neglected in the model. Due to the random nature of piece selection, the completion time can vary using the same setup. This is highlighted with the variance metric.

The speedup chart in Fig. **??** illustrates the relative computing speed improvement, which increases with the number of peers. It further shows the speedup increase with the number of pieces a task is split into. The results are exactly in line with our hypothesis. The experimental results are very close to the expected results.

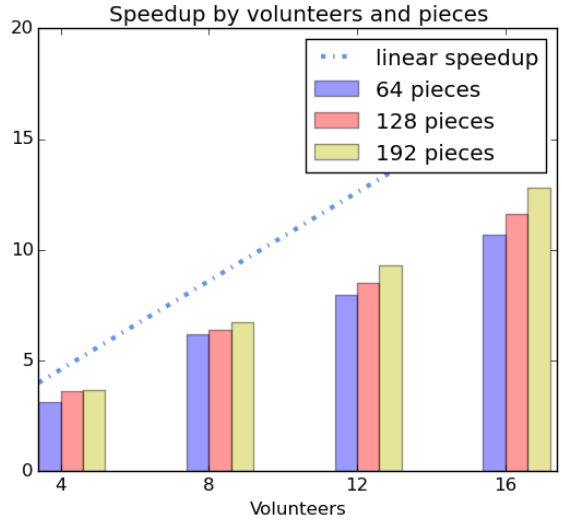

Fig. 2: Speedup

6 Future Work

6.1 Collision Avoidance

The proposed solution does not address collisions occurring when selecting a new piece to be computed. A collision is a piece computed by more than one peer. This collision probability increases when the number of pieces still to be computed decreases. Since pieces to be computed are selected randomly among the missing pieces, many peers could select the same piece. While the number of collisions decreases with a large number of pieces, collisions can be avoided, at least by collaborating with neighbour nodes. To address this problem, signaling mechanisms can be implemented: when a peer receives a "generating piece N" message, it could give a lower priority to this piece and prefer to generate pieces for which there are no hints whether other peers are working on them. The

proposed solution would be, however, only optimal for a small number of peers - in a fully connected swarm - as a torrent peer only communicates with its direct neighbours. In the case of a large number of peers, the connections of each peer are limited to a user-defined threshold value to avoid network and CPU congestion. This results in disjoint networks or low connectivity in extreme cases. In such situations, signaling to every other peer requires flooding, which results in high communication overhead.

6.2 Computation Verification

The peers do not verify downloaded pieces. The system does not provide any protection against data corruption or malicious peers sharing garbage data - the poisoning attacks. This can be avoided using extra security mechanisms such as a peer reputation system and piece verification. There have been some works [**?**,**?**] on reputation systems but the proposed solutions are not applicable if the verification is computationally too expensive. Thus, piece recomputation or verification mechanisms can be implemented at the cost of increasing the time to complete a task. Some specific problems can be verified much faster than solved but this is not always the case as most problems require full recomputation. Using recomputation or verification would not only protect against poisoning but also against data corruption.

7 Conclusions

In this paper we presented BitWorker, a fully distributed computing system. Our approach is fully based on BitTorrent and supports new functionalities through an extension of the BitTorrent protocol. Compared to other solutions such as GPU or CompTorrent, BitWorker is fully decentralized. In our experiments, we measured the speedup with varying numbers of volunteers and piece size. In these experiments, we demonstrated that the system works as expected, with a less than linear speedup due to collisions.

From our experiments, BitWorker proved to be scalable and to work autonomously. Compared to the other distributed systems, the only configuration required by BitWorker is the creation of the meta-info file and the installation of our client. Volunteers willing to contribute to a specific computation only have to launch the task using BitWorker. BitWorker then handles data management as well as work distribution. Furthermore, our prototype allows to perform any kind of computation that is splittable into pieces and can run in the shell command. We propose a fully working prototype available to the community [**?**]. DHT contributes heavily to enhance the robustness of the system by making it fully decentralized.

Acknowledgements

We would like to thank Maxime Petazzoni (Software Engineer at SignalFuse, Inc), original author, main developer and maintainer of the Ttorrent library, a

Java implementation of the BitTorrent protocol. The library was used as a basis for our implementation.

References

1. "BitWorker, https://github.com/DurandA/bitworker."
2. "http://www.jatit.org/distributed-computing/grid-vs-distributed.htm."
3. "http://folding.stanford.edu/."
4. "http://milkyway.cs.rpi.edu/milkyway/."
5. "http://boinc.berkeley.edu/."
6. D. HARTMAN, T. GLASS, S. SINHA, B. BERNHARD, O. Kiselev, and J. MATTLY, "Decentralized distributed computing system," Mar. 12 2015, uS Patent App. 14/535,850. [Online]. Available: http://www.google.com/patents/US20150074168
7. "http://www.rasterbar.com/products/libtorrent/extension_protocol.html."
8. "http://wiki.vuze.com/w/Azureus_messaging_protocol."
9. "Loewenster, A. (2008). BitTorrent DHT protocol. BitTorrent BEP, 5."
10. F. e. a. Costa, "Optimizing the data distribution layer of boinc with bittorrent," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.
11. B. Goldsmith, "Enabling grassroots distributed computing with comptorrent," in *Agents and Peer-to-Peer Computing*, 2010.
12. "Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." Consulted 1.2012 (2008): 28."
13. "http://setiathome.ssl.berkeley.edu/."
14. "http://gpu.sourceforge.net/docs/gpu_p2p.pdf."
15. "T. Mengotti, W. P. Petersen and P. Arbenz, "Distributed computing over Internet using a peer to peer network", September 2002."
16. W. P. Petersen and P. Arbenz, *Introduction to Parallel Computing.* Oxford University Press, 2003.
17. "http://planetside.co.uk/."
18. "https://wiki.theory.org/BitTorrentSpecification#Bencoding."
19. "http://www.ffmpeg.org/."
20. "http://project-rainbowcrack.com/."
21. P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology-CRYPTO*, 2003.
22. "Graham et al. 1994; Knuth 1997, p. 65."
23. K. Aberer and Z. Despotovic, "Managing trust in a peer-2-peer information system," in *ACM Proceedings of the tenth international conference on Information and knowledge management*, 2001.
24. S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *ACM Proceedings of the 12th international conference on World Wide Web*, 2003.