Supporting Task-level Fault-Tolerance in HPC Workflows by Launching MPI Jobs inside MPI Jobs

Matthieu Dorier, Justin M. Wozniak, Robert Ross
Argonne National Laboratory
Lemont, IL
mdorier@anl.gov,wozniak@mcs.anl.gov,rross@anl.gov

ABSTRACT

While the use of workflows for HPC is growing, MPI interoperability remains a challenge for workflow management systems. The MPI standard and/or its implementations provide a number of ways to build multiple-programs-multiple-data (MPMD) applications. **These methods present limitations related to fault tolerance**, and are not easy to use. In this paper, we advocate for a novel `MPI_Comm_launch` function acting as the parallel counterpart of a `system(3)` call. `MPI_Comm_launch` allows a *child* MPI application to be launched *inside* the resources originally held by processes of a *parent* MPI application. Two important aspects of `MPI_Comm_launch` is that it pauses the calling process, and runs the child processes on the parent's CPU cores, but in an isolated manner with respect to memory. This function makes it easier to build MPMD applications with well-decoupled subtasks. We show how this feature can provide better flexibility and better fault tolerance in ensemble simulations and HPC workflows. We report results showing 2× throughput improvement for application workflows with faults, and scaling results for challenging workloads up to 256 nodes.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**;

KEYWORDS

MPI, MPMD, Workflows, Ensemble Simulations, Swift/T, Cram, MPI_Comm_launch

ACM Reference Format:
Matthieu Dorier, Justin M. Wozniak, Robert Ross. 2017. Supporting Task-level Fault-Tolerance in HPC Workflows by Launching MPI Jobs inside MPI Jobs. In *WORKS'17: WORKS'17: 12th Workshop on Workflows in Support of Large-Scale Science, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3150994.3151001

1 INTRODUCTION

Multiple-programs-multiple-data (MPMD) applications are becoming more and more common in today's HPC landscape. Simulations can be coupled with analysis and visualization tools, enabling direct communication between software components. Workflows run serial and parallel tasks potentially written in different programing languages and whose data dependencies need to be managed by the workflow management system (WMS). Ensemble simulations, which consist of running many instances of the same simulation using different input to compute statistics on their aggregated results, can also be put under the workflow umbrella, since they pose similar challenges in terms of process management and fault tolerance.

While WMS in the cloud computing area are usually built with task-level fault-tolerance in mind, HPC workflows are generally not fault tolerant. They are still hand-written using MPI or rely on WMS such as Swift/T [1, 16] that are themselves built on top of MPI. While waiting for better alternatives, supporting task-level fault tolerance in HPC workflows through MPI is therefore critical.

Solutions within the current MPI standard and/or its implementations include using MPMD mode in the MPI launcher's command line (`mpiexec`, `aprun`, etc., depending on the implementation in use); spawning new processes by using functions such as `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`; connecting independently started applications using `MPI_Comm_accept`,`connect`, `join`; and rewriting each program as a function, using MPI communicator management to split the set of processes according to different roles. All these methods have limitations and development overhead, and make fault tolerance difficult if not impossible to manage.

Several requirements can be highlighted to better support MPMD in an MPI context.

(1) **Dynamicity:** One must be able to start and stop tasks (that is, components of an MPMD application) during the run time of a job, on a selected set of resources, without having to specify those tasks statically at job submission time.
(2) **Resource isolation:** Tasks must be able to run in an isolated manner, with their own virtual memory space, to avoid problems with shared symbols, memory leaks, etc.
(3) **Fault tolerance:** The failure of a task due to software defect should not impact other independent tasks. On the contrary, workflow logic should be able to respond to such a fault condition and react accordingly.
(4) **Ease of use:** The mechanism should improve the programmability of workflow applications. In particular, it should allow for the composition of existing MPI applications with little to no modifications in their initial code.

These requirements advocate for a simple solution: enable an MPI application to be launched "inside" another MPI application. In this paper, we propose a new MPI primitive, `MPI_Comm_launch`, to ease MPMD programming and enable better fault tolerance. `MPI_Comm_launch` acts as the parallel counterpart of a `system(3)` call, which, in serial programs, pauses the calling program to start the requested command (through a shell) and returns the exit status of this command. Similarly `MPI_Comm_launch` allows a *child* MPI application to be launched from a set of processes in a *parent* MPI application. While the processes in the parent application are paused, waiting for the exit status of their child, the child application runs on the same CPUs as the paused processes and executes in an isolated manner with respect to memory. Additionally, the child application does not share communication means with its parent. In case of a *software failure* of the child application (e.g., segmentation fault, or even the application calling MPI_Abort following inconsistent computations) the parent application is not affected and is made aware of the failure through the child's return status, offering the workflow manager a chance to react to the failure by either restarting the task, or notifying the user of the task's failure.

This paper makes the following contributions.

- We propose the `MPI_Comm_launch` function along with its semantics, and we compare its advantages with existing MPMD functionalities (Section 3).
- We provide an implementation of `MPI_Comm_launch` and recommend better ways of integrating it in existing MPI implementations such as MPICH (Section 3.3).
- We exemplify the use of `MPI_Comm_launch` in two contexts: the Swift/T parallel scripting language and its associated ADLB-based [11] runtime, and the Cram library from Lawrence Livermore National Laboratory (LLNL), designed to run ensemble simulations (Section 4).
- We evaluate the benefits of `MPI_Comm_launch` in terms of programming overhead, performance, and fault tolerance using CODES-ESW, a real HPC workflow used for design-space exploration of network topologies using the CODES network simulator and synthetic tests designed to analyze raw performance (Section 5).

2 BACKGROUND AND RELATED WORK

In this section we first explicit our fault model. We then detail ways of implementing MPMD applications and workflows using MPI. Table 1 summarizes how each of them satisfies the requirements presented in Section 1.

2.1 Note on the failure model

In this work, we follow the reliability definitions from Hennessy and Patterson [10]. In short, a software defect or other anomaly causing a program to crash is a *fault*; this leads to an *error* state in software; unhandled errors result in service *failures* that propagate up to higher levels of the system architecture, ultimately to the user. We use the term *abort* to indicate other software components that fail due to the failure of a peer component.

The current MPI standard is vague when it comes to fault tolerance. For example, it does not specify what should happen in case of a hardware failure. For this, some tools such as BLCR[1] allow to checkpoint and restart MPI applications, and migrate processes. Some work has been done to propose new functions for supporting fault tolerance at user-level [4], but they have not yet been included in the standard.

Another place where the standard remains imprecise is when calling `MPI_Abort` from a process, in which case the standard only recommends that the processes in the provided communication group be aborted, while, to the best of our knowledge, all implementations abort the entire application. In this situation tools like BLCR are hardly useful, since the application has explicitly aborted itself.

We call *task-level fault-tolerance* the fact of being able to contain a fault to a subset of processes logically representing a well-defined component, or task. The reason for the failure is only software-related, may have been requested explicitly (e.g., with `MPI_Abort`) or not (e.g., segmentation fault). A framework is task-level fault-tolerant if, when a logical subset of processes fails, the framework is notified so as to take action, and the hardware running the failed processes remains available for running other processes in the future.

While our contribution enables task-level fault-tolerance in MPI-based HPC workflows, we do not claim to provide a solution to *process-level* fault-tolerance (e.g., reconfiguring the application after a process failed and rolling it back to a consistent state).

2.2 MPMD techniques using MPI

2.2.1 MPI launchers MPMD mode. Although the MPI standard does not define an interface that the process launcher (mpiexec, aprun, etc.) must comply to, its implementations can generally be used to enable MPMD by specifying multiple executables and their arguments, as well as the number of processes on which each should run. The executables then share a common `MPI_COMM_WORLD` communicator. This mode is therefore convenient for coupled codes, provided that each component of the application is aware that `MPI_COMM_WORLD` is being used by other components.

This MPMD mode is, for instance, used in the Decaf middleware,[2] which couples components and establishes communications between them. The Python script that describes the software components and their data flows is used to issue an `mpiexec` command under the cover, with the proper settings to enable each component to run on the appropriate locations.

In other contexts, however, MPI launcher's MPMD mode presents more limitations. Ensemble simulations require launching thousands of instances with different input parameters. One would need to make sure that the code of the simulation does not use `MPI_COMM_WORLD`, since this communicator is shared among all instances. Also, the run time of the set of simulation instances is that of the slowest instance, leaving resources idle with no possibility for scheduling more work. A better approach would be to launch instances in parallel but also one after the other as resources are released by completed instances; unfortunately MPI launchers MPMD mode cannot do this, since they are inherently supposed to start all the provided executables at once. LLNL's Cram library,

[1] http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/
[2] https://bitbucket.org/tpeterka1/decaf

Table 1: Summary of the existing approaches to MPMD and how they satisfy the requirements presented in the introduction. (The notion of "programmability" denotes whether code changes *in the child application* are necessary.)

Approach	Dynamicity	Memory Isolation	Fault tolerance	Programmability
MPI Launcher's MPMD mode	✗	✓	✗	some code changes
Communicators management	✓	✗	✗	many code changes
MPI_Comm_spawn	✓	✓	✗	some code changes
MPI_Comm_launch (this work)	✓	✓	✓	no code changes

described in more detail in Section 4.1, overcomes the issue of isolating communications, but it does not allow starting instances one after the other.

In the context of HPC workflows, the MPI launcher's MPMD mode simply does not fit: tasks corresponding to individual programs have to be started and stopped dynamically in a way that cannot be predicted when writing the command line.

We also note that on many supercomputers this method of running MPMD programs does not allow multiple executables to reside on the same node (all the cores of a node have to execute the same program).

In terms of fault tolerance, this method does not allow limiting a fault to the faulty component, unless a process-level fault-tolerance mechanism is used; such a mechanism is yet unstandardized.

2.2.2 Communicator management. Another way of implementing MPMD applications and workflows is to make sure each software component is written as a library, with an entry function that at least takes the communicator to use in place of `MPI_COMM_WORLD`. Distinct applications can then be bound together by calling them from a program that builds the appropriate communicators between groups of processes. The MPI functions that help achieve this are `MPI_Comm_split` and `MPI_Comm_create`.

This method is particularly useful for codes coupled for in situ data processing. As an example, Damaris [8], is a middleware that enables in situ analysis and visualization through dedicated cores or dedicated nodes. The initialization function of Damaris calls `MPI_Comm_split` to assign to some processes the tasks related to in situ analysis. The client application is, however, required not to use `MPI_COMM_WORLD`; hence, some development effort is necessary to adapt legacy codes, namely, finding all occurrences of `MPI_COMM_WORLD` in the code and changing it to a global variable initialized with a smaller communicator. While such changes seem reasonable, they can be troublesome when libraries upon which the code depends also use `MPI_COMM_WORLD` directly. This is the case in the workflow presented in Section 5.1, where ROSS, the library upon which CODES is built, uses `MPI_COMM_WORLD`.

`MPI_Comm_create_group` is used by the Swift/T WMS, described in more detail in Section 4.2, to call workflow tasks from a subset of worker processes. Again, the workflow tasks have to be written as libraries exposing an entry function that accepts a communicator. They should not use `MPI_COMM_WORLD`. Converting an existing, standalone MPI program into a library that can be used as a task therefore requires development overhead, as exemplified in Section 5.2.

This method of implementing MPMD programs is not fault tolerant beyond what a process-level fault-tolerance mechanism would provide. Since tasks are called as functions, a software fault in a function leads to the abort of all the components of the application and workflow beyond the function itself. In addition, contrary to MPI's launcher MPMD mode and to `MPI_Comm_spawn`, tasks are not isolated in their own processes; hence the developer has to make sure that the function does not leak memory and does not have symbols in common with other software components.

2.2.3 MPI_Comm_spawn(_multiple). `MPI_Comm_spawn` and `MPI- _Comm_spawn_multiple` are potentially the most appropriate functions from the current standard for implementing workflows and more generally MPMD applications. They act as the parallel equivalent of the `fork(2)/execv(3)` sequence by creating new processes and using them to run a given MPI application. A communicator is created that lets the spawned application communicate with its parent.

`MPI_Comm_spawn` could potentially help implement coupled codes, workflows, and ensemble simulations, but it has several limitations. The first is that, contrary to a `fork(2)/execv(3)` sequence that is usually followed by a `wait(3)` in the parent process, there is no function to check whether the spawned application has terminated. The only way the parent application can "wait" for the child application is by having the child send a termination message to the parent using `MPI_Send`, and call `MPI_Comm_disconnect` on the communicator that binds it to its parent. The reusability of the resources associated with a spawned application that terminated is not discussed in the MPI standard.

The second limitation of `MPI_Comm_spawn` is that of resource management. The MPI 3.1 standard states in its section 10.1 that *"The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers."* In other words, *where* the child processes are started is up to the resource manager, with the help of user-provided attributes from the passed `MPI_Info` argument (e.g. *hosts* and *hostfile* keys). This means that to start processes on previously idle cores, the resource manager should allow to reserve a set of node, launch an MPI program on some of them, while informing the MPI runtime that other nodes are available for spawning processes later. To start processes on cores where an MPI process already runs, the operating system must support oversubscription, which is often not the case on supercomputers because of the limited support for preemptive process scheduling. For these reasons IBM and Cray's implementations of MPI do not support `MPI_Comm_spawn`. IBM

BlueGene/Q, in particular, has hardware limitations that make it impossible to implement.

The third limitation is, again, fault tolerance. In all current implementations of MPI, because the spawned application and its parent share a communicator, a fault in the spawned application will propagate to the parent, and abort the entire set of software components.

3 MPI_COMM_LAUNCH

In this work we advocate for a parallel counterpart of the *system(3)* function, which blocks the calling process until the provided command is run and which returns the exit status of this command upon termination. We propose the `MPI_Comm_launch` function, with the following C prototype:

```
int MPI_Comm_launch(char* command, char* argv[],
                    MPI_Info info, int root,
                    MPI_Comm comm, int* status);
```

And the following Fortran prototype:

```
MPI_COMM_LAUNCH(COMMAND, ARGV, INFO, ROOT,
                COMM, STATUS, IERROR)
  CHARACTER*(*) COMMAND, ARGV(*)
  INTEGER INFO, ROOT, COMM, STATUS, IERROR
```

Like the first two parameters of `MPI_Comm_spawn`, the `command` argument provides the path to the MPI program to be executed and `argv` (null-terminated array of strings) provides its arguments. The `info` parameter can be used to provide options that would normally be passed to the MPI launcher's command line, such as forwarded environment variables, standard output formatting and redirection for the executed processes. The `root` argument is the rank of the process in which previous arguments are examined. The `comm` communicator gathers all the processes collectively calling this function. When the child application terminates, the `status` parameter is set to the child application's exit code.

Listing 1 shows an example usage of `MPI_Comm_launch` where a child application is launched on half the processes of the parent.

```
MPI_Comm childcomm;
int color = rankInWorld < sizeOfWorld/2 ? 0 : 1;
int key = rankInWorld;
MPI_Comm_split(MPI_COMM_WORLD, color, key, &childcomm);
if(color) {
  int status;
  MPI_Comm_launch("./child", NULL, MPI_INFO_NULL, 0,
                  childcomm, &status);
}
```

Listing 1: Example C code splitting MPI_COMM_WORLD and running a child application on half of the parent processes.

3.1 Semantics

`MPI_Comm_launch` is a collective operation over its provided communicator argument. Its semantics are similar to calling the given executable as a collective function taking an array of arguments and a communicator, and in which `MPI_COMM_WORLD` is replaced with the provided communicator. Contrary to a function call, however, the execution of the command is isolated in new processes with their own virtual memory.

More specifically, all processes calling `MPI_Comm_launch` transition to a *blocked* state. On each core where the function is called, a new process is created and executes the provided command. The new processes form a single MPI application with its own `MPI_COMM_WORLD`. The order of processes (that is, their rank in `MPI_COMM_WORLD`) in this new application follows that of the parent processes in the communicator provided as argument. Once the child application terminates, the parent processes switch back to a *running* state. On all parent processes the *status* parameter is set to the exit status of the child application as defined in the following.

Exit status. If all processes in the child application call `MPI- _Finalize`, the exit status is 0. If however one process calls `MPI_Abort` with an error code X, or calls `exit(`X`)`, or terminates with a particular error code (e.g. 11 for a segmentation fault), this error code is set as status for the entire child application. If multiple processes fail with a different error code, one of them is chosen as exit status and is consistent across all parent processes.

3.2 `MPI_Comm_launch` is not `MPI_Comm_spawn`

Why block the parent processes? As explained earlier, `MPI_Comm- _spawn` creates new processes but does not block the calling ones. The location of these processes depends on user-provided attributes and on the resource manager, hence the standard remained vague and major vendors simply don't support `MPI_Comm_spawn`. `MPI_Comm- _launch`, however, does not require any additional resources, hence it does not depend on the resource manager. Also the number of *running* processes/threads remains constant before, during, and after its execution. Hence, supporting `MPI_Comm_launch` is less challenging, even on platforms where an OS without a preemptive scheduler enforces the existence of exactly one process or thread per computing element. A precise description of `MPI_Comm_launch` can thus be added to the standard without depending on the resource manager or the OS. It will allow implementors to handle the complexity of launching the new processes in their various operating systems and environments in a portable manner.

Why restrict communication? With respect to communication, `MPI_Comm_spawn` creates an intercommunicator shared by the calling processes and the spawned application, whereas `MPI_Comm- _launch` does not create such a communication mechanism. While this can be seen as a limitation, it is actually an advantage. As explained in Section 2.1, the MPI standard remains vague about propagating software failures. Implicitly opening potentially unnecessary communicators gives a chance for failures to propagate beyond a single launched task. `MPI_Comm_launch` ensures no communicator is created between the parent and the child, hence making it easier define failure containment to the child application. In case of a software failure or an abort, the child application will simply terminate with a nonzero exit code, and its parent will be notified of the failure through the returned status. Section 6 provides more thoughts about initiating communication between tasks.

Workflows, ensemble simulations, and coupled models The use of `MPI_Comm_launch` greatly simplifies the implementation of workflows, coupled models, and ensemble simulations. It enables starting subapplications dynamically and in an isolated manner, letting developers setup the necessary communication and fault tolerance mechanisms they need to react to task faults. In workflows in particular, it enables tasks to be decoupled from the workflow itself, in terms of both programmability (the tasks can be designed as independent programs) and reliability (the tasks execute in an isolated manner). In Section 4 we present two use cases illustrating the benefits of `MPI_Comm_launch` in the context of ensemble simulations and workflows.

3.3 Implementation

We implemented a prototype[3] of `MPI_Comm_launch` as an MPI extension function, `MPIX_Comm_launch`. This function works as follows (its signature is in the beginning of §3).

(1) Each process in `comm` calls `MPI_Get_processor_name` to retrieve the name of the host on which it runs.
(2) `MPI_Gather` is called to gather all the host names at the *root* process.
(3) The *root* process formats the array of hostnames as a comma-separated list, as well as other potential arguments provided in the *info* variable.
(4) The *root* process calls the *system(3)* function with a command that invokes `mpiexec` on the specified executable and with the arguments previously formatted.
(5) Once the *system* call completes in the *root* process, `MPI_Bcast` is called to broadcast the exit status of the program to all other processes.

The root process blocks in the `system(3)` call while all other processes block in the `MPI_Bcast` during the execution of the child application. Because of the use of `system` and `mpiexec`, two more processes are created in the node running the `root` process: one shell instance to invoke the command and one `mpiexec` process (plus potentially other processes created by `mpiexec`).

This implementation works on traditional Linux clusters in which process management is not an issue. On supercomputers with microkernels, however, a different implementation is required. In particular, we envision a direct use of the Process Management Interface [2] (PMI[4]) to reuse resources of the parent application (such as the KVS space) without calling `mpiexec` or a shell. Such an implementation would work as follows.

(1) Each process in `comm` calls the PMI interface to put key information required for the child application to bootstrap.
(2) Each process in `comm` calls `fork(2)` then `execv(3)` to create a process of the target executable, followed immediately by `wait(3)`.
(3) The newly created processes use the PMI interface to get the necessary information to bootstrap their communications. This include information put by parent processes as well as information related to the parent application that is inherited by the child.
(4) Once the `wait()` calls complete, `MPI_Bcast` is called by the parent processes to broadcast the exit status of the program to all other processes.

[3]See https://bitbucket.org/mdorier/mpix_launch
[4]https://wiki.mpich.org/mpich/index.php/PMI_v2_API

4 CASE STUDIES

In this section, we present two practical use cases motivating the addition of `MPI_Comm_launch` to the MPI standard: Cram and Swift/T workflows.

4.1 LLNL's Cram

Cram[5] [9] is a tool that lets users pack and launch many small instances of an application as a single MPI program. It was developed by LLNL as a way to avoid overloading the job scheduler on the Sequoia supercomputer when running millions of small MPI jobs, and because running a single job script with a million *mpiexec* calls would cause the frontend to run out of resources such as memory or number of processes.

In order to use Cram, the application should be linked with *libcram.a*. The user then generates a *cram.job* file containing the list of instances to run, along with their arguments. When the application runs, the initial call to `MPI_Init` reads the Cram job file and splits `MPI_COMM_WORLD` into as many smaller communicators as necessary to run all the instances of the application in parallel. The library overloads all the MPI functions such that any later reference to `MPI_COMM_WORLD` is replaced with the appropriate smaller communicator. Each MPI process thus "believes" it is part of a much smaller group of processes than the actual `MPI_COMM_WORLD` really is. Cram also redirects each instance's standard output and standard error into individual files.

4.1.1 Cram's limitations.

Since Cram is implemented as a library against which the desired application is linked, it can run instances of only that application. Distinct executables cannot be packaged using Cram, thus making MPMD impossible.

Additionally, if one instance of the application fails, all instances will abort. Cram does not provide any fault tolerance mechanism to safely terminate one instance without impacting other instances, let alone the ability to restart failed instances.

A Cram job should run on at least as many processes as the sum of the processes required by each of its individual instances. If a smaller number is provided, Cram is not able to schedule instances one after the other. Thus *all* the instances run in parallel, and the entire job takes as much time to complete as the time required by the longest instance.

Moreover, because Cram relies on the MPI profiling API (PMPI), the application cannot use tracing libraries such as DUMPI,[6] IPM[7] [18], or Darshan[8] [5], in conjunction with Cram.

4.1.2 Cram with MPI_Comm_launch.

`MPI_Comm_launch` provides a good opportunity to reimplement Cram in a way that overcomes its current limitations. Using `MPI_Comm_launch`, Cram would be implemented as a standalone MPI program that reads a *cram.job* file and calls `MPI_Comm_launch` for each

[5]https://github.com/LLNL/cram
[6]http://sst.sandia.gov/using_dumpi.html
[7]http://ipm-hpc.sourceforge.net/
[8]http://www.mcs.anl.gov/research/projects/darshan

Mode	Task Type	
	In-memory	External program
Sequential	Scripting langs [15]	`app` function [12]
Parallel	`@par` function [17]	`MPI_Comm_launch` (this work)

Table 2: Categorization of task types in Swift/T.

instance on the required number of processes. Because the Cram program would be used instead of the user's application, it would be able to run instances of different executables, thus effectively enabling MPMD.

Since an application executed using `MPI_Comm_launch` can fail without causing the failure of its parent application, Cram would immediately support fault tolerance. Additionally, failed instances would not make other instances abort.

A Cram executable relying on `MPI_Comm_launch` could also implement scheduling techniques to run a set of instances on any number of processes, provided that this number is at least as large as the largest instance to schedule. This would allow for a better resource utilization.

Moreover, because Cram would not need to overload all the MPI functions to catch and replace `MPI_COMM_WORLD`, the launched instances (as well as Cram itself) could use the PMPI API for other purposes such as communication and I/O tracing.

One could note that Cram's motivation was to prevent the creation of hundreds of thousands of processes when calling series of `mpiexec`, and that our proposed solution using `MPI_Comm_launch` also creates new processes. However, such process creation is spread across all resources allocated to the job, instead of being located at the frontend.

All these considerations motivate the addition of `MPI_Comm_launch` as a way to simplify packing many instances of potentially multiple executables as a single MPI job.

4.2 Swift/T workflows

Swift[9] [12] is a programming language to support massively scalable compositional programming. It has implicitly parallel data flow semantics, in which all statements are eligible to run concurrently, limited only by the data flow. Swift emphasizes a hierarchical programming model, in which *leaf tasks* linked to external libraries, programs, and scripts in other interpreted languages [13, 15] execute concurrently, coordinated by logic expressed in Swift code. Swift is typically used to express scientific workflows [19], controlling execution of relatively large tasks (seconds to hours); however, its high performance (1.5 billion tasks/s on 512K cores [1]) allows it to be used as a high-performance computing language as well.

The Swift/T implementation [1, 16] translates Swift scripts into MPI programs. Swift/T programs run on the Turbine runtime [14], which implements a small number of data-flow primitives that enable Swift semantics on a scalable system without bottlenecks. Turbine is based on the Asynchronous Dynamic Load Balancer (ADLB) [11], a scalable master-worker system based on MPI.

[9]http://swift-lang.org

4.2.1 The Swift/T task model. Swift/T has multiple ways of invoking a user code as tasks, as categorized in Table 2. The traditional and most common method in practice inherits from the legacy of Swift as a grid workflow language. This consists of putting the command required to call the executable in an `app` leaf function. For example, Listing 2 shows how to wrap the `cat` command into a function that Swift can call. This method allows *a single worker* to call the command. Swift/T extends this model with the ability to call into in-memory, embedded script language interpreters that are optionally compiled into the Swift/T runtime. Using this, for example, a user can wrap C/C++/Fortran code with Python using SWIG [3] or `f2py`[10] and invoke it very conveniently and efficiently via a Swift/T function. Swift/T currently supports Python, Tcl, Julia, and JVM language (JavaScript, Groovy, Scala, and Clojure) interfaces.

```
app (file out) cat (file inputs[]) {
  "/bin/cat" inputs @stdout=out
}
```

Listing 2: Wrapping the `cat` command into a Swift function.

Parallel tasks. Enabling the executable to run on multiple workers incurs a higher development overhead. First, the executable has to be rewritten as a library, exposing an entry function whose first argument is the communicator to use in place of `MPI_COMM_WORLD`. The developer should make sure that the application does not use `MPI_COMM_WORLD` but uses the provided communicator instead. Additionally, any use of global variables in the executable becomes dangerous. The second step is to generate a Tcl interface to this function that calls the Turbine Tcl API to retrieve input data and store the results. This is not a trivial task and benefits from the help of a Swift developer.

When it comes to calling external programs, the main limitation of Swift/T is its lack of support for fault tolerance. If a command wrapped in an *app* leaf function returns a nonzero exit code to Swift/T, the execution of the Swift/T workflow will abort, returning that exit code (a typical approach is to handle such faults in a wrapper shell script). As for C functions invoked as libraries, any failure inside these functions will make the entire workflow abort as well.

4.2.2 Swift/T with MPI_Comm_launch. We constructed an interface for `MPI_Comm_launch` accessible from Swift/T workflows called `launch`. This function has the following prototype:

```
(int status) launch(string cmd, string argv[])
```

It can be called in a Swift script as follows:

```
c = @par=32 launch("/path/to/mpi/program", args);
```

This script will launch the desired MPI program on 32 workers, with the provided arguments, and store the program's exit code in the variable `c`.

The `launch` function is different from calling an `app` leaf function that embeds `mpiexec` (or a bash script calling `mpiexec`). The latter would create new processes on targeted nodes,

[10]https://docs.scipy.org/doc/numpy-1.10.0/f2py

even though Swift/T workers are running on those nodes and could potentially run other tasks. As of today's Swift/T API, adding `launch` was the only way to run the parallel program *inside* the workers. It could not have implemented with currently available functions of the MPI standard, in particular because `MPI_Comm_spawn` creates new MPI processes instead of using the workers.

Compared with rewritting executables as libraries to enable them to run as parallel tasks of a Swift workflow, our `launch` function completely alleviates the development overhead required to call external programs in parallel from a Swift/T script. This use case again advocates for the adding `MPI_Comm_launch` to the MPI standard.

We also provided a `launch_turbine` function that, instead of launching an MPI program, launches another Swift/T workflow. This function allows to treat a subworkflow as a single task from the point of view of the parent workflow and is therefore very useful to prioritize subtasks and enforce execution locality.

5 EVALUATION

In this section, we assess the benefits of `MPI_Comm_launch` in the context of a real Swift/T workflow: CODES-ESW.[11] This workflow aims to perform ensemble simulations of high-performance networks. It was built to enable design-space exploration of collective algorithms on various network topologies. We also use a synthetic workflow to evaluate performance in terms of launch time.

5.1 CODES-ESW workflow

At its core, CODES-ESW relies on CODES [7], an HPC network simulator based on ROSS [6], a parallel discrete-event simulator that executes on multiprocessor systems and supercomputers. CODES can simulate networks such as torus, dragonfly, fattree, or slimfly. It relies on the CoRtEx library[12] to simulate MPI collective algorithms. These algorithms are written in Python.

CODES-ESW aims to run many instances of CODES, varying input parameters such as link bandwidth, number and size of messages, and size of the network. Each execution of CODES is preceded by the execution of a Python script that generates the required input file. When an instance of CODES completes, a shell script is called to clean up the resulting files and extract useful statistics.

CODES instances may sometimes fail for various reasons. It may run out of memory, in which case a call to `malloc` will fail, making the instance fail. In this situation the task could be restarted on more nodes, to benefit from larger memory. CODES may call `MPI_Abort` if the simulation runs into an incorrect state, in which case the error should be report to the user so that trace files could be examined. In all these cases, CODES-ESW should be able to restart submitting instances, until all instances have successfully run to completion or failing tasks have been reported to the user.

We implemented two versions of CODES-ESW. The **library version** relies on Swift/T's leaf functions implemented in C/Tcl: CODES is recompiled as a library that gets called from Swift/T. The **launch version** relies on the Swift/T-level `launch_turbine` function that itself internally relies on the new `MPI_Comm_launch`. This second version launches a subworkflow for every instance of CODES. We compare both versions, first in terms of development effort, then in terms of performance in the presence of faults.

5.2 Qualitative evaluation

We wrote both versions of our workflow starting from a common version that was missing only the CODES invocation. It took three weeks for a Swift/T specialist and a CODES specialist to implement the library version. It raised a plethora of challenges and issues that we expect many developers would encounter with other simulations.

CODES as a function: CODES is initially an executable. We had to replace its `main` function with a function that could be called from Swift/T. This also involved writing a Tcl interface to this function.
Global variables: Both ROSS and CODES include many global or static variables spread in many files. These global variables need to be reset to their original value before CODES runs again. Otherwise, some incorrect values pollute subsequent runs.
MPI communicators: The ROSS and CODES systems use `MPI_- COMM_WORLD` for communication. Yet inside a Swift/T workflow all communications have to go through a custom communicator gathering only workers involved in a particular task. `MPI_COMM_WORLD`, which gathers all workers plus the ADLB scheduler, should not be used.
Standard output: CODES outputs its data in files, but also on the standard output and standard error. To prevent mixing output from concurrently running CODES instances, we had to hard-code standard output redirections directly in C inside CODES.
Python interpreters: Both Swift/T and CODES use a Python interpreter. While we could not instantiate isolated interpreters for each CODES instance and for Swift/T, we had to make sure the only interpreter that exists in the memory of each Swift/T worker is properly used by CODES instances and that CODES instances do not interfere with one another at this level.
Memory leaks: The biggest challenge consisted of finding all memory leaks in CODES. Indeed, when CODES runs as an executable, it does not matter if a segment of memory allocated at the beginning for the purpose of storing a large data structure is not freed at the end. When running multiple instances one after the other from the same program, however, memory leaks accumulate and eventually cause the entire workflow to abort.

In contrast, the **launch version** of CODES-ESW, which relies on `MPI_Comm_launch`, required barely an hour of development. Furthermore, it does not present any of the aforementioned issues. Thanks to our `launch` and `launch_turbine` functions, no Tcl interface is required. There is no need to change a single line of CODES, since CODES run as an executable. There also is no need to reset global variables, change the communicator, fix memory leaks, or solve any issue related to Python interpreters.

This experience in developing two versions of the same workflow is what motivated us to propose the addition of

[11] https://bitbucket.org/mdorier/codes-esw
[12] https://xgitlab.cels.anl.gov/mdorier/dumpi-cortex

MPI_Comm_launch in the MPI standard. Discussions with other researchers trying to simplify ensemble simulations with their own code led us to the conclusion that there is a broader need for such a functionality, as they also face the same issues related to global variables, memory leaks, and more generally the lack of resource isolation that comes with rewriting an executable as a library function.

5.3 Quantitative evaluation: fault tolerance

The goal in this section is to show that MPI_Comm_launch enables task-level fault tolerance and to quantify the benefits of such fault tolerance compared with restarting the entire workflow whenever one task fails.

Experimental methodology. We ran our experiments on 25 nodes of the *parasilo* cluster of Grid'5000.[13] Each node is a Dell PowerEdge R630 with 16 cores and 128 GB of memory. These nodes are connected through a 10 Gigabit Ethernet network. We used 1 node to run Swift/T's ADLB scheduler and 24 nodes to run the workflow's tasks. Although each instance of CODES could be deployed on several nodes, we ran them on one node here for the sake of simplicity. For each instance of CODES started by the workflow, we report its start time and end time. We do not monitor other tasks of the workflow such as those that create the input files for each CODES instance, since these tasks' run time is negligible compared with the run time of CODES. The run time of the workflow is the difference between the end time of the last task and the start time of the first task.

Fault injection. We injected faults by "corrupting" one of the core functions of ROSS: tw_event_new. This function is repeatedly called by CODES to create its network events. When the task starts, a random number R is drawn from a uniform distribution in the range $[0, 2^X]$ (for some chosen values of X). R is then the number of time tw_event_new will succeed before causing a segmentation fault. Depending on input parameters, a CODES instance will call tw_event_new from a hundred times to ten million times.

Arguably, other fault injection methods and distributions could be used (such as issuing a Bernoulli trial at every call to tw_event_new). However, the pattern of faults is linked to their causes (for example, faults induced by a lack of available memory do not follow a memoryless process, unlike faults caused by radiation-induced memory corruption). The goal in this evaluation is simply to illustrate the workflow's reaction to failures in one example of failure pattern.

Workflow configuration. We configured the workflow to run 1,152 instances of CODES. Each instance executes the simulation of a series of broadcast operations using a binomial tree algorithm on a torus network. We varied a number of input parameters of these simulations: the number of nodes participating in the broadcast, the link bandwidth, the amount of data sent, the number of time the broadcast is repeated, the buffer size in routers, and the network packet size. When injecting failures, we used the fault injection parameter X of 26, 27, 28, and 29. We found that 26 is the lowest value that allows failures to have a noticeable impact on the run time while still keeping this run time reasonably low for each execution of the workflow (up to 10 minutes). Each execution of the workflow was repeated 5 times with different seeds for the random number generator.

Results. Figure 1 shows the timeline of two executions of the workflow when setting the fault injection parameter to 26. Figure 1(a) corresponds to the library version of the workflow. In this version, any task fault causes the entire workflow to abort, so all tasks must be restarted. Red tasks represent tasks in which a fault was injected, leading them to fail. Yellow tasks represent tasks that aborted as a result of another task failing. Failed and aborted tasks must be restarted. Figure 1(b) corresponds to the launch version of the workflow. We can see that in this version, the failure of an task does not cause the entire workflow to abort.

Figure 2 shows the run time of both versions of the workflow as a function of the fault injection parameter. It reports the median over 5 runs for each configuration, along with the minimum and maximum as error bars. We can see that in the launch version, the run time is barely affected by faults (a few seconds of overhead not visible in the figure). In the library version, however, a high number of faults dramatically increases the run time. Overall, even without faults, the launch version performs better than the library version.

Figure 3 shows the number of tasks that failed during the execution of each version of the workflow. This does not include the tasks that have aborted as a result of a fault in another task. For a fault injection parameter of 26, we see that although the method used to inject faults is the same in both workflows, we obtain a higher number of task failures. This is due to the fact that in the library version, a task failure aborts many other tasks. These tasks have to be restarted, which gives them a second opportunity to fail, and so on.

5.4 Quantitative evaluation: performance

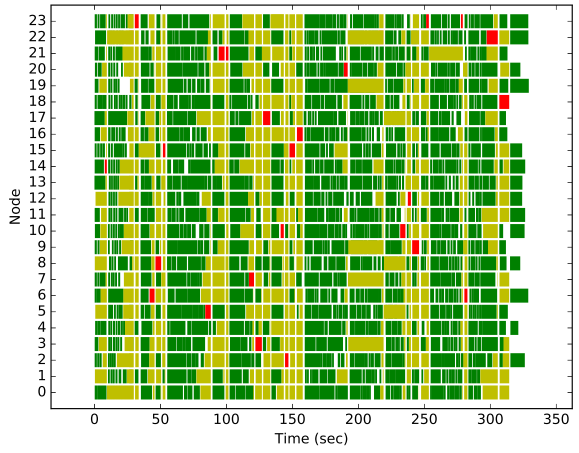
In this section, we evaluate the ability of our system to launch many parallel MPI tasks inside a workflow. Note that this is an evaluation of our prototype, which is in no way optimally implemented, as explained in Section 3.3.

Experimental methodology. We ran our experiments on the *Blues* cluster at Argonne National Laboratory.[14] In the queue batch that was used here, each node contained a Sandy Bridge Xeon E5-2670 at 2.6GHz with 16 cores and 64 GB of memory. These nodes are connected through a QLogic QDR InfiniBand Interconnect (fat-tree topology). We used 1 node to run Swift/T's ADLB scheduler and the remaining nodes to run the workflow tasks. Thus, in our largest tests, $256 \times 16 = 4096$ cores were available to the workflow.

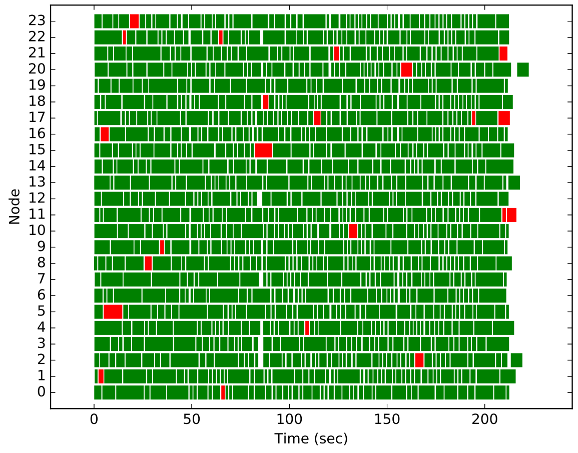
Workflow pattern: Fixed-size tasks. In the the first set of measurements, we used fixed-size parallel tasks. Fixed-size tasks are scripted as shown in Listing 3. On line 1, the integer P is set to 1, and used in the @par annotation to run the given task on that many MPI processes (MPI_Comm_size(MPI_COMM_WORLD) == P). On line 2, the integer N sets the number of loop iterations to that number; in this case, there are enough tasks for each worker to perform 10. On line 7, the new launch() feature runs the given

[13]https://www.grid5000.fr

[14]http://www.lcrc.anl.gov/about/blues



(a) Workflow version 1: tasks running as functions



(b) Workflow version 2: tasks running as MPI programs

Figure 1: Timeline of the workflow's execution (1,152 tasks completed) by using both approaches: tasks as function, and tasks as isolated MPI programs. Green tasks completed successfully. Red tasks were intentionally targeted for failure (fault injection parameter set to 2^{26} here) and were later restarted. Yellow tasks aborted because another task failed and were restarted as well.

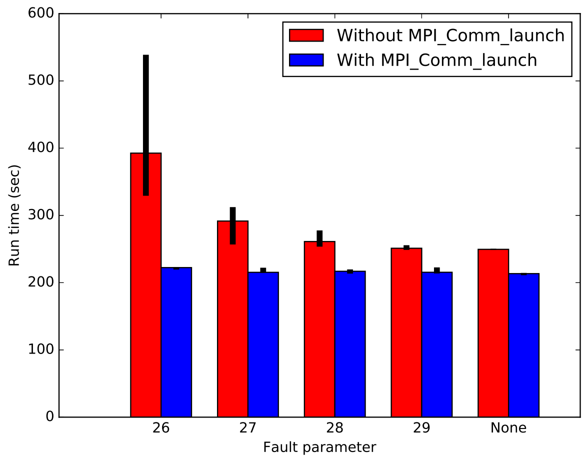


Figure 2: Run time of the workflows under different values of the fault injection parameter. Median over 5 runs. Error bars correspond to minimum and maximum.

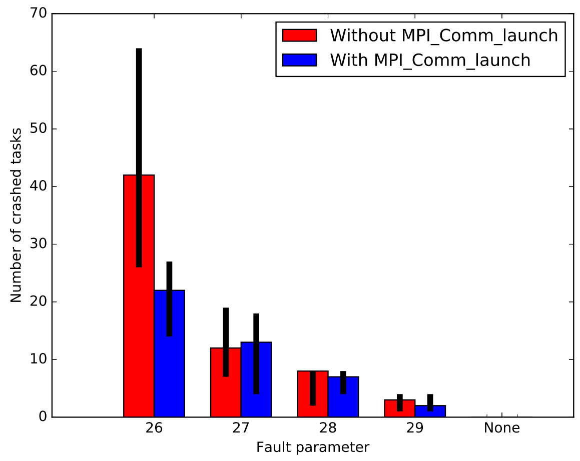


Figure 3: Number of task failures as a function of the fault injection parameter for the two implementations of the workflow.

`program` on `P` processes with the given input `args`. On line 8, the exit code is then checked for an error condition. The user executable program `example.x` simply performs the MPI `Init` and `Finalize` operations, and exits with code 0. (The program is modified in our final case to sleep for 1 second.) The only communication in the system, once running, is that workers must retrieve work from the Swift/T ADLB server (1 task at a time).

Results: Fixed-size tasks. We ran the fixed-size task workflow on successively larger allocations on *Blues*. The task rate is obtained by dividing the total number of tasks by the total wall clock makespan time. Thus, the performance is penalized by any straggling tasks.

For single process tasks, `P`=1. The user program is ideally a 0-second (0s) task, so all time is consumed by system overheads. Task rate results are shown in Figure 4. As shown, the task rate increases approximately linearly with the number of nodes, indicating that the system overheads are dominated

```
P = 1;
N = turbine_workers() * 10;
program = "example.x";
printf("swift: launching: %s", program);
foreach i in [0:N-1] {
  args = ["abc", "defg"];
  exit_code = @par=P launch(program, args);
  if (exit_code != 0) {
    printf("The launched application failed!");
}}
```

Listing 3: Workflow script for fixed-size tasks.

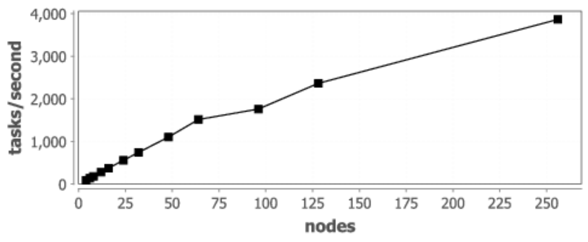


Figure 4: Task rate for 0s 1-process tasks.

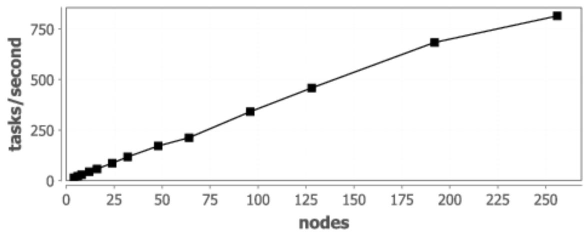


Figure 5: Task rate for 0s 4-process tasks.

by local operations (forking and loading `example.x` from the file system). In our largest case, 2,865 single-process tasks were launched per second across 256 nodes.

For the 4-process tasks, the workflow script was modified so that `P`=4. In this case, launching `example.x` involves coordinating 4 nodes, both by Swift/T and internally by `mpiexec` and `example.x` itself. Task rate results are shown in Figure 5. As shown, the task rate increases approximately linearly with the number of nodes, indicating that the system overheads are dominated by operations local to the task. In our largest case, 815 4-process tasks were launched per second across 256 nodes. This means that extremely fine-grained MPI tasks with sub-second total run times can be coupled together by our system into a workflow application at cluster scale.

Workflow pattern: Mixed-size tasks. In the first set of measurements, we used mixed-size parallel tasks. Mixed-size tasks are scripted as shown in Listing 4. In these scripts, the task parallelism `P` varies from 1 to half the system size. The values of `P` step through powers of 2 (although this is not required by the programming model). On line 1, the integer `c` is set to the $log_2()$ of the system size, this is used to step through the powers of 2 with the parallel loop written on line 7. The loop index `j` is exponentiated on line 8 to obtain the `P` value that is used to launch the parallel task on line 9. In the performance

```
1  c = floor(log2(turbine_workers()));
2  N = c * 10;
3  printf("W=%i c=%i N=%i tasks=%i",
4         turbine_workers(), c, N, c*N);
5  program = "example.x";
6  foreach i in [0:N-1] {
7    foreach j in [1:c] {
8      P = 2**(j-1);
9      exit_code = @par=P launch(program, a);
10     if (exit_code != 0) ...
11 }}
```

Listing 4: Workflow script for mixed-size tasks.

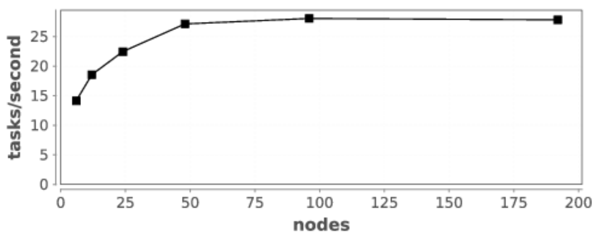


Figure 6: Task rate for 0s mixed-process tasks.

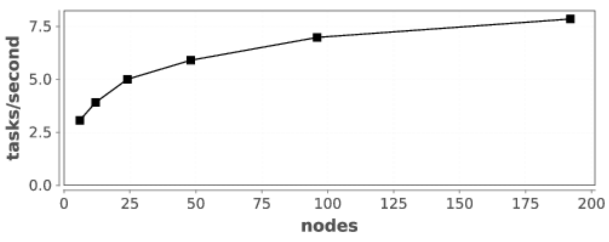


Figure 7: Task rate for 1s mixed-process tasks.

results, we use numbers of workers that are between powers of 2 (6, 12, 24, ...) to demonstrate the ability of the system to put small tasks in the resulting schedule gaps.

Results for 0s mixed-size tasks are shown in Figure 6. In our biggest case, we launched 28 tasks/second across 192 nodes of *Blues*. In accordance with the script logic, the largest tasks in this workflow ran on 64 nodes. Performance seems to level off at that point, probably due to the ability of the cluster to launch 64 task jobs at that rate. This is lower than the rate for the 1 and 4 process workloads above, but still enables workflows to use tasks with subsecond runtimes. Note that while the task rate apparently begins to dip, the amount of work per task is increasing with the node count; this workflow is dominated by launching 64-node MPI tasks.

Results for 1s mixed-size tasks are shown in Figure 6. In these cases, `example.x` sleeps for 1 second before calling `MPI_Finalize` and exiting. In our biggest case, we launched 8 tasks/second across 192 nodes of *Blues*. In accordance with the script logic, the largest tasks in this workflow ran on 64 nodes. Performance does not seem to level off yet at that point, probably because with the 1 second delay, the system still has capacity to start/stop additional tasks per unit time. This demonstrates that our system enables workflows to use tasks with short (∼1 second) runtimes on systems like *Blues*, even with a wide mix of task sizes.

6 DISCUSSION

In this section, we discuss the semantics of MPI_Comm_launch and potential additional functions following similar semantics.

Launching multiple applications. Just as MPI_Comm_spawn has its MPI_Comm_spawn_multiple to spawn processes from multiple executables, we may be tempted to provide an MPI_Comm_launch- _multiple. It is debatable, however, whether this function would be useful. One can argue that allowing each process calling MPI_Comm- _launch to provide a different command and different arguments would be sufficient to effectively start an MPMD subapplication.

MPI_Comm_launch_multiple can be justified if we want to explicitly forbid MPI_Comm_launch from accepting different commands and arguments for each process; the command and arguments in MPI_Comm_launch would be relevant only in the root process.

Communicating outside a task. If communication is required between processes inside and outside a launched task, functions such as MPI_Comm_connect/accept/join can be used, as well as other communication mechanisms such as local storage, burst buffers, and shared memory.

We note that while MPI_Comm_spawn gives the calling processes an intercommunicator allowing them to communicate with the spawned processes, MPI_Comm_launch blocks the calling processes during the execution of the child application. Hence there is no reason to establish a communication mechanism between the child application and the calling processes. The latter will not be able to perform any communication during the lifetime of the child application anyway.

One could argue that the child application may want to communicate with the rest of the processes in the parent application that are not paused (those that did not call MPI_Comm_launch). This notion "the rest of the processes" cannot be well defined, however, given that an MPI application always has the possibility of *spawning* new processes (with MPI_Comm_spawn) or to connect to other processes (with MPI_Comm_connect/accept).

One possibility would be to offer an alternative function, MPI_Comm- _launch_and_connect, with the following prototype:

```
int MPI_Comm_launch_and_connect(char* cmd, char* argv[],
  MPI_Info info, int root, MPI_Comm comm,
  int launch, MPI_Comm* newcomm, int* status)
```

On processes where launch $= 0$, this function would have the semantics of MPI_Comm_spawn, and newcomm would become a new intercommunicator connected to the child application. The cmd, argv, info, and status arguments would not be used on these processes. On processes where launch $= 1$, this function would have the semantics of an MPI_Comm_launch, and newcomm would be left unused. However, we think that such an asymmetric semantics should be avoided altogether.

We advocate for the use of MPI_Comm_accept/connect to connect child applications, or processes of a child application with other processes in its parent application.

Note on Multithreading. When MPI is used in a multithreaded environment (that is, MPI_THREAD_MULTIPLE is used), only the thread calling MPI_Comm_launch is blocked. Other threads of the parent application continue running. In this particular case (which we do not discuss further in this paper), both the child and parent application processes continue running, but the number of running *processing entities* (threads of processes) still remains constant. It is worth noting that on most supercomputers, the OS limits the use of threads to one per core, for the same reason they limit the number of processes to one per core. Our MPI_Comm_launch function still ensures one running processing entity per core at all time.

7 CONCLUSION

We have proposed MPI_Comm_launch to enable an MPI application to run inside another MPI application. This function overcomes the limitations of existing ways of implementing MPMD programs using MPI. In particular, we illustrated its advantages in two practical use cases, using Swift/T and LLNL's Cram. The evaluation of this functionality with a real HPC workflow, CODES-ESW, showed that MPI_Comm_launch (1) enables faster development (from 3 weeks of development using Swift/T existing interface to barely an hour using an interface based on MPI_Comm_launch), and (2) makes the workflow fault tolerant at the task level, considerably improving performance and resource utilization. We plan to propose a more efficient implementation of MPI_Comm_launch inside the process management part of MPICH, first as an MPI extension (MPIX_Comm_launch), and to propose this new function to the MPI Forum.

ACKNOWLEDGMENTS

We thank Pavan Balaji and Rajeev Thakur for their insights about the MPI standard and its implementations, Misbah Mubarak for her help setting up CODES simulations, and the ROSS developers for quickly reacting to our feature requests. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357 We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. 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] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. 2014. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC'14*.

[2] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. 2010. PMI: A scalable parallel process-management interface for extreme-scale systems. In *European MPI Users' Group Meeting*. Springer, 31–41.

[3] D.M. Beazley. 2003. Automated scientific software scripting with SWIG. *Future Generation Computer Systems* 19, 5 (2003), 599–609. DOI:http://dx.doi.org/10.1016/S0167-739X(02)00171-1

[4] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. 2012. A proposal for User-Level Failure Mitigation in the MPI-3 standard. *Department of Electrical Engineering and Computer Science, University of Tennessee* (2012).

[5] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 characterization of petascale I/O workloads. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 1–10.

[6] Christopher D Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *J. Parallel and Distrib. Comput.* 62, 11 (2002), 1648–1669.

[7] Jason Cope, Ning Liu, Sam Lang, Phil Carns, Chris Carothers, and Robert Ross. 2011. CODES: Enabling co-design of multilayer exascale storage architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*, Vol. 2011.

[8] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. 2016. Damaris: addressing performance variability in data management for post-petascale simulations. *ACM Transactions on Parallel Computing (ToPC)* (2016). http://dl.acm.org/citation.cfm?id=2987371

[9] John Gyllenhaal, Todd Gamblin, Adam Bertsch, and Roy Musselman. 2014. Enabling high job throughput for uncertainty quantification on BG/Q. *ser. IBM HPC Systems Scientific Computing User Group (SCICOMP)* (2014).

[10] John L. Hennessy and David A. Patterson. 2002. *Computer Architecture: A Quantitative Approach (3rd Edition)*. Elsevier Science.

[11] Ewing L. Lusk, Steve C. Pieper, and Ralph M. Butler. 2010. More scalability, less pain: a simple programming model and its implementation for extreme computing. *SciDAC Review* 17 (2010).

[12] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. 2011. Swift: a language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011). DOI:http://dx.doi.org/10.1016/j.parco.2011.05.005

[13] Justin M. Wozniak, Timothy G. Armstrong, Daniel S. Katz, Michael Wilde, and Ian T. Foster. 2014. Toward computational experiment management via multi-language applications. In *DOE Workshop on Software Productivity for eXtreme scale Science (SWP4XS)*.

[14] Justin M. Wozniak, Timothy G. Armstrong, Ketan Maheshwari, Ewing L. Lusk, Daniel S. Katz, Michael Wilde, and Ian T. Foster. 2013. Turbine: a distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae* 28, 3 (2013).

[15] Justin M. Wozniak, Timothy G. Armstrong, Ketan C. Maheshwari, Daniel S. Katz, Michael Wilde, and Ian T. Foster. 2015. Interlanguage parallel scripting for distributed-memory scientific computing. In *WORKS '15: Proceedings of the 10th Workshop in Support of Large-Scale Science*.

[16] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. 2013. Swift/T: Scalable data flow programming for distributed-memory task-parallel applications. In *Proc. CCGrid*.

[17] Justin M. Wozniak, Tom Peterka, Timothy G. Armstrong, James Dinan, Ewing L. Lusk, Michael Wilde, and Ian T. Foster. 2013. Dataflow coordination of data-parallel tasks via MPI 3.0. In *Proc. Recent Advances in Message Passing Interface (EuroMPI)*.

[18] Nicholas J Wright, Wayne Pfeiffer, and Allan Snavely. 2009. Characterizing parallel scaling of scientific applications using IPM. In *The 10th LCI International Conference on High-Performance Clustered Computing*. Citeseer, 10–12.

[19] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Ioan Raicu, Tiberiu Stef-Praun, and Mike Wilde. 2007. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. Workshop on Scientific Workflows*.