Efficient Process Replication for MPI Applications: Sharing Work Between Replicas

Thomas Ropars, Arnaud Lefray, Dohyun Kim, André Schiper

Efficient Process Replication for MPI Applications: Sharing Work Between Replicas

Thomas Ropars*, Arnaud Lefray†‡, Dohyun Kim§ and André Schiper*

* *École Polytechnique Fédérale de Lausanne (EPFL), Switzerland*
† *University of Lyon - LIP Laboratory UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668, France*
‡ *INSA Centre Val de Loire, France*
§ *Dept. of Computer Science and Engineering, Seoul National University, Korea*
Email: thomas.ropars@eplf.ch, arnaud.lefray@ens-lyon.fr, dohyun21@snu.ac.kr, andre.schiper@epfl.ch

***Abstract*—With the increased failure rate expected in future extreme scale supercomputers, process replication might become a viable alternative to checkpointing. By default, the workload efficiency of replication is limited to 50% because of the additional resources that have to be used to execute the replicas of the application's processes. In this paper, we introduce intra-parallelization, a solution that avoids replicating all computation by introducing work-sharing between replicas. We show on a representative set of benchmarks that intra-parallelization allows achieving more than 50% efficiency without compromising fault tolerance.**

***Keywords*-High performance computing; fault tolerance; replication.**

I. Introduction

Fault tolerance is a major concern for future Exascale High Performance Computing (HPC) systems. In the context of MPI (Message Passing Interface), checkpointing and replication are the two main fault tolerance strategies. Replication has always been considered as a non-practical solution for dealing with crash-stop failures because duplicating all processes of an MPI application results in doubling the resource requirements for a job to be executed. Some studies showed that at extreme scale the legacy checkpointing technique based on coordinated checkpoints stored on a parallel file system (PFS) could incur such large overheads that replication could become a viable alternative [1]. However, many techniques including multi-level checkpointing [2], [3], hierarchical protocols [4], [5] and advanced message logging [6] have been proposed in the meantime to improve checkpointing-based fault tolerance. Considering the performance that can be achieved by such solutions, it seems that replication, with its 50% efficiency cannot compete.

In this paper, we propose a solution to improve the performance of replication for MPI HPC applications in the context of crash-stop failures beyond the hard limit of 50% efficiency. To achieve such a result, we propose the following idea. Instead of having all replicas of a process executing the whole application, we propose to have replicas collaborating during computational-intensive phases to get the results faster. To do so, we divide computational-intensive phases into tasks (similar to tasks in parallel programming models such as OpenMP) so that each replica of a logical process only executes a subset of the tasks and gets the results of other tasks from the other replicas of this logical process. The solution remains fault tolerant because if one replica fails, the others are still able to compute locally the tasks that were assigned to the failed replicas.

We implemented our solution, called intra-parallelization, in the Open MPI library, and tested it over a high performance network (Infiniband) with a representative set of HPC workloads. Results show that intra-parallelization can achieve up to 99% efficiency on computational-intensive kernels. With respect to full application execution time, the efficiency can get beyond 70%. Such results lead us to think that replication with intra-parallelization could be an attractive solution for fault tolerance at extreme scale.

II. Background

The metric we consider in this paper to evaluate a fault tolerance technique is the workload efficiency E, defined as

$$E = T_{solve}/T_{wallclock}$$

where T_{solve} is the time to solution in a fault-free system, and $T_{wallclock}$, the actual execution time for a given amount of computing resources [7].

To deal with crash failures, the main approach that has been used until now in HPC systems is global (coordinated) checkpoint-restart (cCR): The state of the application is checkpointed periodically and, if a failure occurs, the application is restarted from the last checkpoint. However with the low mean time between failures (MTBF) expected for future exascale systems, the efficiency of cCR becomes questionable. The performance of cCR depends on the time required to checkpoint the state of the application and to restore this state after a failure. Considering the performance of PFS and the size of applications, it was shown that more time could be spent dealing with failures than doing useful computation on future exascale systems [8], *i.e.*, that the efficiency could get below 50%.

Replication had always been considered much too costly for large HPC applications. Indeed, if one applies replication to an MPI application, two replicas of each MPI process have

to be executed (for a replication degree of 2). It implies that the amount of computing resources required to execute the application is doubled, *i.e.*, the efficiency can be at most 50% – assuming that (i) the application features perfect scaling, (ii) the replication protocol introduces no overhead on communication, (iii) the application is never interrupted because the two replicas of the same process fail at the same time. However, as pointed out in [1], considering the performance of the legacy cCR technique at extreme scale, replication could become attractive. Note that in the context of HPC MPI applications, most work focus on state-machine replication (also called active replication).

Since this observation was made, many efforts have been put in improving fault tolerance at extreme scale [9]. These contributions span multiple techniques including algorithmic-based fault tolerance (ABFT). ABFT seems especially attractive to deal with undetected soft-errors such as silent data corruption (SDC) [10], [11], that are expected to become more frequent in future exascale systems.

With respect to system software approaches, many solutions have been proposed to improve the efficiency of checkpointing techniques to deal with crash failures. Multi-level checkpointing [2], [3] provides means to checkpoint the application state very efficiently. New checkpointing protocols have been proposed to avoid global synchronization [12], provide failure containment [4], [13], [5], and allow efficient recovery [6], [14]. Pro-active strategies relying on failure prediction have also been proposed to improve checkpointing efficiency [15]. Obviously, all these improvements will allow checkpointing techniques to reach an efficiency higher than 50% at exascale.

Some work have also focused on replication. A theoretical analysis showed that, at scale, the mean number of process failures until the application gets interrupted, assuming independent failures and no process recovery, is large even with a replication degree of 2 [16]. It implies that replication can be combined with cCR very efficiently (the checkpointing frequency can be very low), as the probability to have to restart from a checkpoint is very low. With respect to communication protocols, we proposed a solution that relies on the partial determinism of HPC applications to provide replication with almost no overhead on communication [17].

To break down the *50%-efficiency-wall* of replication, one can envision partial redundancy, that is, replicating only some of the processes. It has been shown that if the replicated processes are chosen randomly, partial replication does not pay off [18]. However, by taking advantage of a failure predictor to choose the processes to replicate, more than 50% efficiency can be achieved [19]. In this paper, we propose an alternative solution called intra-parallelization that does not rely on failure prediction to break down the *50%-efficiency-wall*. With intra-parallelization, all processes are replicated but not all computation is executed twice.

Finally, we should mention that replication can also be used to detect and correct SDC by comparing the output of multiple replicas [20], [21]. Since our approach tries to avoid replicating computation, it cannot be used in this context. However, as mentioned previously, application-level techniques have been proposed to deal with SDC without relying on process replication [22], [10], [11].

III. Intra-parallelization

Intra-parallelization is introduced in the context of state-machine replication: It is a technique that allows sharing work between the replicas of a logical process. In this paper, we consider distributed applications where processes communicate by exchanging messages using MPI. We assume that a state-machine replication protocol for MPI processes is available [1], [17]. Each *logical* process (*i.e.*, each MPI rank) in the application is replicated, a replica being a *physical* process executed on some computing resource. In a non-replicated application, physical processes and logical processes are the same thing.

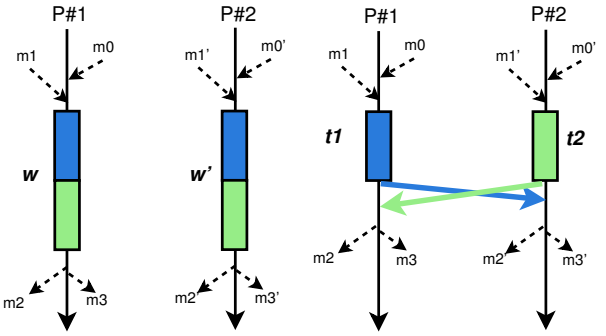
A. The main idea

The goal of intra-parallelization is to improve the performance compared to full replication while requiring only little changes to the application source code. To achieve this goal, it introduces collaboration between replicas to speed-up computation.

The execution of an MPI rank can be seen as a sequence of computation and communication steps (that might be overlapped). When an MPI rank is replicated, both computation and communication steps are replicated on all replicas. With intra-parallelization, we want to avoid fully replicating computation steps. Instead, a computation step is divided into tasks. A task is executed only by one replica, and the result of the execution is sent to the other replicas. If tasks are balanced between replicas and can be executed in parallel, then a speedup of the execution can be expected. Of course, the speedup will depend on the amount of data generated by a task that will have to be sent to the other replicas through the network.

We illustrate intra-parallelization and compare it to *classic* state-machine replication in Figure 1. In this figure, two replicas $P\#1$ and $P\#2$ of a logical process P start by executing a communication step where they receive two messages, $m0$ and $m1$ (resp. $m0'$ and $m1'$). Then, in *classic* state-machine replication (Figure 1a), they execute a computation step w (resp. w'). And finally, they execute another communication step where two messages, $m2$ and $m3$ (resp. $m2'$ and $m3'$), are sent[1]. With intra-parallelization (Figure 1b), communication steps remain the same as with *classic* replication. But the computation step w is divided into two tasks ($t1$ and $t2$) that are executed in parallel on

[1]Describing how processes are synchronized during communication steps to ensure consistency is outside the scope of this paper. We refer the reader to [17] for more information



(a) Classic replication (b) Intra-parallelization

Figure 1. State-machine replication: without and with intra-parallelization

the two replicas, allowing to reduce the execution time. After executing their task, each replica sends an update to the other replica with the result of its task, to ensure that they both have a complete and fully consistent state before moving to the next communication step. In the following, we detail our parallelization technique and how fault tolerance is ensured.

B. The parallelization technique

Intra-parallelization is based on task-parallelism [23]. The programmer is provided with the possibility to divide a computation *section* into *tasks* that can be scheduled on either replica of a logical process. In the following, we define the notion of intra-parallel *section* and *task*. Then, we discuss fault-tolerance-related issues.

1) Sections and Tasks: Intra-parallelization is applied to *sections* that are defined as follows:

***Definition** 1 (Intra-parallel section):* An intra-parallel section is a block of instructions executed by a logical process. It cannot include message-passing communication. The state of process' replicas is consistent when entering and leaving a section (the states are consistent if any variable that might be read after the execution of the section has the same value on all replicas).

We design intra-parallelization to be independent from the replication protocol used to ensure replica consistency after communication steps. This is why sections should not include message-passing communication. If communication were allowed inside a section, the replication protocol would have to be modified to take into account the fact that some communication calls might be executed inside tasks, and so, potentially only by one replica of a process. Note that this definition of sections correspond in most cases to the way OpenMP parallel blocks are used in hybrid MPI+OpenMP applications, implying that, even if communication are excluded from parallel sections, intra-parallelization can be widely applied. We define tasks as follows:

***Definition** 2 (Intra-parallel task):* An intra-parallel task is a block of instructions executed sequentially by a physical process. One or several intra-parallel tasks define an intra-parallel section. The execution of a task can be scheduled independently of any other task defined in the context of the same section.

This definition implies that the only kind of data dependence allowed between tasks belonging to the same section is *input dependence*: Two tasks are input-dependent if they both read the same variable [24]. This constraint simplifies scheduling and, as we will see, still allows applying intra-parallelization to a large number of HPC codes. The model could be extended in the future to handle more complex dependencies between tasks if needed.

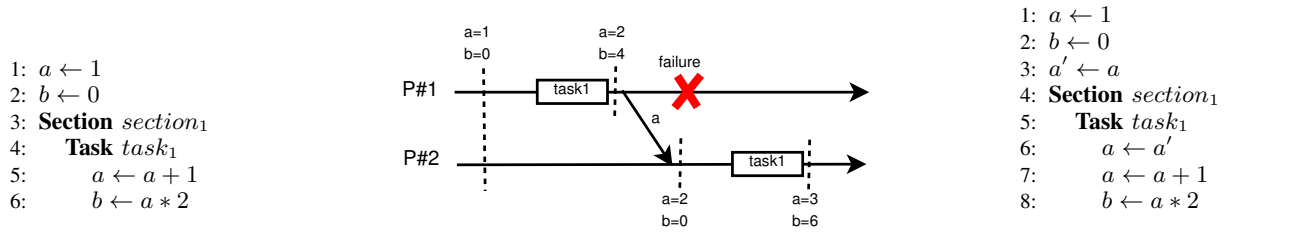
Since all replicas of a logical process have to be consistent at the end of an intra-parallel section, a replica that executes a task needs to send the update corresponding to the execution of this task to all other replicas before the section can be considered as terminated. An update has to include all variables written during the execution of the task that might be read after the end of the section.

2) Failure management: In the following, we consider the set of replicas of one logical process. When dealing with the crash of a replica, we have to distinguish between two cases: the failure occurs either inside or outside intra-parallel sections[2]. If a replica fails outside sections, no specific action is required. Since replicas are consistent outside intra-parallel sections, the remaining correct replicas can continue running and executing the state-machine replication protocol. During the next intra-parallel sections, tasks would be scheduled on the remaining replicas.

If a replica fails during the execution of a section, the other replicas have to ensure that all the tasks of this section have been executed before terminating the section. Thus if the failure of a replica is detected, all tasks that were assigned to this replica have to be assigned to another replica. Since all replicas are consistent before starting a section, each replica can execute any of the tasks. Things are actually more complex depending on when the failure occurs with respect to the task that was currently executed by the failing replica. Three cases need to be considered:

- The failure occurs before the replica has finished executing the current task or, more precisely, before it has started sending any update corresponding to the execution of this task. In this case, the task can simply be executed by another replica as mentioned before. Recall that the only kind of data-dependence allowed between tasks of one section is *input dependence*, which implies that tasks can be executed in any order.
- The failure occurs when the replica has managed to send the full update corresponding to the task execution to some but not all replicas of the logical process. In this case, the replicas that did not receive the update can either execute the task locally or get the update

[2]Failure detection is outside the scope of this paper.



(a) Initial task code (b) Incorrect execution (c) Avoiding the true dependence

Figure 2. Re-executing failed tasks: avoiding true dependence problems

from the replicas that already got it.

- The failure occurs during the send of the update such that other replicas might have received a partial update. In this case, the task has to be executed by another replica, but one problem needs to be solved: Re-executing the task might induce a *true dependence* between the executions of this task.

To explain the problem, we consider the simple example in Figure 2. In Figure 2a, we present a code snippet defining an intra-parallel section $section_1$ with one task $task_1$ to execute. This code is based on a simplified representation of sections and tasks. The actual intra-parallelization API is defined in Section III-C.

In this example, $task_1$ reads and writes variable a and writes variable b. In Figure 2b, replica $P\#1$ of logical process P executes $task_1$ and starts sending the corresponding update to the other replica, but $P\#1$ fails after sending the update of variable a and before sending the update of variable b. Because it did not receive the full update corresponding to the execution of $task_1$, the replica $P\#2$ has to execute it again. However, it does not start with the correct initial value for variable a, and so, the result of the execution of $task_1$ is incorrect.

To avoid true dependence between multiple executions of the same task, we propose to make an extra copy of the variables that could create those dependencies before starting executing a section. This is illustrated by Figure 2c: Variable a is copied into a' before entering the intra-parallel section, and the value of a' is loaded into a at the beginning of $task_1$. There is no true dependence anymore between the different executions of $task_1$ since $task_1$ does not read a anymore. In the scenario of Figure 2b, replica $P\#2$ will now get the correct result for $task_1$ since it will start by loading the value stored in a' before executing the computation.

Note that an alternative solution would be to fully avoid the problem by ensuring that any replica update corresponding to the execution of one task would be applied atomically. This can easily be implemented as well, since it would only require the replicas to store the update in a temporary buffer until the full data has been received, and only then, apply the update. In practice, both solutions have a similar cost, since an extra copy of each problematic variable is either made when entering the section (with our solution) or at the time an update is received (with the alternative).

C. The API

In this section, we introduce a basic interface that allows programmers to apply intra-parallelization. It requires defining functions that can be registered to be executed in the context of a task. Hence, it requires more modifications of the application source code compared to other programming models for task-parallelism such as OpenMP [25] or OmpSs [26]. This interface should be seen as a proof-of-concept, as the main goal of this paper is to show that intra-parallelism can be an alternative to checkpoint-restart at extreme scale. In the future, the approach could be improved to limit the amount of changes made to the application source code, by reusing compiler-based solutions typically used in OpenMP implementations of tasks.

In the current version of the API, the following functions define the beginning and the end of an intra-parallel section.

```
Intra_Section_begin()
Intra_Section_end()
```

Tasks' types have to be declared in the context of a section using the following function that returns an identifier for the new type:

```
id Intra_Task_register( f_ptr,
                        tag type arg,
                        ...)
```

The user provides a function pointer *f_ptr* that defines the code to be executed by such tasks, and defines the list of parameters of this function. Each parameter should be specified with its type and a tag that can be `in`, `out` or `inout`, depending whether the variable is going to be read, written or read and written by the function. All `out` and `inout` variables will be transferred to the other replicas of the logical process after the execution of the task. An extra-copy of all `inout` variables has to be made at the time a task is instantiated, as discussed in Section III-B2.

The following function is provided to instantiate new tasks that have been previously registered:

```
Intra_Task_launch( id,
                   data_ptr,
                   ...)
```

It takes as input the id of a previously declared task type, and a set of pointer to variables to be used as input and output parameters for the newly instantiated task. An example of intra-parallelized code is provided in Section IV.

D. Protocol description

The protocol implemented by the run-time system for intra-parallelization is described in Algorithm 1. This code is specified for a replica r of a logical process P. It includes the four functions defined in the API, plus two internal functions: `execute_task` (line 29) run a task with some given parameters; `receive_task_update` (line 36) receives the updates for one task executed by another replica. The code assumes a scheduling algorithm decides which correct replica of P executes each task (line 24). It also assumes that trying to receive an update from a failed replica returns an error (lines 41).

For the sake of simplicity, this description does not deal with the case where only a subset of the replicas would not have received the full update for one task before a failure (case discussed in Section III-B2): It only considers the case where no correct replica received a full update. In this case, the scheduler is simply asked to choose another replica to execute the task (lines 21-24). The problem with true dependencies on task re-execution described in Section III-B2, is solved by copying the corresponding variables before trying to receive the updated values for these variables (line 38) and restoring them if needed before executing a task (line 31).

IV. Intra-parallelization examples

To illustrate the use of intra-parallelization, we describe how we apply it to one computational kernel of the HPCCG mini-application from the Mantevo Suite[3]. More details about HPCCG are given in Section V.

The kernel, called *waxpby*, computes the sum of two scaled vectors:

$$W = alpha \times X + beta \times Y$$

with X and Y the input vectors, W the output vector, $alpha$ and $beta$ two scalars. Figure 3 and 4 shows the *waxpby* original code and the intra-parallelized version respectively. Some optimizations included in HPCCG have been omitted for the sake of simplicity. As it can be seen in Figure 4, intra-parallelizing the code is trivial. A task function is created to execute the *waxpby* main loop, and N tasks are launched, each one executing n/N iterations, n being the total vector size (to avoid clutter, we assume n is dividable by N).

In this example, none of the variables are read and written (no `inout` variables), which implies that no variables have to be copied to avoid problems in case of failures. It would have been different if the new value of W was computed based on its previous value. We encountered such a case in the GTC application[4]. GTC is used for gyrokinetic particle simulation of turbulent transport in burning plasmas. In this 3D Particle-in-cell code, the new position of particles has to be computed at the end of each iteration (*push* method). Applying intra-parallelization to the *push* method required us to declare particles position as `inout` variables since the new position depends on the current one.

[3]http://mantevo.org/

[4]http://www.nersc.gov/systems/trinity-nersc-8-rfp/draft-nersc-8-trinity-benchmarks

Algorithm 1 Protocol for replica r of logical process P

```
1: type TaskDef{id: int, func: pointer, args: list⟨Arg⟩}
2: type Task{def: TaskDef, vars: list⟨pointer⟩, done: bool, rId: int}
3: type Arg{ptr: pointer, argType: type, tag: Tag, copy: pointer}
4: enum Tag{ in, out, inout}

Local Variables:
5: my_rId: int ← r {id of the replica in the set of replicas of P}
6: task_defs: list⟨TaskDef⟩ ← ⊥
7: tasks: list⟨Task⟩ ← ⊥
8: id ← 0 {id for TaskDef}

9: Intra_Section_begin()
10:    task_defs ← ⊥
11:    tasks ← ⊥
12:    id ← 0

13: Intra_Task_register(f: pointer, argList: list⟨Arg⟩)
14:    id ← id + 1
15:    task_defs.insert(new TaskDef(id, f, argList))
16:    return id

17: Intra_Task_launch(id: int, vars: list⟨pointer⟩)
18:    t: Task ← new Task(task_defs.getCopy(id), vars)
19:    tasks.insert(t)

20: Intra_Section_end()
21:    while ∃t ∈ tasks such that t.done = false do
22:       t_active: list⟨Task⟩ ← {t ∈ tasks|t.done = false}
23:       for all t ∈ t_active do
24:          schedule t {call to the scheduler: assign the task to a correct
           replica by updating t.rId}
25:       for all t ∈ t_active such that t.rId = my_rId do
26:          execute_task(t)
27:       for all t ∈ t_active such that t.rId ≠ my_rId do
28:          receive_task_update(t)

29: execute_task(t: Task)
30:    for all i such that t.def.args[i].tag = inout ∧
       t.def.args[i].copy ≠ ⊥ do
31:       t.vars[i] ← t.def.args[i].copy {memory copy}
32:    call t.def.func(t.vars)
33:    for all i such that t.def.args[i].tag ≠ in do
34:       send t.vars[i] to all other correct replicas
35:    t.done ← true

36: receive_task_update(t: Task)
37:    for all i such that t.def.args[i].tag = inout ∧
       t.def.args[i].copy = ⊥ do
38:       t.def.args[i].copy ← t.vars[i] {memory copy}
39:    for all i such that t.def.args[i].tag ≠ in do
40:       recv t.vars[i] from t.rId
41:    if no recv failed then
42:       t.done ← true
```

```
int WAXPBY (int n, double alpha, double *x,
      double beta, double *y,
      double *w)
{
  for(int i = 0; i < n; i++){
    w[i] = alpha * x[i] + beta * y[i];
  }
}
```

Figure 3. Original waxpby code

```
int task_function(int task_size, double alpha,
                  double *x, double beta,
                  double *y, double *w)
{
  for(int i = 0; i < task_size; i++){
    w[i] = alpha * x[i] + beta * y[i];
  }
}

int WAXPBY (int n, double alpha, double *x,
      double beta, double *y,
      double *w)
{
  Intra_Section_begin();
  int task_id= Intra_Task_register(task_function,
        {in int}, {in double}, {in double},
        {in double}, {in double}, {out double});
  int t_size= n/N;
  for(int i=0; i<N; i++){
    Intra_Task_launch(task_id, t_size, alpha,
          &task_x[i*t_size], beta,
          &task_y[i*t_size], &task_w[i*t_size]);
  }
  Intra_Section_end();
}
```

Figure 4. Intra-parallelized waxpby with N tasks

V. Evaluation

In this section, we present an evaluation of the intra-parallelization technique. In a first part, we study a set of simple micro-kernels to understand how the trade-off between amount of computation and size of updates in intra-parallel tasks impacts the efficiency of the technique. In a second part, we present results for a set of applications selected to be representative of future exascale workloads. We start by describing our implementation of intra-parallelization in the Open MPI library.

A. Implementation

To evaluate intra-parallelization, we implemented it as an Open MPI extension. Open MPI extensions are means to extend the top-level interface available to user-level applications. Our extension, written in C with Fortran bindings, implements the interface described in Section III-C.

Our prototype is based on SDR-MPI [17], our optimized implementation of active replication for MPI applications. SDR-MPI is a patch to the Open MPI library. It relies on the partial determinism of HPC applications to implement an efficient replication protocol. All applications used in this paper comply with the partial-determinism requirements imposed by SDR-MPI. We refer the reader to [17] for a detailed evaluation of the performance of SDR-MPI. Note that the partial-determinism requirement only applies to SDR-MPI. Intra-parallelization does not require partial determinism and could be used with other MPI active replication solutions [1].

SDR-MPI allows sending messages between the replicas of a logical MPI process by simply using MPI functions over a dedicated communicator. Hence, sending updates after the execution of a task is done using MPI messages. Moreover, it has been optimized to overlap sending the updates for the finished tasks and running the computation for the remaining tasks. Namely, a replica posts reception requests for the updates of all tasks it is not going to run locally when it enters a section. As soon as a task is completed, all sending requests for the corresponding updates are also posted. All these requests are only completed at the end of the section using `MPI_Waitall`.

In the current prototype, a simple static scheduling strategy has been implemented. Assuming a section that features N tasks, and a replication degree of 2, the $N/2$ first *launched* tasks of a section are executed by replica 1 and the $N/2$ last ones are executed by replica 2. In the future, more complex strategies could be designed if needed, for instance to deal with load imbalance between replicas. Note that we did not observe such load-imbalance problems in the experiments presented in this paper.

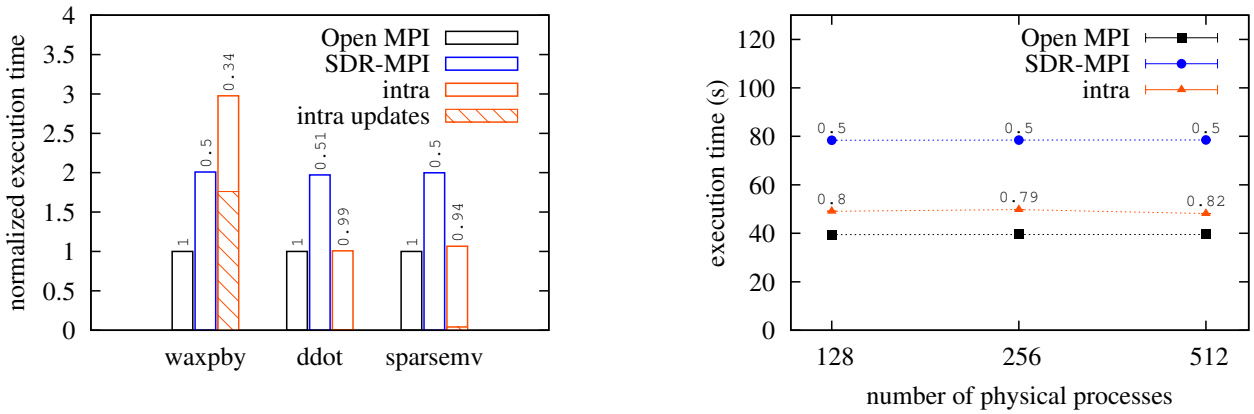
B. Experimental setup

Tests are run on a 128-node cluster of the Grid'5000 testbed. Each node is equipped with a 2.53 GHz 4-core Intel Xeon CPU and 16GB of memory. Nodes communicate over InfiniBand 20G. Operating system is Linux (kernel 3.0.0-2). Our prototype is based on Open MPI 1.7.

The applications used in the experiments are taken from sets of mini-applications and proxy applications targeting very large scale HPC systems: HPCCG and MiniGhost are mini-applications from the Mantevo suite; GTC is an application included in the NERSC 8 benchmark suite, and AMG2013 is a proxy application developed as part of LLNL Advanced Simulation and Computing program[5].

All results presented in the rest of the section are averaged over 10 executions. Standard deviation is not included in the graphs as it is always below 1%. In all experiments with replication, a replication degree of 2 is used. As discussed in Section II, this is the most appropriate replication degree when dealing with crash failures. The two replicas of a logical process are always located on different nodes. Finally, all experiments with intra-parallelization use a granularity of 8 tasks per section, *i.e.*, 4 tasks per replica. Based on our experiments, this task granularity provides good performance for most applications. Having fewer tasks reduces the opportunities of overlapping updates transfer and computation. Having more tasks can create overhead because it increases synchronization between replicas.

[5]https://codesign.llnl.gov/proxy-apps.php



(a) Basic kernels

(b) Application performance (weak scaling)

Figure 5. Performance of HPCCG with intra-parallelization (the value displayed above a data point is the corresponding efficiency).

C. Analyzing the performance of intra-parallelization

First we focus on HPCCG to study in details the performance of intra-parallelization. HPCCG is a simple conjugate gradient benchmark working on a 27-point 3D-grid-based structure. It includes three main computation kernels called *waxpby*, *ddot* and *sparsemv*. The *waxpby* operation is introduced in Section IV. The *ddot* operation computes the dot product of two vectors, *i.e.*, $s = \sum_{i=1}^{n} X[i] \times Y[i]$ with X, Y two vectors and s a scalar. The *sparsemv* operation computes a matrix vector product, *i.e.*, $y = A \times x$ with A a matrix and x, y two vectors. Studying these three kernels is interesting as they provide different trade-offs between amount of computation and size of the output data, *i.e.*, size of the updates for intra-parallelization.

To compare performance with and without replication, we fix the total number of physical processes and we adapt the per-logical-process problem size. Begin able to compare performance for a given amount of physical resources makes plots more readable. Since the number of logical processes is divided by two with replication, each of them has to handle 2 times more data for the global problem size to remain constant and allow a fair comparison. In the following, the per-logical-process problem size is set to $128 \times 128 \times 128$ for runs without replication (corresponds to a per-process memory footprint of 1.5 GB).

Figure 5a shows the performance of each kernel individually. The experiment is run with 512 cores. The graph shows the average amount of time spent by a process inside each computation kernel. It compares the time with an unmodified version of Open MPI, *i.e.*, without replication (labeled *Open MPI*) to the performance with active replication (labeled *SDR-MPI*) and with intra-parallelization (labeled *intra*). To simplify the presentation, results are normalized with the Open MPI performance as reference. On top of each box, the corresponding efficiency (defined in Section II) is displayed.

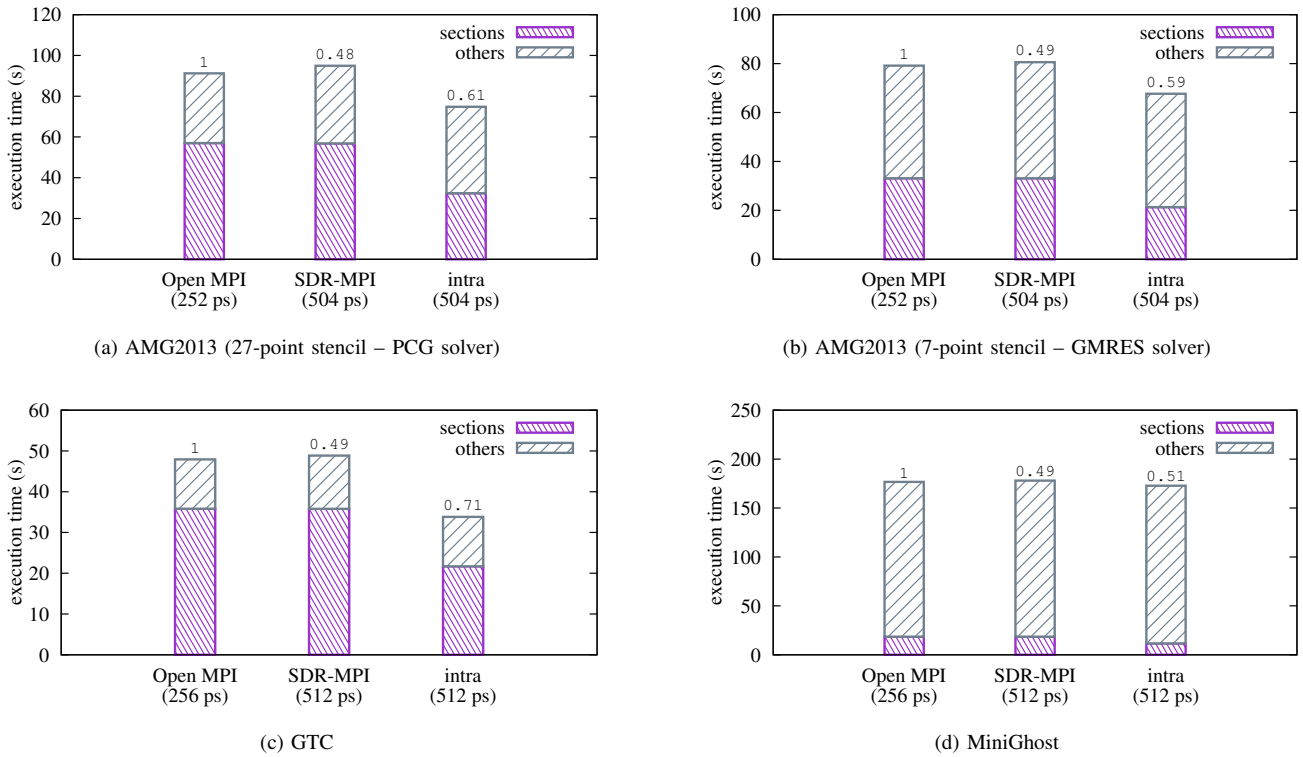
As expected, with all kernels the efficiency of active replication is 50%. Indeed, our measurements only consider the parts of the code where we applied intra-parallelization, that is parts that do not include MPI communication[6].

With intra-parallelization, results are very different depending on the kernel. For *ddot* and *sparsemv*, the efficiency gets very close to 100% whereas for *waxpby*, the efficiency is even worse than with only active replication. To better understand what is happening, we show on the graph the amount of time spent sending and receiving updates for executed tasks (dashed area). More precisely, it shows the time spent by a replica finishing transferring updates after it has finished executing all assigned tasks for the section, *i.e.*, some data transfers might have been overlapped with tasks' execution. It shows that whereas no additional time is spent transferring updates with *ddot* and whereas this time is close to zero for *sparsemv*, most of the time is spent on updates transfer with *waxpby*. This explains the bad performance of intra-parallelization with *waxpby*.

Such results were foreseen based on the ratio between computation and updates' size featured by the kernels. On one hand, the output of a *ddot* task is a single scalar, and so, the intra-parallel version of ddot performs extremely well. On the other hand, the output of a *waxpby* task is a vector of the same size as the input, and so, sending updates takes more time than running the computation. Still it is interesting to notice that, although the output of *sparsemv* is also a vector, intra-parallelization performs well as the amount of computation is larger (operations on a matrix). We should also point out that results could have been better with *waxpby* if the number of computing operations required to generate the output vector would have been higher: We can relate intra-parallelization efficiency to the number of floating-point operations required to compute each output.

Figure 5b presents the application's total execution time for different number of physical processes. The per-logical-

[6]The *ddot* routine includes a reduction step, but this step was excluded from the intra-parallel section, and so, is also not included in our time measurements.



(a) AMG2013 (27-point stencil – PCG solver)

(b) AMG2013 (7-point stencil – GMRES solver)

(c) GTC

(d) MiniGhost

Figure 6. Performance of intra-parallelized applications (`sections`: parts of the code where intra-parallelization has been applied; `others`: unmodified parts of the code; the value displayed above a data point is the corresponding efficiency; `ps`: physical process).

process problem size remains constant and is the same as in the previous experiment. Of course in this experiment, intra-parallelization is only applied to *ddot* and *sparsemv*, since it does not provide good performance with *waxpby*. Results show that intra-parallelization allows us to greatly improve the replication efficiency that reaches more than 80%.

D. Application performance

We evaluate the performance of intra-parallelization on three applications, namely, AMG2013, GTC and MiniGhost. In these applications, we did not apply intra-parallelization to all computational kernels, but we focused on the main kernels where intra-parallelization could be applied efficiently. Hence, the results are not necessarily the best performance that could be achieved with intra-parallelization, but they aim at illustrating the potential performance benefit that can be gained with intra-parallelization.

Figure 6 presents the results. In these experiments, the comparison between the native performance of the application (labeled *Open MPI*) and the performance with replication (labeled *SDR-MPI* for active replication and *intra* for the intra-parallelized version) is done in a different way compared to the experiments in Section V-C. With the application considered in this section, it is not always obvious how to double the per-process problem size. Hence, instead of keeping the number of physical resources constant and doubling the per-logical-process problem size when going from a non-replicated execution to a replicated one, we keep the problem size constant and we double the number of physical resources used: If the execution time is the same in a run with the native application and in a run with replication, the efficiency of replication is 50%, as replication uses two times more resources. Runs of the native AMG2013 code span 252 physical processes (AMG2013 has to be run on n^3 logical processes), and runs with replication span 504 processes. For the two other applications, 256 and 512 physical processes are used. The performance of SDR-MPI is included in the figure to allow the reader to assess the overhead induced by the active replication protocol.

Figures 6a and 6b show results for AMG2013, which is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. We present results for two problems. In Figure 6a, the preconditioned gradient method is applied to a Laplace-type problem using a 27-point stencil. In Figure 6b, the GMRES method is applied to a Laplace-type problem using a 7-point stencil. In the two experiments, the per-logical-process problem size is $100 \times 100 \times 100$. In both cases, intra-parallelization achieves around 60% efficiency although the parts of the code where intra-parallelization has been applied only represent 62% in the first case and 42% in the second case of the total execution time with the native application.

Figure 6c presents results for GTC, a 3D Particle-in-cell code. The experiment is run with $mzetamax = 64$,

$npartdom = 4$ and $micell = 200$. As mentioned in Section IV, GTC includes one example of computational kernel that includes `inout` variables requiring an extra-copy to ensure correctness. In this application, we applied intra-replication to the two main computational kernels (*charge* and *push*), accounting for 75% of the total execution time in the native code. Hence, intra-parallelization efficiency is more than 70%. Extra copy of `inout` variables induces 6% of extra overhead on the affected tasks.

Finally, Figure 6d illustrates a case where intra-parallelization cannot be efficient. MiniGhost is designed to study boundary exchange strategies using stencil computations. In this experiment, it runs a 3D 27-point stencil with a per-process problem size of $128 \times 128 \times 64$. The main computational kernel is the 27-point stencil. We could not apply intra-parallelization efficiently to this kernel as the output is a new 3D matrix: the performance with intra-parallelization were around the same as without intra-parallelization. Hence, we could only apply intra-parallelization efficiently to a function computing a summation of the grid elements that accounts for 10% of the total execution time, leading to low performance increase.

VI. Discussion

Efficiency of the proposed technique: The results presented in Section V only evaluate the efficiency of intra-parallelization in failure-free scenarios. To evaluate its real efficiency, experiments with realistic failure distributions would have to be conducted. With intra-parallelization, it is important to restart failed replicas as soon as possible, since speed-up of a logical process execution can only be achieved if tasks are shared among multiple replicas. Another study of MPI replication shows that the cost of starting a new replica is low in general [19]. This result makes us think that intra-replication will perform well in real test-case scenarios including failures. Analyzing the exact efficiency of intra-parallelization at extreme scale would deserve its own study.

Scalability of the approach: One might argue that the testbed used for experiments does not allow assessing the scalability of the approach. Unfortunately, since intra-parallelization relies on a modified version of an MPI library, it is hard to get access to large scale production machines to run experiments. On the other hand, the experiment presented in Figure 5b provides some evidence that intra-parallelization can scale as the efficiency remains constant when the number of processes increases. Furthermore, one should notice that the only communication introduced by intra-parallelization are messages exchanged between the replicas of a logical process. This communication pattern is scalable by nature. The main scalability issue that could arise is network contention: It could appear if the replicas of each logical process are mapped to too distant nodes in the physical network topology, leading to many messages crossing the network. Hence, replicas should be positioned on neighboring nodes to avoid network contention but at the same time, they should be placed in such a way that the probability of correlated failures is low [2]. This would be an interesting optimization problem to study.

About task-parallelism: As described in Section IV, the current intra-parallelization API allows applying it with only minor changes to the original source code. Implementing a compiler-based approach, such as in OpenMP for instance, could simplify even more the intra-parallelization of codes. We should mention that for applications that already combine MPI and OpenMP, as it was the case for the applications considered in this paper[7], creating tasks is simple since the work required to identify potential parallelism inside the code as already been done. Finally, it should be noticed that intra-parallelization can be used in combination with OpenMP or any other parallelization solution for shared-memory systems: OpenMP can be used inside intra-parallel tasks to take advantage of multi-core processors.

VII. Conclusion

This paper proposes a new fault tolerant technique for MPI HPC applications: Intra-parallelization improves the efficiency of replication-based fault tolerance by introducing collaboration between the replicas of a logical process to execute computational intensive kernels more efficiently. We described the intra-parallelization algorithm as well as our API to define intra-parallel sections and tasks. Minor modifications of existing applications are required to take advantage of this new technique.

Our experiments with a representative set of HPC applications show that for many of them, the *50%-efficiency-wall* of replication techniques can be broken down thanks to intra-parallelization. For some applications, the efficiency can even get beyond 70%. These results illustrate that, at least in some cases, intra-parallelization could be a viable alternative to checkpointing-based fault tolerant solutions at extreme scale. Further studies need now to be conducted to assess the efficiency that can actually be achieved by this new technique on future exascale machines.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *IEEE/ACM SuperComputing 2011 (SC11)*, 2011, pp. 44:1–44:12.

[7]The applications used for evaluation include OpenMP pragmas, but they were not compiled with OpenMP activated.

[2] L. Bautista-Gomez, N. Maruyama, D. Komatitsch, S. Tsuboi, F. Cappello, S. Matsuoka, and T. Nakamura, "FTI: high performance Fault Tolerance Interface for hybrid systems," in *IEEE/ACM SuperComputing 2011 (SC11)*, Seatle, USA, November 2011.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *IEEE/ACM SuperComputing 2010 (SC10)*, 2010, pp. 1–11.

[4] A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Correlated Set Coordination in Fault Tolerant Message Logging Protocols," in *Proceedings of the 17th international conference on Parallel processing (Euro-Par'11)*, 2011, pp. 51–64.

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

[6] T. Ropars, T. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello, "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing," in *IEEE/ACM SuperComputing 2013 (SC13)*, 2013.

[7] M. Snir, R. W. Wisniewski *et al.*, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[8] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, "Modeling the Impact of Checkpoints on Next-Generation Systems," in *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 30–46.

[9] F. Cappello, A. Geist, S. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience: 2014 Update," *Supercomputing Frontiers and Innovations*, vol. 1, pp. 1–28, 2014.

[10] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based Fault Tolerance Applied to High Performance Computing," *Journal of Parallel and Distribed Computing*, vol. 69, no. 4, pp. 410–416, Apr. 2009.

[11] T. Davies and Z. Chen, "Correcting soft errors online in lu factorization," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*, 2013, pp. 167–178.

[12] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications," in *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011)*, Anchorage, USA, 2011.

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

[14] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges, "Alleviating scalability issues of checkpointing protocols," in *IEEE/ACM SuperComputing 2012 (SC'12)*, 2012, pp. 18:1–18:11.

[15] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyama, "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*, 2013, pp. 501–512.

[16] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Using group replication for resilience on exascale systems," INRIA, Tech. Rep. RR-7876, 2012.

[17] A. Lefray, T. Ropars, and A. Schiper, "Replication for Send-Deterministic MPI HPC Applications," in *3rd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, 2013.

[18] J. Stearley, K. Ferreira, D. Robinson, J. Laros, K. Pedretti, D. Arnold, P. Bridges, and R. Riesen, "Does partial replication pay off?" in *2012 IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2012, pp. 1–6.

[19] C. George and S. Vadhiyar, "Fault tolerance on large scale systems using adaptive process replication," *IEEE Transactions on Computers*, 2014.

[20] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *IEEE/ACM SuperComputing 2012*, 2012, pp. 78:1–78:12.

[21] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection," in *IEEE/ACM SuperComputing 2013*, 2013, pp. 7:1–7:12.

[22] L. Bautista Gomez and F. Cappello, "Detecting Silent Data Corruption Through Data Dynamic Monitoring for Scientific Applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, 2014, pp. 381–382.

[23] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A Proposal for Task Parallelism in OpenMP," in *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, 2008, pp. 1–12.

[24] J. R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. dissertation, Houston, TX, USA, 1983.

[25] OpenMP Architecture Review Board, "Openmp application program interface, version 4.0," www.openmp.org, July 2013.

[26] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.