



MPI jobs within MPI jobs: A practical way of enabling task-level fault-tolerance in HPC workflows[☆]

Justin M. Wozniak^{a,*}, Matthieu Dorier^a, Robert Ross^a, Tong Shu^a, Tahsin Kurc^{b,d}, Li Tang^c, Norbert Podhorszki^d, Matthew Wolf^d

^a Argonne National Laboratory, 9700 S Cass Ave., Lemont IL 60439, United States

^b Stony Brook University, Stony Brook, NY 11794, United States

^c Brookhaven National Laboratory, Upton, NY 11973, United States

^d Oak Ridge National Laboratory, Oak Ridge, TN 37831, United States

HIGHLIGHTS

- A novel MPI-compatible function to support fault tolerance in HPC workflows.
- Support for this function in a high-level scripting language.
- A novel workflow coordination model for coordinated bundles of MPI tasks.
- Demonstration of these concepts in multiple applications.

ARTICLE INFO

Article history:

Received 31 March 2018

Received in revised form 26 February 2019

Accepted 6 May 2019

Available online 17 May 2019

Keywords:

MPI

MPMD

Workflows

Ensemble simulations

Swift/T

Cram

MPI_Comm_launch

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.

© 2019 Published by Elsevier B.V.

1. Introduction

Multiple-programs-multiple-data (MPMD) applications are becoming more and more common in today's high-performance computing (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 programming 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.

For example, consider a case in which a researcher desires to study a materials simulator S and the impact of simulated temperature T on noisy simulation outputs Y . The user runs some number N of simulations for each temperature over some domain of temperatures, and averages the N results. Many potential workflows could be performed, such as running higher accuracy simulations using more processors when the simulation is very

[☆] The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. <http://energy.gov/downloads/doe-public-access-plan>.

* Corresponding author.

sensitive to temperature (high finite differences $\frac{dy}{dt}$), or adaptive refinement between samples when dy is high. In this workflow, a great many parallel jobs can be run concurrently, and most input temperatures and processor counts are not determined until the initial samples have completed. Thus, the user needs an efficient, easily programmable, and portable way to launch many parallel calculations based on emerging runtime data.

Solutions within the current MPI standard and/or its implementations include using MPMD mode in the MPI launcher's command line (`mpirexec`, `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.

`MPI_Comm_launch` allows the WMS to be an MPI application that runs other MPI applications as tasks in a perfectly isolated manner.

Example use cases. Ensemble simulations can be written very effectively using `MPI_Comm_launch`. The parent application is a small harness code that splits its set of processes and have groups of processes call `MPI_Comm_launch` to execute instances of the simulation in parallel, making sure that any error in one

of the instances will not propagate to the other. In coupled simulation models (e.g., simulations comprising of a land, atmosphere, and ocean component), the parent application can be in charge of launching the components on the right processes as child applications, setting up non-MPI communication mechanisms beforehand (e.g. through a coupler). Any failure of one of the component is reported to the parent application, which can restart the failed component without affecting the other components.

While WMS in the cloud computing area are usually built with task-level fault-tolerance in mind, MPI programs that use `MPI_Comm_spawn` or invoke external code via library interfaces are not fault tolerant with respect to process failure. A workflow hand-written using MPI or using an MPI-based WMS such as Swift/T [1,2] or Chiron [3] is thus not fault tolerant even with respect to simple application-level job crashes. Thus, supporting task-level fault tolerance in HPC workflows through MPI is critical.

Contributions. 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 3.4).
- We exemplify the use of `MPI_Comm_launch` in two contexts: the Swift/T parallel scripting language and its associated ADLB-based [4] runtime, and the Cram library from Lawrence Livermore National Laboratory (LLNL), designed to run ensemble simulations (Section 4).
- We describe a complex in situ workflow in molecular dynamics and Voronoi tessellation, and describe advanced MPI workflow features to handle this workflow structure and layout complexities of this and similar cases.
- 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).

This paper extends a previously published workshop paper [5] in the following ways: Besides minor revisions following the advice of WORKS reviewers, the main addition to this paper consists of a new use case: code coupling with LAMMPS and Voro++ as a motivating application. LAMMPS is a widely used molecular dynamics simulation framework. Voro++ is a library carrying out 3D Voronoi tessellation. Contrary to the use cases presented in our WORKS paper (Swift/T workflow and Cram ensemble simulations), where the tasks launched have no dependencies between one another and do not communicate with one another, LAMMPS+Voro illustrates a situation where communication between independently launched tasks is established through a third-party middleware, namely ADIOS. We designed an additional workflow capability that enables such additional workflow structure and is fully compatible with the previously presented work. These additions may be found in the presentation of the LAMMPS-Voro++ application (Section 4.2) and the new coordinated launch feature (Section 4.4).

2. Background and related work

In this section we first describe 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 [6]. 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 BLCR¹ 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 [7], 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 recommends that the processes only 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 launcher MPMD mode

Although the MPI standard does not define an interface that the process launcher (`mpirun`, `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 `mpirun` command under the cover, with the proper settings to enable each component to run on the appropriate locations.

In other contexts, however, the 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 simulation code 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, the MPI launcher's MPMD mode cannot do this, since they are inherently supposed to start all the provided executables at once. LLNL's Cram library, 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 4.3, 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.

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

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 nodes, 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 Blue Gene/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.

```
1 MPI_Comm childcomm;
2 int color = rankInWorld < sizeOfWorld/2 ? 0 : 1;
3 int key = rankInWorld;
4 MPI_Comm_split(MPI_COMM_WORLD, color, key, &childcomm);
5 if(color) {
6     int status;
7     MPI_Comm_launch("./child", NULL, MPI_INFO_NULL, 0,
8                     childcomm, &status);
9 }
```

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 do not 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 to 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 set up 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. *MPI_Comm_launch* for fault tolerance

The main advantage of *MPI_Comm_launch* lies in its ability to contain faults. If one of the child application processes crashes because a software-related error (e.g., segmentation fault, erroneous computation, I/O error), the parent application will not only continue running, it will be informed of the nature of the fault through the status parameter of *MPI_Comm_launch*, giving it a chance to react appropriately. Additionally, any communication fault in either the child or the parent application will not propagate to the other because they are not sharing a communicator. *MPI_Comm_launch* also contains memory usage in the child application to the duration of its execution. If the child application suffers from memory leaks, these leaks do not contaminate the parent application's memory, and do not propagate beyond the execution of the child.

An example of the use of the status code for fault tolerance and response is shown in Listing 2.

```

1 | int status = -1, max_tries = 3;
2 | for (int try = 0; try < max_tries; try++) {
3 |     MPI_Comm_launch("./child", NULL, MPI_INFO_NULL, 0,
4 |                     childcomm, &status);
5 |     if (status == 0) break;
6 | }
7 | if (status != 0) { // Handle failure...

```

Listing 2: Example C code to respond to faults in child MPI job.

In this example, the child MPI program *./child* is re-run up to *max_tries*=3 times, until it returns status code 0. These faults can be caused explicitly by the application, or induced by bugs (segmentation violations) or other system faults. When the number of tries is exhausted, the loop exits, and the user checks for and handles the resulting failure (line 7). Note that *MPI_Comm_spawn* does not allow for this kind of fault handling.

3.4. Implementation

We implemented a prototype³ of *MPI_Comm_launch* as an MPI extension function, *MPiX_Comm_launch*. This function works as follows (its signature is in the beginning of Section 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 *mpirexec* 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 *mpirexec*, two more processes are created in the node running the *root* process: one shell instance to invoke the command and one *mpirexec* process (plus potentially other processes created by *mpirexec*).

This implementation works on traditional Linux clusters in which process management is not an issue. On supercomputers, however, a different implementation could be required. In particular, we envision a direct use of the Process Management Interface [9] (PMI⁴) to reuse resources of the parent application (such as the KVS space) without calling *mpirexec* 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 includes 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 three practical use cases motivating the addition of `MPI_Comm_launch` to the MPI standard: LLNL's Cram, code coupling in LAMMPS and Voro, and Swift/T workflows.

4.1. LLNL's *cram*

Cram⁵ [10] 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 IPM⁷ [11], or Darshan⁸ [12], 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 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. Code coupling: LAMMPS+Voronoi tessellation

This use case represents scenarios in which codes, which may potentially be developed by different groups, are coupled for on-line analysis. The example here is a coupling of the LAMMPS molecular dynamics simulator framework [13] with a Voronoi tessellation analysis code, Voro++ [14]. LAMMPS is implemented as a classical molecular dynamics code and can be used to model particles at the atomic and continuum scales. It has been used in a variety of application domains for simulation of solid-state materials, soft materials (e.g., biomolecules) and mesoscopic systems. Voro++ is a C++ library designed to carry out efficient 3-dimensional computation of the Voronoi tessellation [15]. It can be used to analyze a system of particles and their statistics, such as particle neighborhood information, cell volumes, and cell surface area. In our case, LAMMPS outputs particle data at user-specified intervals (e.g., at every 10 simulation time step). LAMMPS output consists of the locations and velocities of particles (atoms) in a 3-dimensional mesh at a given time step. The particle data at each output time step is processed for Voronoi tessellation, and a number of statistics are computed for other analyses.

LAMMPS is a parallel simulation framework that employs domain decomposition methods and MPI for message passing for execution on large parallel machines. Voro++ can be set up to run as a multi-threaded library using multiple cores on a computation node to speed up the Voronoi computations. We implemented a multi-threaded MPI program using the Voro++ library for computation of Voronoi tessellations on LAMMPS output. This program employs a master-worker pattern. One MPI process (the master process) reads particle data at each time step of the simulation output and sends it to an idle worker process dynamically. Each worker processes the received data using multiple CPU cores.

LAMMPS and the Voronoi tessellation program are separately developed MPI programs. A typical scientific study in LAMMPS will perform many simulations with different temperatures or other configuration changes, and observe the visual outputs. Each program can be allocated different numbers of nodes in order to optimize resource utilization and reduce overall execution time. Thus, a single Swift/T script is used to run many such invocations as a logical workflow.

In order to enable in situ data exchange between the two programs, we used the ADIOS framework [16,17]. ADIOS provides interfaces and runtime support to create, read/write, and manage very large datasets in self-describing file structures and to enable efficient staging and in-transit manipulation of such structured data. ADIOS also contains a module called Flexpath [18] that

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>.

Table 2
Categorization of task types in Swift/T.

Mode	Task type	
	In-memory	External program
Sequential	Scripting langs [21]	app function [19]
Parallel	@par function [24]	MPI_Comm_launch (this work)

allows parallel applications to exchange data directly over TCP/IP, without delays due to accessing persistent storage.

In our use case implementation, we integrated ADIOS methods with LAMMPS I/O layer so that output from LAMMPS can use ADIOS methods. Similarly, we incorporated ADIOS methods in the Voro++-based Voronoi tessellation program so it could read data from LAMMPS on the fly. With MPI_Comm_launch, our implementation coordinates the concurrent execution of LAMMPS, the Voronoi program and (optionally) the ADIOS staging processes.

In order to carry out on-line analysis, both programs are executed concurrently in MPMD mode and exchange data directly over the network. Thus, in our experimental layout, LAMMPS data is read directly by Voro++ to produce the tessellations, and the LAMMPS data is not stored on disk. Using MPI_Comm_launch we can launch the two programs concurrently without having to pre-allocate resources, as described in Section 4.4.

4.3. Swift/T workflows

Swift⁹ [19] 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 [20,21] execute concurrently, coordinated by logic expressed in Swift code. Swift is typically used to express scientific workflows [22], controlling execution of relatively large tasks (seconds to hours); however, its high performance (1.5 billion tasks/s on 512K cores [2]) allows it to be used as a high-performance computing language as well.

The Swift/T implementation [1,2] translates Swift scripts into MPI programs. Swift/T programs run on the Turbine runtime [23], 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) [4], a scalable master-worker system based on MPI.

4.3.1. 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 shell command required to call the executable in an *app leaf* function. 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 [25] or f2py¹⁰ 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.

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.

4.3.2. 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, 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 external 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 rewriting 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.

4.4. In situ code coupling with coordinated launch

In many cases, such as the LAMMPS-Voro++ use case described previously (Section 4.2), it is desirable to launch multiple tasks in a coordinated fashion. Consider a workflow in which many LAMMPS runs must be performed at differing temperatures, and each must be connected to a Voro++ instance for in situ tessellation. If the workflow system is not informed about the coupling requirements, then each *pair* of LAMMPS and Voro++ could be run at different times in the workflow, making it impossible for them to communicate. An illustration is shown in Figs. 1 and 2.

⁹ <http://swift-lang.org>.

¹⁰ <https://docs.scipy.org/doc/numpy-1.10.0/f2py>.

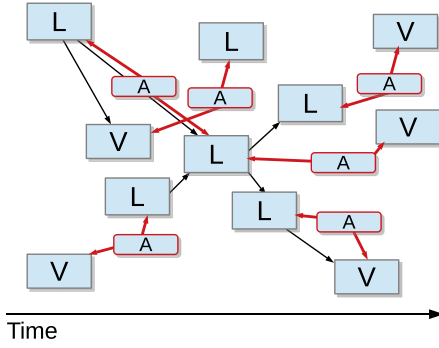


Fig. 1. Depiction of uncoordinated launches. Workflow diagrams. L is LAMMPS, A is ADIOS, V is Vorop++. Additional workflow structure is needed to capture the in situ communication between matching L-A-V instances. Black arrows depict dataflow dependency in the workflow script, red arrows depict bulk in situ data transfer over ADIOS. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

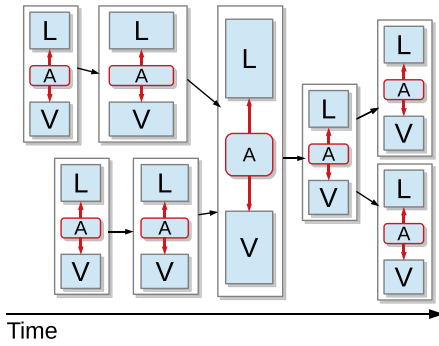


Fig. 2. Depiction of coordinated launches. Workflow diagrams. L is LAMMPS, A is ADIOS, V is Vorop++. Additional workflow structure is needed to capture the in situ communication between matching L-A-V instances. Black arrows depict dataflow dependency in the workflow script, red arrows depict bulk in situ data transfer over ADIOS. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Fig. 1 illustrates a naive user attempt to couple LAMMPS and Vorop++ with pure dataflow. The user is defeated by the load balancing runtime, which does not schedule the task pairs together, meaning the tasks cannot connect with ADIOS and communicate. What is desired is shown in Fig. 2, in which additional structure, shown as bounding boxes around each L-A-V group, is scheduled together (ADIOS may or may not require additional processes to run, depending on the underlying data transfer library).

We added the `launch_multi()` function to Swift/T to support the desired progress model. It has the signature shown in Listing 3.

```

1 // Definition:
2 @par (int status) launch_multi(int procs[], string cmd[],
3                               string argv[] [], string envs[] [],
4                               string color_setting="");
5 // Invocation:
6 int procs[] = [ 6, 2 ];
7 string cmd[] = [ "program1", "program2" ];
8 ... // Other settings ...
9 @par=sum(procs) launch_multi(procs, cmd, ...);

```

Listing 3: Definition and invocation of `launch_multi()`.

For each index i in `procs`, `procs[i]` is the number of MPI processes, `cmd[i]` is the program command to run, `argv[i]` is the array of string arguments to that program, and `envs[i]` is the array of string environment variables for that program. `color_setting` is optional and discussed below.

From the perspective of Swift/T, an invocation of `launch_multi()` is *atomic*: it has a finite set of inputs and

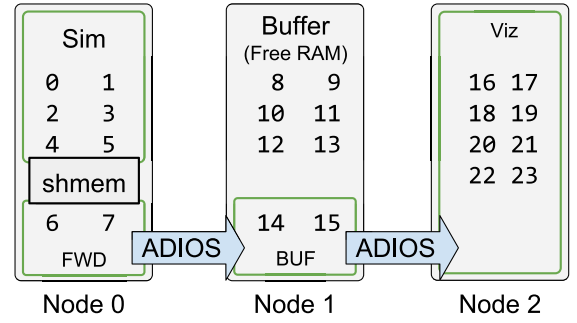


Fig. 3. Rank layout result from Listing 4.

outputs, and is managed as a unit. For example in the case shown in Listing 3, `program1` will have 6 processes, `program2` will have 2 processes, and the programs will be launched together when all inputs are ready and 8 worker processes are available. The user has specified `@par=sum(procs)` to prevent mistakes with the process counts. Providing fewer processes here is detected by Swift/T as a runtime error.

We implemented `launch_multi()` without modification to the `MPI_Comm_launch()` signature. `launch_multi()` is simply a Swift/T `@par` function (cf. Table 2) that performs an `MPI_Comm_split()` and runs `MPI_Comm_launch()` on the resultant subcommunicators.

The MPI Standard provides for “colors,” actually integer categories, as input to `MPI_Comm_split()`. The colors specified to determine how the parent ranks propagate into the new communicator. The default for `launch_multi()` is to simply group them in order, so that nearby ranks in the parent are grouped together in the new communicators. In the case of a more complex data transfer pipeline, a more specific layout can be given, as in the synthetic Listing 4.

```

1 string color_settings_array[] = [ "0-5", "6, 7", // Node 0
2                                "8-13", "14,15", // Node 1
3                                "16-23" ]; // Node 2
4 color_settings = join(color_settings_array, ",");
5 @par=sum(procs)
6 launch_multi(procs, cmds, a, e, color_settings);

```

Listing 4: Invocation of `launch_multi()` with custom color settings.

The resulting layout is depicted in Fig. 3. The user has a simulation `Sim` connection to an ADIOS forwarding component `FWD`. These communicate via shared memory, not seen by Swift/T. Data is transferred from `FWD` to `BUF`, another forwarding component that uses the RAM on a nodes as a buffer and leaves many cores unused. Data from `BUF` is sent to a visualization task `Viz`, that uses all cores on the node and produces graphics.

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.

¹¹ <https://bitbucket.org/mdorier/codes-esw>.

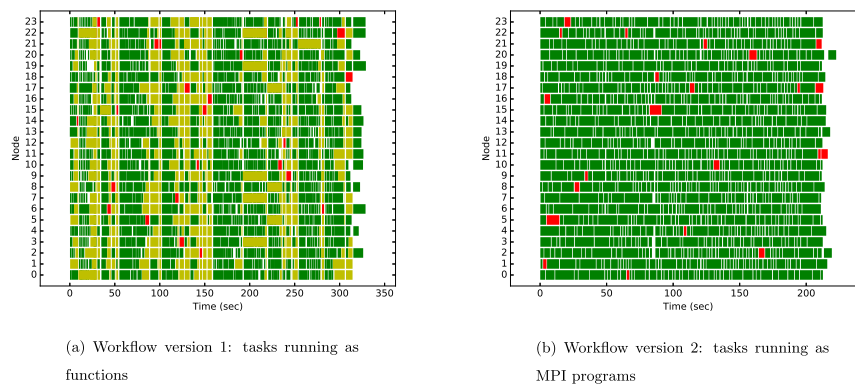


Fig. 4. Timeline of the workflow's execution (1152 tasks completed) by using both approaches: tasks as function, and tasks as isolated MPI programs. Green tasks completed successfully. Red tasks were intentionally targeted for failure (fault injection parameter set to 2^{26} here) and were later restarted. Yellow tasks aborted because another task failed and were restarted as well. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.1. CODES-ESW workflow

At its core, CODES-ESW relies on CODES [26], an HPC network simulator based on ROSS [27], a parallel discrete-event simulator that executes on multiprocessor systems and supercomputers. CODES can simulate networks such as torus, dragonfly, fat tree, or slimfly. It relies on the CoRtEx library¹² 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. They 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 reported to the user so that trace files can 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. The library version raised a plethora of challenges and issues that we expect many developers would encounter with other simulation-based workflows that intend to call parallel applications via a library interface.

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`, does not present any of the aforementioned issues; it simply requires the use of the Swift/T `launch()` function as described in Section 4.3.2. 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

¹² <https://xgitlab.cels.anl.gov/mdorier/dumpe-cortex>.

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 `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 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 times `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 a failure pattern.

Workflow configuration. We configured the workflow to run 1152 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 times 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 min). Each execution of the workflow was repeated 5 times with different seeds for the random number generator.

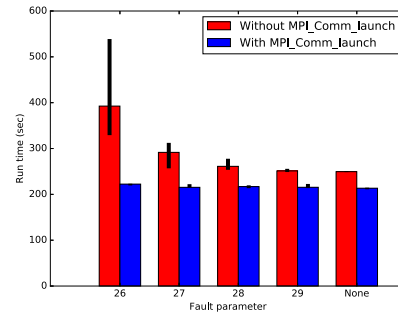


Fig. 5. Run time of the workflows under different values of the fault injection parameter. Median over 5 runs. Error bars correspond to minimum and maximum.

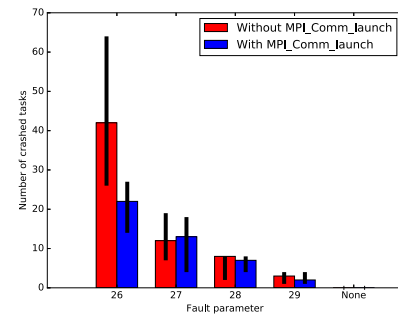


Fig. 6. Number of task failures as a function of the fault injection parameter for the two implementations of the workflow.

Results. Fig. 4 shows the timeline of two executions of the workflow when setting the fault injection parameter to 26. Fig. 4(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. Fig. 4(b) corresponds to the launch version of the workflow. We can see that in this version, the failure of a task does not cause the entire workflow to abort.

Fig. 5 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.

Fig. 6 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 3.4.

¹³ <https://www.grid5000.fr>.

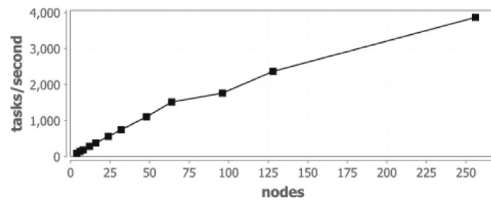


Fig. 7. Task rate for 0 s 1-process tasks.

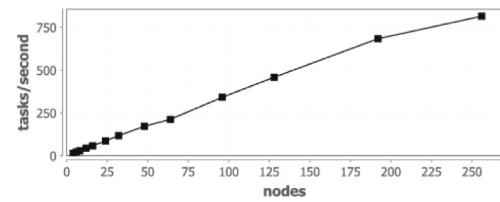


Fig. 8. Task rate for 0 s 4-process tasks.

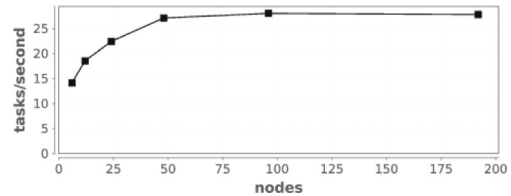


Fig. 9. Task rate for 0 s mixed-process tasks.

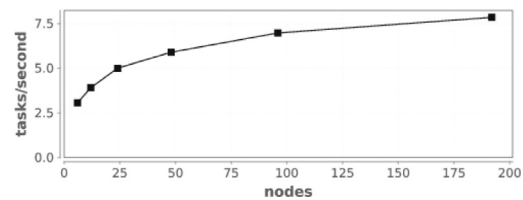


Fig. 10. Task rate for 1 s mixed-process tasks.

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 2.6 GHz 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 first set of measurements, we used fixed-size parallel tasks. Fixed-size tasks are scripted as shown in Listing 5. 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 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 s.) 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).

```

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

```

Listing 5: Workflow script for fixed-size tasks.

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 (0 s) task, so all time is consumed by system overheads. Task rate results are shown in Fig. 7. As shown, the task rate increases approximately linearly with the number of nodes, indicating that the system overheads are dominated by local operations (forking and loading `example.x` from the file system). In our largest case, 2865 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 Fig. 8. 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.

```

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 6: Workflow script for mixed-size tasks.

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 6. 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 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 (see Fig. 10).

Results for 0 s mixed-size tasks are shown in Fig. 9. 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

¹⁴ <http://www.lcrcl.gov/about/blues>.

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 1 s mixed-size tasks are shown in Fig. 10. In these cases, `example.x` sleeps for 1 s 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 s 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 of “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 connecting 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 (`MPICH_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 research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. 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>).

Declaration of competing interest

The authors declared that they had no conflicts of interest with respect to their authorship or the publication of this article.

References

- [1] J.M. Wozniak, T.G. Armstrong, M. Wilde, D.S. Katz, E. Lusk, I.T. Foster, Swift/T: scalable data flow programming for distributed-memory task-parallel applications, in: Proc. CCGrid, 2013.
- [2] T.G. Armstrong, J.M. Wozniak, M. Wilde, I.T. Foster, Compiler techniques for massively scalable implicit task parallelism, in: Proc. SC'14, 2014.
- [3] E. Ogasawara, J. Dias, V. Silva, F. Chirigati, D. de Oliveira, F. Porto, P. Valduriez, M. Mattoso, Chiron: a parallel engine for algebraic scientific workflows, *Concurr. Comput. Pract. Exper.* 25 (16) (2013) 2327–2341.
- [4] E.L. Lusk, S.C. Pieper, R.M. Butler, More scalability, less pain: a simple programming model and its implementation for extreme computing, *SciDAC Review* 17.
- [5] M. Dorier, J.M. Wozniak, R. Ross, Launching MPI applications inside MPI applications, in: Proc. WORKS @ SC, 2017.
- [6] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, third ed., Elsevier Science, 2002.
- [7] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, J. Dongarra, A proposal for user-level failure mitigation in the MPI-3 standard.
- [8] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, L. Orf, Damaris: addressing performance variability in data management for post-petascale simulations, *ACM Trans. Parallel Comput. (ToPC)* (2016) URL <http://dl.acm.org/citation.cfm?id=2987371>.
- [9] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, R. Thakur, Pmi: A scalable parallel process-management interface for extreme-scale systems, in: European MPI Users' Group Meeting, Springer, 2010, pp. 31–41.
- [10] J. Gyllenhaal, T. Gamblin, A. Bertsch, R. Musselman, Enabling high job throughput for uncertainty quantification on BG/Q, ser. IBM HPC Systems Scientific Computing User Group (SCICOMP).
- [11] N.J. Wright, W. Pfeiffer, A. Snively, Characterizing parallel scaling of scientific applications using ipm, in: The 10th LCI International Conference on High-Performance Clustered Computing, Citeseer, 2009, pp. 10–12.
- [12] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 characterization of petascale I/O workloads, in: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, 2009, pp. 1–10.
- [13] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, *J. Comput. Phys.* 117 (1) (1995) 1–19.
- [14] C. Rycroft, Voro++: A three-dimensional voronoi cell library in c++, *Chaos* 19 (4) (2009) 041111.
- [15] G. Voronoï, Nouvelles applications des parametres continus a la theorie des formes quadratiques, *J. Reine Angew. Math.* 133 (1907) 97–178.
- [16] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorski, J.Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al., Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks, *Concurr. Comput. Pract. Exper.* 26 (7) (2014) 1453–1473.
- [17] J.F. Lofstead, S. Klasky, K. Schwan, N. Podhorski, C. Jin, Flexible io and integration for scientific codes through the Adaptable IO System (ADIOS), in: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, ACM, 2008, pp. 15–24.
- [18] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, N. Podhorski, Flexpath: Type-based publish/subscribe system for large-scale science analytics, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014, pp. 246–255, <http://dx.doi.org/10.1109/CCGrid.2014.104>.
- [19] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, I. Foster, Swift: a language for distributed parallel scripting, *Parallel Comput.* 37 (9) (2011) <http://dx.doi.org/10.1016/j.parco.2011.05.005>.
- [20] J.M. Wozniak, T.G. Armstrong, D.S. Katz, M. Wilde, I.T. Foster, Toward computational experiment management via multi-language applications, in: DOE Workshop on Software Productivity for EXtreme Scale Science (SWP4XS), 2014.
- [21] J.M. Wozniak, T.G. Armstrong, K.C. Maheshwari, D.S. Katz, M. Wilde, I.T. Foster, Interlanguage parallel scripting for distributed-memory scientific computing, in: WORKS '15: Proceedings of the 10th Workshop in Support of Large-Scale Science, 2015.
- [22] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, reliable, loosely coupled parallel computation, in: Proc. Workshop on Scientific Workflows, 2007.
- [23] J.M. Wozniak, T.G. Armstrong, K. Maheshwari, E.L. Lusk, D.S. Katz, M. Wilde, I.T. Foster, Turbine: a distributed-memory dataflow engine for high performance many-task applications, *Fundamenta Inform.* 28 (3) (2013).
- [24] J.M. Wozniak, T. Peterka, T.G. Armstrong, J. Dinan, E.L. Lusk, M. Wilde, I.T. Foster, Dataflow coordination of data-parallel tasks via MPI 3.0, in: Proc. Recent Advances in Message Passing Interface (EuroMPI), 2013.
- [25] D. Beazley, Automated scientific software scripting with SWIG, *Future Gener. Comput. Syst.* 19 (5) (2003) 599–609, [http://dx.doi.org/10.1016/S0167-739X\(02\)00171-1](http://dx.doi.org/10.1016/S0167-739X(02)00171-1).
- [26] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, R. Ross, Codes: Enabling co-design of multilayer exascale storage architectures, in: Proceedings of the Workshop on Emerging Supercomputing Technologies, Vol. 2011, 2011.
- [27] C.D. Carothers, D. Bauer, S. Pearce, ROSS: A high-performance, low-memory, modular time warp system, *J. Parallel Distrib. Comput.* 62 (11) (2002) 1648–1669.



supercomputers such as Theta and Summit.

Matthieu Dorier is a software development specialist at Argonne National Laboratory since 2017. He received his Ph.D. in Computer Science from Ecole Normale Supérieure de Rennes (France) in December 2014, and followed with postdoc at Argonne National Laboratory from 2015 to 2017. His research interests include efficient I/O, data management and storage for high-performance simulations, data-intensive computing, data analytics and visualization. He has published several research papers in top HPC conferences, including SC, IPDPS and Cluster, and journals such as IEEE TPDS, Elsevier FGCS, and ACM ToPC.



Rob Ross is a senior fellow in the Northwestern-Argonne Institute for Science and Engineering and in the University of Chicago/Argonne Computation Institute. He is also an adjunct assistant professor in the Department of Electrical and Computer Engineering at Clemson University.

Rob received his Ph.D. in computer engineering from Clemson University in 2000. He currently holds several leadership positions at Argonne and in the U.S. DOE computing community, including serving as deputy director of the Scientific Data Management, Analysis, and Visualization Institute, and he co-lead development of initial plans for the Data Management component of the DOE Office of Science Exascale Computing activity.



Tong Shu received her B.S. degree in information management and system from Peking University, and her Ph.D. degree in computer science from New Jersey Institute of Technology. She is a postdoctoral appointee at Argonne National Laboratory. Her research interests include scientific workflow optimization, machine learning, big data processing, energy efficiency, cloud computing, high-performance computing and networking, and wireless networks.



Tahsin Kurc is a Research Associate Professor in the Department of Biomedical Informatics at Stony Brook University and holds a Joint Faculty Appointment in Scientific Data Group at Oak Ridge National Laboratory. He received his Ph.D. in computer science from Bilkent University, Turkey, in 1997. His research seeks to develop informatics platforms that enable scalable management and analysis of large volumes of high-resolution, high-throughput scientific data on high end computing systems.



Li Tang is a Research Associate of Brookhaven National Laboratory. Before joining Brookhaven National Laboratory, he graduated with his Ph.D. from the Department of Computer Science and Engineering at University of Notre Dame. His research interests include hardware/software co-design, big data, performance/energy analysis and optimization, and heterogeneous computing.



Norbert Podhorszki is a senior research scientist in the Scientific Data Group in the Oak Ridge National Laboratory. He is one of the key developers of ADIOS that won an R&D100 award in 2013. His main research interest is in creating I/O and staging solutions for in-situ processing of data on leadership class computing systems. He received his Ph.D. in Information Technology from the Eotvos Lorand University of Budapest. He has worked in the field of logic programming, performance monitoring and analysis of message-passing programs, application monitoring in Grid environments

and application development in Desktop Grids, scientific workflow technologies in supercomputing projects, and high performance I/O. He is author and co-author of more than 100 scientific papers and book chapters. He has given many tutorials on ADIOS, including several times at the Supercomputing and the International Supercomputing conferences.



Matthew Wolf