Hierarchical Clustering Strategies for Fault Tolerance in Large Scale HPC Systems

Leonardo Bautista-Gomez*, Thomas Ropars†, Naoya Maruyama‡, Franck Cappello§, Satoshi Matsuoka*
*Tokyo Institute of Technology, Japan
†École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
‡RIKEN AICS, Japan
§INRIA, France

Abstract*—Future high performance computing systems will need to use novel techniques to allow scientific applications to progress despite frequent failures. Checkpoint-Restart is currently the most popular way to mitigate the impact of failures during long-running executions. Different techniques try to reduce the cost of Checkpoint-Restart, some of them such as local checkpointing and erasure codes aim to reduce the time to checkpoint while others such as uncoordinated checkpoint and message-logging aim to decrease the cost of recovery. In this paper, we study how to combine all these techniques together in order to optimize both: checkpointing and recovery. We present several clustering and topology challenges that lead us to an optimization problem in a four-dimensional space: reliability level, recovery cost, encoding time and message logging overhead. We propose a novel clustering method inspired from brain topology studies in neuroscience and evaluate it with a Tsunami simulation application in TSUBAME2. Our evaluation with 1024 processes shows that our novel clustering method can guarantee good performance for all of the four mentioned dimensions of our optimization problem.

I. Introduction

In high performance computing (HPC), application executions can last several days and in some cases several weeks. Such executions need to be protected against possible failures using fault tolerance (FT) techniques. In this work we focus on tightly-coupled parallel applications based on the message passing programing model, e.g. MPI (Message Passing Interface [18]) applications. Checkpoint-Restart (CR) is usually used to provide FT for these applications because it is much less resource consuming than replication techniques. However, CR suffers from several issues that need to be addressed in order to be efficient at large scale. The first issue is the checkpointing overhead. Indeed, future systems with hundreds of thousands of sockets will fail at a higher frequency than current systems and at the same time the amount of data to save will be bigger. Thus, storing the state of a large execution reliably will be increasingly challenging. Another issue is the cost of recovery. In the standard CR technique, all the processes restart from the last checkpoint in the event of a failure. This is not always necessary and so, it is a waste of resources. In systems where applications can easily spawn millions of processes, confining the failure to a single node, or a small set of nodes, and forcing only these nodes to restart can represent a substantial gain in resources.

Several works [20], [22], [3], [2] reduce the checkpointing overhead by using local storage in combination with erasure codes. While using these techniques, the system is partitioned in groups or clusters of processes and each cluster can tolerate a given number of failures. On the other hand, hybrid CR protocols [6], [13], [17], [27] try to reduce the cost of recovery by using group-coordinated checkpointing plus message logging. In these protocols, the system is partitioned in clusters. Checkpoints are coordinated within clusters, but are not coordinated between clusters. This is combined with message logging to provide failure containment. Only inter-cluster messages are logged, decreasing the amount of data to be logged compared to full message logging.

In this paper, we combine all these techniques in order to create a framework that reduces the checkpoint overhead and simultaneously reduces the recovery cost. To reach this goal, we start by combining all these techniques in a naïve approach and study how our framework performs for different aspects. During our study several challenges are raised and lead us to an optimization problem in a four-dimensional space. We propose a solution for this optimization problem and evaluate it using a tsunami simulation application in TSUBAME2.

A. Contributions

This paper is, to the best of our knowledge, the first study to combine fast checkpointing and failure containment techniques together in order to reduce the checkpoint overhead and the recovery cost for large scale HPC systems. Our contributions are summarized as follows.

- We study the fast checkpointing and failure containment clustering requirements, which raise several clustering challenges. We explain how combining the previously mentioned techniques lead us to an optimization problem in a four dimensions space.
- We propose a novel hierarchical clustering as solution to our optimization problem. Our solution allows systems to couple fast checkpointing and failure containment.
- We evaluate our hierarchical clustering in TSUBAME2 using a tsunami simulation application on 1024 processes. We analyze the implemented technique and we show that our solution scores high in all four dimensions of the presented optimization problem.

The rest of this paper is organized as follows. In Section II we explain the background and motivations for this work. Section III details our study and the optimization problem.

Section IV describes our hierarchical clustering solution and Section V shows the results of our evaluation on TSUBAME2. Finally, Section VI reviews the related work and Section VII presents our conclusions.

II. Background and motivations

In this section we first present the main issues related to fault tolerance in large scale HPC systems. Then, we describe the state-of-the-art solutions proposed to improve either checkpointing or recovery performance, and discuss the advantages of combining the approaches. Finally, we point out that both approaches are based on clustering, but that their clustering strategy are very different.

A. Challenges for Fault Tolerance at Extreme Scale

To provide fault tolerance for parallel HPC applications, the usual approach is to use a coordinated checkpointing protocol implemented at application or system level. All processes are coordinated at checkpoint time to ensure that the recorded state is consistent, and the checkpoints are transferred to the parallel file system (PFS), which is assumed to be highly reliable. In the event of a failure, all processes rollback to the last coordinated checkpoint available on the PFS. Studies show that such an approach is not suitable at extreme scale because a large part of the execution time would be spent saving checkpoints or recovering from a failure [21], [10].

Indeed future Exascale systems raise new challenges for fault tolerance. The mean time between failures (MTBF) of such systems is expected to be low, requiring a high checkpointing frequency to allow applications to progress. At the same time, the amount of data to save as part of the application state is increasing much faster than the I/O bandwidth provided by the PFS, leading to a longer checkpoint time. To mitigate the cost of fault tolerance at extreme scale, two main research directions are investigated: i) improving the techniques used to save checkpoints data to reduce the checkpointing time; ii) designing more efficient checkpointing protocols to reduce the cost of handling failures. In the next section, we discuss these two aspects.

B. Advanced Fault Tolerant Solutions

1) Fast Checkpointing and Erasure Codes: To sustain high checkpointing frequency, the time required to save a checkpoint needs to be reduced. Several solutions have been designed to circumvent the I/O bottleneck [20], [3], [2]. First, models with several levels of reliability have been proposed [20], [3]. Multi-level checkpointing takes advantage of nodes local storage devices to avoid saving all checkpoints on the PFS. Since local storage devices provide better performance than the PFS, the application can be checkpointed much more frequently. Such techniques are based on an important observation: Most failures in current supercomputers affect only a small fraction of the system, where the affected part is often one single node or a small set of nodes [3].

By checkpointing in local storage, an application is able to tolerate transient failures affecting the data integrity of the application. Soft-errors are expected to become one of the main source of failures in future systems [5]. Using classic CR to tolerate soft-errors is a waste of resources since saving the checkpoint data on the PFS is not necessary to restart the execution. To deal with node failures, local checkpointing solutions need to be complemented with erasure codes [20], [3]. Such techniques can improve resiliency by several order of magnitude. Parity data are generated through distributed encoding performed after the checkpoints have been stored locally. Upon a failure, the lost data is rebuild using the parity data saved on other nodes. Several encoding techniques, such as bit-wise XOR or Reed-Solomon, exist and provide different encoding complexities and different reliability levels [7], [20].

2) Failure Containment: To get a scalable CR protocol, hybrid protocols combining coordinated checkpointing and message logging have gain attention [6], [13], [17], [27] because they can limit the consequence of a failure to a small subset of the processes, *i.e.*, they provide failure containment. One of the main drawback of coordinated checkpoints is that a single failure makes all processes rollback to the last coordinated checkpoint. By providing failure containment, hybrid protocols i) reduce the amount of rolled back computation, and so reduces the amount of wasted energy, and ii) can speed up the recovery [23], [12].

Contrary to coordinated checkpointing protocols, message logging protocols can efficiently limit the consequences of a process failure. Using a causal or a pessimistic message logging protocol, only the failed processes have to rollback after a failure [11]. However, message logging protocols require to log all messages payload in the nodes memory during failure free execution [14]. This logging can impair communication performance. More importantly, it imposes a high memory footprint that increases with the communication rate of the application. Thus, such a protocol is not suitable at very large scale. Hybrid protocols aim at providing failure containment without the drawbacks of pure message logging protocols. Coordinated checkpointing is used within clusters of processes. Message logging is only used for inter-cluster communication to ensure that if a process in one cluster fails, only the processes in this cluster have to rollback. Thus, the consequences of a failure are limited to a small subset of the processes while logging only a small fraction of the messages.

C. Combining the Approaches

In this paper, we study how hybrid CR protocols can be combined with multi-level checkpointing and erasure codes for fault tolerance at extreme scale. We focus on our previous works: FTI [3], a checkpointing library based on multi-level checkpointing and erasure codes, and HydEE [13], an hybrid CR protocol implemented in the MPICH2 library. Considering the features they provide, the two approaches are complementary. If HydEE only relies on the PFS to save checkpoints, checkpoint scheduling strategies, *i.e.*, checkpointing the clusters of processes at different time, have to be used to avoid the I/O bottleneck. Implementing such strategies has two main drawbacks. First it prevents from taking advantage of application-level checkpointing since system-level checkpointing has to be used to be able to control when a checkpoint is taken. Second, tightly-coupled MPI applications performance might be significantly affected by the *noise* introduced by non-coordinated checkpoints.

Combining HydEE with FTI will allow to checkpoint all application processes *"at the same time"*[1] at a high frequency, and so, avoid the need to implement scheduling strategies. Also, this solution will provide failure containment whereas FTI combined with coordinated checkpointing requires all processes to rollback in the event of a failure. However, combining the two approaches is not straightforward. As mentioned above, HydEE relies on process clustering for failure containment. But FTI also relies on clustering to decrease the encoding time in erasure codes [3]. As we will see, the clustering techniques used in FTI and HydEE are different.

1) Clustering for Erasure Codes: Encoding is done using mathematical operators on a *distributed* set of data, in our case, multiple checkpoint files distributed among the compute nodes. Its complexity is directly proportional to the amount of data to encode. Clusters are defined to independently use erasure codes within each cluster. It is important to keep the clusters size small enough to guarantee fast encoding [3]. Using small clusters that can be encoded in parallel significantly improve the checkpointing performance [2].

Erasure codes can only work if data and parity data are distributed among multiple different physical locations (i.e. multiple compute nodes). If all the data and parity data is stored in the same *physical container*, the failure of one *container* leads to an unrecoverable failure, and the time spent encoding is wasted. It is then necessary to use clusters with a good distribution through multiple *physically distant containers*. The efficiency of erasure codes for FT in HPC is directly linked to the distribution of the checkpoint and parity data among multiple physical compute nodes. As we can see in Figure 1, in the case of FTI [3], the clusters are created in such a way that all the processes of a cluster belong to different physical nodes and all the processes in a same physical node belong to different virtual clusters.

2) Clustering for Failure Containment: Processes clustering for a hybrid coordinated checkpointing/message logging protocol should satisfy two requirements: i) minimizing the number of processes to rollback after a failure; ii) limiting the amount of data to log. To minimize the amount of data to log, the amount of inter-cluster communication should be minimized. The simplest solution is to choose large clusters. However, it would imply that a large a number of processes would rollback in the event of a failure. To solve this issue, the communication pattern of the application has to be taken into account. Indeed, the communication graph of most HPC applications shows a low degree of connectivity [15]. It has been shown that a good trade-off can be found between the size of the clusters and amount of data to log for most MPI HPC applications [24].

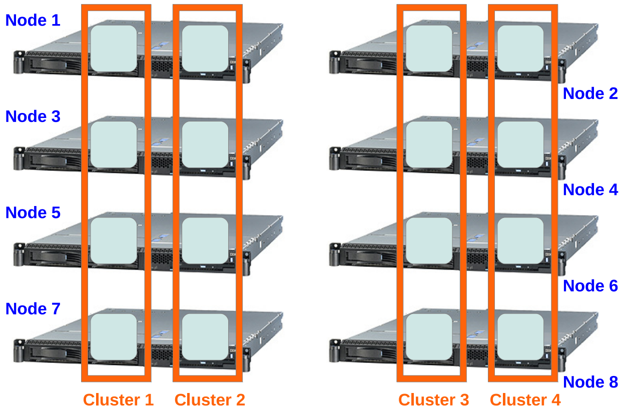


Fig. 1. Erasure codes clustering

In addition to the communication pattern of the application, the probability that multiple processes fail simultaneously can be taken into account. When a process fails, all processes belonging to the same cluster have to rollback, since coordinated checkpointing is used within clusters. If two processes have a very high probability to fail simultaneously, they should be put in the same cluster to ensure that their failures impact only one cluster. In [6], all processes running on the same node belong to the same cluster. This solution could be extended to deal with correlated node failures. For instance, two nodes sharing a power supply should be located in the same cluster, so that only one cluster restarts after a power supply failure.

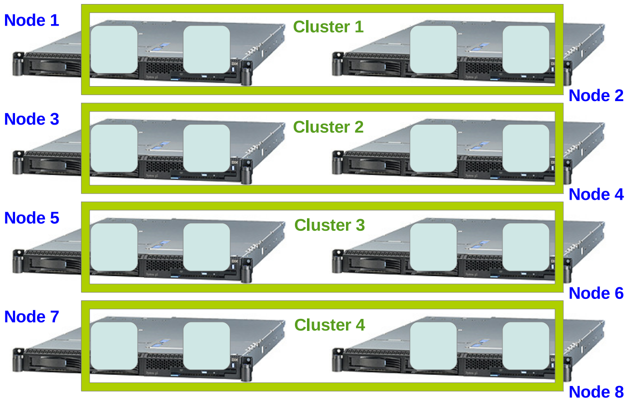


Fig. 2. Failure containment protocol clustering

To optimize communication performance, processes communicating frequently together should be located as physical neighbors in the machine [26]. In this case, optimizing clustering based on the application communication pattern provides also a good solution with respect to correlated failures, as illustrated by Figure 2.

[1] It does not imply that a coordinated checkpointing protocol is executed between all processes in the application.

Combining FTI and HydEE raises a non trivial problem since the optimal clustering for the two techniques are conflicting. This paper explores techniques and proposes a novel approach to solve this multi-criteria optimization problem.

III. Topology Study and Clustering Challenges

To couple the FT techniques presented in the previous section, a suitable clustering strategy has to be found. We study and evaluate several clustering solutions in this section.

In order to evaluate the hybrid CR protocol, we use the communication graph obtained by executing a tsunami simulation application [1] with 1024 processes in TSUBAME2. Details on the platform and the technique used to get the communication graph are provided in Section V. This tsunami simulation is a good example of stencil applications which are widely used in HPC [16]. It performs a 2-dimensional decomposition of a sea region and each process computes the fluid dynamics of its segment. Processes communicate with their neighbors to share *ghosts regions*. To maximize intra-node communications, and so performance, consecutive process ranks are placed on the same node. Each node hosts 16 processes. The data was collected for a short execution of 100 iterations.

We start by defining a *baseline* of requirements that a clustering should reach in order to be efficiently used for large scale HPC systems. These requirements are established using the cost function, reliability model and performance model proposed in our previous studies [3], [24]. They can be used to model different configurations. First, the system should not log more than 20% of the messages. We consider that 20% of the communications is already a large amount of data and since future systems are expected to have less memory space per process it is important to reduce the memory overhead generated by message logging. Second, the system should be capable of encoding 1GB of data in less than one minute. One minute per GB is already a slow encoding and as the amount of memory per node is increasing with time, it is mandatory to encode data as fast as possible in order to decrease the checkpoint overhead. Our third requirement is to have a system where only one in several thousand failures is unrecoverable. Since failures will be more frequent in large systems it is important to have a very low probability of unrecoverable failure. Finally, the system should avoid restarting more than 20% of processes after a failure. Since most failures affect only one or a small set of nodes, restarting 20% of nodes after a failure is already a significant waste of resources.

As described in the previous section, a hybrid CR protocol uses coordinated checkpointing within clusters. When a process fails, it forces all processes in the same cluster to rollback. On the other hand, the encoding technique requires that the processes in the same encoding cluster coordinate after the checkpoint data is saved locally because the encoding algorithm itself requires coordination. Furthermore, the decoding algorithm also requires coordination before restarting the execution. Encoding checkpoint files of processes that do not checkpoint at the same time would be highly inefficient. Thus, the processes of the encoding clusters must checkpoint in a coordinated fashion and restart together after a failure. This observation lead us to use the same clustering for both, the failure containment protocol and the encoding algorithm, guaranteeing that the processes of a same cluster will checkpoint and encode in a coordinated fashion. In this section, we explore and evaluate several clustering, comparing their overhead with the *baseline* introduced above.

A. *Naïve Clustering*

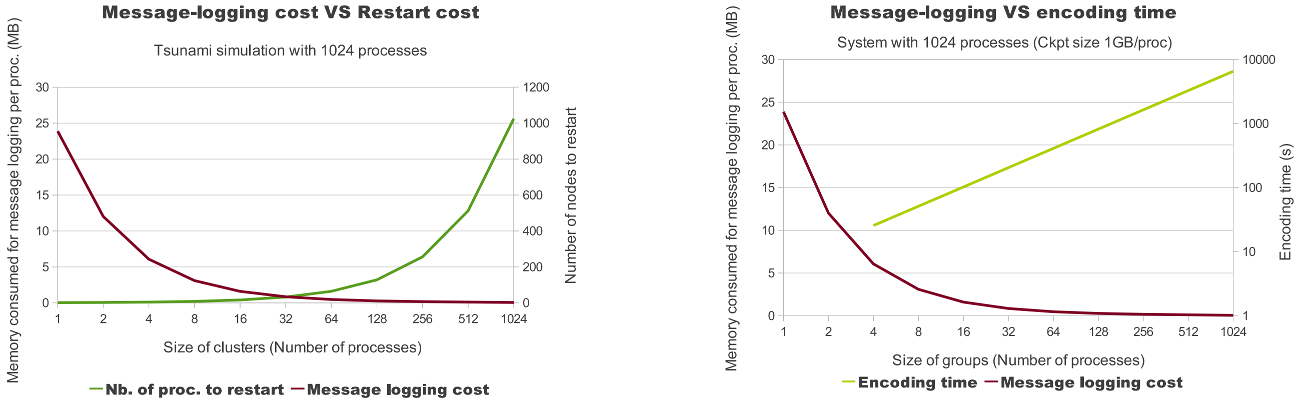
The first clustering challenge comes while choosing the optimal cluster size for the failure containment protocol. As we explained in Section II-C2, it is necessary to make clusters in such a way that we limit the number of messages logged but at the same time we limit the number of processes to restart in the event of a failure. We study the impact of the clusters size on message logging and recovery cost using the communication graph of the tsunami simulation. The influence of the communication patterns of different applications on the results of the failure containment protocol has already been studied [24] and it is out of the scope of this paper.

Figure 3a shows the trade-off between message logging overhead and recovery cost. In this evaluation, each cluster gathers a set of consecutive process ranks. The message logging overhead is shown using the left-handed axis while the restart cost is shown using the right-handed axis. This figure presents results for a short execution, but for long executions each process will communicate many GBs of data, making logging prohibitively expensive for small clusters. However, the message logging overhead can be reduced using larger clusters. As we can see, there is a *sweet spot* for clusters of 32 processes where less than 4% of the messages are logged and only 3% of the processes needs to restart after a failure.

Thus, we use clusters of 32 processes in order to optimize the message logging vs. recovery cost trade-off. We call this *naïve clustering*. However, experiments using the *naïve clustering* produced a very poor encoding performance. Indeed, *naïve clustering* does not take into account the impact of the cluster size on the encoding speed. In fact, such large clusters will produce a highly time consuming encoding that becomes prohibitively expensive at high checkpointing frequency.

B. *Size-guided Clustering*

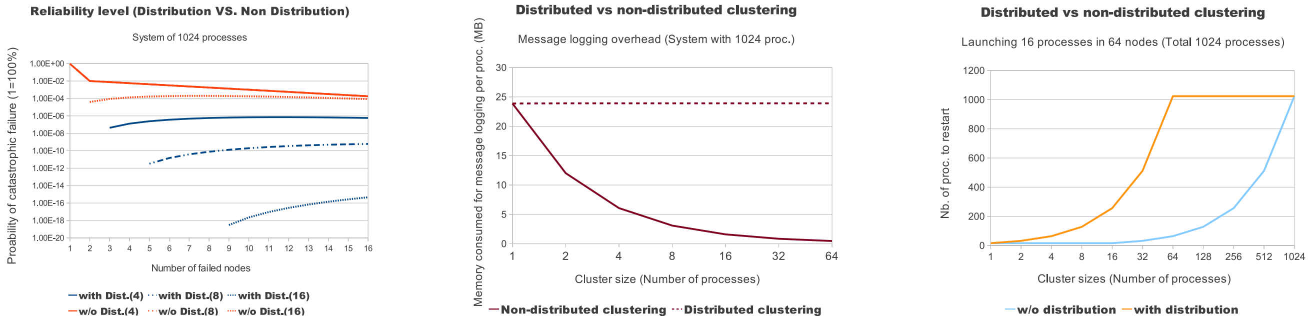
In order to solve the issue mentioned above we study the impact of the cluster size on the encoding time and compare it with the message logging overhead. Figure 3b shows this comparison, using the left-handed axis for the message logging overhead, and the right-handed axis for the encoding time. Please notice that the right-handed axis uses a logarithmic scale. The encoding time measure starts at clusters of size 4 since it does not make sense to use erasure codes to tolerate multiple failures in clusters of one or two processes. As we can see, while increasing the cluster size from 4 processes to 32 processes, the encoding time increases by almost one order of magnitude. In clusters of 32 processes, encoding 1GB of checkpoint data takes more than three minutes while it could take less than half-minute with clusters of 4 processes. In other



(a) Recovery cost VS. message logging overhead

(b) Encoding time VS. message logging overhead

Fig. 3. Cluster size study



(a) Reliability level

(b) Message logging overhead

(c) Restart cost

Fig. 4. Distribution study

words, encoding 20GBs of data will take more than one hour while it could take less than five minutes. Because of this inadmissible loss in performance, we search for a trade-off between all three: encoding time, message logging overhead and recovery cost. We call this strategy *size-guided clustering*. Clusters of size 8 comply with the first two requirements (See Section III-A) by logging only 13% of messages and encoding at a 1GB/50s rate. In comparison, clusters of size 4 would result in 25% of logged communications and clusters of size 16 would take almost 2 minutes to encode 1GB of data.

However, this configuration lacks reliability. In some cases one node failure could lead to an unrecoverable failure. Indeed, while using the failure containment protocol, one want to create clusters in such way that the number of intra-cluster communications is maximized and the inter-cluster communications get minimized. In parallel, users usually implement topology-aware positioning techniques [4], [26] that maximize data locality and intra-node communication in order to optimize the use of resources and get better performance. In the tsunami simulation application, most of the communications done during the execution are boundary exchanges between neighbor regions. Thus, intra-node communications are maximized by placing neighbor processes in the same compute node. As a result, clusters of 8 consecutive processes are grouping processes located on the same compute node. This is the worst scenario for erasure codes. As explained in Section II-C1, erasures codes work by distributing data and parity data among distant physical nodes. Instead, by grouping all the data and parity data in the same node, the erasure codes cannot tolerate hard node failures, losing all their benefits and making the encoding useless. In other words, locality enhances performance, while distribution enhances reliability. As we can see, in the context of this topology study, performance and reliability are conflicting.

C. Distributed Clustering

It is necessary to find a configuration where the erasure codes allow the system to tolerate multiple simultaneous node failures. Evidently, by grouping all the processes of the same node in a cluster the system cannot guarantee this, so we test a clustering where all the processes of a cluster belong to different nodes, we call this technique *distributed clustering*. We perform a reliability study comparing distributed and non-distributed clustering methods. We use our catastrophic failure model presented in [3]. We assume a system of 128 nodes with 8 processes per node (1024 processes in total). We study the reliability for clusters of 4, 8 and 16 processes. In the distributed clustering all the processes of a cluster belong to different nodes while in the non-distributed clustering all the processes of a cluster are hosted by one or two nodes. As we can see in Figure 4a, non-distributed clustering is several orders of magnitude less reliable than distributed clustering. For non-distributed clusters of 4 or 8 processes, one single node failure could lead to an unrecoverable failure.

Unfortunately, *distributed clustering* also raises new issues. The first one is the message logging overhead. Indeed, since the clusters are composed of processes belonging to different nodes and since the processes are located in order to maximize the intra-node communications, it is expected to see a high percentage of messages logged even while using large clusters. Figure 4b shows a comparison between distributed and non-distributed clustering. Combining *distributed clustering* and topology-aware process positioning results in a very high number of messages logged. This configuration impacts so badly the message logging technique that the size of the clusters lose all their influence in the performance trade-off.

This is not the only issue of *distributed clustering*. The recovery cost also grows faster while using *distributed clustering*. The reason is that one single process failure in a cluster forces all the processes in the cluster to restart. For instance, when a node with 16 processes fails and the 16 processes of the node belong to different clusters of 16 processes, each one of the 16 failed processes will force other 15 to restart; as a result one single node failure forces 16 nodes to restart. Figure 4c shows a comparison between *distributed clustering* and non-distributed clustering for a system with 64 nodes of 16 processes (1024 processes). As we can see, the impact of *distributed clustering* on recovery cost is so large that for clusters of 32 processes the recovery cost grows from 3% with non-distribution to 50% with distribution.

In summary, two factors (cluster size and process distribution) are affecting four different parameters: encoding time, recovery cost, reliability level and message-logging overhead. All these four parameters are correlated, going sometimes in opposite directions. At this stage, none of the proposed clustering was able to reach our four requirements. All of our previously mentioned clustering techniques perform very poorly in at least one of these four dimensions.

IV. Hierarchical Clustering for HPC

As previously presented, physical distance between processes of the same cluster enhances reliability and decreases performance. In contrast, proximity enhances performance and decreases reliability. These conflicting goals lead us to design more elaborated clustering schemes capable of ensuring both performance and reliability. In this section, we start with a very short overview of clustering techniques developed in neuroscience, as this was the inspiration for our proposed solution. Then, we present and develop our proposal.

A. Brain Segregation, Distribution and Modularity

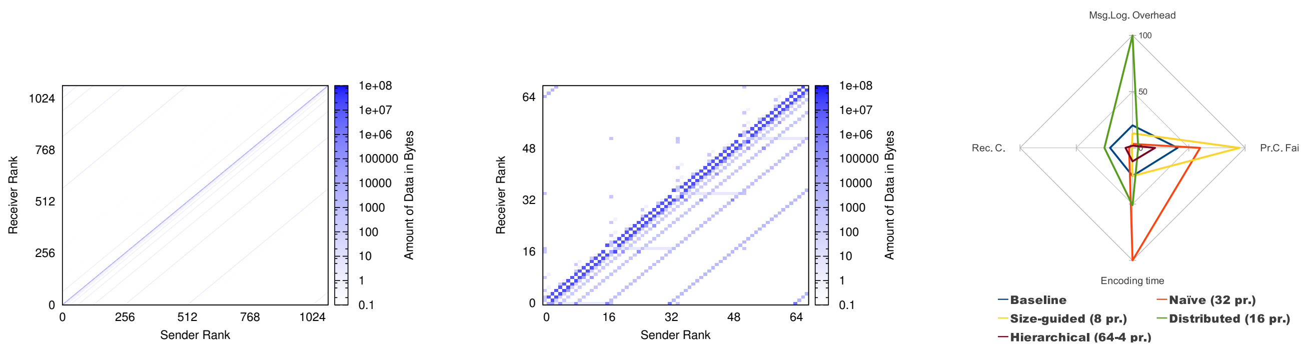
Systems with highly computational tasks and high reliability exist across different domains. A clear example is the brain itself. Brain networks are known to share multiple properties with other complex non-biological networks and they have important characteristics. One of these characteristics is called functional segregation. Indeed, it has been found that densely interconnected clusters of regions exist in the brain network. Such clusters are capable of specialized processing indicating statistical dependencies between regions. These community structures are revealed by partitioning the brain network into clusters that maximize the number of intra-cluster links and minimize the number of inter-cluster links [19], [28]. As we can see, this is exactly the same strategy we use to create highly efficient message logging clusters.

Another important characteristic of brain networks is the degree distribution [25]. The degree of a single node can be measured counting its number of links or neighbors. The degree distribution in a brain network is obtained by adding the degree of all the nodes in the network. The degree distribution is an important marker of network evolution and resilience. In addition to these brain network characteristics, other works have studied the hierarchical optimization on human brain networks using functional magnetic resonance imaging (fMRI) [19], [8]. It has been established that there is a clear evidence of hierarchical modularity in human brains. Hierarchical modularity allows systems to combine densely interconnected regions with resilient distribution for faster adaptation or evolution in rapidly changing external conditions. Based on the presented evidence of the robustness of hierarchical modularity in brain networks, we propose a clustering approach with a hierarchical scheme, aiming to optimize the four dimensions of our optimization problem.

B. Hierarchical Clustering implementation

We propose a hierarchical clustering composed of two levels. The first level (L1) aims to reduce the message logging overhead vs. restart cost. The second level (L2) aims to ensures fast encoding and high reliability. Building the hierarchical clustering includes the following steps. First, it is required to obtain the application's communication matrix. From the obtained process communication graph, it is simple to construct a node-based communication graph. Then, we apply the partitioning algorithm and cost function presented in [24] over the node-based communication graph. By using node-based instead of process-based communication graphs we guarantee that all the processes of each node belong to the same cluster, so that at most one cluster needs to restart after a node failure.

Once the L1 clustering is done, we apply the L2 clustering inside the L1 clusters, using the following criteria: Larger L2 clusters lead to more reliability, but smaller L2 clusters improve encoding speed. However, clusters of 4 or 8 processes are already highly reliable if the processes are distributed in different compute nodes, as presented in figure 4a. In order to apply failure distribution techniques inside L1 clusters, we need L1 clusters large enough to implement such failure distribution scheme. Therefore, we set the minimum number of nodes per L1 cluster to 4 in the partitioning algorithm. This guarantees that the systems will be able to apply erasure codes



(a) Communication Pattern

(b) Zoom on the 68 First Processes

(c) Overall clustering comparison

Fig. 5. Hierarchical clustering study and comparison

inside each L1 cluster. In systems with thousands of nodes, message logging overhead decreases by grouping multiple nodes in a cluster, thus the partitioning algorithm for L1 clusters is very likely to produce L1 clusters of more than 4 nodes, even if the minimum was not a requirement.

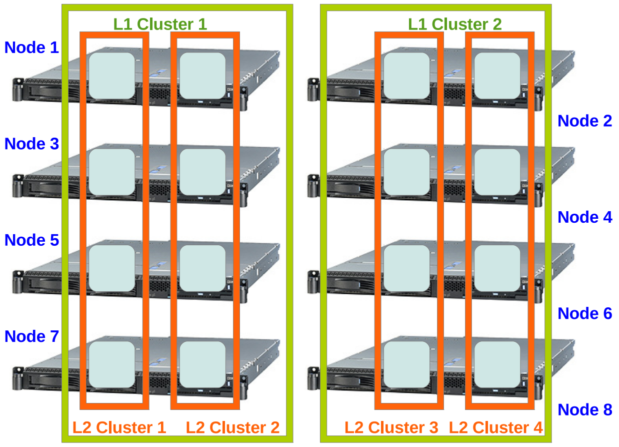


Fig. 6. Hierarchical clustering

Now that we have guaranteed that failure distribution is possible inside L1 clusters, we just need to keep the size of the L2 clusters as low and homogeneous as possible: low to have fast encoding and homogeneous to have all the clusters spending about the same time in encoding. Thus, we divide the L1 clusters in groups of 4 nodes (or more), and then we group the i^{th} process of each node in the group in a L2 cluster, leading to x L2 clusters inside the group, where x is the number of processes per node. Figure 6 shows a simple scenario where hierarchical clustering has been applied to a 8-node cluster. The reliability and performance models presented in our previous work [3] can be used to guide the L2 clustering by predicting the probability of catastrophic failure and encoding time for a given cluster size.

V. Evaluation

We evaluate our proposed hierarchical clustering on TSUBAME2 using the tsunami simulation application described in section III, launching from 64 to 1024 processes. In this section we study the case of 1024 processes. The experimental platform is given in Table I. The application is run with a modified version of the MPICH2 library including HydEE CR protocol. We made a minor modification to the library to collect data on communications.

Nodes	1408 High BW Compute Nodes
CPU	2 Intel Westmere-EP 2.93GHz 12Cores/node
Mem	55.8GB or 103GB (Total: 80.55TB)
GPU	NVIDIA M2050 515GFlops, 3GPUs/node (Total: 4224 NVIDIA Fermi GPUs)
SSD	60GB x 2 = 120GB (55.8GB node) 120GB x 2 = 240GB (103GB node) (Total : 173.88TB) Write speed : 360MB/s (RAID0)
Network	Dual rail QDR IB (4GB/s x 2)
File system	5 DDN DFA10000 units (3 Lustre and 2 GPFS) with 600 2TB HDDs each Measured Lustre write troughput (10GB/s)
OS	Suse Linux Enterprise + Windows HPC

TABLE I
TSUBAME2 ARCHITECTURE

TSUBAME2 nodes have 12 cores and it uses hyperthreading, so it allows a maximum of 24 processes to be launched per node. Since the application requires a power-of-two number of processes, we launch the application on 64 nodes with 16 application processes per node. The extra 8 processes *available* per node could not be used for the application, so they can be used for other purposes, such as fault tolerance. We link the application with FTI and we use FTI for multi-level checkpointing using the solid-state drives (SSDs) in the compute nodes and the Reed-Solomon encoding algorithm implemented in FTI. As explained in section IV-B, FTI uses one extra process per node for fast encoding, so we launch 17 processes per node for a total of 1088 MPI processes.

This raises a technical issue. The encoding process is a MPI process that communicates with the application processes in the same node at every checkpoint and with other encoding processes during the encoding. FTI guarantees the correctness of the application by replacing the global communicator with a new communicator during the initialization. However, HydEE is not able to make the difference between encoding processes and application processes. Thus, all the encoding processes are grouped in a single L1 cluster to avoid logging the communications related to encoding. Then, we divide the rest of the system in L1 clusters of 4 or more nodes using the approach presented in Section IV-B. Since the processes of the tsunami simulation exchange boundary regions with their neighbors, communications get optimized by placing consecutive MPI ranks in the same physical node. As a result, the L1 clusters of 4 nodes correspond to 64 consecutive MPI processes. Following the introduced technique, we create 16 L2 clusters of 4 processes, where all the processes of a L2 cluster belong to different compute nodes of the same L1 cluster. We then launch the execution, logging inter-cluster communications, checkpointing several times using SSDs and encoding the application checkpoints with FTI.

Figure 5a shows the communication graph produced by the execution, where dark blue means a high amount of data communicated. In order to analyze these results in detail, Figure 5b zooms in the first 4 nodes (i.e. 64 application processes and 4 encoding processes). The first pattern we notice is the blue double diagonal in the middle which corresponds exactly to the communication pattern of the tsunami simulation (i.e., exchange of boundary regions). We also notice that the diagonals get interrupted for ranks 0, 17, 34 and 51 which are the four encoding processes. It is important to notice that most data communicated between these groups of processes is located in these two diagonals (dark blue) and none of these communications are logged because the first 64 processes of the application belong to the same L1 cluster.

We also notice four short horizontal lines in light blue again at 0, 17, 34 and 51 (y axis) which correspond to the few communications done between the application processes and the encoding process of the respective node. There are also some isolated points at the intersections of processes 0, 17, 34 and 51 which correspond to the communications done between the encoding processes during the encoding phase. Finally, we can observe other diagonals in light blue starting in the x axis from processes with a power-of-two rank . These diagonals correspond to the communication pattern of the *MPI_Allgather* implementation in MPICH2. Indeed, FTI uses *MPI_Allgather* during its initialization.

Clustering method	Msg.Log. overhead	Recovery cost	Encoding time(1GB)	Prob.cat. failure
Naïve (32 pr.)	3.5%	3.1%	204 s	1^{-4}
Size-guided (8 pr.)	12.9%	0.7%	51 s	0.95
Distributed (16 pr.)	100%	25%	102 s	1^{-15}
Hierarchical (64-4 pr.)	1.9%	6.25%	25 s	1^{-6}

TABLE II
CLUSTERING COMPARISON

We measure the performance of our proposed hierarchical clustering in all four dimensions and compare to the clustering strategies proposed in Section III. A detailed comparison of all the studied clustering techniques is presented in table II. As we can see, the hierarchical clustering logs less than 2% of the messages, restarts less than 7% of the processes after a failure, encodes checkpoints at 25s/GB and its failure distribution guarantees a very high reliability level. Let us remember that none of the other studied clustering could be efficiently used for large scale HPC systems. Figure 5c presents a comparison between all the clustering strategies and the *baseline*. The *baseline* is the normalized maximum overhead in all four dimensions that a clustering can have in order to be used at large scale (See Section III). Any clustering going outside the area delimited by the *baseline* is not suitable for FT in future large scale HPC systems. The hierarchical clustering complies with all the requirements and performs well in the four studied dimensions, providing a complete CR solution for large HPC systems. The same results are expected for other HPC applications, except in the case of all-to-all communications, applications using collective communication patterns, can also be correctly partitioned [24].

VI. Related Work

Several works have achieved high checkpointing performance by using local checkpointing in combination with erasure codes or checkpoint replication [20], [3], [7], [22], [9]. These works propose various techniques achieving different levels of reliability and performance, but all of them require data distribution on the compute nodes as the main strategy to guarantee reliability and performance at high checkpoint frequency. However, none of these techniques limit the cost of restart after failures. We complement these works by presenting how to combine such techniques with failure containment techniques that reduce the cost of restart.

On the other hand, several works [6], [13], [17], [27] aim to reduce the cost of restart by using hybrid protocols for failure containment. These studies do not limit the time spent in checkpointing which is crucial to limit the overhead on the application executions. Although hybrid protocols often use uncoordinated checkpoint between different clusters and coordinated checkpoint inside clusters, it is necessary to reduce as much as possible the checkpoint time spent for each cluster checkpoint. Indeed, while a cluster is checkpointing, processes from other clusters may depend on data from the checkpointing cluster. In particular, HPC application processes are tightly coupled and any slowdown in one single process will have an important negative impact on the overall execution performance. In this work, we complement such approaches by coupling hybrid protocols with fast checkpointing techniques.

Some works have studied the performance impact of topology-aware process positioning on different architectures such as 3D torus or fat tree networks [4], [26]. Although these works provide clever solutions for enhancing performance in multiple topologies, they do not study the clustering issues for fault tolerance. In this work, we partially use such approaches by implementing a topology-aware positioning strategy that optimizes resources usage in TSUBAME2 and increases the performance of the tested tsunami application. Also, we study the clustering challenges for fault tolerance and we propose a hierarchical clustering that guarantees performance and reliability even while using topology-aware positioning techniques.

This work is, to the best of our knowledge, the first attempt to combine fast checkpointing techniques using local storage and erasure codes with hybrid protocols using partially coordinated checkpoint and message logging for failure containment. Furthermore, this is, (again) to the best of our knowledge, the first time that all these techniques have been implemented and evaluated together with a scientific HPC application on a large HPC system, such as TSUBAME2.

VII. Conclusions

In this work we have introduced several existing techniques that partially solve known issues of CR at large scale. In order to combine all the introduced techniques together, we have analyzed the clustering approaches that they implement. We show clustering challenges that lead us to an optimization problem in a 4-dimensional space and we propose a hierarchical clustering inspired from studies of brain networks in neuroscience. We implement our hierarchical clustering strategy and we evaluate it using a tsunami simulation on 1024 processes on TSUBAME2. We perform a detailed analyze of the implemented solution and we demonstrate its feasibility. Our results show that our proposal highly optimizes the four dimensions of the presented optimization problem and is the only technique that reaches all the requirements needed for large systems. The hierarchical clustering proposed in this paper allows us to combine all the previously mentioned techniques in order to build, for the first time, a complete CR solution that minimize both, the checkpointing overhead and the recovery cost, for future large scale HPC systems.

References

[1] M. Arce-Acuna and T. Aoki. Multi-gpu computing and scalability for real-time tsunami simulation. In *HPCS '10: Proceedings of the International Conference on High Performance Computing & Simulation*, pages 125–132, January 2010.

[2] L. A. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 63 –72, may 2010.

[3] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *SC11*, page 32. ACM, 2011.

[4] A. Bhatelé, L. V. Kalé, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 110–116, New York, NY, USA, 2009. ACM.

[5] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25:10–16, November 2005.

[6] A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *Euro-Par 2011*, pages 51–64, 2011.

[7] Z. Chen and J. Dongarra. A scalable checkpoint encoding algorithm for diskless checkpointing. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 71 –79, dec. 2008.

[8] Z. J. Chen, Y. He, P. Rosa-Neto, J. Germann, and A. C. Evans. Revealing modular architecture of human brain structural networks by using cortical thickness from mri. *Cerebral Cortex*, 18(10):2374–2381, 2008.

[9] C. da Lu, C. da Lu, and D. A. Reed. Scalable diskless checkpointing for large parallel systems. Technical report, Ph.D. Dissertation, Univ. of Illinois at Urbana-Champain, 2005.

[10] E. N. Elnozahy et al. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[12] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. Hydee: An energy and memory efficient cluster-based hybrid checkpointing protocol for mpi applications. Technical Report TR-JLPC-11-05, INRIA-UIUC Joint Laboratory for Petascale Computing, 2011.

[13] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications. In *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS2012)*, Shanghai, China, 2012.

[14] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[15] S. Kamil, J. Shalf, L. Oliker, and D. Skinner. Understanding ultra-scale application communication requirements. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 178–187, 2005.

[16] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[17] E. Meneses, C. L. Mendes, and L. V. Kale. Team-based Message Logging: Preliminary Results. In *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010).*, May 2010.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. www.mpi-forum.org, 1995.

[19] D. Meunier, R. Lambiotte, and E. T. Bullmore. Modular and hierarchically modular organization of brain networks. *Frontiers in neuroscience*, 4(December):11, 2010.

[20] 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*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[21] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[23] S. Rao, L. Alvisi, and H. M. Vin. The Cost of Recovery in Message Logging Protocols. In *Symposium on Reliable Distributed Systems*, pages 10–18, 1998.

[24] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. In *Euro-Par 2011*, pages 567–578, 2011.

[25] M. Rubinov and O. Sporns. Complex network measures of brain connectivity: uses and interpretations. *NeuroImage*, 52(3):1059–1069, 2010.

[26] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 77:1–77:11, New York, NY, USA, 2011. ACM.

[27] J.-M. Yang, K. F. Li, W.-W. Li, and D.-F. Zhang. Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery. *Concurrency and Computation : Practice and Experience*, 21:819–853, April 2009.

[28] C. Zhou, L. Zemanová, G. Zamora, C. C. Hilgetag, and J. Kurths. Hierarchical organization unveiled by functional connectivity in complex brain networks. *Physical Review Letters*, 97(23):238103, 2006.