Original Article

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–17
© The Author(s) 2016
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342016669416
hpc.sagepub.com
SAGE

Unified fault-tolerance framework for hybrid task-parallel message-passing applications

Omer Subasi[1,2], Tatiana Martsinkevich[3], Ferad Zyulkyarov[1], Osman Unsal[1], Jesus Labarta[1,2] and Franck Cappello[4]

Abstract

We present a unified fault-tolerance framework for task-parallel message-passing applications to mitigate transient errors. First, we propose a fault-tolerant message-logging protocol that only requires the restart of the task that experienced the error and transparently handles any message passing interface calls inside the task. In our experiments we demonstrate that our fault-tolerant solution has a reasonable overhead, with a maximum observed overhead of 4.5%. We also show that fine-grained parallelization is important for hiding the overheads related to the protocol as well as the recovery of tasks. Secondly, we develop a mathematical model to unify task-level checkpointing and our protocol with system-wide checkpointing in order to provide complete failure coverage. We provide closed formulas for the optimal checkpointing interval and the performance score of the unified scheme. Experimental results show that the performance improvement can be as high as 98% with the unified scheme.

Keywords

Fault-tolerance, message logging, checkpoint/restart, task-based programming model, optimal checkpointing interval

1. Introduction

As high-performance computing (HPC) systems continue to grow, so does their fault rate. Applications running on these systems must deal with rates on the order of hours or days; in the future the situation is not expected to improve (Cappello et al., 2014).

Amongst the variety of faults that HPC systems experience, in particular *transient* faults, such as, for example, multiple-bit upsets that cannot be corrected by ECC mechanisms, are expected to become more frequent due to the increased complexity and size of node memory (Snir et al., 2014). Whilst it is costly to correct multi-bit flips, they could be detected through practical Error Correcting Code (ECC) implementations. For example, most high performance DRAM memory units now provide so-called Single-Error Correction and Double-Error Detection (SEC-DED) ECC. Recent Intel Xeon chips even support Triple-Error Detection in memory lockstep mode. Those detected faults typically lead to *detected uncorrected errors* (DUE). Unfortunately, DUEs caused by transient faults usually raise a hardware exception which eventually causes an application's abort and global restart. This approach is not very efficient, and other solutions to deal with such situation are desirable for large-scale executions.

Meanwhile, the task-based parallel programming model (PM) is becoming a credible paradigm in HPC applications, and attention has been drawn to it for designing a fault tolerance solution that could gracefully handle the nondeterministic nature of such applications. In task-based parallel programming, an application code is divided into *tasks*, or *taskified.* These blocks of code can be executed in parallel to boost the performance. Commonly in task-based programming, tasks are synchronized at global program points; this is based on the fork-join execution model. Such global synchronization limits the level of the achieved parallelism due to unnecessary blocking of tasks whose computations may be independent of others (Amer et al., 2013). This style of synchronization hinders implicit or irregular parallelism. Hence,

[1] Barcelona Supercomputing Center, Spain
[2] Universitat Politecnica de Catalunya, Spain
[3] INRIA, University of Paris Sud, France
[4] Argonne National Laboratory, USA

Corresponding author:
Omer Subasi, Torre Girona, c/ Jordi Girona, 31, 08034, Barcelona Supercomputing Center Barcelona, Spain.
Email: omer.subasi@bsc.es

programming models based on dataflow semantics have been proposed (Duran et al., 2011; Gajinov et al., 2014; Stavrou et al., 2007). Among them, OmpSs PM (Duran et al., 2011) extends the OpenMP's tasking constructs by introducing data dependency clauses. Using them, the programmer describes task dependencies; the OmpSs runtime then transparently handles this information. Thus, OmpSs and similar models use a data-driven control flow rather than global synchronization barriers. Consequently, a higher level of parallelism can be achieved due to the asynchronous task execution. Moreover, such an execution model offers opportunities for designing a fault tolerance protocol that can profit from knowledge about data dependencies.

In order to execute on a large scale, task-based parallel models can be combined with distributed memory PMs, such as the message passing PM. The hybrid MPI+OmpSs PM is an example of such integration. However, to the best of our knowledge, no resiliency solution has been proposed yet that is designed specifically for such hybrid models. In this work we introduce such a solution. In particular, we extend an existing checkpointing protocol for pure OmpSs applications called NanoCheckpoints (Subasi et al., 2015a), with a message logging protocol adjusted to match the task-parallel execution model. The combined scheme is used to tolerate transient faults in the system and limits the consequences of such faults on the task that experienced them. It allows fast and asynchronous recovery that is more efficient than the conventional full application rollback-restart. Our protocol has a negligible fault-free execution overhead and is highly scalable. NanoCheckpoints is implemented in the OmpSs runtime, and our message-logging protocol is in the PMPI profiling layer. However, it is worth noting that the combined protocol is applicable to any hybrid task-parallel message passing programming model that has a dataflow execution model.

In addition, we develop a mathematical model that unifies our fault-tolerant protocol with system-wide checkpointing. There is no previous work that studies and proposes task-level checkpointing with message logging on top of system-wide checkpointing. That is, there is no previous work that demonstrates how to set the checkpointing period of the unified scheme and whether or how much performance improvement will be gained if the unified scheme is adopted.

Moreover, other than being motivated by the lack of previous research, this unification has significant benefits and implications in terms of fault-tolerance. First, since our fault-tolerant protocol mitigates a fraction of failures, the total number of expensive system-wide checkpoints is reduced. Therefore, overall the checkpointing overheads are decreased. Secondly, failure recovery is mostly faster with the unified scheme thanks to the in-memory task checkpoints and message logs. Thirdly, the unified model has a better failure containment; that is, the scope of failures becomes a single task rather than the complete application. Consequently, with the unified scheme the amount of lost computation is less than a system-wide only scheme, as in the unified scheme the amount of lost computation is the computation since the beginning of a task. This is typically much less than the amount of lost computation in a system-wide only scheme, which is the computation since the beginning of the checkpointing period. With the unified model these benefits can be gained without sacrificing the complete failure coverage of system-wide checkpointing.

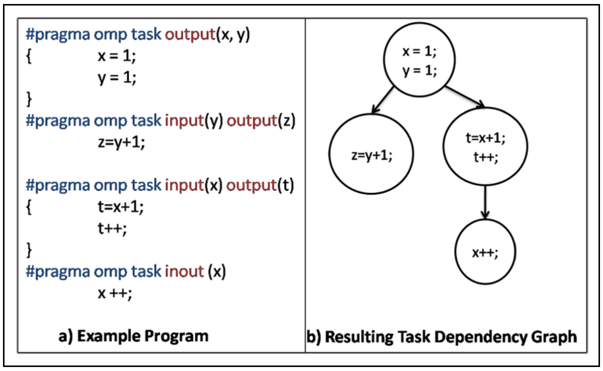
To the best of our knowledge, our model is the first that unifies task-level checkpointing and message logging with system-wide checkpointing. Moreover, we derive closed formulas for the optimal checkpointing interval and the performance score of the unified model. Results indicate that the performance gain (score) can be as high as 98% over system-wide-only checkpointing when the unified model is adopted. Our main contributions are as follows.

1. A scalable fault tolerance protocol for task-parallel message-passing applications for mitigating transient faults. The protocol has a reasonable fault-free overhead and transparently handles MPI calls inside tasks in recovery. To the best of our knowledge, this is the first resiliency solution for such a hybrid model.
2. A mathematical model that unifies our fault-tolerant protocol with system-wide checkpointing.
3. Closed formulas for the optimal checkpointing interval and the performance score of the unified model.
4. An extended evaluation of fault-free execution, execution with faults, model validation and the performance score of the benchmarks.

The rest of this paper is organized as follows. First we provide background information and related work. Then, we discuss our deterministic model and application requirements for the proposed protocol. Next, we move onto explaining the fault model. We then describe the design of the proposed protocol. After that we present the unified model formalization. In our experimental evaluation, we evaluate our protocol in the fault-free execution and execution with faults. In addition, we validate our unifying model and evaluate the performance score of our model. Finally we summarize our work and briefly discuss future research avenues.

2. Background

In this section we review the pure OmpSs PM and its runtime and elaborate on the hybrid OmpSs + MPI programming model.



```
#pragma omp task output(x, y)
{           x = 1;
            y = 1;
}
#pragma omp task input(y) output(z)
            z=y+1;

#pragma omp task input(x) output(t)
{           t=x+1;
            t++;
}
#pragma omp task inout (x)
            x ++;
```

Figure 1. Sample OmpSs code and its task dependency graph.

2.1. OmpSs and Nanos

OmpSs is a task-based parallel programming model derived from OpenMP. OmpSs applications utilize the source-to-source Mercurium[1] compiler and the Nanos runtime (Teruel et al., 2007).

OmpSs uses a thread-pool execution model: the master thread starts the execution and all other threads cooperate to complete the work it generates. OmpSs programs are parallelized by declaring and instantiating tasks — pieces of code that can be executed on parallel resources. The programmer specifies task inputs and outputs to express the data dependencies. Based on this information, Nanos dynamically generates a task dependency graph of the program.

A task is created in a waiting state. Once all its input dependencies are satisfied, it is ready and is executed by a thread from the thread pool as soon as a core is available. After it finishes, the dependency graph is updated so that its dependent tasks become ready for execution. Thus, the order of task executions follows the dependency graph but can be out of order from the point of view of the program. Figure 1 shows a sample program code and its task dependency graph generated at runtime.

As experiments have shown (Gajinov et al., 2014), the performance of OmpSs and Nanos is on a par with, or is superior to that of other programming models and their associated runtimes, because it exploits implicit and irregular parallelism. Because OmpSs is an extension of OpenMP, OmpSs compiler and runtime can handle almost all OpenMP programs. In other words, there is no need to convert OpenMP programs to OmpSs. OmpSs additionally introduced directionality annotations for task inputs and outputs. OpenMP now also supports these annotations starting from version 4.0.

2.2. OmpSs+MPI hybrid programming model

Similarly to the hybrid MPI+OpenMP model, OmpSs can be combined with MPI the standard system that defines syntax and semantics of a core of library routines for message passing programs to allow execution on distributed-memory systems.

```
i = n*my_rank; /*First C block*/
for( it = 0; it < nodes; it++ ) {
#pragma omp task input(A, B) inout(C)
  mxm((double *)A, (double *)B, (double *)&C[i][0]);

   if (it < nodes-1) {
#pragma omp task input(A) output(receivebuf)
      SendRecv((double *) A, (double *)receivebuf);
   }
   i = (i+n)%m;   /*Next C block*/
      ...
}
```

Figure 2. Sample MPI+OmpSs code.

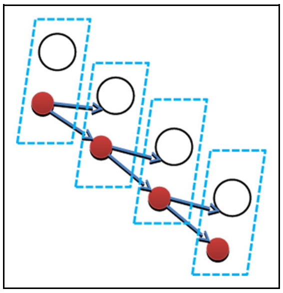


Figure 3. Communication/computation overlap between tasks.

In an MPI+OmpSs application, the programmer is encouraged to taskify parts of code that include communication calls. This action creates an opportunity to overlap communication and computation, as such tasks can run in parallel with other nondependent tasks. If a task includes an MPI call, the programmer must list the buffer used for the call among the task input or output parameters so that the runtime knows which tasks depend on communication.

Figure 2 is a simplified code example taken from the matrix multiplication application that we use as one of our benchmarks in this paper. Each process multiplies blocks of a matrix and exchanges results with other processes. Figure 3 schematically shows the overlapping of tasks inside one process. The red nodes represent SendRecv tasks with MPI communication calls inside, and the white nodes represent the mxm computation tasks. The parallelograms demonstrate the overlap of communication and computation tasks.

3. Related work

In this section we describe existing fault tolerance approaches for shared-memory applications as well as message-passing applications. We finally discuss state-of-the-art modeling checkpointing systems.

3.1. Fault tolerance techniques for shared-memory applications

Checkpoint-based rollback recovery is a common way to tolerate a fail-stop failure in which a process crashes.

Solutions proposed for shared-memory applications are usually on a system level (Prvulovic et al., 2002; Sorin et al., 2002) or compiler level (Bronevetsky et al., 2009). However, user-level checkpointing is still more common due to its lower cost and portability.

On the other hand, multicore architectures are convenient for redundant executions and detection of transient faults. Thread-level redundancy can be used to detect and recover from transient errors by replicating thread computations (Fu and Ding, 2010; Rashid and Huang, 2008). Similarly, task-based applications can replicate tasks for this purpose. For example, in Cao et al. (2015), authors use re-execution of a subgraph of the task-dependency graph, coupled with checkpointing of the task data, to recover from soft errors, or silent data corruption. In Tahan and Shawky (2012), triple modular redundancy with majority voting is used to in task-based OpenMP programs for the same purpose.

3.2. Fault tolerance techniques for message-passing applications

Checkpointing and message logging are two rollback-recovery techniques for mitigation of stop-fail failures that have been extensively studied in the past two decades (Treaster, 2005).

Checkpointing implies periodical saving of the application state reliably. If a failure occurs, all processes restart from their recent checkpoints. The two most common types are coordinated and uncoordinated checkpointing. In the first case, processes coordinate to reach a consistent global state before taking a checkpoint. User-level coordinated checkpointing is currently widely used in practice.

In uncoordinated checkpointing, the checkpoints are taken independently, thus avoiding expensive coordination. However, the application may end up in an inconsistent state after the restart, causing some processes to roll back to their previous checkpoints trying to reach a globally consistent state but, instead, invoking other processes also to roll back because of the communication dependencies. Eventually, the application may have to roll back to the beginning, losing all the work. This situation is called a *domino effect*.

Message logging protocols achieve consistency by logging communication-related data. In this case, only the failed process has to restart and all communication dependencies are resolved with the help of message logs.

Basic checkpointing and message logging algorithms often suffer from low scalability and large memory footprint. Considerable work has been done to address these issues (Riesen et al., 2012). To alleviate the high overhead of storing checkpoints to a parallel file system that impacts scalability, multi-level checkpointing has been proposed (Bautista–Gomez et al., 2011; Moody et al., 2010). Some researchers suggest considering the knowledge of correlation between hardware faults in order to lower the memory overhead of message logs (Bouteiller et al., 2011; Meneses et al., 2012); others combine coordinated checkpointing with message logging for the same purpose (Ropars et al., 2013). To the best of our knowledge, however, no prior work has tackled message logging for hybrid PMs.

Message logging is advantageous as it can improve failure containment, since it is often enough to restart a limited number of processes to recover from a fault. Other schemes has been also proposed to achieve a good failure containment. In Chung et al. (2012), the authors describe a concept of containment domains that employs semantics of nested transactions: the programmer defines containment domains by using a special API. If a fault is detected, the runtime either rolls back to the beginning of a current domain and re-executes it, or passes the error handler to a higher level containment domain in case the fault cannot be handled on this level. Similar to fault tolerance solutions for task-based PMs, in object-based programming models like Charm++, faults can be handled by migrating objects to healthy nodes and re-executing them there (Meneses et al., 2015).

3.3. Modeling in checkpointing systems

There has been a significant body of work that models checkpointing systems. The primary aim is to optimize the checkpoint period so that the total execution time is minimized. Young (1974) and Daly (2006) study sequential jobs, with Daly (2006) providing higher order estimates for the optimal checkpoint period. In addition, with higher order estimates Daly (2006) shows that restart time has no contribution on the optimal checkpoint period which was predicted by low order approximations. The studies of Jin et al. (2010), Wang et al. (2005) and Zheng and Lan (2009) focus on parallel jobs. Meneses et al. (2015) develop a performance model to study object-oriented parallel programming models.

Moreover, there is extensive research that studies hierarchical checkpointing systems such as Di et al. (2014b) and Di et al. (2014a). These studies investigate how to model and analytically determine the optimal checkpoint interval of each level in the hierarchy. In these studies Di et al. (2014b) and Di et al. (2014a) characterize the overhead of a multilevel scheme with analytically found checkpoint intervals. Bautista–Gomez et al. (2014) addresses the question of optimizing the checkpoint intervals while considering the energy consumption of a multilevel scheme.

However none of these models can be leveraged to address the unified scheme of task-level fault-tolerance with system-wide checkpointing. The unified model proposed by Bosilca et al. (2013) cannot model the uncoordinated task-level checkpoints and the asynchronous task-level message logs with system-wide checkpointing, because there is no hierarchical relation between task-level fault-tolerance and system-wide checkpointing. The model proposed by Subasi et al. (2015b) does not take into account message logging at the task level. Moreover Subasi et al. (2015b) define the failure coverage of task-level checkpoints theoretically which may not hold for all systems. Instead we experimentally find the failure coverage of task-level checkpoints by fault injection experiments. Considering the state of the art, our mathematical framework is the first to study the unification of task checkpointing and message logging with system-wide checkpointing.

4. Deterministic model and application requirements

The MPI+OmpSs hybrid model is a multithreaded dataflow-based execution model where any thread can execute communication calls at any time. Such an execution model implies a certain amount of nondeterminism as seen from the process's point of view.

Traditional models order events occurring during the execution using Lamport's *happened-before* relation (Lamport, 1978). This relation does not suit our execution model well, because the thread concurrency does not allow assumptions to be made about the order of events with respect to the process. For example, according to the MPI v3.0 standard[2], if two send operations are executed by two distinct threads concurrently where these two threads are running on the same node, no assumption can be made about the relative order of completion of the two operations.

However, the distinctive property of the task-based execution model is that it is *dataflow driven*. This means that we can partially order events using the knowledge about task dependencies. We use the notion of the *always-happens-before* relation defined in Ropars et al. (2013) to do so. Briefly, the always-happens-before relation orders the events not with respect to the program but with respect to different executions of the same program: if two events e_1 and e_2 are present in all executions of a program and e_1 *happened-before* e_2 in all executions, then we say that e_1 *always-happens-before* e_2, or $e_1 \xrightarrow{A} e_2$.

Given two tasks A and B, let e_A and e_B denote any event in task A and task B, respectively. If task B is reachable from task A in the dependency graph (i.e. a path in the graph connects A with B), these tasks will always be executed in succession and, therefore, $e_A \xrightarrow{A} e_B$. If task B is unreachable from task A, however, no assumption can be made about the order of events because these tasks can be executed concurrently. For example, in the dependency graph in Figure 4 task 5 can run concurrently with task 2 since there are no dependencies between them; but task 4 depends on tasks 2 and 3, so in any execution any event of tasks 2 and 3 will always precede any event of task 4.

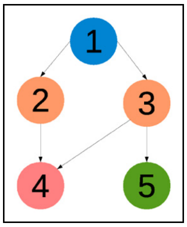


Figure 4. Example of a task dependency graph.

The always-happens-before relation is typical for *channel deterministic* (Ropars et al., 2013) applications. Channel determinism implies that the order of sends and receives in the application may change from execution to execution but, as long as the same set of messages is exchanged in any execution and the order of sends per channel is always the same, the application is channel deterministic. We cannot directly impose channel determinism on MPI+OmpSs hybrid applications because, as explained earlier, the thread concurrency introduces nondeterminism that may violate the requirement about the order of sends per channel.

However, because in this work we treat a task as the unit that may fail and be recovered, we will consider the deterministic model not from the point of view of the process but from the point of view of the task.

Therefore, as part of the proposed fault tolerance solution, we design a message logging protocol that can be applied to any MPI+OmpSs application in which all the tasks as stand-alone units are channel deterministic.

5. Fault model

The extended protocol for hybrid MPI+OmpSs applications proposed in this work is used to handle transient faults that can cause detected uncorrected errors (DUE), such as, for example, multiple bit-flips that cannot be corrected by ECC. Such errors usually cause the system to raise an exception with consequent application crash (or abort) and restart from the latest checkpoint, in case checkpointing is provided. However, we assume that the runtime can catch such an exception and, instead of aborting the application, it tries to take recovery measures.

If the fault happened inside a task, the runtime can restart it from the checkpointed state of that task and use message logging to recover MPI communication state. We note, however, that our protocol can protect only tasks. If a transient fault occurs during execution of nonprotected code, such as, nontaskified portion of the application code, execution of the runtime itself or during message logging, the application will abort, or conventional application rollback-restart will be triggered in case application-level checkpointing is available. Therefore, to maximize the benefit from the task recovery and avoid the whole application restart, it is better to taskify the application code as much as possible in order to reduce the ratio of protected:unprotected code.

6. Fault tolerance protocol for the hybrid MPI+OmpSs model

In this section we first summarize the existing task-level checkpointing scheme for pure OmpSs applications called NanoCheckpoints. We then discuss the additional steps required for fault recovery with the hybrid MPI + OmpSs model, followed by details of the message logging protocol that, combined with NanoCheckpoints, constitutes a fault tolerance solution for hybrid message-passing task-parallel applications.

6.1. NanoCheckpoints: Baseline task checkpoint/restart design

The baseline task-level checkpoint/restart, called *NanoCheckpoints*, targets pure OmpSs applications and is designed to mitigate transient faults that occur during the execution of a task. NanoCheckpoints is implemented in the Nanos runtime and it is completely transparent to the user.

Briefly, when a task is ready to execute, the runtime automatically saves the task inputs in the main memory. If a fault occurs during the task execution, it will restore the input parameters of the task and re-execute it. If after several trials the task keeps failing, the runtime aborts the application and, in that case, whole application rollback-restart has to be used. Task checkpoints in Nanos are not deleted upon task completion but are kept in memory in case this task will be executed again. Then, the runtime finds the previous checkpoint and overwrites it with the most recent input parameters in case they have been modified. The purpose is to avoid costly memory allocation and freeing. This approach has proven to significantly decrease the runtime overhead of checkpointing (Subasi et al., 2015a).

The rollback-recovery of a task is possible because of the notion that any data which constitutes the application's global state and that the task is going to modify must be declared as an input and, hence, will be checkpointed. It is the same if the task was working with a private copy of this data. Therefore, if a transient fault occurs in a task, only the task-local data is lost, but the process's global state, including the message logs, stays untouched.

We refer the reader to Subasi et al. (2015a) for further information about NanoCheckpoints.

6.2. Message logging protocol for MPI+OmpSs applications

To allow for a task to be recovered with an MPI call inside we use a message-logging protocol. We opted for receiver-base logging because we can benefit from the fact that with transient faults in tasks we lose only the task-local data, such as results of its computations, but the process state data, such as message logs, persists because the process itself keeps running.

Algorithm 1 presents a pseudo-code of our protocol. Incoming messages are logged in the main memory during the execution of a task and are garbage collected after the task has successfully completed. Logs are kept in the context of a task in order to prevent other tasks from matching a message from the log by mistake.

We differentiate messages by their *msg_id* defined as a tuple $\{rank, tag, comm\}$. Here, *rank* is the id of the process with whom this process communicates, *tag* is the MPI message tag, and *comm* is the communicator. For every incoming and outgoing *msg_id* we keep a sequence number *seqnum* which is incremented every time a communication involving this *msg_id* takes place. The pair $\{msg\_id, seqnum\}$ can uniquely identify every received message and can be used to restore the order of receives for each channel. We keep two values: *seqnum.current* for the current value and *seqnum.commited* for the latest value of *seqnum*. Both are incremented simultaneously in fault-free execution. If a task restarts, *seqnum.current* of all sequence numbers is reset to zero (Line 9).

For anonymous receive calls, i.e. calls with `MPI_ANY_SRC` or `MPI_ANY_TAG`, we additionally store a list *anonlist* of *msg_id*-s of messages that were received during the fault-free execution, so that in recovery we could receive them from log in the same order (Line 24).

In recovery, that is when $seqnum.current < seqnum.commited$ for current *msg_id*, in the case of a send call, the process simply increments *seqnum.current* and skips sending the message, as it had been already sent before the fault (Lines 29 to 31). In the case of a receive call, it copies the message with the corresponding *seqnum.current* value from the log to the receive buffer (Lines 15 to 17). If an anonymous receive call is executed in recovery, the protocol matches the next *msg_id* from *anonlist* (Line 11). Once it matches the last item of the list, the recovery of anonymous communication is considered finished.

There is a trade-off in using the receiver-based logging - it usually has a higher failure-free overhead than does the sender-based logging (Elnozahy and Zwaenepoel, 1994). On the other hand, receiver-based

Algorithm 1 Fault tolerance protocol for task A.

```
Upon starting task
    Checkpoint input parameters
    $in\_seqlist = out\_seqlist = anonlist = NULL$
Upon finishing task
    Delete all logs and seqnum lists.
Upon recovering task
    Restore input parameters
    for all seqnums in in_seqlist, out_seqlist do
        $seqnum.current = 0$ /*Reset current values for all seqnums*/
Upon receiving
    if Anonymous recv & Recovering anonymous communications then
        /*Get next msg to match in recovery*/
        $msg\_id \leftarrow next\_item(anonlist)$
     /* Get msg_id's seqnum. If not found, create new and add to in_seqlist*/
    $seqnum \leftarrow find(in\_seqlist, msg\_id)$
    if $seqnum.current < seqnum.commited$ then
        $seqnum.current \leftarrow seqnum.current + 1$
        $msg \leftarrow extract\_log(msg\_id, seqnum.current)$
        if Anonymous recv & Reached the last item in anonlist then
            Finished recovery of anonymous communications
    else
        $seqnum.current \leftarrow seqnum.current + 1$
        $seqnum.commited \leftarrow seqnum.current$
        $log(payload, msg\_id, seqnum.current)$
        if Anonymous recv then
            /*Log the order of anonymous receives*/
            $add\_item(anonlist, msg\_id)$
Upon sending
    $seqnum \leftarrow find(out\_seqlist, msg\_id)$
    if $seqnum.current < seqnum.commited$ then
        $seqnum.current \leftarrow seqnum.current + 1$
        Skip sending the message
    else
        $seqnum.current \leftarrow seqnum.current + 1$
        $seqnum.commited \leftarrow seqnum.current$
```

logging allows us to perform better in recovery, because the task will receive the message directly from the log and will not need to wait for it. Furthermore, it simplifies garbage collection: we can delete logs immediately after the task finishes. Sender-based message logging would require the sender to keep the log until the corresponding task on the receiver has completed and, hence, may cause large memory usage during runtime.

7. Unified model for task-level fault tolerance and system-wide checkpointing

We develop a mathematical model to combine task-level fault-tolerance (i.e. checkpointing and message logging) with a system-wide checkpointing scheme. If an error cannot be handled by task-level fault-tolerance, the whole application will have to roll back to the last system-wide checkpoint. Our model targets DUEs. It does not consider undetected errors and proposals for these errors are complementary.

Table 1. 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 DUEs
T_{overall}	Total execution time including fault-tolerance
$CovTL$	Failure coverage of task-level fault-tolerance from DUEs
T_{TL}	Total amount of time used for task-level fault-tolerance
T_{sys}	Total amount of time used for system-wide fault-tolerance
T_{overhead}	Total amount of time (overhead) used for fault-tolerance
W_{TL}	Total overhead per unit of time of task-level fault-tolerance
W_{sys}	Total overhead per unit of time of system-wide scheme

Table 1 shows the model parameters. We first introduce the total time equations

$$T_{\text{overall}} = T_{ff} + T_{\text{overhead}} = T_{ff} + T_{TL} + T_{\text{sys}} \qquad (1)$$

$$= T_{ff} + (W_{TL} + W_{\text{sys}})T_{\text{overall}} \qquad (2)$$

The first equation shows the overall execution time, which includes the overhead time due to task-level fault-tolerance and the overhead time due to system-wide checkpointing. The second equation shows the breakdown of the overall execution time with respect to the total overhead 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 DUEs. It is the percentage of time task-level fault-tolerance successfully recovers from failures. $CovTL$ is experimentally obtained for an application.

In the next section, we detail the overhead per unit of time of system-wide checkpointing, i.e. W_{sys}, and then the overhead per unit of time of task-level fault-tolerance, i.e. W_{TL}. After that, we analytically find the optimal checkpoint intervals of the system-wide scheme and of the unified scheme. Finally, we conclude the model formalization by defining the performance score for an application using the unified model.

7.1. The overhead of a system-wide scheme

The overhead time of a system-wide scheme is the sum of the checkpointing, rework and restart time. Checkpointing time is the overhead of taking checkpoints. The rework time is the lost computation from the last checkpoint to the moment of the failure. The restart time refers to the time to restore the latest checkpoint. We include down time, if any, in restart time since it does not require any special treatment. Now we formalize checkpointing, rework and restart overheads. Since we minimize the total overhead per unit of time, which is a rate, we will define rates for checkpointing, rework and restart time and work with these rates.

The checkpoint rate 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 rate is

$$W_{\text{syschk}} = \frac{c_s}{\tau_s} \tag{3}$$

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

$$W_{\text{sysrew}} = \frac{\mu_s \tau_s}{2} \tag{4}$$

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

$$W_{\text{sysres}} = \mu_s r_s \tag{5}$$

Therefore, the total overhead per unit of time (rate) of a system-wide scheme without task-level fault-tolerance is

$$W_{\text{sys}} = \frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + \mu_s r_s \tag{6}$$

7.2. The overhead of a task-level scheme

The overhead of a task-level scheme is constituted by the checkpoint overhead of all tasks, the rework and restart overheads of the failed tasks, and the message logging overheads of receiver tasks during the program execution. We note that our model assumes, like our protocol, a receiver-based message logging protocol. Let FTs be the set of the failed tasks during the program execution due to DUEs. Let RTs and STs be the set of the receiver and sender tasks of the program execution respectively (note that a task can be both a sender and a receiver task). In addition, let α be a factor showing the fraction of the total wasted time of task-level fault-tolerance 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 fault-tolerance 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 $0 <= \alpha <= 1$. α captures the parallelism in application executions and scales down the total overhead to the reflected overhead in the wall-clock time. We will now define rates for checkpoint, rework, restart and message-logging for our model.

The checkpoint overhead per unit of time, or rate, W_{TL}^{chk}, is the rate of the overhead of taking task-local checkpoints. Hence the checkpoint time of task-level fault-tolerance is

$$W_{TL}^{\text{chk}} = \sum_{\forall i,\, Tk_i} Tk_i^{\text{chk}} \tag{7}$$

where Tk_i^{chk} is the checkpoint rate of task Tk_i.

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

$$W_{TL}^{\text{rew}} = \mu_s(\sum_{Tk_i \in FTs} Tk_i^{\text{rew}} - \sum_{Tk_i \in STs \cap FTs} Tk_i^{\text{noSend}}) \tag{8}$$

where Tk_i^{rew} is the lost computation rate of the failed task Tk_i since the beginning of its computation. Tk_i^{noSend} is the rate saved because the recovering task does not need to re-send messages to non-recovering tasks.

The restart overhead per unit of time, W_{TL}^{res}, is the rate of overhead of restoring the task-local checkpoints. Thus the restart rate of task-level fault-tolerance is

$$W_{TL}^{\text{res}} = \sum_{Tk_i \in FTs} \mu_s \times Tk_i^{\text{res}} \tag{9}$$

where Tk_i^{res} is the restart rate of task Tk_i.

The message logging overhead, W_{TL}^{log}, is the sum of the message logging overhead of all receiver tasks. Hence the message logging rate is

$$W_{TL}^{\text{log}} = \sum_{Tk_i \in RTs} Tk_i^{\text{log}} \tag{10}$$

where Tk_i^{log} is the message logging rate of task Tk_i.

Finally the total overhead per unit of time (rate) of a task-level fault-tolerance scheme is

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

In the next section we combine the two models.

7.3. The overhead and the optimal checkpoint interval of the unified model

If task-level fault-tolerance has the failure coverage $CovTL$, then for system-wide checkpointing μ_s is decreased as

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

Let us denote the overhead per unit of time of system-wide checkpointing under failure coverage $CovTL$ with $W_{\text{sys}}(CovTL)$. As a result, note that $W_{\text{sys}}(0)$ denotes the overhead per unit time of system-wide checkpointing without task-level fault-tolerance. The overhead per unit time of system-wide checkpointing under failure coverage $CovTL$ is

$$W_{\text{sys}}(CovTL) = \frac{c_s}{\tau_s} + \frac{\mu_s' \tau_s}{2} + \mu_s' r_s \tag{13}$$

Then the total overhead per unit time of the unified model is

$$W = W_{TL} + W_{\text{sys}}(CovTL) \tag{14}$$

We now compute the optimal checkpoint intervals of both the system-wide only checkpointing scheme and the unified scheme. To find the optimal interval of the system-wide only scheme, we take the derivative of equation (6) with respect to τ_s and equate it to zero to find the global minimum. We compute it as

$$\tau_s^* = \sqrt{\frac{2c_s}{\mu_s}} \tag{15}$$

which is similar to Young's formula (Young, 1974).

The overhead per unit of 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 rates are independent from the checkpoint interval of the system-wide scheme. W_{TL} can therefore be treated as a constant - with respect to τ_s - such that the total overhead rate of the unified scheme in equation (14) is still a convex function and thus has a global minimum. The first derivative of equation (14) with respect to τ_s is given by

$$\frac{dW}{d\tau_s} = \frac{\mu_s'}{2} - \frac{c_s}{\tau_s^2} \tag{16}$$

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

$$\tau_{\text{unified}}^* = \sqrt{\frac{2c_s}{\mu_s'}} \tag{17}$$

Hence, if the task-level scheme has coverage $CovTL$, then τ_s^* is increased by a factor of $\sqrt{\frac{1}{1-CovTL}}$, according to equation (17). That is, the optimal solution of the unified model is

$$\tau_{\text{unified}}^* = \sqrt{\frac{1}{1-CovTL}}\sqrt{\frac{2c_s}{\mu_s}} \tag{18}$$

7.4. Performance score of the unified model

The performance score of the unified model can be intuitively understood as whether the unified model provides any performance gain over system-wide-only checkpointing. If the score is positive, then there is some performance gain, otherwise some additional performance overhead is incurred. The latter can happen when the system-wide checkpointing overhead is very low and the task-level fault-tolerance overhead is high. However, most often in the unified model, even though task-level fault-tolerance incurs an overhead, the unified model increases the optimal checkpoint interval and thereby decreases the number of system-wide checkpoints. This effectively reduces the checkpoint overhead of the system-wide-only checkpointing scheme, which is much higher than that of task-level fault-tolerance. This way the overhead of the unified scheme is reduced.

Formally, the performance score of the unified model can be computed by calculating the difference between the overhead of the system-wide only scheme and the overhead of the unified scheme. The *performance score S* of the unified model is

$$S = W_{\text{sys}}(0) - [W_{\text{sys}}(CovTL) + W_{TL}] \tag{19}$$

S can be further simplified as

$$S = (1 - \sqrt{1 - CovTL})\sqrt{2c_s\mu_s} + (CovTL)\mu_s r_s - W_{TL} \tag{20}$$

An important observation is that the performance score is increasing (linearly) with the failure rate. This is especially critical considering the expected increase in failure rates in the exascale era.

7.4.1. Energy considerations for the unified scheme.

Considering the issue of energy consumption of the unified scheme, firstly, since the checkpoint and performance overheads are minimized, and some performance improvement is achieved, the unified scheme becomes more energy efficient than system-wide-only schemes. Secondly, and more importantly, with our protocol the fault recovery is mostly confined to a single task within an application process. This means in case of a failure, with our protocol, it is most likely that only a task will recover. However in the absence of our protocol in system-wide-only schemes, a whole application will have to recover which is clearly much more energy consuming. Our protocol and task-level checkpoints provide better and fine-grain failure containment, which has the potential to make fault recovery more energy efficient.

Table 2. Benchmark parameters.

Matrix multiplication (mxm)	Matrix size 32,768
Nbody	524,288 particles
Himeno	1408 × 704 × 704

8. Evaluation

In this section, we first present the experimental setup and our benchmark set. We next discuss the scalability, performance, and memory overheads of the proposed protocol in fault-free execution and in the presence of faults. Finally we provide the experimental evaluation of the proposed unified model.

8.1. Experimental setup

We conducted experiments with three hybrid MPI+OmpSs benchmarks[3]: Himeno, which performs the incompressible fluid analysis; Nbody, which simulates a dynamical system of particles; and a matrix multiplication benchmark. All three benchmarks can be found in the core source code of many HPC applications.

The sizes of input parameters of the benchmarks are given in Table 2.

The experiments were carried out on the MareNostrum III supercomputer located in the Barcelona Supercomputing Center. Each of its total 3056 nodes has two Intel SandyBridge-EP E5-2670 processors with 8-cores at 2.6 GHz and 32 GB DDR3 memory (16 cores per node). InfiniBand FDR10 interconnect is used for the application communication and Gigabit Ethernet for GPFS. The nodes run Linux SuSe Distribution. We used OpenMPI 1.6.4 in combination with our PMPI wrapper library.

All the tests were run with 64 MPI processes, one process per node and 16 threads per process. For each configuration we execute the application three times. All the overheads are computed against the pure execution time without our protocol.

8.2. Fault-free execution

First, Figures 5 and 6 show the impact of our protocol on the original strong and weak scalability efficiency of the applications. We measured the scalability efficiency in the following manner. If t_1 is the time to execute a unit of work on one process and t_N is the time to execute the same amount of work on N processes, then the strong-scalability efficiency is computed as $S_{\text{strong}} = \frac{t_1}{Nt_N} * 100\%$.

Respectively, if t_1 is the time to execute a unit of work on one process and t_N is the time to execute N units of work on N processes, then the weak scalability efficiency is computed as $S_{\text{weak}} = \frac{t_1}{t_N} * 100\%$.

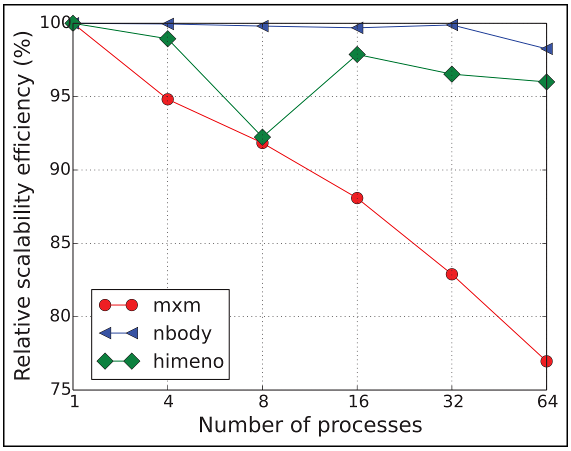


Figure 5. Relative strong scalability of the fault tolerance execution compared with nonresilient execution.

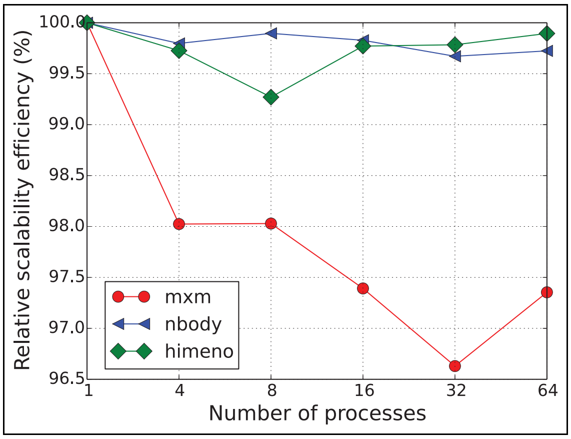


Figure 6. Relative weak scalability of the fault tolerance execution compared with nonresilient execution.

The tests show that checkpointing and message logging may decrease the scalability to different extents depending on the application. The impact for the matrix multiplication benchmark was the biggest in both cases. Its strong scalability with 64 processes dropped by 24%, because at larger scale the input data size per process was so small that the checkpointing and message logging overheads became more noticeable and difficult to hide by task overlapping. This was not the case in the weak scalability test and so the scalability dropped by only 2.7% at a respective scale. The scalability of Nbody and Himeno did not change significantly.

Next, the runtime overhead of using our protocol measured at 4.45% for the matrix multiplication application, 0.31% for Nbody and 0.89% for Himeno. Note that these overheads correspond to W_{TL} in equation (11) in our model. To understand these results, one needs to look at the memory overhead of the protocol presented in Table 3.

Table 3. Memory overhead.

	Checkpoint size (MB)	Total message log (MB)	Peak message log (MB)
mxm	512	8064	128
Nbody	0.59	0.75	0.25
Himeno	1202	448	1.20

Column one of Table 3 shows the total memory size that was occupied by NanoCheckpoints: 512 MB for the matrix multiplication, around 1 GB for Himeno, and less than 1 MB for Nbody. The size of a checkpoint depends on how the program was taskified. For instance, Nbody has a small checkpoint size because every process works only on the portion of particles that it is responsible for. Conversely, in Himeno, additional arrays hold the data, and whole arrays are passed to the tasks; therefore, more data is checkpointed.

The total amount of logged messages are presented in column two: 448 MB for Himeno, 7.8 GB for the matrix multiplication benchmark and less than 1 MB for Nbody. We note that Nbody is a computation-intensive application in which communication occurs only when the processes exchange particle data at the end of a step; therefore, the message log is small for this application. We tried to increase the input number of particles for the Nbody application to determine whether the checkpointing and message logging overheads would noticeably rise, but they did not grow significantly compared with the greatly increased compute time. Therefore, we decided to leave the current input parameter size.

Analyzing the numbers, we concluded that message logging was the main contributor to the execution overhead. In particular, the results for the matrix multiplication application show a lot of communication, which is clear from the total size of all logged messages.

Next, we point out that the memory occupied by the protocol at any given moment is much smaller since message logs are kept only until the corresponding task is finished. Column three in Table 3 shows the peak message log size per process observed during the whole execution in all tests. It is clear that the peak log size is much smaller than the total message log size.

Finally, despite the small scale of our experiments we think that the proposed scheme should scale well to much larger executions, due to the fact that (i) our approach is completely distributed and does not require synchronization neither between processes nor between threads; (ii) most application algorithms and MPI implementations are designed in such a way that no matter what is the scale of execution, the degree of connectivity of each process in the communication graph stays approximately the same. In other words, as the number of processes grows, the number of communication channels for which each process will apply the logging protocol will likely to stay the same. Therefore, as long as the scale and the input problem size are increased in a balanced way, we expect the overhead of using our approach to stay reasonably small. This is supported by the results of the weak scalability tests in Figure 6 where we observed a relatively small impact of the protocol on the scalability.

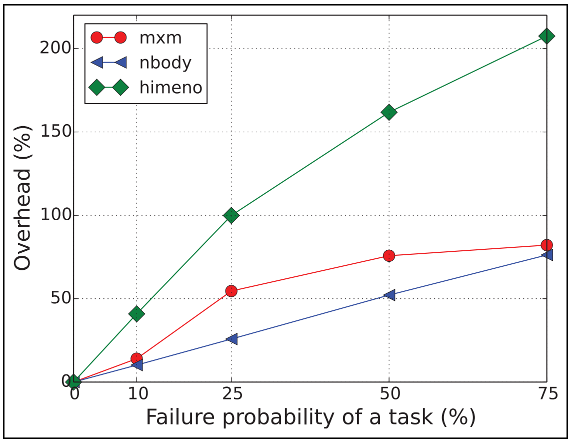


Figure 7. Runtime overhead in the presence of faults.

8.3. Execution with faults

To evaluate how much time it would take for an application to finish when transient faults are present, we simulate faults in the following manner. We first set a parameter for probability of a task failure. At the end of a task, a random number between 0 and 100 is generated. If the number is less than the probability of a failure, we consider that a fault has occurred, restore the task input parameters, and re-execute it. For example, if the probability is set to 50%, we expect approximately half the tasks to re-execute.

Because we simulate faults at the end of the task, we imagine the worst-case scenario and measure the upper bound for the execution time with failures.

Figure 7 shows the benchmark execution overhead for different failure probabilities. Additionally, Table 4 gives a general idea about the total number of tasks executed by each process and the relation between the probability of a task failure and the actual number of failed tasks and corresponding fault rates.

Nbody has a linear dependency between the number of task re-executions and the program complete time. This is because Nbody is tightly coupled: In each step, processes must exchange particle information; therefore, if one task from the step has to re-execute, all the other processes that have already finished computing for this step will have to wait for the delayed process.

In the matrix multiplication benchmark, recovering a quarter of the total number of tasks delays the

Table 4. Task statistics with corresponding fault rates (faults/second).

Himeno	Total 600 tasks				Nbody	Total 383 Tasks				mxm	Total 126 Tasks			
Fault probability	10	25	50	75	Fault probability	10	25	50	75	Fault probability	10	25	50	75
Avg. recovered tasks	60	151	298	449	Avg. recovered tasks	43	104	193	287	Avg. recovered tasks	12	32	63	93
Fault rate	1.59	2.82	4.24	5.44	Fault rate	0.30	0.64	0.99	1.27	Fault rate	0.16	0.31	0.53	0.76

execution by 50%. Studying the source code of the program suggested that this benchmark does not have fine task granularity. Coarse task granularity along with tight data dependency results in worse parallelization. Therefore, less overlapping with task recovery is possible since dependent tasks can not be executed until the current one recovers and finishes. On the other hand, when there are more independent tasks, they can be executed while the failed task is recovering to fill in the gaps and hide the time lost on task recovery.

Similar effects of the coarse granularity and task dependency were observed in the tests for Himeno: recovering about half of the tasks delayed the execution by 1.5 times. After examining the source code we noticed that whole arrays of data are passed in virtually all taskifying pragmas in Himeno. Hence, all tasks become tightly dependent on each other; and if one task has to re-execute, no other tasks will be able to make other threads busy. Everything will have to wait for the recovery of this one task.

8.4. Unified model experiments

In this section, we first present experimental results regarding the validation of our model. Then we study the failure coverage of task-level fault-tolerance which is needed for the calculation of the performance score of the unified scheme. Finally, we present the performance score experiments of our model.

8.4.1. Model validation. We validate our model through Monte Carlo simulations. To perform simulations, we implement a simulator that generates failure sequences where failure arrivals follow a Weibull distribution. In our experiments, even though we study different Weibull distributions, we report realistic scenarios where the shape parameter is 1 (exponential distribution) and the scale parameter is the number of tasks to have reasonable failure distributions in the simulations. We feed the model and the simulator with the same parameter values. We set parameter values to be on par with the predictions of studies Dongarra et al. (2011) and Amarasinghe et al. (2009). We compare the total execution time predicted by our model to the total execution time of simulations in the presence of failures. We perform 10,000 simulations for each experiment.

Figure 8 shows the model compared with the simulations under different parameters. On the left, the x-axis shows the average absolute difference in total execution time as the system-wide checkpointing time (y-axis) changes. As checkpointing time increases, the divergence between the model and the simulation increases. However, the deviation is very low and less than 0.7%. The relation between checkpointing time and the deviation can be due to the tacit assumption that no failures are expected to occur during the checkpointing itself

Table 5. Task-level failure coverages of benchmarks.

	Task-level failure coverage (%)
mxm	98%
Nbody	73%
Himeno	86%

Table 6. FTI checkpointing time of benchmarks.

	System-wide checkpointing time (seconds)
mxm	1.92
Nbody	40.44
Himeno	45.79

however, in simulations this assumption may - though rarely - not hold.

On the right, again the x-axis shows the average absolute difference in total execution time and y-axis shows the task-level failure coverage. As the task-level coverage increases, the deviation between the model and the simulations decreases. This is because the task-level coverage reduces the failure rate of the system which is positively correlated with the deviation between the model and the simulation. This positive correlation stems from the fact that the approximations in the model, such as equation (4), become less accurate as the failure rate increases. From the figure, we can see that the deviation is always less than 0.35%. We omit the effects of other parameters - which are similar - such as restart time for brevity.

Overall, validation experiments demonstrate that our model accurately reflects the real-world behavior of the unified checkpointing system.

8.4.2. Failure coverage of task-level fault-tolerance. To see the failure coverage of task-level fault-tolerance we have performed fault injection experiments. In these experiments each workload was executed 10,000 times with task checkpoints enabled. During each workload execution only one fault is injected into the dataset of the application. The fault injection occurs randomly both in time and space. The moment when a fault is injected is a random event with exponential distribution. The memory where a fault is injected is a random event with uniform distribution. Depending on how the fault is injected, a workload may crash (segmentation fault), complete with a wrong result, or complete with a correct result. In these experiments we compare how many times the executions of a workload complete successfully. The more successfully completed executions the better coverage.

Table 5 shows the failure coverage of task-level fault-tolerance for each benchmark; that is, the percentage of time task-level fault-tolerance successfully recovered from the injected faults. The failure coverage of task-level fault-tolerance is high for all benchmarks showing its effectiveness.

8.4.3. Performance score experiments. In order to assess the performance score of the unified model for the benchmarks under different failure rates, the system-wide checkpointing times are needed for equation (20). We use FTI (Bautista–Gomez et al., 2011) checkpointing library as the system-wide checkpointing scheme to measure checkpointing times. We perform experiments to obtain the FTI checkpointing time of benchmarks. We used the following parameters for FTI library during our experiments. We used four checkpointing levels. The checkpointing intervals were 1, 2, 4 and 8 minutes for each level respectively. We did not use any dedicated core for checkpointing. The group size was 4 and the block size was 1024. Other parameters were set at their default values.

Table 6 shows the system-wide checkpointing times of benchmarks per process when running with a total of 64 processes.

Figure 9 shows the performance score of the unified model (over system-wide-only checkpointing) for the benchmarks under different failure rates. We see that Nbody and Himeno almost always have performance gain with the unified model and matrix multiplication has performance gain after the failure rate is 10^{-3} Hz (1 failure every 1000 seconds). Only matrix multiplication has a relatively low system-wide checkpointing time. As discussed before, in matrix multiplication the per process input data size is very small which leads to a low system-wide checkpointing time.

Another important observation is that as the failure rate increases, so does the performance score of the unified model. This is especially important in the exascale era, since failure rates are expected to increase dramatically.

9. Conclusions and future work

In this paper we proposed a unified framework that provides fault-tolerance for hybrid task-parallel message-passing HPC applications. Our framework can be adopted for any dataflow task-based message-passing programming model. With our framework this study had two main contributions.

We first introduced a message logging protocol that, combined with task checkpointing, provides a resiliency solution for mitigating DUEs in task-parallel message-passing applications. The protocol allows the avoidance of costly application rollback-recovery by asynchronously restarting only tasks in which a fault has occurred, and transparently resolves recovery of tasks

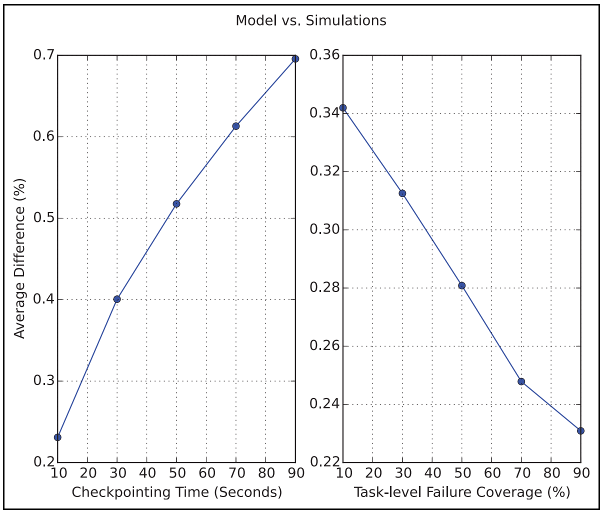


Figure 8. Model validation results.

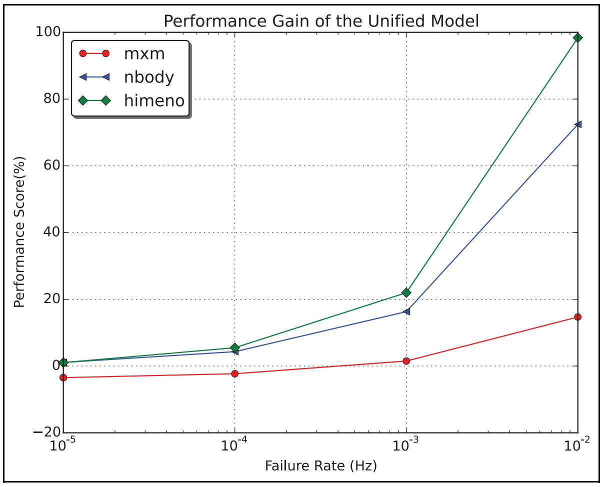


Figure 9. Performance score results.

that have MPI calls inside, thanks to the message logging.

Evaluation of the execution in the presence of faults showed that task granularity and coupling play an important role in hiding task recovery. The higher the number of tasks that can be executed independently while some other task is recovering from a fault, the less the impact faults will have on the total execution time. If the program was not taskified well, however, recovery of even one task may slow the program significantly.

Secondly, we introduced a unified model that combines task-level fault-tolerance with the system-wide checkpointing. We analytically derived the optimal checkpointing interval (equation (18)) and the performance score (equation (20)) of the unified model. Moreover, our results show that with the unified scheme, the performance gain can be as high as 98% over system-wide-only checkpointing.

Our future work will investigate whether the task checkpointing mechanism can make use of the message logs. Since the received messages are often used as inputs for other tasks, potentially we can avoid checkpointing this data and decrease the protocol overhead.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the FI-DGR 2013 scholarship and 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 TIN2015-65316-P.

Notes

1. Available at: http://pm.bsc.es/mcxx
2. MPI: A message passing interface standard, version 3.0, 2012. Available at: http://www.mpiforum.org/docs/mpi-3.0/mpi30-report.pdf
3. Application repository: https://pm.bsc.es/projects/bar/wiki/Applications

References

Amarasinghe S, Campbell D, and Carlson W. (2009) Exa-Scale software study: Software challenges in extreme scale systems. Available at: http://users.ece.gatech.edu/mrich ard/ExascaleComputing StudyReports/ECSS%20report %20101909.pdf (accessed 1 July 2015).

Amer A, Maruyama N, Pericas M, et al. (2013) Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM. In: *Proceedings of the 28th international supercomputing conference*, Leipzig, Germany, 16–20 June 2013, pp.255–266. Berlin, Heidelberg: Springer.

Bautista–Gomez L, Balaprakash P, Bouguerra M, et al. (2014) Hovland: Energy-performance tradeoffs in multilevel checkpoint strategies. CLUSTER 2014. pp.278-279. Poster.

Bautista–Gomez L, Tsuboi S, Komatitsch D, et al. (2011) FTI: High performance fault tolerance interface for hybrid systems. In: *Proceedings of the 2011 international conference for high performance computing, networking, storage and analysis*, Seattle, Washington, 2011, pp.32:1–32:32. New York, NY: ACM.

Bosilca G, Bouteiller A, Brunet E, et al. (2013) Unified model for assessing checkpointing protocols at extreme-scale. *Journal of Concurrency and Computation: Practice and Experience* 26: 2772–2791.

Bouteiller A, Herault T, Bosilca G, et al. (2011) Correlated set coordination in fault tolerant message logging protocols. In: *Proceedings of the 17th international conference on parallel processing*, vol. 2, Bordeaux, France, 29 August–2 September 2011, pp.51–64. Berlin, Heidelberg: Springer.

Bronevetsky G, Marques D, Pingali K, et al. (2009) Compiler-enhanced incremental checkpointing for openmp applications. In: *IEEE international symposium on parallel distributed processing*, Salt Lake City, UT, May 2009, pp. 275–276. New York, NY: ACM.

Cao C, Herault T, Bosilca G, et al. (2015) Design for a soft error resilient dynamic task-based runtime. In: *29th IEEE International parallel and distributed processing symposium*, Hyderabad, India, 2015, pp.765–774. Washington, DC: IEEE.

Cappello F, Geist A, Gropp W, et al. (2014) Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations* 1(1): 5–28.

Chung J, Lee I, Sullivan M, et al. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis*, Salt Lake City, Utah, pp.58:1–58:11. Los Alamitos, CA. IEEE Computer Society Press.

Daly JT (2006) A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems* 22(3): 303–312.

Di S, Bautista–Gomez L and Cappello F (2014a) Optimization of a multilevel checkpoint model with uncertain execution scales. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*, New Orleans, Louisana, pp.907–918. Piscataway, NJ: IEEE Press.

Di S, Bouguerra M, Bautista–Gomez L, et al. (2014b) Optimization of multi-level checkpoint model for large scale HPC applications. In: *Proceedings of the 2014 IEEE 28th international parallel and distributed processing symposium*, PHOENIX (Arizona), USA, pp.1181–1190. Washington, DC: IEEE Computer Society.

Dongarra J, Beckman P, Moore T, et al. (2011) The international exascale software project roadmap. *The International Journal of High Performance Computing Applications* 25(1): 3–60.

Duran A, Ayguade E, Badia RM, et al. (2011) OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21(2): 173–193.

Elnozahy EN and Zwaenepoel W (1994) On the use and implementation of message logging. In: *24th international symposium on fault-tolerant computing*, 1994, pp.298–307. IEEE.

Fu H and Ding Y (2010) Using redundant threads for fault tolerance of OpenMP programs. In: *2010 International Conference on Information Science and Applications ICISA*, pp.1–8. Seoul, Korea: IEEE.

Gajinov V, Stipić S, Erić I, et al. (2014) Dash: A benchmark suite for hybrid dataflow and shared memory programming models with comparative evaluation of three hybrid dataflow models. In: *Proceedings of the 11th ACM conference on computing frontiers*, Cagliari, Italy, 2014. pp.4:1–4:11. New York, NY: ACM.

Jin H, Chen Y, Zhu H, et al. (2010) Optimizing HPC fault-tolerant environment: An analytical approach. In: *Proceedings of the 2010 39th international conference on parallel processing*, 13–16 September 2010, pp.525–534. San Diego, CA: IEEE Computer Society.

Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7): 558–565.

Meneses E, Ni X and Kale LV (2012) A message-logging protocol for multicore systems. In: *Proceedings of the 2nd workshop on fault-tolerance for HPC at extreme scale*, Boston, MA, June 2012.

Meneses E, Ni X, Zheng G, et al. (2015) Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems* 26(7): 2061–2074.

Moody A, Bronevetsky G, Mohror K, et al. (2010) 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*, New Orleans, Louisiana, pp.1–11. Washington, DC: IEEE Computer Society.

Prvulovic M, Zhang Z and Torrellas J (2002) Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: *Proceedings of the 29th Annual International Symposium on Computer Architecture ISCA*, Anchorage, Alaska, 2002, pp.111–122. Washington, DC: IEEE Computer Society.

Rashid MW and Huang MC (2008) Supporting highly-decoupled thread-level redundancy for parallel programs. In: *IEEE 14th international symposium on high performance computer architecture*, Salt Lake City, UT, 2008, pp.393–404. IEEE.

Riesen R, Ferreira K, Da Silva D, et al. (2012) Alleviating scalability issues of checkpointing protocols. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis.* Salt Lake City, Utah, pp.18:1–18:11. Los Alamitos, CA: IEEE Computer Society Press.

Ropars T, Martsinkevich TV, Guermouche A, et al. (2013) SPBC: Leveraging the characteristics of MPI HPC applications for scalable checkpointing. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis.* Denver, Colorado, pp.8:1–8:12. New York, NY: ACM.

Snir M, Wisniewski RW, Abraham JA, et al. (2014) Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* 28(2): 403–421.

Sorin DJ, Martin MMK, Hill MD, et al. (2002) Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, ISCA*, Anchorage, Alaska, 2002, pp.123–134. Washington, DC: IEEE Computer Society.

Stavrou K, Kyriacou C, Evripidou P, et al. (2007) Chip multiprocessor based on data-driven multithreading model. *International Journal of High Performance Systems Architecture* 1(1): 34–43.

Subasi O, Arias J, Unsal O, et al. (2015a) NanoCheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: *Proceedings of the 23rd euromicro international conference on parallel, distributed and network-based processing*, Turku, Finland, 2015, pp.99–102. IEEE.

Subasi O, Zyulkyarov F, Unsal O, et al. (2015b) Marriage between coordinated and uncoordinated checkpointing for the exascale era. *High Performance Computing and Communications* 470–478. Newyork, USA: IEEE.

Tahan O and Shawky M (2012) Using dynamic task level redundancy for openmp fault tolerance. In: *Architecture of Computing Systems* (*Lecture Notes in Computer Science*, vol. 7179).

Teruel X, Martorell X, Duran A, et al. (2007) Support for OpenMP tasks in Nanos v4, in CASCON 2007. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, Richmond Hill, Ontario, Canada, 2007, pp.256–259. Riverton, NJ: IBM Corp.

Treaster M (2005) A survey of fault-tolerance and fault-recovery techniques in parallel systems, *ACM Computing Research Repository* 501002: 1–11.

Wang L, Pattabiraman K, Kalbarczyk Z, et al. (2005) Modeling coordinated checkpointing for large-scale supercomputers. In: *International conference on dependable systems and networks*, Yokohama, Japan, 28 June–1 July 2005, pp.812–821. Washington, DC: IEEE Computer Society.

Young JW (1974) A first order approximation to the optimum checkpoint interval. *Communications of the ACM* 17(9): 530–531.

Zheng Z and Lan Z (2009) Reliability-aware scalability models for high performance compuing. In: *IEEE international conference on cluster computing and workshops*, New Orleans, LA, 31 August–4 September 2009, pp.1–9. IEEE.

Author Biographies

Omer Subasi Omer Subasi received his BS degree in Computer Engineering and Mathematics in Koc University, Istanbul, Turkey in 2009. He finished his Master's degree on formal verification of concurrent data structures in Koc University, Istanbul, Turkey in 2012. He is currently studying reliability for HPC and Exascale systems for his PhD degree at Barcelona Supercomputing Center and Universitat Politecnica de Catalunya in Spain. His research interests are reliability and fault-tolerance for HPC and exascale systems and applications, programming models, dataflow runtimes, error detection and correction codes, task-parallelism, and reliability modelling.

Tatiana Martsinkevich Tatiana Martsinkevich received her PhD degree from the Department of Informatics at the University of Paris Sud XI in 2015. She is currently a postdoctoral researcher at RIKEN AICS. Her research interests include fault tolerance in distributed systems, message passing libraries and parallel runtime systems.

Ferad Zyulkyarov Ferad Zyulkyarov is a research scientist at Barcelona Supercomputing Center. He leads the resiliency and fault-tolerance work package in a large EU funded research project, Mont-Blanc (www.montblanc-project.eu). His current research work is related to software-based techniques for reliability in future Exascale systems. Previously, he worked on non-volatile memory technologies (NVM) focusing on using the emerging storage-class NVMs as main memory and enabling persistence at the system and application level. His work on NVM and 2 patents, for which he is the main inventor, relate to the CLWB and PCOMMIT x86 instructions. In total, he is a co-inventor of 6 patents related to NVM.

Osman Unsal Osman Sabri Unsal is the co-leader of the Architectural Support for Programming Models group at the Barcelona Supercomputing Center. Dr. Unsal was also a researcher at the BSC-Microsoft Research Centre from 2006 to 2014. He holds BS, MS, and PhD degrees in electrical and computer engineering from Istanbul Technical University, Brown University, and University of Massachusetts, Amherst, respectively.

Jesus Labarta Jesus Labarta has been a full professor of Computer Architecture at the Technical University of Catalonia (UPC) since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, performance analysis and prediction tools. Since 2005 he has been responsible for the Computer Science research Department within the Barcelona Supercomputing Center (BSC). The major directions of his current work relate to performance analysis tools, programming models, in particular developing the StarSs model (with implementations for different multicore and accelerator nodes such as Cells, GPUs, SMP) interconnection networks, node architecture and resource management.

Franck Cappello Franck Cappello is the Project Manager of Research on Resilience at the Extreme Scale at Argonne. He received his PhD from the University of Paris XI in 1994 and joined CNRS, the French National Center for Scientific Research. In 2003, he joined INRIA, where he holds the position of permanent senior researcher. In 2009, Cappello also became a visiting research professor at the University of Illinois. He has developed and directed several R&D projects, including the MPICH-V project to provide a message-passing interface based on the Argonne-developed MPICH software. He also initiated the G8

"Enabling Climate Simulation at Exascale" project. Moreover, as a member of the executive committee of the International Exascale Software Project, he led the roadmap and strategy efforts for projects related to resilience at the extreme scale. His research interests are resilience and fault tolerance at the extreme scale, design and development of experimental platforms for grid computing, petascale computing and message passing applications.