2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

Marriage Between Coordinated and Uncoordinated Checkpointing for the Exascale Era

Omer Subasi, Ferad Zyulkyarov, Osman Unsal, Jesus Labarta
Barcelona Supercomputing Center, Universitat Politecnica de Catalunya
name.surname@bsc.es

***Abstract*—The state-of-the-art checkpointing techniques are projected to be prohibitively expensive in the Exascale era. These techniques are most often holistic in nature which prevents them to leverage programming model and paradigm specific advantages so as to be viable for the Exascale era. In this work, we present a unified non-hierarchical model to combine uncoordinated checkpointing with coordinated system-wide checkpointing to capitalize on programming model specific advantages. We develop closed-form formulas for performance improvement and the optimal checkpoint interval of the unified model in our analytical assessment. As an instantiation of our model, we propose to unify task-level checkpointing with a system-wide checkpointing scheme for task-parallel HPC applications. This instantiation has three distinct advantages: first it reduces performance overheads by decreasing the frequency of checkpoints in the unified system; second it features fast failure recovery by using in-memory task-local checkpoints instead of on-disk global checkpoints; and third it does not compromise from the high failure coverage typical of system-wide checkpointing.**

***Keywords*—Fault tolerance, Coordinated and Uncoordinated Checkpointing, Task-based dataflow programming, HPC and Exascale.**

I. Introduction

Recent High Performance Computing (HPC) Roadmaps agree that fault-tolerance will become indispensable in the Exascale Era [9] [15] and [16]. As HPC systems grow in capability and capacity, which are anticipated to have millions of cores and components, the state-of-the-art checkpoint/restart techniques will be prohibitively expensive. These techniques are either coordinated or uncoordinated with message logging accompanying checkpointing. While coordinated schemes suffer high computation overheads due to synchronization to achieve a consistent global state, uncoordinated schemes incur high amount of memory consumption due to message logging. Moreover, the coordinated techniques can cause high pressure on I/O such as Parallel File Systems (PFS) thereby creating congestion. Furthermore, these techniques are mostly holistic, i.e. system-wide, in nature. As a result, they do not take into account programming model and paradigm specific aspects to leverage so as to decrease the performance overheads to be viable for the Exascale era. In this work, we propose a unified approach to combine uncoordinated checkpoint/restart, or checkpointing, with coordinated checkpointing. Our objective is to decrease the system-wide checkpointing overheads and to obtain a faster recovery mechanism while not sacrificing the complete failure coverage provided by a system-wide scheme.

To achieve our objective, we unify task-level uncoordinated and system-wide coordinated checkpointing. Task-level checkpointing provides the following advantages when unified with a system-wide scheme: i) Since it mitigates a fraction of failures that was to be handled by system-wide checkpointing, the checkpoint frequency can be decreased. This way, the total number of system-wide checkpoints is decreased and as a consequence, the checkpointing overheads are reduced. ii) The failure recovery and restart become orders of magnitude faster with task-level checkpoints because of two reasons. First, task-level checkpoints are in-memory and fast as opposed to the global checkpoints that are stored on the disk if not on the PFS. Second, most often the task-level checkpoints are able to mitigate errors avoiding expensive global recovery. iii) With the unified approach, the scope of the failure becomes the task computation. As a result, task-level checkpointing provides fine-grain failure containment than system-wide checkpoints. Consequently, the amount of useful computation lost due to a failure becomes the computation since the beginning of a task rather then the beginning of a system-wide checkpoint interval. Because system-wide checkpoint intervals are much longer than a single task computation, the amount of useful computation lost is reduced significantly.

To materialize these observations, we provide a unified mathematical model to optimize the checkpoint interval in which the system-wide optimal checkpoint interval is increased. Our model quantifies this increase. Moreover, we analytically formulate the performance gain that is achieved by the unified approach. Figure 1 provides an overview of our analytical assessment and its implications. We see that as failure rate and task-level failure coverage increase, the performance gain increases.

Our main contribution is a unified approach that combines uncoordinated and coordinated checkpointing. *To the best of our knowledge, this is the first mathematical framework that unifies task-level uncoordinated checkpointing with system-wide coordinated checkpointing.* Briefly, our contributions are:

- Development, validation and evaluation of a unifying mathematical model for task-parallel HPC applications. We analytically derive a closed formula for the optimal checkpointing interval of the unified model.
- A closed formula to compute the performance gain when the task-level checkpointing is combined with a system-wide holistic checkpointing scheme. Simulations demonstrate that the performance gain can be as high as 306%. Our results for actual runs of real-world HPC benchmarks indicate that the performance gain is 282% of the total execution time on average and can be as high as 539%.

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.150

- A closed formula to compute the decrease in the checkpoint frequency of a system-wide checkpointing scheme. Our benchmark results show that the checkpoint frequency can be decreased 5.3× while not sacrificing complete failure coverage.

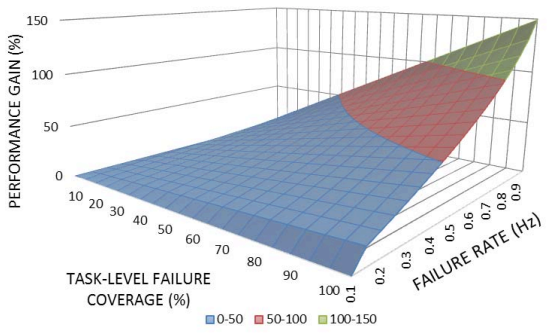

Fig. 1. Performance gain predicted by our analytical model as a function of task-level failure coverage and failure rate. As seen, increasing failure rate - expected for the Exascale era - is the dominating factor for the performance gain (improvement).

The rest of this paper is organized as follows: Section II presents background, related work and our motivation. Section III provides the formalization of our model. Section IV presents the formulation of performance gain. Section V presents the experimental evaluation. Finally, Section VI summarizes this work.

II. Background

In this section, we first provide background on checkpoint/restart and task-based parallelism (Section II-A). Then we present our motivation for this study (Section II-B). Finally, we discuss the related work (Section II-C).

A. Background

1) Coordinated and uncoordinated checkpointing: Checkpoint/restart, or checkpointing, is a well-known *rollback recovery* technique deployed to mitigate fail-stop errors [17]. In checkpointing the application state, called checkpoint, is saved and the application is restarted by using the checkpoint when a failure occurs. The most common types of checkpointing are coordinated and uncoordinated checkpointing for distributed applications. In coordinated checkpointing, processes coordinate with each other to reach a consistent global state before taking a checkpoint. The main drawback of coordinated checkpointing is the expensive coordination. Uncoordinated checkpointing was proposed to avoid the expensive coordination [17]. In uncoordinated checkpointing, the states of processes are saved independently, which avoids the coordination but supplementary application data is also logged such that a consistent global state could be formed in a restart. The logging of supplementary data is called message logging which is a technique used to prevent domino effect [17]. Domino effect is a condition in which, in the lack of message logging, the restart of a process causes other processes to restart; and in the end all processes have to restart losing all useful computation. Uncoordinated checkpointing has its own disadvantages. The log size, communication overhead, and fault-free execution overheads can be prohibitive.

To address these issues, hierarchical coordinated checkpointing schemes are proposed such as [11] and [18]. These schemes partition the application processes into groups in which each group can checkpoint independently. They limit the coordination among the processes in a single group and they perform costly global checkpoints less frequently than those within each group. However message logging is performed across the groups to be able for a global restart. Moreover the coordination among the processes belonging to the same group can still be expensive.

2) Task-based parallelism: The aforementioned studies are fit-for-all holistic approaches. Consequently, programming model or paradigm specific features cannot be incorporated to achieve affordable fault tolerance. Recently task-parallel data-driven programming is becoming mainstream in HPC. We see the adoption of data-driven task parallelism in Open4.0 [1] and Intel Building Blocks [2], and in programming models such as the ADF model [3], the Argobots [4] and OmpSs [6] since it has been successful in increasing programmer productivity and expressing parallelism. Coupled with data-driven execution model, this paradigm is shown to be on par or better in terms of performance than the conventional fork-join parallelism [10], [21] and [22]. Moreover, task-level parallelism is a good fit for distributed programming [25], [19] and [20].

In this paradigm, programs are parallelized by declaring and instantiating tasks. Tasks are pieces of code that can be executed on parallel resources. When an application is taskified, i.e. source code sections are defined as tasks, task inputs and outputs are used to express the data dependencies. Then the runtime executes tasks asynchronously based on solely data dependencies and out-of-order. There are well-established tools that not only automate the specification of tasks but also help the programmer to find the best parallelization strategy [23] and [24]. Moreover, we will see some properties of task-parallel programming paradigm that make fault tolerance for future HPC systems affordable in the next section.

B. Motivation

We propose combining uncoordinated task-level checkpointing with coordinated system-wide checkpointing. This combination is *non-hierarchical* since task-level and system-wide checkpointing are not coupled together such as multilevel systems from Bautista-Gomez et al. [11] and Sato et al. [18]. One cannot enforce and specify a periodic checkpoint interval at the task level. Due to the data-driven semantics, tasks are executed out-of-order and asynchronously where only data dependencies are obeyed.

There are several advantages of the integration of task-level checkpointing with system-wide checkpointing. First, since task-level checkpointing mitigates a fraction of failures, the checkpoint frequency of system-wide checkpointing can be decreased to improve the overall application performance. Second, the restart overhead of task-level checkpointing is significantly lower than that of system-wide checkpointing since task-level checkpointing only restores in-memory task inputs while system-wide checkpointing will likely need to

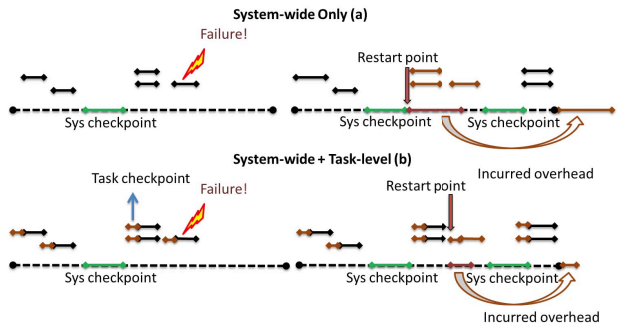

Fig. 2. System-wide vs. System-wide + Task-level Checkpointing: Short line segments show task computations. In the system-wide only case, in (a), the restart point is the last global checkpoint. In the combined scheme, in (b), the restart point is the beginning of the failed task computation. Note that in the system-wide only case, we lose all computation since the last global checkpoint whereas in the combined scheme, it is only the computation since the beginning of the failed task.

read a checkpoint from the parallel file system (PFS) or a local disk. Third, the combined model can be instantiated with application and system specific parameters, such as the system-wide checkpoint and restart times for an application, so that the optimal checkpoint interval is tuned for the application and the total waste is minimized particularly for the application.

Task-parallel dataflow programming has its own inherent merits that enable affordable fault tolerance for future HPC systems. First, task-level checkpointing offers fine-grain *failure containment*, that is, limits the scope of failures to task computations instead of the entire application. This effectively means that task-level checkpointing recovers to the point of failure much faster since tasks are usually much shorter than the system-wide checkpointing interval. As a result, the amount of lost work is smaller for task-level checkpointing compared to system-wide checkpointing. Second, the asynchronous nature of data-driven execution enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s). This means that task-level checkpointing is uncoordinated. Figure 2 visualizes these observations. Third, task inputs are available through annotations which represent the minimal state of the tasks needed to be checkpointed for the failure recovery. In contrast, holistic system-wide solutions typically checkpoint the entire address space of the application since they are not aware of the minimal state. Finally, the performance of dataflow parallelism is shown to be higher than the traditional fork-join parallelism due to amount of implicit and irregular parallelism achieved [10].

C. Related Work

There has been a significant body of work to model checkpointing systems in order to find the optimal checkpoint interval to minimize the total waste. Young and Daly study the sequential jobs [12] [13]. For parallel jobs, studies such as [26], [27] and [28] model the coordinated checkpointing protocols.

Considering hierarchical and partially coordinated checkpointing systems, the studies of Di et al. [29] and [30] investigate how to model and analytically determine the optimal checkpoint interval of each level in the hierarchy. In these studies Di et al. characterize the overheads of a multilevel scheme under analytically found checkpoint intervals. Balaprakash et al. [31] address the question of optimizing the checkpoint intervals while also considering the energy consumption of a multilevel scheme. As with the models for the coordinated schemes [26], [27] and [28], these hierarchical models cannot be used for non-hierarchical unified uncoordinated and coordinated checkpointing systems such as combined task-level and system-wide checkpointing. The unified model proposed by Bosilca et al. [32] models a range of checkpointing systems from fully coordinated scheme on one extreme to partially coordinated hierarchical schemes on the other extreme. However their model does not fit to the case of the non-hierarchical uncoordinated and coordinated unification since for the uncoordinated task-level checpointing, a periodic checkpointing interval cannot be imposed. The data-driven and task-parallel execution model prevents periodicity which in fact turns out to be an enabler for affordable overall fault tolerance. Moreover, their assumption of tightly-coupled applications is not needed for our unified model and is not the case for a task-parallel application. If one of the computing resources, such as a processor, fails, then a task can simply be rescheduled to another processor whereas a tightly-coupled application would have to recover and restart. As a result, *our mathematical framework is the first to study the non-hierarchical unification of the uncoordinated checkpointing with the coordinated checkpointing.*

III. Model Formalization and Analytical Assessment

We build a mathematical model to combine task-level checkpointing with a system-wide checkpointing scheme. If task-level checkpointing cannot handle the errors occurred, such as those that corrupt kernel, runtime and MPI library, then the system-wide scheme will recover from those errors. The model targets the detected errors that prevent the successful completion of the application execution, i.e. fail-stop errors. The undetected errors, called silent errors, are out of the scope of this study.

TABLE I. Model Parameters

Parameter	Description
τ_s	Checkpoint interval of system-wide checkpoints
c_s	Time to take a system-wide checkpoint
r_s	Time to restart with system-wide scheme
T_{ff}	Failure-free computation time without fault tolerance
μ_s	Expected rate for fail-stop errors
$T_{overall}$	Total execution time including fault-tolerance
$CovTL$	Failure coverage of task-level checkpointing from fail-stop errors
T_{TL}	Total amount of time used for task-level checkpointing
T_{sys}	Total amount of time used for system-wide checkpointing
T_{wasted}	Total amount of time used for task-level and system-wide scheme
W_{TL}	Total waste per unit of time of task-level checkpointing
W_{sys}	Total waste per unit of time of system-wide scheme
C	Fraction of time that an application is performing task computations

Table I shows the model parameters. We first introduce the total time equations:

$$\begin{aligned} T_{overall} &= T_{ff} + T_{wasted} \\ &= T_{ff} + T_{TL} + T_{sys} \quad (1) \\ &= T_{ff} + (W_{TL} + W_{sys})T_{overall} \quad (2) \end{aligned}$$

The first equation shows the overall execution time which includes the wasted time due to task-level checkpointing and the wasted time due to system-wide checkpointing. The second equation shows the breakdown of the overall execution time with respect to the total waste per unit of time which we will minimize. We note that the *failure coverage* $CovTL$ refers to the fraction of time that a task-level scheme is able to recover from fail-stop errors. Formally, it is defined as

$$CovTL = C \times e^{-\int_0^t \lambda(\theta)\, d\theta} \tag{3}$$

where C is a factor showing the fraction of time that a task-parallel application is performing task computations, that is, the execution time except the time consumed in runtime and MPI library, and in system calls. During the time in runtime and MPI library and in system calls, task-level checkpointing cannot mitigate errors which we need to factor out. In addition, $e^{-\int_0^t \lambda(\theta)\, d\theta}$ is the probability of the successful execution of a task-parallel computation from the beginning of the computation until time t with some failure distribution $\lambda(\theta)$ [34]. Our model does not assume any particular distribution for $\lambda(\theta)$.

We detail the waste per unit of time of system-wide checkpointing, i.e. W_{sys}, in Section III-A and then the waste per unit of time of task-level checkpointing, i.e. W_{TL}, in Section III-B. In Section III-C, we analytically find the optimal checkpoint intervals of the system-wide scheme and of the combined scheme. Finally, we conclude the model formalization with Section III-D where we discuss the generality of our model.

A. The waste time of a system-wide scheme

The waste time of a system-wide scheme, i.e. W_{sys}, is the sum of the checkpointing, rework and restart time. Checkpointing time captures the overhead of taking checkpoints. The rework time is the lost computation from the latest checkpoint to the moment of the failure. The restart time refers to the time to restore the latest checkpoint. We include down times, if any, in restart times since they do not require any special treatment. Now we formalize checkpointing, rework and restart time.

The checkpoint overhead of a system-wide scheme, denoted by W_{syschk}, is the product of the time to checkpoint c_s and the number of checkpoints which is $\frac{1}{\tau_s}$. Hence the checkpoint overhead is

$$W_{syschk} = \frac{c_s}{\tau_s} \tag{4}$$

When a failure occurs, on average, half of the computation is lost since the last checkpoint. Thus, the expected value of rework time of a system-wide scheme, denoted by W_{sysrew}, is

$$W_{sysrew} = \frac{\mu_s \tau_s}{2} \tag{5}$$

Restart time, denoted by W_{sysres}, is the product of the failure rate and the time to restore the checkpoint. Thus,

$$W_{sysres} = \mu_s r_s \tag{6}$$

Therefore the waste time of a system-wide scheme without task-level checkpointing is

$$W_{sys} = \frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + \mu_s r_s \tag{7}$$

B. The waste time of task-level scheme

The wasted time of a task-level scheme, W_{TL}, is constituted by the checkpoint overhead of all tasks, and the rework and restart overheads of the failed tasks during the program execution. Let $FailTs$ be the set of the failed tasks during the program execution due to fail-stop errors. In addition let α be a factor showing the fraction of the total wasted time of task-level checkpointing and recovery that is reflected in wall-clock time of the application execution. So for instance if $\alpha = 0$, then the application perfectly overlaps the computation with the task-level checkpointing and recovery and there is no overhead reflected in the wall-time of the application. On the other hand if $\alpha = 1$, then it means that task-level fault tolerance is sequential and fully reflected in the wall-time of the application. Note that $0 <= \alpha <= 1$.

The checkpoint time, W_{TL}^{chk}, is the overhead of taking task-local checkpoints. Hence the checkpoint time of task-level checkpointing is

$$W_{TL}^{chk} = \sum_{\forall i, Task_i} Task_i^{chk} \tag{8}$$

where $Task_i^{chk}$ is the checkpoint overhead of task $Task_i$.

The rework time for a single task is the computation lost since the beginning of the task. Thus the rework time, W_{TL}^{rew}, of task-level checkpointing is

$$W_{TL}^{rew} = \sum_{Task_i \in FailTs} \mu_s \times Task_i^{rew} \tag{9}$$

where $Task_i^{rew}$ is the lost computation time of the failed task $Task_i$ since the beginning of its computation.

The restart time, W_{TL}^{res}, is the overhead of restoring the task-local checkpoint. Thus the restart time of task-level checkpointing is

$$W_{TL}^{res} = \sum_{Task_i \in FailTs} \mu_s \times Task_i^{res} \tag{10}$$

where $Task_i^{res}$ is the restart time of task $Task_i$.

Finally the wasted time of a task-level checkpointing scheme is

$$W_{TL} = \alpha(W_{TL}^{chk} + W_{TL}^{rew} + W_{TL}^{res}) \tag{11}$$

In the next section we combine the two models.

C. The waste time of the combined model

If the task-level scheme provides the failure coverage $CovTL$, μ_s is decreased as

$$\mu_s' = (1 - CovTL) \times \mu_s \tag{12}$$

which leads us to parametrize the waste time of the system-wide scheme with respect to the expected failure rate.

Let $\mu_s' = (1 - CovTL) \times \mu_s$. We then denote the waste time of the system-wide scheme under failure coverage $CovTL$ with $W_{sys}(\mu_s', CovTL)$. As a result, note that $W_{sys}(\mu_s', 0)$ denotes the waste time of the system-wide scheme without

task-level checkpointing. The waste time of the system-wide scheme under failure coverage $CovTL$ is

$$W_{sys}(\mu'_s, CovTL) = \frac{c_s}{\tau_s} + \frac{\mu'_s \tau_s}{2} + \mu'_s r_s \quad (13)$$

Then the total waste per unit time of the combined scheme is

$$W = W_{TL} + W_{sys}(\mu'_s, CovTL) \quad (14)$$

We now compute the optimal checkpoint intervals of both the system-wide only and the unified scheme. To find the optimal interval of the system-wide only scheme, we take the derivative of Equation 7 with respect to τ_s to and equate it to zero to find the global minimum. We compute it as

$$\tau_s^* = \sqrt{\frac{2c_s}{\mu_s}} \quad (15)$$

which is similar to the Young's formula [12].

The waste time of the task-level scheme W_{TL} is independent from τ_s of the system-wide scheme. That is, the task-level overheads and wasted time is independent from the checkpoint interval of system-wide scheme. W_{TL} then can be treated as a constant - with respect to τ_s - such that the total waste time of the combined scheme in Equation 14 is still a convex function and thus we can find a global minimum. The first derivative of Equation 14 with respect to τ_s is given by

$$\frac{dW}{d\tau_s} = \frac{\mu'_s}{2} - \frac{c_s}{\tau_s^2} \quad (16)$$

where the derivative of W_{TL} with respect to τ_s is zero. Setting the derivative to zero, the solution is

$$\tau^*_{comb} = \sqrt{\frac{2c_s}{\mu'_s}} \quad (17)$$

Hence, if the task-level scheme has coverage $CovTL$, then τ_s^* is increased by a factor of $\gamma = \sqrt{\frac{1}{1-CovTL}}$, which we call the *gamma factor*, according to Equation 17. That is, the optimal solution of the combined model is

$$\tau^*_{comb} = \sqrt{\frac{1}{1-CovTL}}\sqrt{\frac{2c_s}{\mu_s}} = \gamma\tau_s^* \quad (18)$$

D. Discussion on the Generality of the Model

The unified model can be applied to any nonhierarchical scheme where there is an uncoordinated fault tolerance scheme, such as task-level checkpointing, that is used with a coordinated checkpointing scheme. The coordinated scheme can be any system-wide scheme where the uncoordinated scheme decreases the optimal checkpoint intervals in accordance to Equation 18. Moreover, this effectively also means that our model is orthogonal to studies done for coordinated or hybrid hierarchical systems such as Di et al. [29] and Bosilca et al. [32].

Our model enables application users tune the optimal checkpoint interval according to their applications. For instance, the system-wide checkpoint and restart times for application data differ from application to application, as evident in Table V from our results. They can tune the optimal checkpoint interval with respect to their applications' checkpoint and restart times. In addition, if their applications exhibit different failure rates as reported by Fang et al. [33], the optimal checkpoint interval can be adapted further.

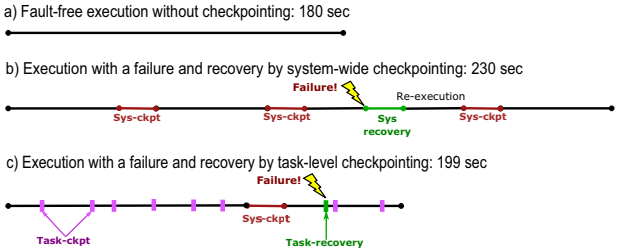

Fig. 3. Comparison of the Unified Model to the System-wide-only Model

IV. Performance Benefits of the Combined Model

The *performance gain* of the combined model can be computed by calculating the difference between the overhead of the system-wide only scheme and the overhead of the combined scheme. In the combined model, even though the task-level scheme incurs overhead due to its task-local checkpointing, the model increases the optimal checkpoint interval and thereby decreasing the number of system-wide checkpoints. This effectively reduces the checkpoint overhead of the system-wide scheme which is much higher than that of the task-level scheme. The *performance gain per unit of time* G_{ut} of the combined model is:

$$G_{ut} = W_{sys}(\mu'_s, 0) - W_{sys}(\mu'_s, CovTL) - W_{TL} \quad (19)$$

Further G_{ut} can be simplified as:

$$G_{ut} = (1 - \frac{1}{\gamma})\left(\frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + (1 + \frac{1}{\gamma})\mu_s r_s\right) - W_{TL} \quad (20)$$

As an example, assume $CovTL = 0.75$, $c_s = r_s = 10$ seconds, and $\mu_s = 1/180$ Hz (1 error every 3 minutes). The system-wide only optimal checkpoint interval, τ_s, is calculated to be 60 seconds. Since $\gamma = 2$, the optimal checkpoint interval for the combined model is 120 seconds. In addition, let the task-level waste time per unit time, W_{TL}, be 5% of the fault-free execution time. Then G_{ut} is calculated to be 11/120 seconds. If, say the fault-free execution is 180 seconds, with the combined model, the execution takes about 199 seconds, thus the overall gain is 19/120 $\times$ 199 = 31.5 seconds which is a significant gain. Figure 3 visualizes this example.

An important observation is that since $\mu'_s = (1-CovTL)\times \mu_s = 1/720$ Hz (a failure every 12 minutes instead of every 3 minutes), the system-wide scheme does not encounter any failure for which it has to recover when combined with the task-level scheme. This way, a system-wide recovery and restart is avoided which is much more expensive than a task-local recovery and restart.

V. Evaluation

We use NanoCheckpoints [8] as our task-level scheme and publicly available FTI [11] as our system-wide scheme for our experimental evaluation. However, our model is applicable to any task-level and system-wide checkpointing scheme.

We present three-fold experimentation. First, in Section V-A, we present the experimentation to evaluate the validity of our unified model. Second, in Section V-B, we assess in detail the implications of the validated model through simulations. Finally, in Section V-C, we evaluate the combined model for a set of task-parallel distributed applications. We will underline important takeaway lessons throughout the evaluation.

A. Model Validation

To validate our model, we perform failure simulation for which we wrote a simulator. The simulator generates failure sequences where failure arrivals follow Weilbull distribution. The shape parameter was 1 (exponential distribution) and the scale parameter was the number of tasks - for which we study a range - to have reasonable failure distributions in the simulations. Both the model and the simulator were fed with the same parameter values. We set parameter values to be on par with the predictions of studies [9] and [14]. We compare the total execution time predicted by our model to the total execution time of simulations in the presence of failures. We perform 10^4 simulations for each single case. Overall, the average difference between the total execution time predicted by the model and by the simulations is less than 0.24% of the execution time. The standard deviation was found to be 6.26 (over application execution time of 10^5 seconds) showing the consistency that the simulations and the model are predicting close values.

Moreover, we investigate the impact of different parameters on the accuracy of our model via simulations. Figure 4 a) shows the effect of the failure rate. The standard deviation is on the left axis and the average difference in percentage is on the right axis. We see that failure rate does not compromise the validity of the model. Likewise Figure 4 b) shows the effect of the system-wide checkpoint time to local storage on the accuracy of the model. We conclude that the system-wide checkpoint time does not interfere with the predictions of the model. Figure 4 c) shows the effect of the task granularity while Figure 4 d) shows the effect of the application execution time on the accuracy on the model. We see that these parameters do not affect the accuracy of our model.

The adaption and usage of our unified model is straightforward. First, given the failure rate of the system, we set the optimal checkpoint interval according to Equation 18 in the configuration file of FTI. Moreover given the failure distribution $\lambda(\theta)$, the runtime dynamically computes the task-level failure coverage as the application execution progresses. We can set the C factor in the Equation to be zero in the beginning. Recall that the C factor, defined in Section III, refers to the fraction of time that the application execution is in a task computation. Then as the application execution progresses, NanoCheckpoints measures and maintains on-the-fly the C factor at runtime. NanoCheckpoints' first update will set the C factor some value other than zero, for instance from 0 to 0.9. As the application progresses, if the C factor changes significantly, such as from 0.9 to 0.8, then NanoCheckpoints simply recomputes the optimal checkpoint interval and resets the value in the FTI configuration file.

TABLE II. BENCHMARK SET

Linpack	Matrix size 131072, block size 256
Heat	Resolution 32768, 2 heat sources, Jacobi
Matrix multiplication (mxm)	Matrix size 9216x9216
Nbody	65536 particles
Specfem3d	Seismic wave propagation, 8x8 grid

B. Analysis of Unified Scheme

In this section, we investigate the impact of different parameters on the performance gain that our model provides via simulations. Figure 5 a) shows the impact of system-wide checkpoint time on the performance improvement. We see that the performance improvement increases as the checkpoint time increases. The impact of system-wide restart time is similar to checkpoint time and omitted. For these simulations, the application execution time is fixed as 10^5 seconds and the failure of the system is fixed as 0.001 Hz. Figure 5 b) demonstrates that the effect of the failure rate with respect to total execution time. We see that as the failure rate increases from every 10^5 seconds to every 2 seconds, the performance gain increases and can be as high as 306%. Given that exascale error rates can be on the orders of minutes or seconds [14] [9], the unified model provides significant reduction in performance overheads as well as decreases the demand for I/O which is a main source of bottleneck. For these simulations, we fix the application execution time as 10^5 seconds and the checkpoint and restart time as 10 seconds. The impact of task-level failure coverage in Figure 5 c), i.e. the fraction of time that task-level checkpointing recovers a failure, is similar to the impact of the system-wide checkpoint time.

Figure 6 presents the amount of performance gain or improvement in seconds as the parameters change. From these simulations, we conclude that it is important to taskify a parallel application as much as possible not only for performance - as a result of parallelism - but also for reducing the checkpointing overheads. This is because the more an application is taskified, the more likely a failure is mitigated by task-level checkpointing. Overall the results show the merit of the unified model for the Exascale era.

Takeaway 1: *Analysis on the simulations points that as the system-wide checkpoint time and failure rates increase, which is expected for the Exascale era, so does the performance gain of the unified approach. This corroborates the adaption of the unified approach so that checkpointing can be a viable technique for the Exascale era.*

Takeaway 2: *It is important to taskify an application not only for the performance gained from parallelism but also for the performance gain from the reduced checkpointing overheads.*

C. Benchmark Evaluation Results

In this section we present the results for our benchmark set. We perform our experiments on Marenostrum III Supercomputer [5]. Table II summarizes our task-parallel distributed benchmarks [7]. In our experiments 1024 cores over 64 nodes are used. Our aim is to evaluate the unified model with real-world applications and assess the practical aspects of the model.

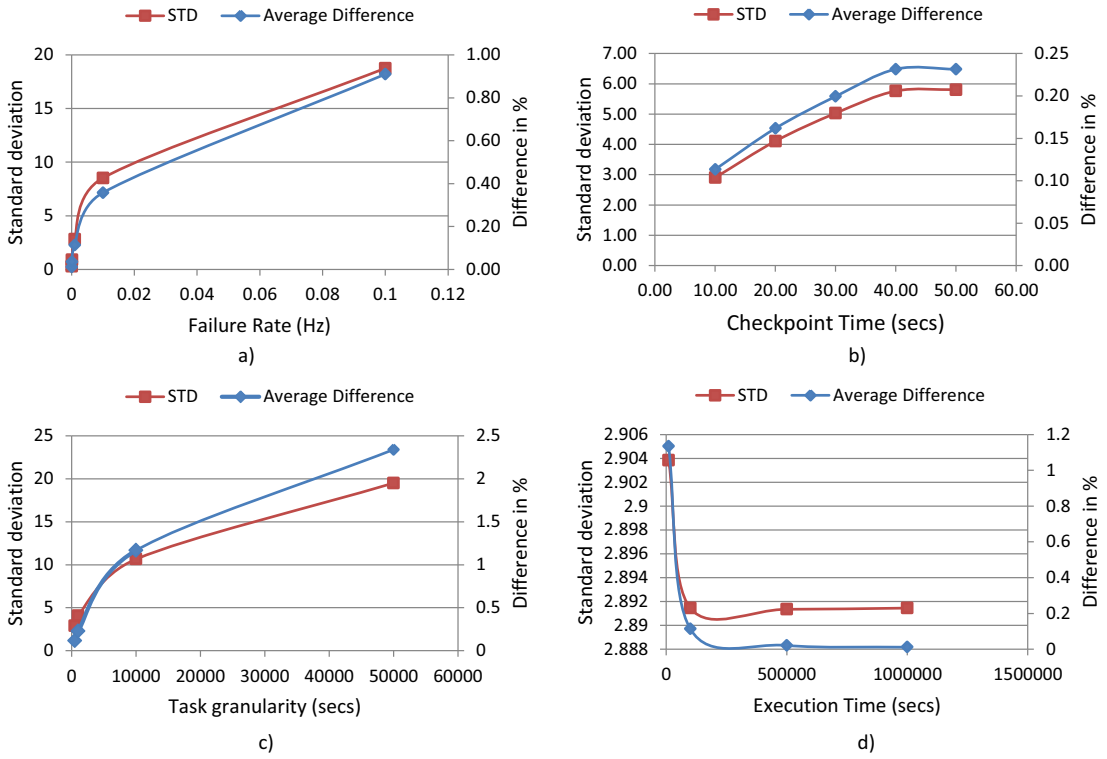

Fig. 4. Model validation results

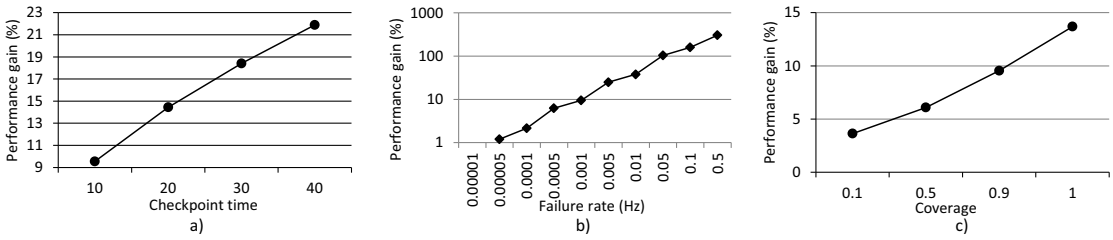

Fig. 5. Performance gain percentages

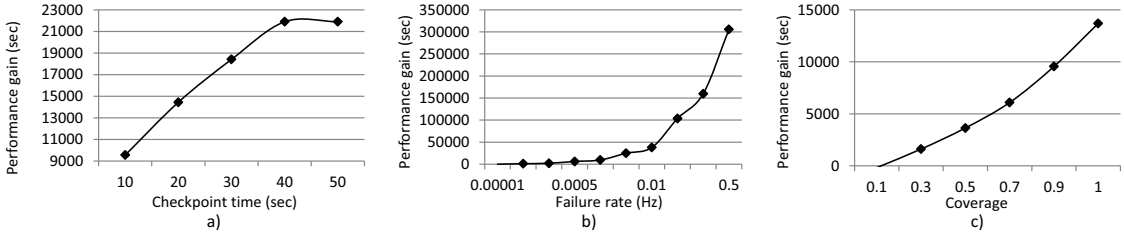

Fig. 6. Performance gain amounts in seconds

TABLE III. COVERAGE COEFFICIENTS: C FACTOR (%)

HPL	mxm	Heat	Nbody	Specfem3d
87.3	91.31	97	98.7	97

TABLE IV. WALLTIME OVERHEAD (%)

HPL	mxm	Heat	Nbody	Specfem3d
7.25	6.01	1.03	0.32	0.3

Table III shows the fraction of time that the application execution is in a task computation. This fraction corresponds to the C factor we defined in Section III. This is important since the higher this fraction is, the more likely that task-level checkpointing will mitigate the failures affecting the application. As seen, for all benchmarks the fraction is high.

Table IV presents the wall-clock time overheads of task-level checkpointing. For HPL and mxm, the overheads are relatively high due to communication across nodes. However, overall, all overheads are low and less than 8%. Note that in our model, these percentage overheads correspond to the W_{TL} values, total waste per unit of time of task-level checkpointing.

Figure 7 shows the performance gain of benchmarks when the combined model is used instead of system-wide scheme only up to 10 failures per minute (1/6 Hz.). First, we see that as the failure rates increase, so do the gains. Second, these gains are crucial given the expected error rates for Exascale and extreme scale HPC systems. Third, the difference in behavior of benchmarks is due to the system-wide checkpoint times for each benchmark. Table V shows the measured system-wide checkpoint (or restart since they are very close) and task-level checkpoint (restart) times. With Figure 7 and Table V together, we see that the higher the system-wide checkpoint time is, the higher the performance gain.

Another observation is that compared to the γ factor (in Figure 8), the factor by which we decrease the system-wide checkpoint frequency, the system-wide checkpoint time is the dominating factor in terms of the performance gain. Even though the values of the γ factor are higher in some benchmarks, such as Nbody, their system-wide checkpoint times are low enough such that their performance gain is less than the gain of other benchmarks having higher system-wide checkpoint times. Moreover, the γ factor values converge to 2 since task-level failure coverages converge to 0.75 as the failure rate converges to 1/6 Hz.

Takeaway 3: *These experiments show that the performance gain is 282% of the application execution time on average and can be as high as 539%. Additionally, the decrease in checkpoint frequency, i.e. the γ values, is 5.3× on average.*

Table V shows the average task restart time and the FTI restart time. From these results we conclude that our model can provide orders of magnitude faster restart if combined with system-wide checkpointing. Our model uses in-memory task-level checkpoints rather than system-wide FTI checkpoints resided on the local disk, if not on the PFS.

Finally, we present our results in which we measure the peak memory sizes that task-level checkpointing incurs in Table VI in a single node. We see that the memory sizes are reasonably low considering HPC systems' memory capacities.

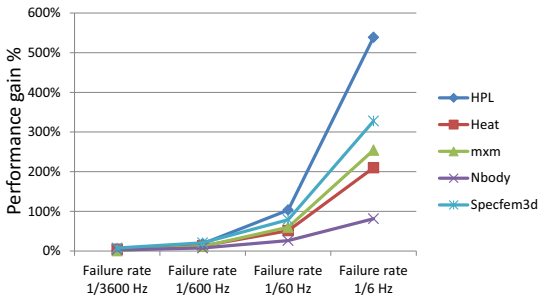

Fig. 7. Performance gain of our benchmarks

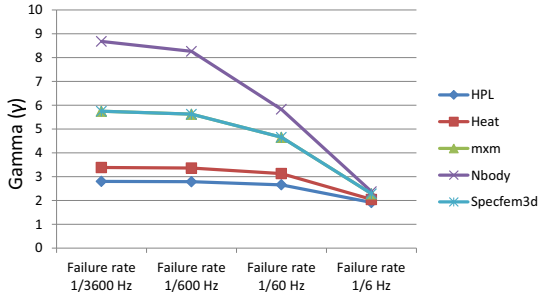

Fig. 8. γ factor values of our benchmarks

TABLE V. CHECKPOINT/RESTART TIME OF TASKS VS FTI

	Task restart time (sec)	FTI restart time (sec)
HPL	0.016	32.13
Heat	0.086	9.48
mxm	0.34	11.25
Nbody	0.08	2.28
Specfem3d	0.02	15

TABLE VI. NANOCHECKPOINTS' PEAK CHECKPOINT SIZE

	Peak checkpoint size (MB)
HPL	34
Heat	128.5
mxm	386
Nbody	0.6
Specfem3d	38

Takeaway 4: *We conclude that the memory overheads of task-level checkpoints are low and become relatively insignificant considering the potential performance improvement that can be achieved with the unified model, while coping with the high failure rates anticipated for the Exascale era.*

VI. CONCLUSION

In this work, we propose to unify task-level checkpointing with system-wide checkpointing for task-parallel HPC applications. We provide closed-form formulas for computing performance improvement and the optimal checkpoint interval of the unified model. Our simulation results show the validity of our model as well as the effects of parameters such as system failure rate and checkpoint time. Furthermore, results show that the unified model can decrease the overheads of system-wide checkpointing up to 539%.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-blanc 2 Project (www.montblanc-project.eu), grant agreement no 610402 and the FI-DGR 2013 AGAUR grant.

REFERENCES

[1] OpenMP Architecture Review Board. OpenMP application programming interface version 4.0, 2013.

[2] Intel Threading Building Blocks (TBB) version 4.3 update 2. http://www.threadingbuildingblocks.org/documentation. Accessed in April 2015.

[3] V. Gajinov, S. Stipic, O. Unsal, T. Harris, E. Ayguade, A. Cristal. Integrating Dataflow Abstractions into the Shared Memory Model. IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). pages 243-251. 24-26 Oct. 2012.

[4] Argonne National Laboratory, Argobots Home Website. https://collab.mcs.anl.gov/isplay/ARGOBOTS/Argobots+Home. Accessed in April 2015.

[5] Marenostrum III system architecture: http://www.bsc.es/marenostrum-support-services/mn3. Accessed in April 2015.

[6] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. Parallel Processing Letters, 2011, pages 173-193.

[7] BSC Application Repository. https://pm.bsc.es/projects/bar/wiki/Applications.

[8] O. Subasi, J. Arias, O. Unsal, J. Labarta, A. Cristal. NanoCheckpoints: A Task-Based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart. 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2015). pp. 99-102, 4-6 March 2015.

[9] Jack Dongarra et al. The International Exascale Software Project roadmap. Int. J. High Perform. Comput. Appl. 25, 1 (February 2011), 3-60.

[10] A. Amer, N. Maruyama, M. Pericas, K. Taura, R. Yokota, S. Matsuoka. Fork-Join and Data-Driven Execution Models on Multi-core Architectures: Case Study of the FMM. Springer Berlin Heidelberg. Supercomputing 2013, pages 255-266.

[11] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11).

[12] J. W. Young. 1974. A first order approximation to the optimum checkpoint interval. Commun. ACM 17, 9 (September 1974), 530-531.

[13] J. T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. Future Gener. Comput. Syst. 22, 3 (February 2006), 303-312.

[14] Saman Amarasinghe et al. ExaScale Software Study: Software Challenges in Extreme Scale Systems, 2009. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf. Accessed in July 2015.

[15] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, (2014). Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations, 1(1). http://www.mcs.anl.gov/publication/toward-exascale-resilience-2014-update-1.

[16] Secretary of Energy Advisory Board, U.S. Department of Energy. Report of the Task Force on Next Generation High Performance Computing 2014. http://energy.gov/sites/prod/files/2014/10/f18/SEAB HPC Task Force Final Report.pdf. Accessed in April 2015.

[17] E. N. Elnozahy, L. Alvisi, Y. Wang, D. B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (September 2002), pp. 375-408.

[18] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. Supinski, S. Matsuoka. 2012. Design and modeling of a non-blocking checkpointing system. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Article 19, 10 pages.

[19] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, J. Labarta. 2011. Productive cluster programming with OmpSs. In Proceedings of the 17th international conference on Parallel processing (Euro-Par'11), Springer-Verlag, pp. 555-566.

[20] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, J. Labarta. Productive Programming of GPU Clusters with OmpSs. IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS 2012). pp. 557-568. 21-25 May 2012.

[21] M. Andersch, C. Ching, B. Juurlink. Using OpenMP superscalar for parallelization of embedded and consumer applications. International Conference on Embedded Computer Systems (SAMOS 2012), pp. 23-32, 16-19 July 2012.

[22] M. Andersch, B. Juurlink, C. Chi. A Benchmark Suite for Evaluating Parallel Programming Models. Workshop on Parallel Systems and Algorithms (PARS), Gesellschaft fr Informatik, Volume 28, Ruschlikon, Zurich, pp. 7-17 (2011).

[23] V. Subotic, E. Ayguade, J. Labarta, M. Valero. Automatic Exploration of Potential Parallelism in Sequential Applications. Springer International Publishing, pages 156-171. Supercomputing 2014.

[24] V. Subotic, S. Brinkmann, V. Marjanovic, R. M. Badia, J. Gracia, C. Niethammer, E. Ayguade, J. Labarta, M. Valero. Programmability and portability for exascale: Top down programming methodology and tools with StarSs. In Journal of Computational Science, February 2013, volume 4, issue 6, November 2013, pages 450-456.

[25] V. Marjanovic, J. Labarta, E. Ayguade, M. Valero. 2010. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10). ACM, pages 5-16.

[26] H. Jin, Y. Chen, H. Zhu, X. Sun. 2010. Optimizing HPC Fault-Tolerant Environment: An Analytical Approach. In Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP '10). IEEE Computer Society, pages 525-534.

[27] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R.K. Iyer, L. Votta, C. Vick, A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. International Conference on Dependable Systems and Networks, 2005. DSN 2005 Proceedings. pp. 812-821. 28 June-1 July 2005.

[28] Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09. pp. 1-9. Aug. 31 2009-Sept. 4 2009.

[29] S. Di, M. Bouguerra, L. Bautista-Gomez, F. Cappello. 2014. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14). IEEE Computer Society, pages 1181-1190.

[30] S. Di, L. Bautista-Gomez, F. Cappello. 2014. Optimization of a multi-level checkpoint model with uncertain execution scales. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14). IEEE Press, pages 907-918.

[31] L. Bautista-Gomez, P. Balaprakash, M. Bouguerra, S. M. Wild, F. Cappello, P. D. Hovland: Energy-performance tradeoffs in multilevel checkpoint strategies. CLUSTER 2014. pages 278-279.

[32] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. Journal of Concurrency and Computation: Practice and Experience, 2013.

[33] B. Fang, K. Pattabiraman, M. Ripeanu, S., Gurumurthi. Evaluating the Error Resilience of Parallel Programs. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014). pp. 720-725. 23-26 June 2014.

[34] A. Verma, S. Ajit, and D. Karanki. Reliability and Safety Engineering. Springer, 2010.