Dynamic State Partitioning in Parallelized Byzantine Fault Tolerance

Bijun Li, Wenbo Xu and Rüdiger Kapitza
TU Braunschweig
{*bli, wxu, kapitza*}*@ibr.cs.tu-bs.de*

***Abstract*—Recent research works have shown that applying parallelization to request processing in Byzantine Fault Tolerance (BFT) can bring significant performance improvement. Based on partitioned service state, parallelism is introduced to both agreement and execution to address performance and scalability limitations caused by the global total order of all requests. However, in case of inefficient state partitioning, expensive synchronization among partitions is expected, which leads to a considerable performance loss. To improve the efficiency of parallel processing, we present Dynamic State Partitioning (DYPART), a framework that maps service state into multiple partitions and periodically reconfigures the partitions for different usage patterns. DYPART relies on the knowledge about relations between the state objects for partitioning, which is obtained by collecting request dependencies. It utilizes a high-performance graph partitioning algorithm to ensure that the resulting state partitions can achieve both workload balance and low synchronization among partitions. Our evaluation of a key-value store shows that compared to a random partitioning, DYPART can improve the performance by at least 40%.**

1. Introduction

Byzantine Fault Tolerance (BFT) protocols [1] have been proven to benefit from allowing *non-conflicting*, *independent* requests to be executed in parallel [2], [3], [4], [5]. Besides parallel execution, recent research works have shown that ordering of independent requests can be parallelized as well in order to further accelerate the processing speed, while still holding strong consistency. For instance, in our previous work SAREK [6], we proposed a parallel ordering framework that *logically* partitions the application's state to exploit parallelism for both agreement and execution in BFT systems. The number of partitions can be equal to the number of replicas of the system. This way, each replica can be the leader of one specific partition, establishing an order upon the requests that access only the objects of this partition. It also supports operations that span multiple partitions, noted as *cross-border requests* and ensures deterministic executions. Evaluation results reveal the facts that using SAREK can gain a throughput increase by a factor of 2, while handling cross-border requests causes extra synchronization overhead and results in performance loss.

Acknowledgments: This research was supported by the German Research Council (DFG) under grant no. KA 3171/1-2 and DI 2097/1-2 ("REFIT"), as well as under grant no. FOR 1800/2 and KA 3171/5-1 ("CCC").

Reducing the probability of cross-border requests requires the system state to be properly partitioned. The knowledge about the request dependencies can contribute to high quality state partitions to support efficient parallel processing. These request dependencies can be derived from the object access pattern: Certain objects are more often accessed together, whereas some other ones are accessed solely. However, on the one hand, it is rather impossible to gain the accurate and entire knowledge of an application's state before actually running it; on the other hand, as long as clients request dependencies vary, by either creating new state objects or changing the objects access pattern, a static partitioning solution can quickly become outdated and a performance loss could be expected for the new requests. Moreover, having balanced workload on each partition also plays an important role in achieving good performance: Objects should be distributed as equal as possible to involve more parallelization to the processing, in order to fully utilize the computing power of modern multi-core machines. In a word, keeping a low cross-border request rate as well as balanced workload should be taken into account when utilizing parallel BFT systems.

In this paper we present DYPART, a dynamic state partitioning framework. DYPART collects and divides each application's state into partitions, then applies and reconfigures the state partitions on the fly for performance improvement. Assume that at the system startup, replicas do not know the objects access patterns of any clients and therefore have no knowledge about request dependencies, resulting in a default state partitioning. When ordering and executing the requests, each replica monitors the request dependencies and maps them to a graph that represents the relations between the state objects. The state partitioning update is associated with the checkpoint mechanism to guarantee its determinism and keep the overhead as low as possible. Once the checkpoint threshold is reached, besides creating a new checkpoint, each replica also invokes a graph partitioning algorithm to partition the graph's vertex set into blocks with minimized weight of edges running in between, and possibly of similar sizes. This ensures that the resulted new state partitions can keep a low cross-border request rate as well as fully utilize the potential of parallel request processing.

Since the checkpoint creation process is invoked periodically, the state partitions can be updated with new request dependencies on a checkpoint-interval cycle. This enables the reconfiguration feature of DYPART and makes it adaptable to any dynamically changing request dependencies.

Moreover, the reconfiguration does not involve any object transferring but only changes the logical partitions, making it efficient to be applied.

We have implemented DYPART on top of the SAREK prototype, by attaching a module of the dynamic partitioning method to the checkpoint mechanism. We evaluated the prototype with microbenchmarks, where requests are generated based on a social network dataset [7] and access multiple objects simultaneously. Evaluation is conducted to show the performance improvement by using DYPART, compared to SAREK's original naive solution.

In particular, this paper makes the following contributions:

- It presents DYPART's approach to creating the knowledge of an application's state with respect to request dependencies, and leveraging a graph partitioning algorithm to divide the state into blocks.
- It introduces the reconfiguration feature for dynamically updating replicas' partitioning knowledge in order to adapt to different requests.
- It implements a prototype of DYPART, which enables the parallel BFT system to handle the trade-off between low cross-border request rate and balanced workload.

The remainder of the paper is organized as follows: Section 2 explains SAREK , the utilized BFT system. Section 3 discusses the system model and presents the design of DYPART with implementation details. Section 4 presents the evaluation result. Section 5 summarizes related works while Section 6 concludes the paper.

2. SAREK Background

DYPART aims to improve the efficiency and flexibility of the partitioning mechanism of the parallel ordering framework SAREK [6]. Thus we firstly give a brief introduction to the base system before explaining the details of DYPART.

2.1. SAREK System Model

Most existing BFT systems assume a total order upon write requests that cause state update, as the service state is considered indivisible that the processing of each write request is dependent on all others. However, for many applications such as key-value stores or web applications, this assumption is too pessimistic. In fact, only those requests accessing shared state should be ordered to guarantee consistency, while the ones do not share state could be executed in parallel. Based on this idea the basic structure of SAREK is built, as shown in Figure 1.

In SAREK, a replica possesses the *entire* state as in many other single-leader based BFT systems. The difference is that in SAREK *multiple BFT agreement instances* are running in parallel within each replica. Each of the instances is responsible for a logical partition of the state and maintains a partial order upon the requests accessing that partition.

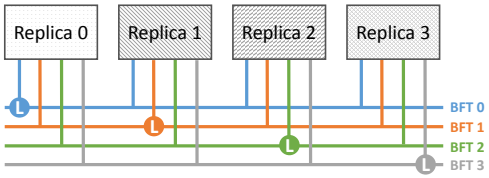

Figure 1. Multi-leader based BFT system.

The leaders of the BFT instances are distributed among the replicas, which leads to a multi-leader based approach for both ordering and execution of independent requests. Figure 2 shows the architecture of a replica in SAREK.

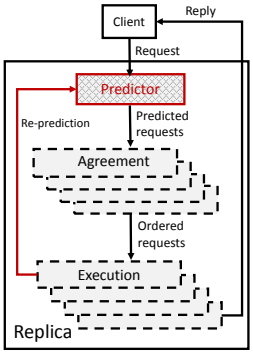

Figure 2. Architecture of the SAREK system.

The state partitioning mechanism of SAREK is implemented with a *predictor* component. It contains an application-specific PREDICT() function that is created by each replica during initialization phase. The original PREDICT() function in SAREK applies a *modulo* calculation upon an accessed object to decide which partition it belongs to. This naive mechanism achieves well balanced partition size, but lacks of efficient means to gain the insight of the relations between the objects.

In SAREK, clients send a request to *all* replicas via broadcast. Each replica then performs the PREDICT() to decide the responsible BFT instance(s) for ordering, according to the objects to be accessed. For a request that accesses a single partition, only one responsible BFT instance across all replicas will handle it. The independently ordered requests can be executed in parallel and their partition accessing behaviors are monitored by SAREK. Once a request attempts to access partitions not predicted, causing a *mis-prediction* due to unforeseen data dependency or insufficient application knowledge, it will get detected and triggers a *re-prediction* process to tackle with the case. The missing partition(s) will be added to the request to ensure that it will be then ordered by the corresponding BFT instance(s). This way, system consistency is guaranteed even if the state partitioning and predictor are not perfect.

2.2. Handling Cross-Border Request

For cross-border requests that access multiple partitions, additional overhead is introduced for consistency guarantee.

At agreement, each cross-border request needs to be ordered by all predicted BFT instances, which considerably increases the workload on the replica network. For execution, only one deterministically chosen BFT instance across all replicas will actually execute the request, while others only synchronize with it once the execution is done. This ensures consistency and determinism by executing a cross-border request for exactly one time, yet at a price of expensive synchronization. To avoid cross-border requests, the objects that are frequently accessed together should be located in the same logical partition.

3. DyPart Design

In this section, we reveal the details of DyPart design and the integrated high-quality graph partitioning algorithm.

3.1. The DyPart Approach

Figure 3 shows an overview of the Sarek system featuring DyPart for state partitioning and illustrates how they conceptually interact. A request is firstly handled by the predictor component (①). The PREDICT() function maps each object to a corresponding partition. In case an object is *unknown* to any partition so far, it will be mapped to the default partition with the smallest ID number. The responsible BFT instances will then order and execute the request by following the rules defined in Sarek.

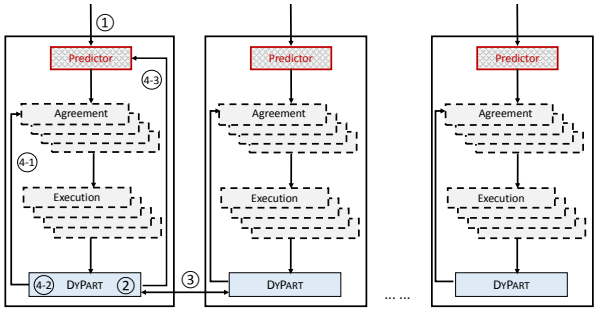

Figure 3. Overview of DyPart-backed Sarek system.

The actual process of DyPart is associated with the checkpoint mechanism of Sarek (②). To acquire the request dependencies of one application, DyPart logs the co-occurrences of the objects and their frequency by recording every successfully executed request. Meanwhile the checkpoint mechanism will check if a predefined threshold has been reached after each request execution. Within a replica, each BFT instance independently maintains a counter to indicate how many requests have been processed since the last stable checkpoint. If the counter of any BFT instance reaches the threshold, a *pre-checkpoint* message with the replica ID and the *expected next* checkpoint sequence number is sent out to all other replicas (③). Meanwhile the next checkpoint sequence number of that instance is increased by one. Each replica keeps the number of the last stable checkpoint, thus the pre-checkpoint messages with a stale number will be discarded. Once a replica collects $2f + 1$ pre-checkpoint messages with the same sequence number, it confirms the attempt to create a checkpoint with this sequence number and generates a *create-checkpoint* request. Due to different execution speeds of the BFT instances across the replicas, it is necessary to have a synchronization before creating the checkpoints. Therefore the create-checkpoint request is resembling a cross-border request that involves *all* logical partitions. All non-faulty replicas will eventually generate this request.

The create-checkpoint request is then ordered and executed in the same way of handling a normal client request ((4-1)): All involved BFT instances order the request but only one instance executes it while others only synchronize once the execution is complete. The execution triggers the creation of a checkpoint and updates the last stable checkpoint number on each replica. As the create-checkpoint request involves all partitions, it works as a "*barrier*" to ensure the checkpoint is created across all instances among all replicas in a deterministic manner, despite different execution speeds.

Besides creating a checkpoint, executing the create-checkpoint request also causes DyPart to update the logical partitions of the state by reconfiguring the predictor. It immediately invokes the graph partitioning algorithm that utilizes the collected request dependencies as inputs and outputs the new partition knowledge ((4-2)). The predictor is instantly provided with this knowledge to update its PREDICT() function for the new incoming requests ((4-3)). It will not cause data moving but only change the mapping between the objects and the partitions. Once the predictor has been reconfigured, the execution of the create-checkpoint request is complete. It is possible that after reconfiguring the predictor, some already-ordered requests are nevertheless predicted with the outdated knowledge, due to the time difference between triggering the state partitioning and applying the new partitions. This will cause *mis-predictions* during execution, and can be handled by Sarek with its *re-prediction* mechanism.

Optimization can be done by applying exponential smoothing technique to smooth the continuously updated partition knowledge, as the partition update is triggered periodically. For example, the later the request dependencies are obtained, the higher smooth factor is given to the new partition knowledge calculation. This way, the new knowledge can always reflect the most recent changes in the objects over time and be resilient against unexpected short-term fluctuations.

3.2. Graph Partitioning Algorithm

The service state is modeled as a graph in DyPart : The individual objects are the vertices, and the weight of an edge indicates how often the two objects are accessed together. Thus the problem of partitioning the state can be converted to the graph partitioning problem, where a graph is cut into smaller pieces, and the aggregated weight of edges running between different pieces should be as small as possible. The

similar idea can be found in high performance computing, which distributes the work to processors in a static manner [8]. DYPART features this technique to minimize the synchronization among partitions and ensure load balance.

In DYPART, we use the amount of requests that solely access an object during a checkpoint interval as the vertex weight of that object. The edge weight, on the other hand, is abstracted from the requests accessing multiple objects: For a set of objects that are accessed together in one request, each pair of the objects contributes to the edge weight between them. As a result, the more frequently a pair of objects are accessed together, the less likely they are assigned to different partitions.

We choose the Karlsruhe High Quality Partitioning (KaHIP) [9], [10], [11] as the graph partitioning algorithm in DYPART for its good performance and simplicity of integration [12]. KaHIP combines the local improvement algorithm based on max-flow min-cut computations and global search strategies to achieve fast graph partitioning. For more details about KaHIP we refer to its documentations.

DYPART provides an interface for integrating the graph partitioning algorithm into the codebase of SAREK as a plug-in service. Each time when the state partitioning is triggered, DYPART immediately transforms its collected request dependencies into a formatted input graph for KaHIP to perform the partitioning. Various options can be added to tune the multi-level graph partitioning program, e. g. to define the number of partitions, to satisfy different quality requirements, or to pursue a balance of the edges among the partitions as well as the vertices. As we aim to achieve both low cross-border request rate and a balanced workload upon the partitions, we choose high quality partitioning as well as balanced vertices and edges for creating the output file. DYPART then reads the partition knowledge from the output file into its predictor to update the PREDICT() function.

4. Evaluation and Discussion

We implemented a prototype of DYPART and evaluated its performance with a comparison to the original partitioning method of SAREK, i. e. the modulo-based partitioning.

4.1. System Setup

We use a cluster of four machines to host the replicas (hence we consider $f = 1$ faults) and a dedicated machine to generate client workloads. Each physical machine is equipped with an Intel Core i7-6700 quad-core processor running at 3.4 GHz with Hyper-threading activated, as well as 24 GB of memory. All the machines are running 64-bit Ubuntu 16.04 with OpenJDK 1.8.

We evaluate and compare the performance of SAREK featuring DYPART (noted as *DyPart*) with the original modulo-based SAREK (noted as *baseline*).

4.2. Microbenchmark Setup

We use a hash-map-based key-value store for the microbenchmark evaluation to measure the performance, including throughput and latency, of the two prototypes. The key-value store provides the following functionality: It receives each request that accesses one or multiple objects in the store, and creates a reply for the request. If a request accesses only one object, it is handled by the *put()* operation for writing data; otherwise it requires a *putall()* operation to simultaneously access multiple data.

Given that the state partitioning of DYPART relies on the underlying request dependencies of different applications, the input must be inherently representing the relations of the accessed objects. Inputs that lead to random object accessing are not suitable for evaluation purpose. We consider social-network like datasets as the source of generating cross-border requests that possibly access more than two objects. More specifically, for the preliminary evaluation we choose a dataset of co-occurrence network from the Stanford Graph Base [7], which represents the interactions between the characters in Victor Hugo's *Les Misérables*. We abstract the co-occurrences of the characters in each scene of each subchapter from the dataset and store them in a list.

Each character is considered as an object of the service state and a vertex of the graph. Between a pair of characters, there exists an undirected edge. The edge of the graph is weighted by the amount of co-occurrence of that pair. Clients randomly select co-occurrence entries from the list to form the requests accessing multiple (up to nine) objects.

The original SAREK prototype relies on the modulo-based partitioning method to get four partitions of approximately equal size. For all experiments we deploy up to 200 clients to saturate the system. The final result is calculated as an average of the results from multiple runs. No batching is used since it is an orthogonal approach that will independently influence the results.

4.3. Microbenchmark Results

To gain an insight of the efficiency of the modulo-based partitioning method and DYPART, we first conduct an experiment to calculate the percentage of cross-border requests that span multiple partitions. We go through the request list to figure out how many partitions are accessed by each request. Figure 4 shows that, for the modulo-based partitioning method, only 45% requests access a single partition, while for DYPART 68% requests are handled by one partition, which indicates a big advantage of DYPART in reducing the cross-border request overhead. When it comes to the requests that access more than one partition, DYPART is able to reduce the 2-partition requests by 14%, and 3-partition requests by 6%. For 4-partition requests that bring significant synchronization overhead to the system, DYPART can totally eliminate this category, while the baseline results in 3%. This proves that DYPART has managed to achieve its goal of keeping a low cross-border request rate.

We also analyze the workload balancing factor of the two prototypes, and demonstrate the results in Figure 5. For the baseline, the modulo operation divides the state into four partitions of approximately the same size. Since no request dependency but only the amount is considered

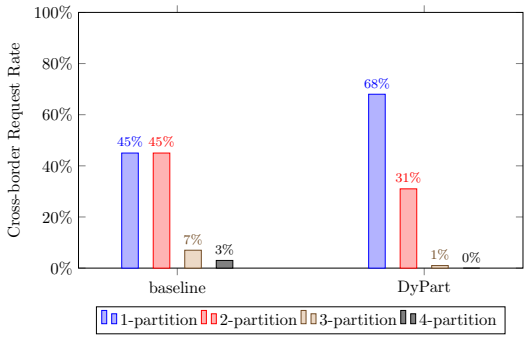

Figure 4. Percentage of different types of requests.

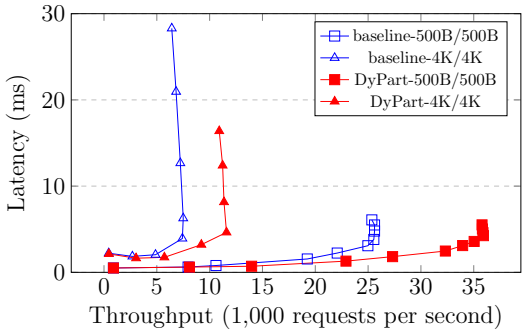

Figure 6. Throughput and latency of handling cross-border requests.

for partitioning, partitions of equal size might instead cause more cross-border requests and eventually become more expensive. On the other hand, DyPart gives priority to partitioning quality as well as balanced vertices and edges, although some partitions might contain slightly less objects than others.

To measure the throughput and latency performance, we apply two request/reply sizes in our evaluations: 500 bytes/500 bytes and 4 kilobytes/4 kilobytes. An increasing amount of workload is generated by up to 200 clients to test against the replicas. From Figure 6 we can learn that by the time that the whole system is saturated (latency grows but throughput not), Sarek with DyPart can achieve a much higher throughput than the baseline. For a smaller message size of 500 B/500 B at least 40% performance improvement can be observed, and for the 4 KB/4 KB case, DyPart shows nearly 50% higher performance.

Figure 7 shows the impact of applying state partitioning and performing the re-prediction mechanism (see Section 3.1) on the latency. The measurements are taken every 20 ms, and we show a part of the result after the warm-up phase. Using KaHIP to partition the input graph (77 vertices with more than 500 edges) takes on average 56 ms, and the possibly resulted re-prediction might also introduce limited overhead. As a result, a high latency spike that lasts for 40-60 ms can be observed each time when a checkpoint is triggered. By comparing the results of different checkpoint intervals, we have learned that increasing the interval can significantly reduce the frequency of having such spikes. Further optimization can be done to reduce the spikes, by decoupling the graph partitioning computation from the deployment of the new partition knowledge. For example, when a checkpoint is triggered, DyPart does not immediately use the results from the current checkpoint interval for the reconfiguration. Instead, it applies the partition knowledge from the last checkpoint and directly completes the checkpoint processing. In parallel, the partitioning algorithm uses the newly collected request dependencies from the latest checkpoint interval to generate new partition knowledge. This way, the graph partitioning computation does not block the executions at each checkpoint.

5. Related Works

DyPart utilizes a graph partitioning algorithm to improve the performance and workload balance in a parallelized BFT system. We discuss related works in a reference to both parallel computing in state machine replication and applications of graph partitioning.

a) Parallel Computing: P-SMR [13] shows that parallelism can be achieved by mapping non-conflict requests to different multi-cast groups, according to application-specific semantics. A follow-up work [14] proposed an optimistic mapping approach as well as a roll-back mechanism to

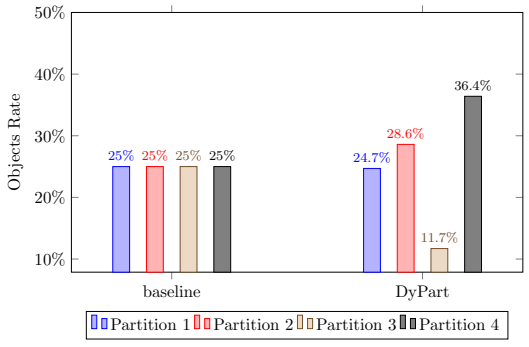

Figure 5. Percentage of objects in each partition.

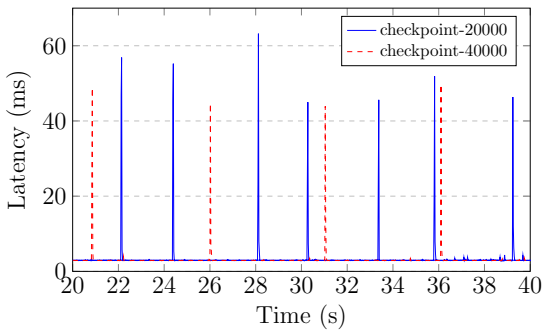

Figure 7. Impact of state partitioning and re-prediction on latency.

resolve the inconsistency issue caused by inaccurate assumptions for the mapping. Compared to them and based on SAREK , DYPART is able to handle arbitrary failures and needs no roll-backs for handling mis-predictions.

Alchieri et al. presents a new way to dynamically reconfigure the degree of parallelism of replicas for different workloads [15]. In case of high conflicting rate operations, it requires fewer threads to reduce the synchronization overhead; and for low conflicting rate operations, more threads are used to maximize the performance. In [16] a high performance recovery method for parallel state machine replication is proposed in order to prevent overhead of using dependency detection for keeping consistency. It includes two techniques: The speedy recovery that allows processing new commands concurrently with old commands if they are independent; and the on-demand recovery that is able to recover only a fraction of the state. The dependency among commands is represented as a dependency graph and is executed when parallel execution is efficient considering resource availability. The targeted system model differentiates DYPART from these works. Besides that, DYPART reconfigures the state partitioning instead of computing resources to adapt to different workloads. This guarantees that DYPART can always perform under full computing power.

b) Graph Partitioning Applications: Many research works have explored the potential applications of graph partitioning. Here we summarize those that share a similar concept as DYPART. Newman shows in [17] how to apply graph partitioning algorithms to solve community detection problems. It is done by mapping the common community inference methods onto the min-cut graph partitioning problem.

The work of Glantz et al. [18] explores efficient static mapping of parallel processes to processing elements of a parallel system. There an application graph is created to represent the application's computations and their dependencies, and partitioned into blocks of equal size. Different algorithms are evaluated to find out an efficient mapping between the blocks and the processing elements with minimized communication costs.

6. Conclusion

In this paper we presented DYPART, a dynamic partitioning framework that utilizes request dependencies for state partitioning in the parallelized BFT system SAREK. It collects and analyzes the application's state with its objects accessing pattern, and models the pattern into a graph to represent the relations of the objects. A high-quality graph partitioning algorithm is integrated into DYPART for dividing the state graph into partitions for the purpose of having the least interactions as well as keeping a workload balance among partitions. Each replica adapts the new partition knowledge in order to achieve good performance under that request pattern. The dynamic update of the partition knowledge is associated with the checkpoint mechanism of SAREK so that it can be applied consistently across all BFT instances of all replicas. We implemented a prototype of DYPART based on SAREK and conducted measurements with microbenchmarks. The results show that by periodically applying DYPART, SAREK can gain at least 40% performance improvement compared to the original modulo-based solution.

References

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. of the third USENIX Symp. on Operating Systems Design and Implementation (OSDI '99)*, 1999, pp. 173–186.

[2] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proc. of the 2004 Int'l Conf. on Dependable Systems and Networks (DSN '04)*. IEEE, 2004, pp. 575–584.

[3] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 295–308.

[4] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin *et al.*, "All about eve: Execute-verify replication for multi-core servers." in *OSDI*, vol. 12, 2012, pp. 237–250.

[5] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. ACM, 2015, pp. 173–184.

[6] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "Sarek: Optimistic parallel ordering in byzantine fault tolerance," in *Dependable Computing Conference (EDCC), 2016 12th European*. IEEE, 2016, pp. 77–88.

[7] "Les misrables co-occurance network," https://people.sc.fsu.edu/~jburkardt/datasets/sgb/jean.dat.

[8] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*. Springer, 2016, pp. 117–158.

[9] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[10] "Kahip: Karlsruhe high quality partitioning," http://algo2.iti.kit.edu/schulz/software_releases/kahipv2.00.pdf.

[11] "Kahip v2.0," https://github.com/schulzchristian/KaHIP/.

[12] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Graph partitioning and graph clustering*. American Mathematical Soc., 2013, vol. 588.

[13] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 368–377.

[14] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE, 2014, pp. 57–66.

[15] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*. IEEE, 2017, pp. 104–113.

[16] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "High performance recovery for parallel state machine replication," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 34–44.

[17] M. E. Newman, "Community detection and graph partitioning," *EPL (Europhysics Letters)*, vol. 103, no. 2, p. 28003, 2013.

[18] R. Glantz, H. Meyerhenke, and A. Noe, "Algorithms for mapping parallel processes onto grid and torus architectures," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 236–243.