Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing

Avinash Maurya, Bogdan Nicolae, M Mustafa Rafique, Thierry Tonellot, Franck Cappello

Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing

Avinash Maurya*, Bogdan Nicolae†, M. Mustafa Rafique*, Thierry Tonellot‡, Franck Cappello†
*Rochester Institute of Technology, USA
†Argonne National Laboratory, USA
‡Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco, Dhahran, Saudi Arabia
Email: *{am6429, mrafique}@cs.rit.edu; †{bnicolae, cappello}@anl.gov; ‡thierrylaurent.tonellot@aramco.com

***Abstract*—Efficient checkpointing of distributed data structures periodically at key moments during runtime is a recurring fundamental pattern in a large number of uses cases: fault tolerance based on checkpoint-restart, in-situ or post-analytics, reproducibility, adjoint computations, etc. In this context, multi-level checkpointing is a popular technique: distributed processes can write their shard of the data independently to fast local storage tiers, then flush asynchronously to a shared, slower tier of higher capacity. However, given the limited capacity of fast tiers (e.g. GPU memory) and the increasing checkpoint frequency, the processes often run out of space and need to fall back to blocking writes to the slow tiers. To mitigate this problem, compression is often applied in order to reduce the checkpoint sizes. Unfortunately, this reduction is not uniform: some processes will have spare capacity left on the fast tiers, while others still run out of space. In this paper, we study the problem of how to leverage this imbalance in order to reduce I/O overheads for multi-level checkpointing. To this end, we solve an optimization problem of how much data to send from each process that runs out of space to the processes that have spare capacity in order to minimize the amount of time spent blocking in I/O. We propose two algorithms: one based on a greedy approach and the other based on modified minimum cost flows. We evaluate our proposal using synthetic and real-life application traces. Our evaluation shows that both algorithms achieve significant improvements in checkpoint performance over traditional multi-level checkpointing.**

***Index Terms*—GPU checkpointing, asynchronous I/O, peer-to-peer collaborative caching, multi-level checkpointing**

I. Introduction

High-Performance Computing (HPC) applications produce massive amounts of distributed intermediate data during runtime that needs to be checkpointed concurrently. This is a fundamental I/O pattern used in a wide range of scenarios: fault tolerance based on checkpoint-restart, offline or in-situ analytics, reproducibility (validate intermediate states in addition to the end result), etc.

Of particular interest is the use of checkpointing for the purpose of revisiting previous states in order to advance a computation. For example, adjoint computations are often used to adjust the parameters of a function used to predict an outcome by minimizing the differences between the actual and expected output. Adjoint computations usually consist of two stages: a forward pass used to obtain the actual output that checkpoints the intermediate states, followed by a backward pass that uses the checkpoints in reverse order to make the adjustments. Such adjoint techniques are ubiquitous, from climate and ocean modeling to seismic imaging in the oil industry. The training of deep learning (DL) models is also an adjoint computation: techniques such as stochastic gradient descent use a forward pass and a backward pass in order to adjust the parameters of a deep neural network (DNN) model composed of multiple layers.

The increasing performance requirements of HPC and DL applications result in the rapid adoption of accelerators such as GPUs. Naturally, this also introduces the need to checkpoint more frequently, especially for scenarios such as adjoint computations, where the checkpoint intervals are in the order of milliseconds. In this context, GPUs are equipped with high bandwidth memory (HBM) capable of keeping up with the massive amount of computational resources without causing data bottlenecks. Unfortunately, HBM is expensive, and thus its capacity is limited. Consequently, applications commonly do not have enough room to keep all checkpoints in HBM for the entire runtime for large problem sizes. Therefore, it is essential to devise checkpointing techniques that combine fast tiers of limited capacity (e.g., HBM of GPUs) with slower tiers of larger capacity (e.g., DDR4 host memory).

In this context, asynchronous multi-level checkpointing techniques are popular. They rely on a simple idea: all processes write their checkpoints to a fast tier, then flush them in the background to a slower tier while the application continues running in the foreground. Given that the slow I/O operations needed to flush the checkpoints from the fast tiers to the slow tiers can often be overlapped with the application runtime, multi-level checkpointing techniques also open up opportunities to mask data movement latency either partially or entirely. However, the constant accumulation of checkpointing data on the fast tiers means there is often not enough free space to write the entire checkpoint on the fast tiers. Therefore, the processes need to directly write to the slow tiers in a blocking fashion, which leads to increased I/O latency.

In a quest to alleviate this issue, various approaches have been explored to reduce the checkpoint sizes, e.g., compression, decimation, and interpolation. These techniques mitigate but do not fully eliminate this challenge. For example, when compression is used, the compression ratio may be high for some checkpoints but low for others. Thus, even if most processes are capable of fitting their checkpoints in the free

space of their fast local tiers, it is enough for some stragglers that were not capable of doing so to delay the whole group especially for tightly coupled applications.

In this paper, we aim to solve the aforementioned challenge. Our approach is based on two key observations. First, if there is a significant difference between the checkpoint sizes, then the processes will fall into two groups: some will have a checkpoint size that does not fit in the free space of their fast local tiers, while others will have free space left even after completely writing their checkpoint to their fast local tier. Second, modern accelerators, particularly GPUs are typically linked with high bandwidth interconnects that are significantly faster than the links between the accelerators and the host memory. This is also true for other classes of fast vs. slow tiers, e.g., the bisection bandwidth of remote transfers between main memories and SSDs is faster than the aggregated bandwidth of a parallel file system. Based on these observations, we propose sending some of the checkpointing data to the processes with spare free space to reduce the amount of data that needs to be written directly to the slow tiers.

While simple as a concept, this is a complex optimization problem, i.e., to obtain a schedule which identifies processes which must transfer a specific amount of data to one or more processes for minimizing execution stalls during checkpoints. We refer to this as the *collaborative multi-level checkpointing* problem. This is non-trivial both because of link heterogeneity, i.e., each fast tier may be connected to multiple other fast tiers with links of different bandwidths, and because these links can be used concurrently, i.e., it is suboptimal to send all data over the fastest link even if the destination has enough capacity to fit the entire data. To this end, we contribute with two algorithms, which we evaluate in a series of simulated scenarios using both synthetic and real-life traces of checkpoints of variable sizes. We summarize our contributions below:

- We formulate the problem of collaborative multi-level checkpointing, introducing a set of general considerations and assumptions for the design of scheduling algorithms (§ II).
- We propose two scheduling algorithms based on a greedy approach and a variation of min-cost max-flow in transport networks, which leverage the differences between the checkpoint sizes and the peer-to-peer interconnects of the fast tiers to minimize the blocking time of collaborative multi-level checkpointing (§ IV).
- We evaluate our proposed algorithms using the performance model in a variety of experimental scenarios using both a real-world application trace and a synthetically generated trace. Our evaluation shows that our algorithm based on min-cost max-flow approach outperforms the greedy and traditional checkpointing approach while incurring comparable execution time (§ V).

II. Problem Formulation

Collaborative multi-level checkpointing is a fundamental I/O pattern applicable in a large variety of scenarios. For simplicity, consider the case of a single node equipped with

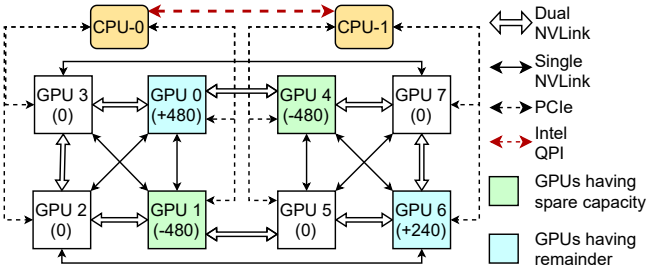


Fig. 1: Nvidia DGX-1 GPU interconnect [1].

N GPUs, each of which is assigned to a process i. At a given moment, all processes need to simultaneously checkpoint some data structures stored on their GPUs that add up to size $Ckpt_i$. Each GPU G_i has a free space F_i. We call the processes for which $Ckpt_i > F_i$ *senders* and those for which $Ckpt_i < F_i$ *receivers*. Each GPU G_i is connected to a subset of the other GPUs through a link with a maximum bandwidth B_{ij} (with $B_{ij} = 0$ if G_i is not connected with G_j), as well as to the host memory H through a link of bandwidth B_h. Each sender can concurrently use any of its peer-to-peer links and the device-to-host link to transfer an arbitrary amount of its *remainder* ($C_i = Ckpt_i - F_i$) to the receivers or to the host memory. The goal is to identify the amount of data that each sender needs to transfer to each receiver or host memory such that the following two conditions are satisfied: (1) the total amount of data sent to each receiver j is smaller or equal to its total *spare capacity* $S_j = F_j - Ckpt_j$; and (2) the total duration of all concurrent data transfers (denoted t_{ckpt} and referred to as *blocking time*) is minimized. We call this the *optimal schedule*.

We elaborate this problem by considering Nvidia DGX-1 [1], which has a hybrid cube-mesh GPU interconnect with 8 GPUs that are interconnected with each other using either single (24 GB/s) or dual NVLinks (48 GB/s), as well as with the host memory through PCIe links (12 GB/s). As shown in Figure 1, the senders have a positive amount of data (remainder), the receivers have a negative amount (spare capacity), and the GPUs whose local checkpoints fit exactly in their free space have a spare capacity of zero. In this case, G_0 could send all the data to G_4 over a fast dual NVLink, but this is suboptimal for two reasons: (1) if G_0 writes $1/4$ of its data to the host memory in parallel, it would finish 25% faster; (2) G_6 is not linked to G_1, therefore it is forced to write its data to the host memory while G_1's spare capacity remains unused. Thus, accurately identifying senders and corresponding receivers, and the amount of data that should be sent to each receiver is critical in reducing the overall checkpoint time.

Once the optimal schedule is determined, collaborative multi-level checkpointing can be easily implemented by splitting the checkpoints into chunks on the senders, which then transfer them to the receivers. Next, the chunks can be transferred from their intermediate locations to the host memory in the background while the application continues its execution.

In this paper, we focus on multiple GPUs on a single node. However, it is important to note that the problem can be

generalized to any number of compute nodes and any other combinations of fast and slow storage tiers, e.g., the main memory of compute nodes and the parallel file system.

III. Related Work

Checkpoint-Restart: Both loosely and tightly coupled applications use checkpoint-restart to support resilience. When I/O bandwidth is a concern (especially for tightly-coupled HPC applications running at a large scale), *multi-level checkpointing* [2] can be used to leverage complementary strategies (partner replication, erasure coding) adapted for HPC storage hierarchies in asynchronous mode [3]. Currently, such techniques use local storage independently on each compute node via a single shared link, but can be complemented to leverage local storage of remote nodes. Additionally, checkpoint-restart techniques are also used for accommodating on-demand jobs with batch jobs [4], [5] and workload migration [6], [7].

Checkpoint size-reduction: There are a variety of techniques that can be used to reduce the checkpoint sizes for specific objectives: space (i.e., within the same checkpoint), time (i.e., across consecutive checkpoints), scope (i.e., individual checkpoints vs. global checkpoints aggregated from all processes). They include lossy [8] and lossless [9], [10] compression, de-duplication [11], [12], incremental checkpointing based on dirty page tracking [13], etc. Such approaches can be used together with our proposal and often lead to differences in the checkpoint sizes leveraged by our proposal.

GPU checkpointing: With the increasing popularity of GPUs, checkpointing techniques are used both for migration [14], [15] and resilience [15]–[20]. System-level checkpoint-restart libraries, e.g., CheCUDA [16] and NVCR [15] transparently record and replay all the memory-based API calls. Efforts such as MLBS [21] and CheckFreq [22] leverage multi-level memory subsystem starting from GPU memory to minimize checkpoint time. However, none of these approaches analyze the imbalance of checkpoint sizes across multiple devices, nor do they leverage peer-to-peer transfers to reduce the checkpointing overheads.

Schedule optimization: Maximum flows in transportation networks have been studied for decades, and the complexity of the algorithms is constantly improving [23]. Such algorithms can be extended to produce the minimum cost for a given flow [24], which is closely related to our problem if we model it using graph theory. However, these algorithms assume that each transported unit incurs a cost that is the sum of the costs of all links it passes through, while in our case the cost (i.e., blocking time) only increases if we transfer more data over the slowest link. Linear programming [25] is a well-researched area for solving complex optimization problems, with a variety of tools available. However, such tools are typically heavyweight and are not designed to be easily used in real-time for system-level runtimes.

To the best of our knowledge, *we are the first to study the problem of optimal schedule for collaborative multi-level checkpointing* for different checkpoint sizes and spare capacities, heterogeneous peer-to-peer links, and concurrent transfers.

Algorithm 1: Greedy-based Scheduling Algorithm

Input : number of processes N, remainder C, spare capacity S, bandwidth matrix B
Output: total blocking time t_{ckpt}, schedule P

```
// GPU G_i with largest remainder
while ∃i = argmax(C|C_i > 0) do
    while C_i > 0 do
        // fastest G_j with spare capacity
        if ∃j = argmax(B_i|B_ij > 0 and S_j > 0) then
            k ← min(C_i, S_j)
            C_i ← C_i − k
            S_j ← S_j − k
            P_i ← P_i ∪ {(k, j)}
            t_ckpt ← max(t_ckpt, k/B_ij)
        else
            break
    if C_i > 0 then
        // Transfer to host
        C_i ← 0
        P_i ← P_i ∪ {(C_i, H)}
        t_ckpt ← max(t_ckpt, k/B_h)
return (t_ckpt, P)
```

IV. Optimal Schedule for Collaborative Multi-Level Checkpointing

In this section, we propose two lightweight algorithms to solve the problem introduced in § II.

A. Greedy-Based Schedule

Our first algorithm is based on a greedy strategy. While it does not guarantee to find the optimal solution, it is easy to implement and has a short runtime due to low computational complexity. The key idea we exploit in this context is to transfer as much data as possible from the sender with the highest checkpoint remainder to the fastest receivers with spare capacity it is connected to. The intuition behind this idea is that the sender with the highest remainder is likely to be forced to transfer most of its remainder to the host memory if other senders occupy the spare capacities of the receivers, thereby it would cause the longest transfer delay in the whole group. Thus, by sorting senders in descending order of their remainder, we minimize the occurrence of this undesired effect.

The complete algorithm is listed in Algorithm 1. The notations used in the algorithm are consistent with those used in § II. Specifically, for each sender i, $C_i > 0$ is the remainder after it partially wrote its checkpoint to G_i, leaving the spare capacity $S_i = 0$. Similarly, for each receiver j, $C_j = 0$ and $S_j > 0$. The algorithm computes the total blocking time t_{ckpt} and the schedule P: for each sender i, P_i is a set of tuples (k, j), where j is a receiver of the checkpoint chunk of size k. In the worst case, each node is connected with every other node, therefore the complexity of this algorithm is $\mathcal{O}(N^2)$

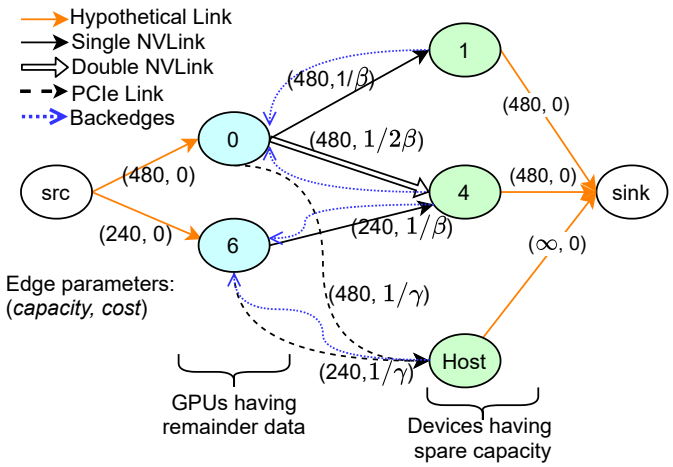


Fig. 2: Min-time, max-flow algorithm: flow graph corresponding to the example in Figure 1, using $\beta = 24\ GB/s, \gamma = 12\ GB/s$.

(which supersedes the complexity of sorting the senders in descending order of their remainder).

For the example illustrated in Figure 1, the greedy approach chooses G_0 first and uses the fastest link to G_4 to transfer all of its remainder, thereby filling the spare capacity of G_4 (i.e., $C_0 = 0$ and $S_4 = 0$). At this point, G_6 is forced to transfer its remainder to the host memory H because none of its peers have any spare capacity left. The transfer time for G_0 is 10 ms (at 48 GB/s dual NVLink speed), and, respectively 20 ms for G_6 (at 12 GB/s PCIe link speed). Thus, t_{ckpt} = 20 ms.

B. Min-Time Flow-based Schedule

The greedy algorithm fails to find the optimal schedule because every transfer decision is irreversible and limits future opportunities for other senders to exploit fast links to their receiver peers. One way to address this limitation is to model the problem as a min-cost max-flow transportation network [24]. In the simplest form, min-cost max-flow aims to find the minimum cost of moving F items (flow) through a graph of N vertices (if possible), starting from a source node to a sink node. In this graph, each directed edge represents a maximum number of items that can be transported between the vertices and the cost of each item. In our case, we can create a virtual source and a virtual sink, then link the senders to the source, the receivers with the sink and finally the senders and receivers with themselves. Specifically, the edges from the source to the receivers have capacity C_i and cost 0, which is needed to limit the amount of data exiting the sender to C_i. Then, the edges from sender i to receiver j have capacity S_j and cost $1/B_{ij}$ (meaning sender i is allowed to transfer up to S_j and each unit of data takes $1/B_{ij}$ time to transfer). Finally, the edges from the receivers to the sink have capacity S_j and cost 0 (to limit the amount of data exiting receiver j from all its senders).

Figure 2 depicts the transportation network corresponding to the example shown in Figure 1. A classic algorithm to solve this problem is to iteratively increase the number of transported items (flow) by K, where K is the minimum capacity among

Algorithm 2: Min-Time Flow-based Schedule

Input : N (number of nodes), $Capacity$ (capacities of edges), $Cost$ (cost of edges)
Output: total blocking time t_{ckpt}, schedule P

while *true* **do**
  $d_i \leftarrow \infty,\ i = 0 \ldots N-1$ // min distance to i
  $p_i \leftarrow -1,\ i = 0 \ldots N-1$ // parent of i along path
  $Q \leftarrow Q \cup \{source\}$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow \text{head}(Q)$
    $Q \leftarrow Q \setminus \{u\}$
    for $v \in B_u | B_{uv} > 0$ **do**
      if $Cost_{uv} > 0$ **then** $f \leftarrow Capacity_{vu}$
      else $f \leftarrow Capacity_{uv}$
      $t \leftarrow max(d_u, abs(f \cdot Cost_{uv}) + Cost_{uv})$
      if $Capacity_{uv} > 0$ ***and*** $d_v > t$ **then**
        $d_v \leftarrow t$
        $p_v \leftarrow u$
        $Q \leftarrow Q \cup \{v\}$
  if $d_{sink} = \infty$ **then**
    break
  $curr \leftarrow sink$
  while $cur \neq source$ **do**
    $pred \leftarrow p_{cur}$
    $Capacity_{pred,curr} \leftarrow Capacity_{pred,curr} - 1$
    $Capacity_{curr,pred} \leftarrow Capacity_{curr,pred} + 1$
    $cur \leftarrow pred$
forall $(u, v) \in Edges | u \neq source\ and\ v \neq sink$ **do**
  $P_u \leftarrow P_u \cup \{(Capacity_{vu}, v)\}$
  $t_{ckpt} \leftarrow max(t_{ckpt}, Capacity_{vu} \cdot Cost_{uv})$
return t_{ckpt}, P

all edges on the minimum cost path from the source to the sink. Note that at each iteration, all edges on the minimum cost path will have their capacity reduced by K. By itself, this is not enough to reverse previous decisions if they proved to be suboptimal. Therefore, the graph is augmented such that for each edge, there is a back-edge whose capacity is increased by K at each iteration and whose cost is the negative cost of the opposite edge. Using this approach, previous decisions that proved suboptimal can be reversed by allowing back-edges in the minimum cost path. Note that it is not possible to use Djikstra's algorithm to calculate the minimum cost path, because of the negative cost of the back edges. However, algorithms, such as Bellman-Ford, can be used.

Unfortunately, using the unmodified classic min-cost max-flow algorithm to solve the optimal schedule problem is not possible, because the cost incurred by the concurrent transfers is the duration of the transfer over the slowest link, not the sum of the durations of the transfers over the links along the min-cost path. This has two important consequences: (1) we need to change how the cost of the minimum cost path is calculated by including both latency gain and loss due to back-edges; and (2) it may be suboptimal to increase the flow by the full capacity of the minium edge along the minimum-cost path as it may increase the overall blocking time. We solve both aspects by extending the classic min-cost max-flow algorithm as listed

in Algorithm 2, which we refer to as *min-time max-flow*. The algorithm needs the capacities and costs of the edges of the augmented graph, which are initialized as discussed above. To incorporate (2), our algorithm increases the flow by one unit in each iteration, which uses a modified form of Bellman-Ford. Therefore, the worst case complexity is $\mathcal{O}(F \cdot N \cdot |E|)$, where $F = \sum_{i=0}^{N-1} C_i$ and $|E|$ is the number of edges in the graph. Note that we used an optimized implementation of Bellman-Ford that prunes the number of attempted relaxations, which can often run in $\mathcal{O}(|E|)$ in practice.

Unlike the greedy algorithm, for the example in Figure 1, the flow based algorithm finds the optimal schedule as follows: G_0 transfers 137 MB to G_1, 275 MB to G_4, and the remaining 68 MB to the host memory. G_6 transfers 160 MB to G_4 and 80 MB to the host. Thus, the blocking time is 6.7 ms, which is 3× faster than the greedy algorithm.

V. Experimental Evaluation

A. Methodology

We base our evaluations on two checkpoint traces: one is obtained from a real run of a reverse time migration (RTM) application used in the oil industry, the other is generated synthetically. Each trace consists of a series of global checkpoints, which we refer to as *snapshots*. In turn, each snapshot records the checkpoint size of each process after applying compression to reduce its size. Additionally, we explore several fixed configurations of free space available on each GPU, which is used to calculate the remainder $C_i = Ckpt_i - F_i$ of each sender and the spare capacity $S_i = F_i - Ckpt_i$ of each receiver.

We compare our algorithms with a baseline approach that uses a standard strategy adopted by multi-level checkpointing approaches where each process i writes its checkpoint to G_i. Then, if a remainder C_i is left, it is written to the host memory. These comparisons are independently performed for each snapshot in order to study the tradeoff between the quality of the checkpoint schedule (blocking time t_{ckpt}) returned by each algorithm and the required runtime to obtain it.

We implemented our proposed algorithms and the baseline in a simulation framework written in Python, which is responsible to parse the traces, calculate the remainder C_i and the spare capacity S_i used by the algorithms, generate the flow graph corresponding to snapshots in the case of the min-time max-flow algorithm, and finally run the algorithms to obtain the blocking time t_{ckpt} and runtimes. The simulations are performed on a Dell PowerEdge C6320 server on the Cloudlab testbed [26], which is equipped with an Intel Xeon CPU E5-2683 v3 CPU and 256 GB of memory, and runs Python 3.8.10 over Ubuntu 20.04.2 distribution.

B. Checkpoint Traces

We start by analyzing the distribution of checkpoint sizes for all snapshots across the runtime of a real-life application. We then use the distribution of checkpoint sizes obtained from the real-life workload to generate synthetic checkpoint traces.

RTM: Adjoint-state methods are commonly used in the oil and gas industry to generate subsurface images from seismic data [27]. Reverse time migration (RTM) [28] is an example of such an application. First, forward and backward propagated seismic wavefields are calculated by solving the three-dimensional direct and adjoint wave equations in a known propagation model. Next, the two wavefields are cross-correlated in time to form the subsurface image. The correlation step requires combining the two wavefields at identical propagation times, and hence one of those wavefields needs to be reversed in time using checkpointing techniques [29], [30]. RTM usually relies on time-domain explicit finite difference (TDFD) solvers of the wave equation in its acoustic or elastic approximation. The wave equation derivatives are commonly approximated with finite-difference stencils of order 2 to 4 in time derivatives and up to 16 in space. The solver generates a snapshot of the wavefield at every step that needs to be stored for time reversal. The total wavefield commonly occupies several terabytes for production-size applications. Compression techniques are thus critical to reducing the overall size and speeding up the transfer from the computation unit to the storage layer. We run the RTM application on Nvidia's DGX-1 platform consisting of 8 Tesla V100 GPUs, each containing 32 GB of HBM2 memory and interconnected through a hybrid-mesh cube topology. The wavefield reversed in time consists of 776 snapshots with an aggregated compressed checkpoint size across all snapshots of ∼53 GB per GPU. We study the variability of checkpoint sizes across different snapshots and show the minimum, maximum, and average checkpoint size across all GPUs for every snapshot in Figure 3. The application starts with smaller average checkpoint sizes, which increase almost linearly up to snapshot 400 and then follow an arbitrary distribution. Figure 4 shows the variability of checkpoint sizes across all GPUs for 5 representative snapshots. Depending upon the compression ratio achieved on each GPU, we observe up to 107 MB (26×) difference in the smallest and largest checkpoint size for a given snapshot.

Synthetic: We perform a statistical analysis of checkpoint sizes of the RTM application to develop a model for generating synthetic checkpoint traces. We use the `rv_histogram` distribution of the `scipy.stats` Python package to obtain a template distribution from the binned data sample of the RTM application. Note that unlike the RTM trace, in this case each snapshot is allowed to capture the full range of checkpoint sizes observed during the RTM runtime. This is done to study a complementary behavior where the evolution of the checkpoint sizes from one snapshot to another is not smooth, and can change significantly. The distribution of the checkpoint sizes corresponding to this case is depicted in Figure 5.

C. GPU Configurations

We explore two dimensions of GPU configurations:

GPU Checkpoint Cache Size: We fix the free space available on the GPUs to 80, 128 and 160 MB, which roughly corresponds to the 50, 75 and 98 percentile of the average checkpoint size of the RTM application. Since the checkpoint sizes of snapshots are increasing (as seen in Figure 3), these configurations are representative of the average checkpoint

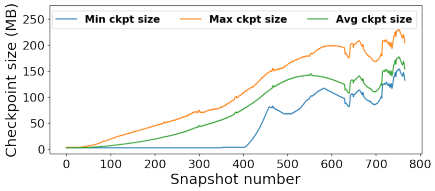


Fig. 3: Distribution of checkpoint sizes for different snapshots for RTM.

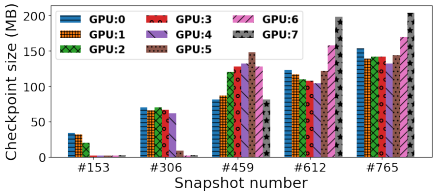


Fig. 4: Variability in checkpoint sizes of each GPU for 5 snapshots.

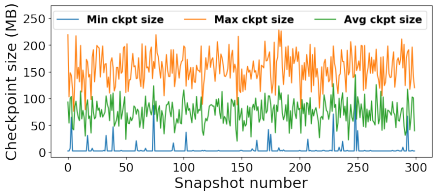
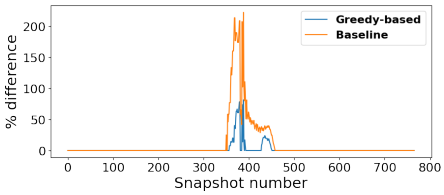
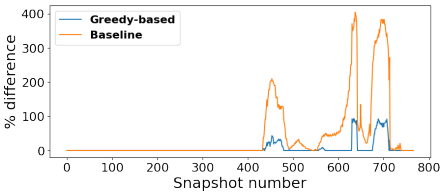


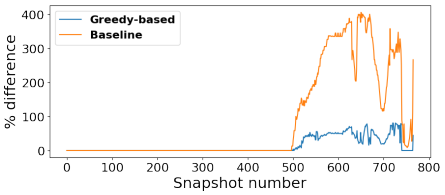
Fig. 5: Distribution of checkpoint sizes for different snapshots for synthetic traces.



(a) GPU free space = 80 MB



(b) GPU free space = 128 MB



(c) GPU free space = 160 MB

Fig. 6: RTM trace: checkpointing overhead (%) for the blocking phase of collaborative multi-level checkpointing based on the greedy and baseline approaches relative to the min-time, max flow approach.

sizes at various runtime stages, which corresponds to the free space that an application can afford for checkpoint caching.

Increasing Number of GPUs: In addition to the DGX-1 configuration with 8 GPUs, shown in Figure 1, we also study the behavior of all three approaches for GPU dense systems by increasing the number of GPUs. Although currently, the maximum number of interconnected GPUs in a single server is only 16 [31], multi-instance GPUs (MIG) and an increasing adoption of GPU dense systems and fast interconnects, e.g., NVLink and NVSwitch, advocate for a scalability evaluation.

D. Results

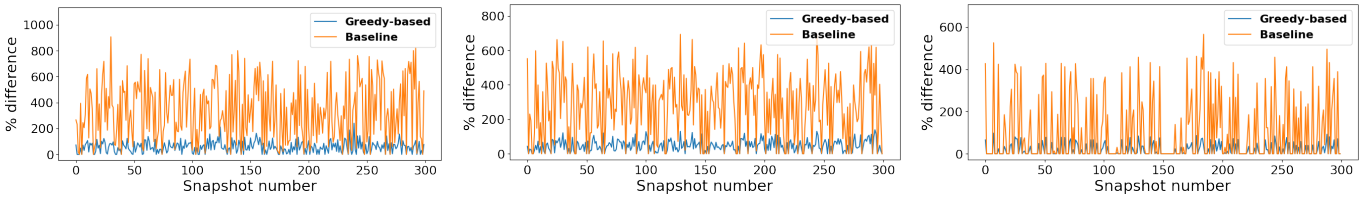
Checkpoint Schedule: For every snapshot of the RTM and synthetic traces, we obtain the blocking time t_{ckpt} returned by each of the three approaches (greedy, min-time max-flow, and baseline) using our Python-based simulator. We assume a granularity of 1 MB for the chunk sizes of the checkpoints, which means the minimum transferable data from a sender to a receiver is 1 MB. To better emphasize the differences between the three approaches, we use the optimal blocking time obtained by min-time, max-flow approach as a reference (ranging from less than 1 ms to 12 ms), and plot the relative percentage increase of the other two approaches in Figure 6 (RTM trace) and Figure 7 (synthetic trace).

Correlating Figure 6a (RTM trace) with Figure 3, for a GPU cache size of 80 MB, we observe a high overhead for the greedy (up to 50%) and the baseline (up to 200%) approaches in the snapshot range 350-450. Outside this range, all the three approaches produce comparable schedules. Specifically, for the snapshots higher than 450, the minimum checkpoint size grows beyond the free capacity, therefore there are only senders and no receivers, which means every process needs to checkpoint their remainder to the host memory. For the snapshots smaller than 350, the opposite is true: the maximum checkpoint size is smaller than 80 MB, therefore there are only receivers and no senders, which means every process can checkpoint directly to their GPUs. With increasing free space available on the GPU checkpoint caches (Figures 6b and 6c), performance differences begin to emerge between the three compared approaches for the higher snapshot numbers. This is expected as the average checkpoint size is increasing with the snapshot number. In these scenarios, greedy and baseline approaches are up to 2× and 5× slower, respectively, as compared to the optimal approach. Based on these results, we can draw two conclusions: (1) a sub-optimal schedule can dramatically lower the performance of multi-level checkpointing, especially when the free space on the GPU checkpoint caches are close to the average (compressed) checkpoint size; and (2) it is non-trivial to choose an optimal fixed GPU cache size, because the average checkpoint size varies during runtime.

For the synthetic trace, due to the high variance in the checkpoint sizes for every snapshot (Figure 5), we observe high overheads, i.e., greedy and baseline approaches experience 200% and 800% higher blocking time, respectively, relative to the min-time, max-flow approach. Since the average checkpoint size does not vary significantly during runtime, we observe a consistent speed-up for the min-time, max flow approach across all snapshots.

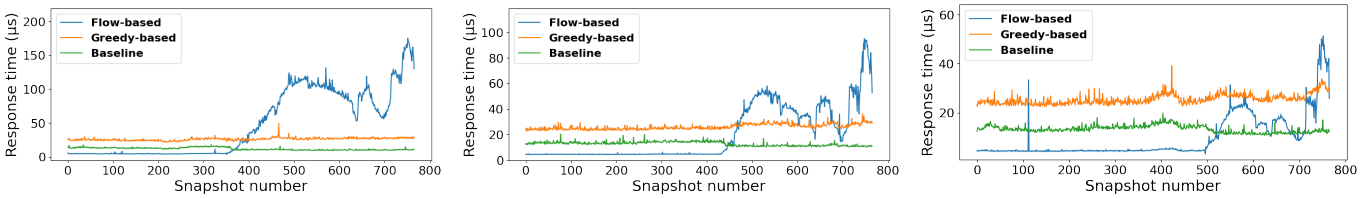
Execution Time: Next, we focus on the runtime overhead of the three approaches necessary to obtain the checkpoint schedule. Figures 8 and 9 depict the execution time of the three approaches for the real-world and synthetic traces respectively.

Due to the limited number of GPUs and therefore graph edges, the min-time, max flow approach runs in nearly same amount of time as the other two approaches despite a higher computational complexity. For the synthetic trace, in Figure 9, we observe that for larger GPU cache sizes, the flow-approach actually runs faster than greedy and the baseline. This is because for increasingly larger GPU cache sizes, the remainder on the GPUs becomes smaller, thereby reducing the total flow. Overall, we observe that the execution time is negligible for all three approaches, ranging between 10 to 170 µs. Thus,



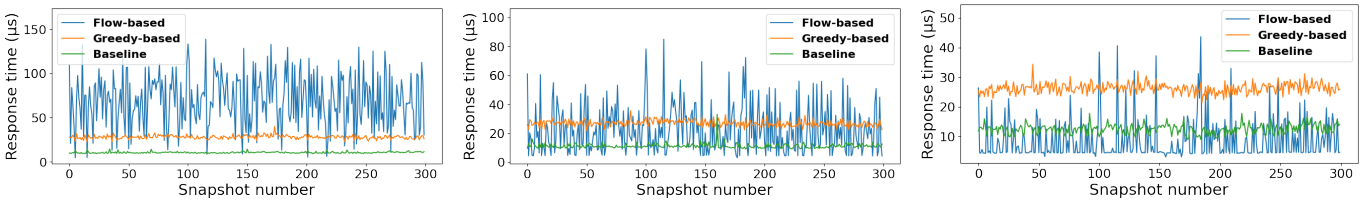
(a) GPU free space = 80 MB **(b)** GPU free space = 128 MB **(c)** GPU free space = 160 MB

Fig. 7: Synthetic trace: checkpointing overhead (%) for the blocking phase of collaborative multi-level checkpointing based on the greedy and baseline approaches relative to the min-time, max flow approach.



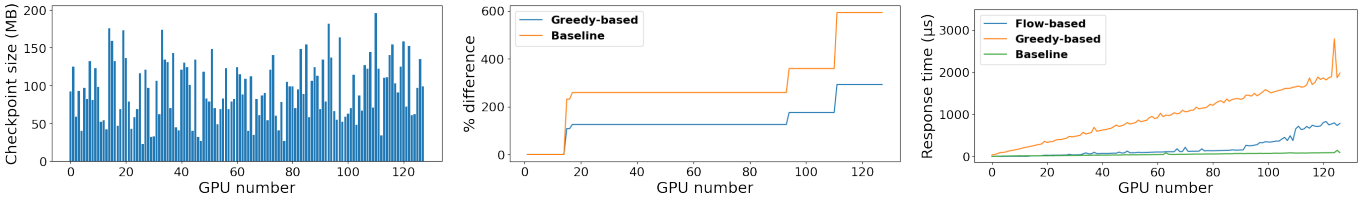
(a) GPU free space = 80 MB **(b)** GPU free space = 128 MB **(c)** GPU free space = 160 MB

Fig. 8: RTM trace: execution times of the baseline, greedy and min-time, max flow approaches.



(a) GPU free space = 80 MB **(b)** GPU free space = 128 MB **(c)** GPU free space = 160 MB

Fig. 9: Synthetic trace: execution times of the baseline, greedy and min-time, max flow approaches.



(a) Checkpoint size on each GPU **(b)** Checkpoint overheads of Greedy and Local-only approaches as compared to flow-based approach **(c)** Response times for Local-only, Greedy and Flow-based approach

Fig. 10: Checkpoint sizes, overheads and response times for different number of GPUs.

the min-time, max flow algorithm has no significant runtime penalty compared with the other two approaches and is an all-around winner given that it always produces an equal or better checkpoint schedule than the other two approaches.

Scalability Study: We evaluate the scalability of the three approaches for an increasing number of GPUs, ranging from 16 (DGX-2) up to 128 GPUs. To this end, we generate a synthetic snapshot with different checkpoint sizes per GPU, whose distribution is depicted in Figure 10a. We fix the free size of each GPU checkpoint cache at 160 MB. The GPUs are assumed to be connected in an all-to-all pattern using an NVSwitch, similar to the DGX-2 topology.

As the number of GPUs is increasing, we observe in Figure 10b, a significant increase in the blocking time for the greedy and baseline approaches relative to the flow-based approach. This is because the likelihood to obtain non-trivial optimal schedules increases for an increasing number of senders and receivers. Figure 10c illustrates the increase in execution times for all three approaches as a function of the number of GPUs. Again, despite the higher complexity, the flow-based approach outperforms the greedy approach and stays below 1 ms.

Overall, we conclude that with increasing scale, the optimal schedule found by the flow-based approach has an increasingly larger impact, while the execution time remains negligible.

VI. Conclusions

In this paper we studied the problem of efficient collaborative multi-level checkpointing, focusing on two algorithms that leverage the spare capacity of the fast local storage of remote peers to minimize the blocking phase needed to cache

all checkpoints such that they can be flushed asynchronously in the background to shared storage.

Based on simulations with two traces that compare our algorithms with a baseline that does not leverage remote local storage, we make the following observations: (1) the effectiveness of remote transfers depends on the imbalance between the checkpoint sizes (which often occurs due to compression) and the spare capacities (which can vary during runtime); (2) our min-time, max-flow algorithm finds the optimal schedule and significantly reduces the blocking time over the baseline and the greedy algorithm; and (3) the execution time of our algorithms is negligible even at scale.

Encouraged by these promising results, in future work we plan to implement the proposed algorithms in production-ready multi-level checkpointing systems such as VELOC [2]. Another interesting direction to consider is the case when the GPU cache accumulates the checkpoints of successive snapshots. In this case, the free space on each GPU is not fixed and can be dynamically adjusted, e.g. by dropping older checkpoints from the cache, which opens additional optimization opportunities.

ACKNOWLEDGMENTS

This work is supported in part by the ARAMCO Services Company and the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research and Argonne National Laboratory, under contract numbers PRJ1008127, Argonne: 0F-60169/DOE: DE-AC02-06CH11357. Results presented in this paper are obtained using the Chameleon [32] and CloudLab [26] testbeds supported by the National Science Foundation.

REFERENCES

[1] NVIDIA, "White Paper: DGX-1 With Tesla V100 System Architecture." [Online]. Available: https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf

[2] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[3] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous i/o interference in hpc applications," *The International Journal of High Performance Computing Applications*, vol. 35, pp. 391–412, 2021.

[4] A. Maurya, B. Nicolae, I. Guliani, and M. M. Rafique, "Cosim: A simulator for co-scheduling of batch and on-demand jobs in hpc datacenters," in *Proc. IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2020.

[5] F. Liu, K. Keahey, P. Riteau, and J. Weissman, "Dynamically negotiating capacity between on-demand and batch clusters," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[6] M. Rodríguez-Pascual, J. Cao, J. A. Moríñigo, G. Cooperman, and R. Mayo-García, "Job migration in hpc clusters by means of checkpoint/restart," *The Journal of Supercomputing*, vol. 75, no. 10, pp. 6517–6541, 2019.

[7] B. Nicolae and F. Cappello, "Blobcr: Virtual disk based checkpoint-restart for hpc applications on iaas clouds," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 5, pp. 698–711, 2013.

[8] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[9] B. Nicolae, "On the Benefits of Transparent Compression for Cost-Effective Cloud Data Storage," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 3, pp. 167–184, 2011.

[10] NVIDIA, "nvCOMP: A CUDA library that features generic compression interfaces." [Online]. Available: https://github.com/NVIDIA/nvcomp

[11] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[12] ——, "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[13] B. Nicolae and F. Cappello, "AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing," in *Proc. ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2013.

[14] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan *et al.*, "Transparent accelerator migration in a virtualized gpu environment," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2012.

[15] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, 2011.

[16] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," in *Proc. International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2009, pp. 408–413.

[17] T. Suzuki, A. Nukada, and S. Matsuoka, "Transparent checkpoint and restart technology for cuda applications," in *Proc. GPU Technology Conference (GTC)*, 2016.

[18] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-restart support for CUDA's unified memory," in *Proc. IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[19] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint restart support for heterogeneous hpc applications," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[20] M. Siavvas and E. Gelenbe, "Optimum checkpoints for programs with loops," *Simulation Modelling Practice and Theory*, 2019.

[21] T. Alturkestani, T. Tonellot, H. Ltaief, R. Abdelkhalak, V. Etienne, and D. Keyes, "MLBS: Transparent Data Caching in Hierarchical Storage for Out-of-Core HPC Applications," in *Proc. IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019.

[22] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained DNN checkpointing," in *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[23] J. B. Orlin, "Max flows in o(nm) time, or better," in *Proc. ACM Symposium on Theory of Computing (STOC)*, 2013.

[24] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by successive approximation," *Mathematics of Operations Research*, vol. 15, no. 3, pp. 430–466, 1990.

[25] E. Oki, *Linear Programming and Algorithms for Communication Networks: A Practical Guide to Network Design, Control, and Management*. CRC Press, Inc., 2012.

[26] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig *et al.*, "The design and operation of CloudLab," in *Proc. USENIX Annual Technical Conference (ATC)*, 2019.

[27] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[28] H.-W. Zhou, H. Hu, Z. Zou, Y. Wo, and O. Youn, "Reverse time migration: A prospect of seismic imaging methodology," *Earth-Science Reviews*, vol. 179, pp. 207–227, 2018.

[29] W. Symes, "Reverse time migration with optimal checkpointing," *Geophysics*, vol. 72, 2007.

[30] J. E. Anderson, L. Tan, and D. Wang, "Time-reversal checkpointing methods for RTM and FWI," *GEOPHYSICS*, vol. 77, no. 4, pp. S93–S103, 2012.

[31] NVIDIA, "DGX 2/2H System." [Online]. Available: https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf

[32] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione *et al.*, "Lessons learned from the chameleon testbed," in *Proc. USENIX Annual Technical Conference (ATC)*, July 2020.