2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-6654-7315-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS56603.2022.00067

iCheck: Leveraging RDMA and Malleability for Application-Level Checkpointing in HPC Systems

Jophin John*, Isaac David Núñez Araya†, Michael Gerndt‡

Chair of Computer Architecture and Parallel Systems
Technische Universität München
Germany

Email: john@in.tum.de, isaac.nunez@tum.de, gerndt@in.tum.de

***Abstract*—The estimate that the mean time between failures will be in minutes in exascale supercomputers should be alarming for application developers. The inherent system's complexity, millions of components, and susceptibility to failures make checkpointing more relevant than ever. Since most high performance scientific applications contain an in-house checkpoint restart mechanism, their performance can be impacted by the contention of parallel file system resources. A shift in checkpointing strategies is needed to thwart this behavior. With iCheck, we present a novel checkpointing framework that supports malleable multilevel application-level checkpointing. We employ an RDMA enabled configurable multi-agent-based checkpoint transfer mechanism where minimal application resources are utilized for checkpointing. The high-level API of iCheck facilitates easy integration and malleability. We have added the iCheck library into the ls1 mardyn application providing performance improvement up to five thousand times over the in-house checkpointing mechanism. LULESH, Jacobi 2D heat simulation, and a synthetic application were also used for extensive analysis.**

***Index Terms*—Fault Tolerance, Adaptive Checkpointing, RDMA, Malleable Checkpointing, MPI**

I. Introduction

As more and more computational power is stacked to create exascale performance, resilience to failure will remain crucial in HPC application development [1]. The massive improvement in performance from PetaScale to Exascale will further amplify and propagate the failure rate [2]. The mean time between failures (MTBF), which now averages around hours in the petascale system, is predicted to be in minutes [3], and application developers should anticipate more frequent failures and provide a recovery mechanism.

We look at this issue from a software engineering perspective, particularly the techniques employed to overcome such failures and continue the application execution gracefully. Since most HPC applications follow the Single Program Multiple Data programming models using the Message Passing Interface (MPI), our primary focus will be to provide fault tolerance to MPI applications, the most commonly used distributed memory programming model in HPC systems.

MPI [4] is inherently non fault tolerant in design, and a single process failure will cascade into a complete job failure. Although there is research [5]–[8] in creating a fault tolerant MPI, it is not yet standardized. Conversely, a redesign of typical HPC applications (for example, multiphysics and multiscale codes) is essential to integrate such fault-tolerant MPI implementations. As a result, they continue to use one of the most popular techniques to handle fail-stop errors, namely `Checkpointing` [9]. It follows a rollback recovery mechanism, where the storage of the current state of an application occurs periodically, and upon failure, the application will restart from the most recently saved state.

The frequently used approaches to implement such a system are application-level checkpointing and system-level checkpointing [10]. In the former, the user determines application-specific data that needs to be saved and restored in case of failures, while in the latter, the whole checkpointing process is transparent to the user. After a failure, the restart of the application happens by restoring the checkpointed data and maintaining job progress. Our work here focuses on application-level checkpointing since it leads to smaller checkpoints, less overhead and faster restart than system-level checkpointing.

One of the core issues in the above approaches is the contention of the parallel file system. Multiple applications attempting simultaneously to write the checkpoints to the file system can induce potential resources contention and leads to the performance degradation of the application [11]. A software system that runs on compute nodes designed to read the checkpoints using remote memory accesses from the application instead of writing into the Parallel File System (PFS) will avoid performance degradation [11]. So, we propose iCheck, a software system running on dedicated compute nodes in an HPC system providing asynchronous multilevel checkpointing services to MPI and non-MPI applications using reconfigurable remote direct memory access (RDMA). With iCheck, we introduce a novel checkpointing system that considers metrics like system-level power usage, available memory and bandwidth, and checkpoint frequency to make resource and data management decisions. iCheck brings novelty by exploiting resource malleability for checkpointing. This work succinctly posits the checkpointing systems as a holistic resource management framework rather than a pure data management framework.

Using malleability, iCheck brings the following two ben-

978-1-6654-7315-6/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS56603.2022.00067

efits to checkpointing in HPC. Firstly, iCheck can scale its checkpointing resources horizontally. By having the horizontal scaling capability (in subsection V-B7), there is no need to specify a dedicated set of nodes for checkpointing. The number and types of nodes allocated to checkpointing services can be varied on-the-fly by a resource manager. Secondly, iCheck can dynamically adjust the checkpointing processes allocated to each application. In subsection V-B5 we demonstrates that dynamically adding checkpointing resources (agents) improves the checkpointing time, and placing it across different nodes can significantly improve the checkpointing bandwidth (subsection V-B6). Additionally, the resource malleability in iCheck can also cater to the necessities of growing trends in malleability in MPI [12]–[14]. For example, a resource increase in malleable MPI applications can trigger a corresponding increase in checkpointing resources.

The key contributions made towards application-level checkpointing in this work are:

- RDMA based malleable Checkpoint/Restart library `iCheck` that supports standard MPI, malleable MPI and non-MPI applications.
- Checkpointing system that scales its resources on-the-fly.
- Multiple checkpoint orchestration strategies.

The outline of the paper is as follows. Section II discusses the related work in this domain. Subsequently, Section III presents the architecture and design of our iCheck library, followed by Section IV which in detail explores nuances in the iCheck system. Section V evaluates the impact of the library in production-level and synthetic applications. The last section (Section VI) summarizes and concludes the paper.

II. Related Work

In a simplistic view, a checkpointing system can be classified as an application-level checkpointing or a system-level checkpointing system. But based on the hierarchies of storage technologies used, it can further be designated as a multilevel or single-level checkpointing system. These systems can be further broken down into blocking and non-blocking checkpointing systems. Furthermore, it can be categorized into process recovery or data recovery, or both. Based on different use cases and programming models, there are even more categories (For example, fault tolerant MPI implementations). There are a plethora of checkpointing works that span across these complex categories [5]–[8], [11], [15]–[24].

Here, we only focus on the works relevant to our checkpointing system - a malleable asynchronous multilevel application-level based in-memory checkpointing system using RDMA. The closest work to our system is by Sato et al. [11]. It is also an RDMA based non-blocking multilevel checkpointing system. It has staging nodes (remote nodes) where a staging server retrieves the checkpoint from the staging client running in the compute nodes of the application. iCheck differs from this work in the following aspects. Firstly, iCheck is a malleable system. It can increase or decrease the iCheck nodes (equivalent to staging nodes) dynamically. It can also dynamically change the agents (equivalent to staging servers) as per the requirement. Secondly, a single iCheck node in iCheck can simultaneously cater to the needs of multiple applications while it is not demonstrated in [11]. Thirdly, iCheck doesn't use a separate staging client. Before triggering the initial data transfer, the application registers its memory regions with the agents in the remote node to perform RDMA. So when a checkpoint is ready, an application can inform the agents by a simple library call and avoids using an extra thread in compute node just for staging.

Another related work, VeloC [21] is an adaptive asynchronous checkpointing system. It is adaptive because it dynamically selects where to store the local checkpoints by exploiting the underlying heterogeneous storage solutions and avoiding local I/O bottlenecks. In iCheck, malleability refers to the dynamism of resources of our system. VeloC uses IO threads inside nodes to flush the data into the underlying storage system, while iCheck employs RDMA based checkpointing.

FENIX [23] and CRAFT [22] offer application-level checkpointing but differ from iCheck in the sense that FENIX checkpoints on neighboring nodes while CRAFT focuses on writing checkpoints into the parallel file system. iCheck system itself is adaptive and performs RDMA based in-memory checkpointing with reconfigurable threads (agents) in remote nodes, which is not covered by these related works. Additionally, these agents perform the second level checkpointing from the remote nodes (not from the compute nodes in which the application is running), thereby reducing the overhead associated with file system transfers.

III. System Design

iCheck is a software system dedicated to providing checkpointing services to applications. Its design objective focuses on providing resources (memory, endpoints) for checkpointing based on the demand from the applications. iCheck has a core system and an iCheck library (See Figure 1). iCheck runs on dynamically configurable number of dedicated nodes (iCheck nodes) of the HPC system and communicates with the application on the machine via the high-performance network. Using the provided library, the application contacts the iCheck for checkpointing. Below subsections explore in detail the different components and functionalities in iCheck.

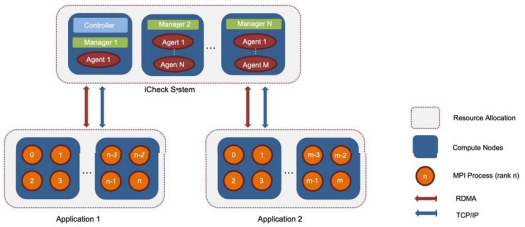


Fig. 1: Components in iCheck system

A. iCheck Core

The core system consists of three components: Controller, Manager, and Agent.

1) Controller

The controller performs multiple roles in iCheck. The primary role is to perform the iCheck node selection and agent scheduling tasks. The monitoring framework in the controller analyzes different metrics (For example, bandwidth, available memory, checkpoint frequency) and aids in deciding the number of agents along with iCheck nodes in which those agents will be launched. The secondary role is to interact with a malleable resource manager. iCheck can be configured as a standalone system or be tightly coupled with a resource manager. If configured and integrated with a malleable resource manager, the controller can request additional nodes or relinquish the existing nodes as per the system state.

2) Manager

It manages the node-level activities of the software, such as launching the agents, monitoring and predicting the node usage parameters (e.g., memory usage, bandwidth usage) in iCheck nodes, and interacting with the controller. The manager frequently updates the controller with the node status information from the monitoring infrastructure.

3) Agent

An agent is the smallest module in the architecture. The agent performs the core functionality of checkpoint read/write. The communication between the agents and an application is performed through RDMA operations and TCP/IP. The agent implements both the synchronous (blocking) and asynchronous (non-blocking) checkpoint operations.

B. iCheck library

iCheck provides a minimalistic API for application-level coordinated checkpointing. A minimum of six iCheck API calls is needed for successfully writing and reading the checkpoint, of which two each are for configuration, checkpoint, and restart.

Pseudocode of a naive iCheck enabled application is shown in the Listing 1. As the first step, `icheck.h` header file is added to the application. The `icheck_init()` function call (in line 5) registers the application with the iCheck controller and passes the hints about the application (if available). `icheck_init` requires the `MPI_COMM_WORLD` communicator, application name as well as status. For a non-MPI application, `NULL` can be used in place of the communicator. Then the application receives the agent information from the controller and registers itself with the agent for future RDMA operations.

`icheck_add` (in line 7) registers the array *data* to the iCheck library, its size, and assigns a label for this data. Multiple data types can be added using this routine. In line 17, the execution of an `icheck_commit()` routine informs the library to copy all the data added using `icheck_add` into a buffer registered for remote access and informs the agents about the checkpoint availability. Agents then retrieve the checkpoint data remotely. The dynamism in iCheck can be triggered by the `icheck_probe_agents()` call (in line 20). Upon calling the probe function, the application receives information about the agent change. `icheck_finalize()` (in line 22) informs the controller about the application finish. The argument in the finalize calls informs the iCheck system whether to retain the checkpoint in the agent memory or not. `IC_PERSIST` informs the system that the checkpoint must be kept in the agent memory. `icheck_restart` (in line 10) will transfer the checkpoint from the agent to the library. The `icheck_restore()` (in line 11) will transfer the data associated with the label to the data structure reference passed in the call.

```
1  #include<icheck.h>
2  int main() {
3      MPI_Init(NULL, NULL);
4      float data[SIZE];
5      icheck_init(argv[1], "sample",
6                  MPI_COMM_WORLD, status);
7      icheck_add("Array", data,
8                      SIZE, ICHECK_FLOAT);
9      if(checkpoint_available) {
10       icheck_restart();
11       icheck_restore("Array",data,SIZE);
12     }
13     /*computation*/
14     for(i = 0; i<N; i++) {
15       /*Read/Write data[]*/
16       if(i%100)
17         icheck_commit();
18       /*computation*/
19       if(i%1000)
20         icheck_probe_agents();
21     }
22     icheck_finalize(IC_PERSIST);
23     MPI_Finalize();
24 }
```

Listing 1: Pseudocode of a sample MPI application using iCheck.

C. Using RDMA in iCheck

We employ the libfabric library [25] to deliver RDMA support into iCheck. RDMA allows a remote process to perform the data access of preregistered memory regions by avoiding CPU interference. This significantly improves the throughput and reduces the roundtrip latency. OpenFabrics Interfaces (OFI) provide the program interfaces to write high performance and scalable remote memory access operations. Libfabric, being a core component of OFI, allows access to the user-space API of OFI services.

There are two ways to perform a Checkpoint and Restart operation in iCheck, and it is based on the type of RDMA operations provided by libfabric, namely read and write. iCheck uses a combination of these to perform checkpoint storage and retrieval. Push and pull are the two techniques iCheck supports to transfer the data during checkpoint/restart. In the former, the application writes/reads the checkpoint to/from the agent's memory, while in the latter, the agent reads/writes the checkpoint from/to the application memory. Additionally, agents can also use multithreading to parallelise the transfers.

iCheck, with its agent-based checkpoint retrieval model, easily integrates the asynchronous checkpoint retrieval capability in its library. Since the agents use RDMA, the application

does not need to use its computing resource for data transfer rather it can continue the execution immediately after notifying the agents about the checkpoints. The agents can remotely retrieve the data without affecting the application performance.

D. Multilevel checkpointing in iCheck

iCheck stores the checkpoint in the agents memory in the iCheck nodes as well as in PFS. PFS acts as the second level in our multilayered checkpointing approach. iCheck performs checkpoint data compression and writes the data into PFS during its idle time. This data can be used in case of an agent failure or checkpoint corruption inside the agent. Furthermore, this backed-up data in the file system can be uploaded into the new iCheck nodes and existing agents can then be migrated into the new manager.

IV. Configuring the iCheck System

The performance of the iCheck depends heavily on how it is configured. In the iCheck ecosystem, the agents interact with the application processes and store the checkpoints. So, the ideal agent placements across the iCheck nodes are crucial and opens a window for ample optimization opportunities. Three control knobs to fine-tune and optimize iCheck are the number of agents, the iCheck nodes in which they are launched, and how they are distributed. The monitoring framework in iCheck collects the data concerning memory usage and power, checkpoint operations, data transfer, and the available cores to decide about the number of iCheck nodes and agents.

A. Agent count

The controller decides how many agents to launch when an application registers. This selection can be based on the application information like the number of processes, the number of nodes, application hints, and iCheck system metrics like the number of agents and the bandwidth usage across current iCheck nodes. This initial selection of agents can significantly improve the performance of checkpointing in the application. We can see from Figure 2c that a larger initial number of agents has better impact on the application. Alternatively, the controller can also begin allocation with a minimum number of agents and increase the agent count as the application execution progresses (See Figure 4a).

B. Agent placement

After the agent count selection, the immediate task by the controller is to find suitable iCheck nodes to launch these agents. iCheck facilitates following different strategies to select the nodes.

1) Bandwidth: Select the iCheck nodes that transfers low amount of data (or has higher bandwidth available for checkpoint transfers). It improves the performance of frequently checkpointing applications.
2) Available Memory: Pick the iCheck nodes with the maximum available memory. It is relevant for an application that checkpoints a large amount of data.
3) Checkpoint Frequency: Select the iCheck nodes that has the least activity. It is relevant for an application that checkpoints a large amount of data.
4) Power Budget: Pick the iCheck nodes with the least power usage. It is relevant if iCheck needs to be within a specific power budget.

After finding the suitable candidates to launch the agents, the controller should next decide on the distribution of agents across these iCheck nodes. iCheck facilitates the following three different strategies to do so.

1) Block Distribution: The agents are distributed in block of specified size across the iCheck nodes.
2) Round Robin: Iterate over available iCheck nodes and allocate one agent per iCheck nodes until all agents are distributed.
3) Fill to completion: Allocate the maximum possible number of agents to each iCheck nodes such that fewer numbers of iCheck nodes are selected.

The impact of different strategies on checkpointing is observable in Section V.

C. Controller plugins for dynamism

At the beginning of the application, the controller might not have enough information to provide the optimal agent placement strategies or be constrained by the resources. By analyzing the performance, the controller can change the agent distribution during and after the application execution to improve the overall checkpointing performance. Two strategies employed by the controller to improve the checkpointing behavior are the following:

1) Horizontal scaling

After an application execution, the controller can migrate the agents from one iCheck node to another iCheck node. This is useful if the initial agent distribution was constrained due to the lack of resources. For example, there were not enough free memory to accommodate all the agents of an application in the same iCheck node. In such a scenario, the controller can request for new iCheck nodes and migrate the agents to the new nodes once it becomes available. This strategy requires a malleable resource manager.

2) Dynamic agents

After the initial assignment of agents, the controller can dynamically increase/decrease the number of agents. The agent change information is passed to an application during `icheck_probe_agents` function call. There are four possible outcomes to a probe call. It can be related to change in agent numbers, i.e., agent expansion or reduction. Alternatively, it can be about maintaining the status quo or triggering an agent migration to new iCheck nodes.

Dynamic agents will only be useful in the following scenario. Suppose the total time taken in an application for dynamic agent reconfiguration is t_{probe}, the time taken to transfer checkpoint with old number of agents is $t_{old\_transfer}$, and the time taken for checkpoint transfer with new number of agents is $t_{new\_transfer}$, then the potential number of checkpoint operations (*ncops*) possible with the old number of agents during the new agent configuration period is

$$ncops = \frac{t_{probe} + t_{new\_transfer}}{t_{old\_transfer}} - 1 \quad (1)$$

The dynamic agents will only give performance improvement from the n^{th} checkpoint transfer operation, where n is defined as

$$n = \frac{ncops \times t_{old\_transfer}}{t_{new\_transfer}} + 1 \quad (2)$$

The dynamic agents will degrade the application performance if the number of remaining checkpoint operations is less than n.

V. Experimental evaluation

A. Evaluation Setup

1) System

The evaluation of iCheck is performed on the high performance system SuperMUC-NG [26] at Leibniz Supercomputing Centre in Germany. A compute node consists of 2 Intel Xeon 8174 processors, with 96 GB DDR4 Memory with Intel OmniPath Adapter and Gigabit Ethernet adapter. Intel OmniPath Interconnect network with fat-tree network topology provides a high bandwidth networking between compute nodes. Sixteen nodes (768 cores) were used for our evaluation, in which iCheck used up to four nodes, and applications were launched in the remaining nodes based on the evaluation objective.

2) Applications

The iCheck system is assessed with a highly scalable molecular dynamics application ls1 mardyn [27], hydrodynamics proxy application LULESH [28], Jacobi 2D heat simulation code, and a synthetic benchmark application that checkpoints the specified size in intervals and duration provided by the user. We launched ls1 mardyn application on one (48 processes) to twelve (576 processes) nodes for the analysis. For data transfer tests (Subsections V-B1,V-B2), we ran ls1 mardyn for 100 time steps with a checkpoint interval of ten time steps. These quick runs were also used to stress test the iCheck infrastructure. For overhead analysis (Subsection V-B3), we ran ls1 mardyn for 10000 time steps, in which iCheck and MPI-IO checkpoints every 1000 time steps. The simulation system was initialized with 65536 molecules and checkpointed eight megabytes during each checkpoint operation for all the tests. Each of the process stores information about molecules like id, position, momentum, velocity, etc. The synthetic application was also run on one to twelve nodes (48 to 576 processes) by writing a checkpoint of size up to 2.3 GB per checkpoint operation. LULESH was run on two nodes and the heat simulation application was run on two to twelve nodes to study the malleability of iCheck.

B. Results

1) iCheck vs MPI-IO in ls1 mardyn

In the following analysis, the blocking checkpointing strategy of iCheck is compared with the in-house MPI-IO based checkpointing provided by ls1 mardyn. Figure 2a (in log scale) and 2b depict the average time taken for a checkpoint transfer operation performed by iCheck and MPI-IO during the execution of ls1 mardyn. Figure 2a shows the time taken by iCheck with agents varying from one to twelve and MPI-IO during the application run on twelve nodes, while Figure 2b plots the iCheck and MPI-IO time while running the application on one, four, eight and twelve nodes with a single agent inside a single iCheck node.

TABLE I: Impact of checkpointing on ls1 mardyn execution

Time	MPI-IO	iCheck
1775.5s	104.383s	0.567538s

It can be observed from both the figures that the iCheck performs better than MPI-IO versions in all the configurations. A comparison with the worst-performing single iCheck agent with the MPI-IO in Figure 2a shows that iCheck is around 400 times faster in writing checkpoints than MPI-IO. The best iCheck configuration with 12 agents performs around 5000 times faster than MPI-IO. Similarly, the same trend is observable in Figure 2b, with iCheck taking .8 to 1 millisecond, while MPI-IO ranges from 100 to 700 milliseconds. Another observation from Figure 2b is that as the application's nodes increase, the checkpointing time by iCheck increases if there are no changes in the number of agents.

2) Blocking vs Non-Blocking checkpointing in ls1 mardyn

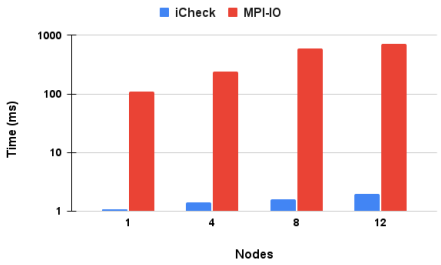
Here, we show the difference between blocking and non-blocking checkpoint transfers of iCheck in ls1 mardyn (see Figure 2c). It compares the performance delivered by different agent configurations. It can be deduced from Figure 2c that the non-blocking mode of iCheck yields better performance than the blocking mode in every agent configuration. The non-blocking agent combinations perform up to 50% faster than the blocking combinations. Furthermore, we can observe that the non-blocking version provides the combined performance of the blocking version plus the improvement gained by an additional agent. Another insight from the graph is that as the number of agents increases, the checkpoint transfer happens more quickly in blocking and non-blocking combinations.

3) iCheck overhead analysis in ls1 mardyn

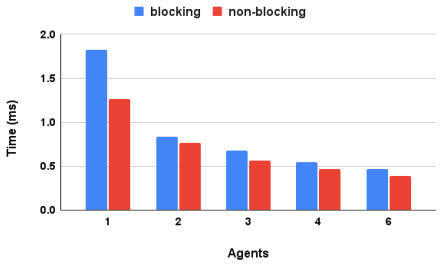
In this subsection, we show the overhead induced by iCheck on ls1 mardyn. The application was run for 10000 time steps with checkpointing performed every 1000 steps. The overhead induced by iCheck on ls1 mardyn can be analyzed from two aspects. The first aspect explores the overall impact of iCheck and MPI-IO on the application run (See Table I).We can infer that the total time spent on iCheck calls is only .03% of the total application time, while MPI-IO took around 5.6%. A significant portion of overhead comes from the initial setup of iCheck. The overall checkpoint performance of the iCheck library improves as the application progresses, which can be attributed to the iCheck commit operation that takes only a minuscule time to transfer checkpoints compared to MPI-IO. Furthermore, ls1 mardyn has six percentage faster execution with a single iCheck agent.

To analyze the impact of the iCheck initial setup, we ran ls1 mardyn on one, four, eight, and twelve nodes (See Figure 2d (in log scale)). A single agent, which delivers the poorest performance in iCheck (as seen in Figure 2a), is used for comparison with MPI-IO. We can observe that as the number of processes increases, configuration time in iCheck increases. A similar trend is visible for all operations plotted in the graph,

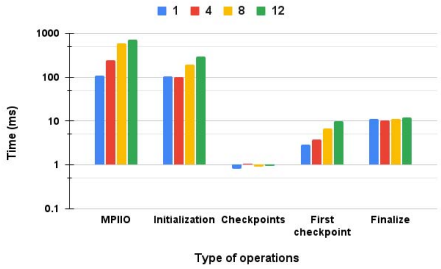
(a) Comparing iCheck and MPI-IO with different agents



(b) Comparing iCheck and MPI-IO with different nodes

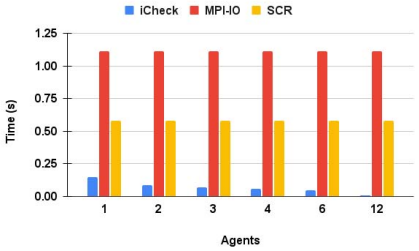


(c) Comparing blocking vs non-blocking strategies in iCheck

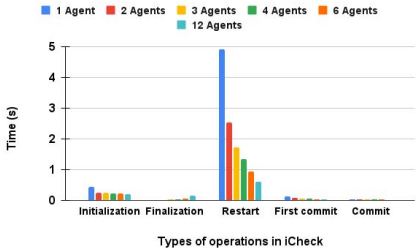


(d) Breakdown of iCheck induced overhead

Fig. 2: Checkpointing analysis on ls1 mardyn using iCheck and MPI-IO



(a) Comparing iCheck, MPI-IO and SCR on checkpointing large data size



(b) Overhead analysis of various iCheck operations

Fig. 3: Checkpointing analysis on synthetic application using iCheck and MPIIO

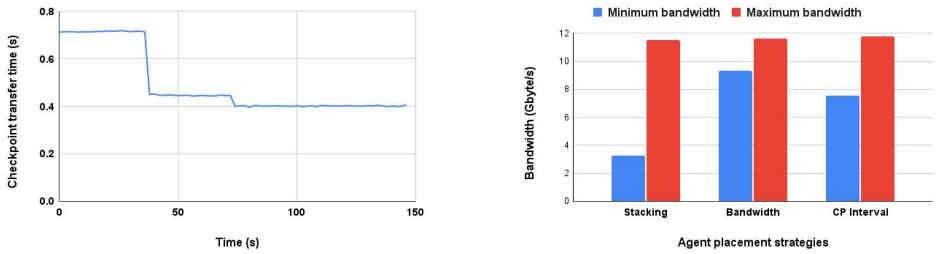
including time taken for MPI-IO. Furthermore, we can observe that the first checkpoint transfer needed additional time than subsequent checkpoint transfers This is for the libfabric memory region registration and shared memory configuration performed inside the library and agents. Nevertheless, the configuration time in iCheck can be reduced significantly by using more agents.

4) Comparing iCheck with SCR and MPI-IO using a synthetic application

In this subsection, we first compare the performance of iCheck against MPI-IO and state-of-the-art Scalable Checkpoint/Restart [24] using a synthetic application. The application was run on 12 nodes (576 processes) and checkpointed 2.3 GB of data. Further, we discuss the overhead induced by iCheck on large checkpoint transfers.

Figure 3a depicts the average time for checkpoint transfer in iCheck and MPI-IO. We can observe from the figure that iCheck performs 3.9 times faster than SCR and 7.6 times faster than MPI-IO while using a single agent. As we increase the number of agents, iCheck provides up to 57 and 100 times faster data transfer than SCR and MPI-IO, respectively.

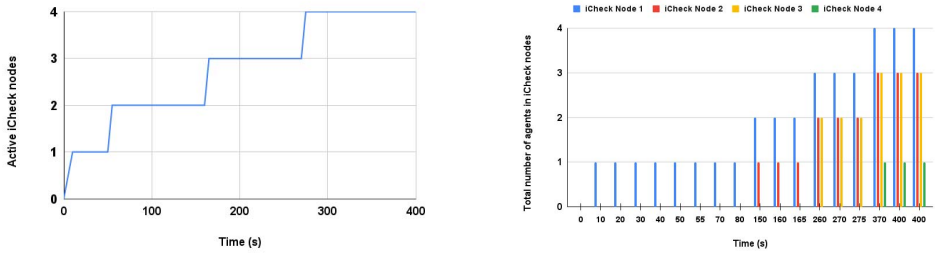
Figure 3b shows the iCheck initialization time, checkpoint data transfer time, finalization time and the time taken to restore the checkpointed data for up to twelve agents. We can observe in the figure that increase in agents can reduce the iCheck induced overhead on the application. The contrast in performance between restart and commit operations is noteworthy since both performs RDMA read and write. In commit, iCheck makes sure that the data is ready to be transferred by performing an intra-node synchronisation at the beginning meanwhile during restart, an inter-node synchronisation occurs which leads to a large checkpoint restart time. This synchronisation is essential to support automatic data redistribution, an extension under development in iCheck.



(a) Impact of dynamic agent adaptation on a 2D heat simulation

(b) Effect of agent placement strategies on synthetic application

Fig. 4: Effect of dynamic agents and agent placement strategies



(a) Number of active iCheck nodes in the system

(b) Distribution of agents across iCheck nodes

Fig. 5: Demonstrating the horizontal scaling in iCheck during a time interval

5) Impact of dynamic agents in heat simulation

We can conclude from the above sections that checkpointing performance improves significantly in applications with a higher number of agents. This section examines whether this holds in the case of dynamically expanding the agents in an application. A 2D Jacobi heat simulation application, launched with 576 processes that checkpoint a total of 1.4GB, is used to evaluate on-the-fly agent adaptation.

The application is started with two agents, and we can observe in Figure 4a that it took around .7 seconds to transfer each checkpoint. After a few seconds, we can see the performance improvement in data transfer. This improvement was the result of agent change performed by the controller. As seen in the figure, a performance improvement of up to 80% is attained as new agents are added to the application. The overhead incurred by the application for a successful agent change was around 2.5 seconds (cost of four commits). This overhead is negligible since we got double the performance improvement with each successful agent change.

6) Agent placement analysis with synthetic applications

This subsection evaluates the placing of agents on iCheck nodes based on different strategies introduced in Section IV. In this analysis, we ran a synthetic application (say app A) in a continuous loop on two nodes (96 processes) with a single agent. Then, we ran an additional five synthetic applications (two nodes per application), placed the agents associated with these applications on different iCheck nodes, and studied their effect on app A's bandwidth with the ensuing agent placement strategies. 1) Stacking: All the agents were placed in the same iCheck node as app A in this strategy. 2) Bandwidth: Agents were placed on iCheck nodes with least activity (higher bandwidth will be available for data transfer). 3) CP interval: Agents are placed in iCheck nodes that have a low checkpoint interval between applications. We can see from Figure 4b that these placement strategies can influence the performance of the application. For example, in the stacking strategy, the data transfer rate for checkpoints between the application and agents fluctuated between 3GB/s to 11.5GB/s. Meanwhile, in the strategies where agents are distributed among different iCheck nodes, the fluctuation is minimal. The strategy that always distributed the agents in iCheck nodes with least activity yielded the best performance (a bandwidth of 9.5GB/s to 11.5 GB/s) for our test app A.

7) Horizontal scaling and agent placement analysis

iCheck is fully malleable and can grow or shrink based on the node availability. To fully utilize this ability, iCheck needs to be integrated with a malleable resource manager that can give/take nodes to/from the iCheck system based on the system status. Hence, a concrete evaluation of this feature is not yet possible and is beyond the scope of this work. However, we can demonstrate the ability of iCheck to scale its nodes horizontally by simulating a scenario were nodes are given to iCheck in predefined time intervals. Then we will launch the ls1 mardyn (on two nodes), LULESH(on two nodes), heat distribution (on four nodes), and synthetic (on four nodes) applications with a prescheduled agent distribution of one, two, four and four agents respectively in predetermined intervals. The agent placement strategy that gave best performance in

Subsection V-B6 is used in this analysis.

We can observe the results of the experiments in Figure 5a and 5b. The former figure depicts the active iCheck nodes during a time interval, and the latter displays the agent to iCheck node placement during the same period. The increase in the iCheck nodes as seen in Figure 5a validates the ability of our system to scale as the resources (nodes) becomes available. Additionally, we can infer from the second Figure 5b that not only was iCheck able to scale horizontally but also used these new iCheck nodes for agent placement. Initially, ls1 mardyn was launched with a single agent, and it is placed in the only available node (See iCheck node 1 in Figure 5b). Later, we can see that a new node (iCheck node 2) was added at around the 40th second (See Figure 5a). When LULESH was started, the two agents assigned to it were distributed among the iCheck node one and iCheck node two. We can observe it in Figure 5b (See 150th second).

VI. Conclusion and Future Work

In this paper, we presented a new malleable two-level application-level checkpointing system called iCheck. iCheck supports malleability by increasing its resources on-the-fly based on the requirements of the application. We demonstrated the working of iCheck on SuperMUC-NG with four applications. We improved application execution time on a production-level molecular dynamics simulation application (ls1 mardyn) by improving the checkpointing performance using iCheck. We also evaluated iCheck against the state-of-the-art SCR system and obtained up to 57% faster checkpointing. Additionally, we showcased the impact of dynamic agents by improving checkpointing performance up to 80% on-the-fly in a heat simulation application. Our experiments showed that the main bottleneck associated with iCheck lies in the remote memory registration with agents. We plan to optimize this process and further improve the overall performance of our system.

VII. Acknowledgment

The research leading to these results was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)-Projektnummer 146371743-TRR 89: Invasive Computing.

References

[1] B. Schroeder and G. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, 09 2007.

[2] M. A. Heroux, "Software challenges for extreme scale computing: Going from petascale to exascale systems," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 437–439, 2009. [Online]. Available: https://doi.org/10.1177/1094342009347711

[3] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of resilience techniques for exascale computing platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 914–923.

[4] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems*, K. M. Decker and R. M. Rehmann, Eds. Basel: Birkhäuser Basel, 1994, pp. 213–218.

[5] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[6] J. Stearley, J. Laros, K. Ferreira, K. Pedretti, R. Oldfield, R. Riesen, and R. Brightwell, "rmpi : increasing fault resiliency in a message-passing environment." 04 2011.

[7] G. Zheng, L. Shi, and L. V. Kale, "Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, ser. CLUSTER '04, 2004, p. 93–103.

[8] G. E. Fagg and J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2000, p. 346–353.

[9] M. Kumar, A. Choudhary, and V. Kumar, "A comparison between different checkpoint schemes with advantages and disadvantages," 2014.

[10] A. Lumsdaine and J. Hursey, "Coordinated checkpoint/restart process fault tolerance for mpi applications on hpc systems," 2010.

[11] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. Supinski, N. Maruyama, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," 11 2012, pp. 1–10.

[12] J. John, S. Narváez, and M. Gerndt, "Invasive computing for power corridor management," *Parallel Computing: Technology Trends*, vol. 36, p. 386, 2020.

[13] M. Chadha, J. John, and M. Gerndt, "Extending slurm for dynamic resource-aware adaptive batch scheduling," in *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2020, pp. 223–232.

[14] L. V. Kalé, S. Kumar, and J. DeSouza, "A malleable-job system for time-shared parallel machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002, pp. 230–230.

[15] N. Sultana, A. Skjellum, I. Laguna, M. Farmer, K. Mohror, and M. Emani, "Mpi stages: Checkpointing mpi state for bulk synchronous applications," 09 2018, pp. 1–11.

[16] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogeneous architectures," 07 1997, pp. 58 – 67.

[17] R. Arora and T. N. Ba, "Italc: Interactive tool for application-level checkpointing," in *Proceedings of the Fourth International Workshop on HPC User Support Tools*, ser. HUST'17, 2017.

[18] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," *TCON*, 12 1995.

[19] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. Supinski, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," 05 2014, pp. 21–30.

[20] D. Kimpe, K. Mohror, A. T. Moody, B. C. V. Essen, M. B. Gokhale, R. B. Ross, and B. R. de Supinski, "Integrated in-system storage architecture for high performance computing," in *ROSS '12*, 2012.

[21] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," 05 2019, pp. 911–920.

[22] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, 08 2017.

[23] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015, pp. 895–906, 01 2015.

[24] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010, p. 1–11.

[25] "Libfabric library." [Online]. Available: https://github.com/ofiwg/libfabric

[26] "Supermuc-ng." [Online]. Available: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG

[27] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and M. Horsch, "ls1 mardyn: The massively parallel molecular dynamics code for large systems," *Journal of Chemical Theory and Computation*, vol. 10, no. 10, pp. 4455–4464, 2014.

[28] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.