A Technical Anatomy Of How OpenMPI Applications Can Inherit Fault Tolerance Using SPM.Python

Minesh B. Amin

Abstract—SPM.Python offers a rich environment for creating and deploying scalable, fault-tolerant, parallel applications to solve problems in domains spanning finance, life sciences, electronic design, IT, visualization, and research. Software developers may use SPM.Python to augment new or existing (Python) serial scripts for scalability across parallel hardware. Alternatively, SPM.Python may be used to better manage the execution of stand-alone (non-Python x86 and GPU) applications across compute resources. In this paper, we shall review how OpenMPI applications (legacy or otherwise) can inherit fault tolerance using SPM.Python.

Index Terms—fault tolerance, parallel closures, parallel exceptions, parallel invariants, parallel programming, parallel sequence points, scalable vocabulary, parallel management patterns

Prologue

Consider the following acid test for general purpose parallel computing. A serial session is depicted on the left, whereas the session on the right describes its parallel equivalent:

```
                    >>> createVirtualCloud -async
>>> cmdA            >>> cmdA -parallel
>>> cmdB            >>> cmdB -parallel
>>> cmdC            >>> cmdC -parallel
>>> cmdD            >>> cmdD -parallel
```

For example, the command **cmdA** **-parallel** may be a parallel make-like capability, while the command **cmdB** **-parallel** may be a map-reduce capability. At the same time, the command **cmdC** **-parallel** may be a fine grain parallel SAT solver that limits itself to resources with specific incarnations of those utilized by the command **cmdA** **-parallel**. Finally, **cmdD** **-parallel** may be a parallel graph-based analytics capability.

Yet, notwithstanding the prosaic serial session, the equivalent parallel session is in fact predicated on solutions to what were several formally open problems, including (a) defining a scalable vocabulary rich enough to capture the essence of a wide range of parallel problems, (b) the ability to utilize a collection of hardware resources in completely different ways, depending on the nature of parallelism exploited by the respective commands within the same session, and (c) the ability to treat the conclusion of each parallel command as a sequence point, thus guaranteeing that there would be no pending side effects post conclusion.

The corresponding author is with MBA Sciences, Inc, e-mail: mamin@mbasciences.com.

Introduction

In this paper, we shall review how OpenMPI applications can inherit fault tolerance using SPM.Python. Our solution is predicated on the supposition that parallelism entails nothing more than the *management* of a collection of *serial tasks*, where *management* refers to the policies by which:

- tasks are scheduled,
- premature terminations are handled,
- preemptive support is provided,
- communication primitives are enabled/disabled, and
- the manner in which resources are obtained and released

and *serial tasks* are classified in terms of either:

- Coarse grain – where tasks may not communicate prior to conclusion, or
- Fine grain – where tasks may communicate prior to conclusion.

We shall review how SPM.Python can single-handedly, without any external dependencies, packages, utilities, or support from IT, launch/track/monitor an OpenMPI application using compute resources under its control. Furthermore, the launching/tracking/monitoring is done in a manner that results in the said application inheriting fault tolerance with robust support for timeouts and self-cleaning attributes without requiring any changes in the source code of the application. This approach differs from most past attempts ([MPI01], [MPI02], [MPI03]) in that they require changes to the source code, or, as in the case of [MPI04], treat the problem as an instance of resource management.

Specifically, we will begin by describing typical flows that illustrate how SPM.Python compliments the strengths of OpenMPI. Next, the context for and solutions to two technical problems will be reviewed:

- design and architecture of the built-in package management sub-system,
- design and architecture of parallel closures that enable any OpenMPI application (legacy or otherwise) to inherit fault tolerance, and robust support for timeouts and self-cleaning by way of SPM.Python.

Finally, we will conclude by reviewing a simple, scalable, fault-tolerant, self-cleaning 60-line Python script that can be used to launch any OpenMPI application in parallel, thus illustrating key concepts.

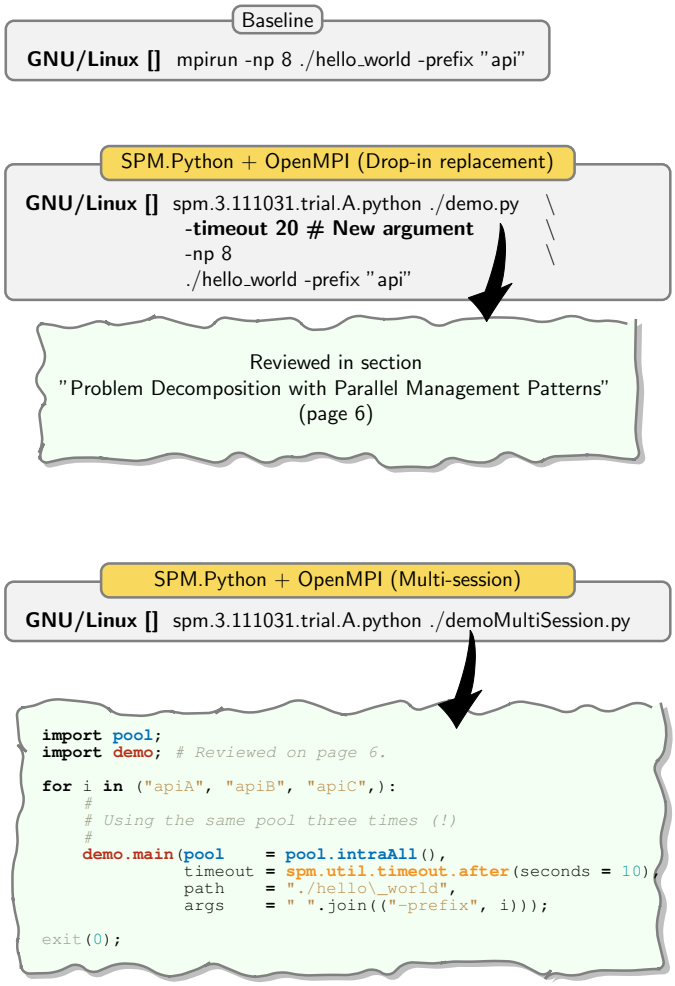

Figure 1: *Two ways by which SPM.Python may be used to compliment the strengths of any OpenMPI application. The first approach is simply a drop-in replacement for mpirun. The second approach leverages the virtual cloud within SPM.Python to invoke multiple sessions of potentially different OpenMPI applications one after the other. Both approaches involve the invocation of existing OpenMPI application(s), but in a fault-tolerant manner with robust support for timeouts.*

Complimenting the Strengths of OpenMPI with SPM.Python

OpenMPI offers many compelling advantages for developing high performance computing parallel and/or distributed applications. However, a common hurdle faced by solution developers includes the lack of built-in support for fault tolerance and timeouts.

SPM.Python offers an environment architected for developing scalable, fault-tolerant parallel solutions. Our integration with OpenMPI was motivated by a desire to enable users to leverage the strengths of both solutions in a complementary and frictionless manner.

To that end, SPM.Python may be used as a drop-in replacement for mpirun to launch, track and monitor any OpenMPI application. Furthermore, the application is executed in a manner so that it inherits fault tolerance and support for timeouts without requiring any changes in the source code.

Furthermore, thanks to the built-in notion of virtual cloud within SPM.Python, resources acquired by SPM.Python may be used to execute multiple sessions of OpenMPI applications, thus maximizing utilization of these resources and minimizing time lost due to the lack thereof.

Figure 1 illustrates the progression of OpenMPI execution flows, from a baseline using mpirun to an OpenMPI + SPM.Python drop-in flow with inherited fault tolerance offering timeout support, and finally an OpenMPI + SPM.Python flow that also offers multi-session support for resource management.

However, these complimented flows are predicated on:

- Built-in package management system
  (within SPM.Python)

  We need to substitute in real-time customized versions of:
  - mpirun that is augmented to acquire compute resources from SPM.Python,
  - orted that is launched by the Spokes of SPM.Python, and is augmented to launch SPM.Python's wrapper executable (in lieu of the OpenMPI application)

  In other-words, a single installation of SPM.Python needs to have access to multiple versions of the augmented OpenMPI environment. This is only possible if SPM.Python's Python interpreter has a built-in package management system.
- SPM.Python's ability to directly track and monitor the OpenMPI application
  This is achieved by having SPM.Python's wrapper executable launch the OpenMPI application in a manner so that critical system and OpenMPI shared library calls are redirected using ELF PLT infection, as described in ([ELF01], [ELF02]). The goal of the redirection is to establish a connection from the OpenMPI node (application) to the SPM.Python Spoke so that SPM.Python can track/monitor the said node.

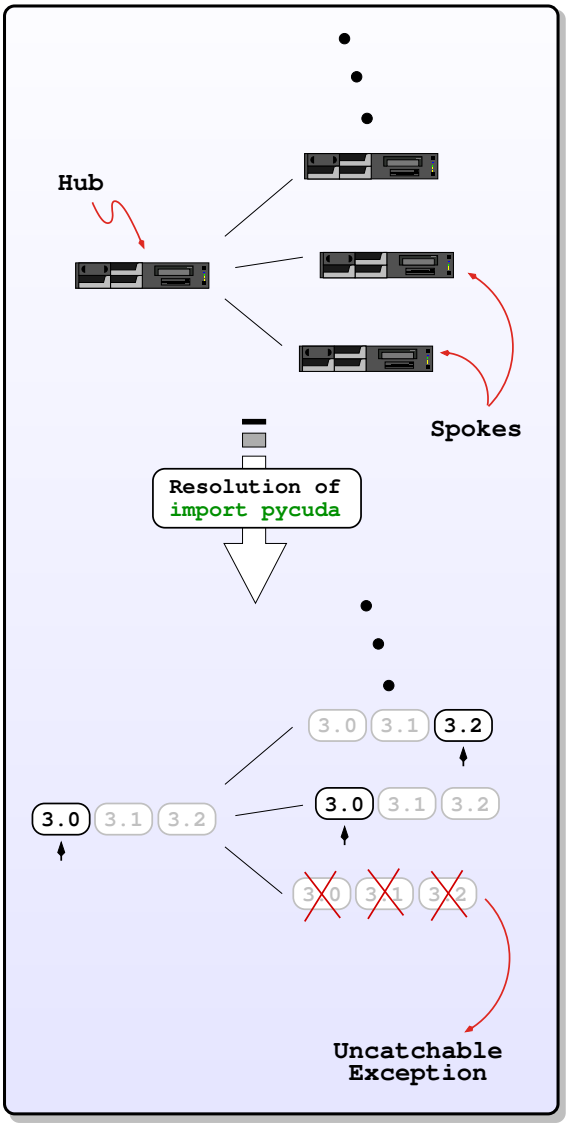

Figure 2: *The resolution of the statement "*import pycuda*" across all resources. The ultimate version (if any) is a function of the CUDA driver install on the respective resources. For the purpose of illustration, the Hub is resolved to pycuda for CUDA driver 3.0, while two spokes are resolved to pycuda for CUDA drivers 3.2 and 3.0 respectively. Finally, the raising of an uncatchable exception (see [USP01], [SPM01] for more details on such exceptions) on the third Spoke is shown; this exception would be raised in situations where the resource in question does not have any CUDA driver that is supported by SPM.Python.*

Built-in Package Management

Most tools used to manage Python packages are designed around the philosophy of "*single-version, externally managed*". The Python developer has to manipulate `sys.path` to pick the correct version of some package, a process that can get out of hand very quickly when, for example, trying to run multiple versions of module(s) in parallel.

SPM.Python's built-in package management sub-system is designed around the philosophy of "*multiple-versions, internally managed*". `sys.path` would point to the root where multiple versions of packages reside. The actual path is determined by the package selected, or inferred (in case of default behavior).

In other-words, the goal of the built-in package management sub-system is to permit the installation of multiple versions of any package under some root directory, while ensuring that only one version may reside in memory during a session.

For the purpose of illustration, consider how SPM.Python's interpreter processes the statement

```
import pycuda
```

Normal resolution of paths to packages is commenced. However, upon reaching the virtual path `"/-@-/pkg.builtin/"` (defined in `sys.path`), the built-in package sub-system takes over, and proceeds to resolve the path in three steps:

- Search for all packages whose name matches our target. If no match exists, the sub-system returns, and the normal resolution continues using any path(s) that follow `"/-@-/pkg.builtin/"` (defined in `sys.path`).
- If a match does exist, if possible, return the path to the resolved path computed sometime in the past.
- If there is no resolved path in memory, invoke the preloading utility for each version of the package, from the oldest version forward. The first utility to return TRUE is deemed to be a successful match, and thus the only version to be loaded during this session for the compute resource in question.

Currently, the built-in package management system does not allow the developer to guide the resolution process. Furthermore, the order in which the preloading utilities are invoked is fixed and as described above.

Thus, our statement

```
import pycuda
```

can be resolved to different paths that take into account the version of CUDA driver installed at each compute resource, as depicted in Figure 2.

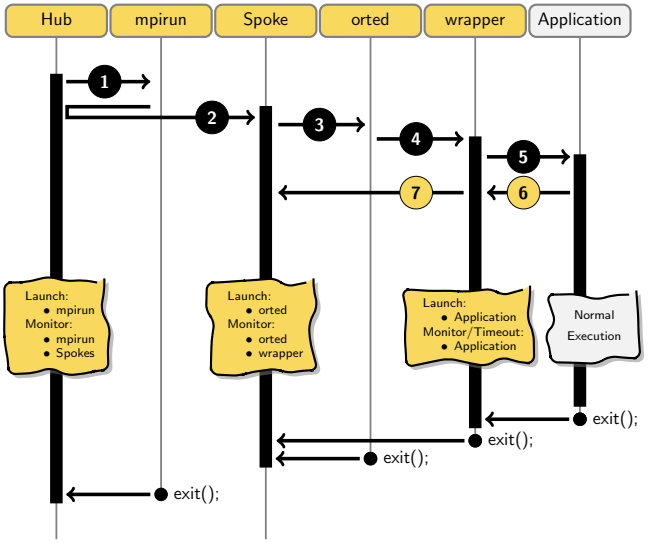

Figure 3: *The overall time-line of various events when launching/monitoring/tracking an OpenMPI application from SPM.Python.*

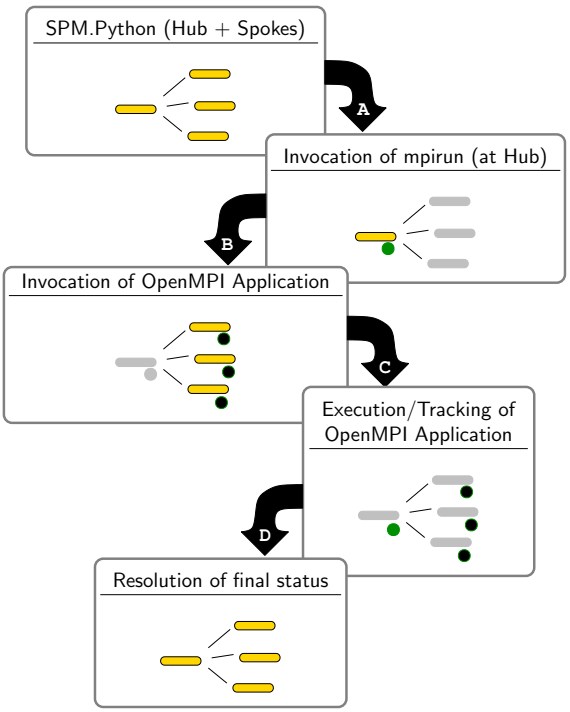

Figure 4: *The four steps by which compute resources under the control of SPM.Python may be leveraged to launch/track/monitor an OpenMPI application in a fault-tolerant manner with support for timeouts and self-cleaning. The actual version of OpenMPI is determined by probing the library dependencies of the application. Next, the application is launched in a manner so that default paths to the OpenMPI installation (typically under /usr/local) are selectively replaced to instead point to the equivalent version of OpenMPI provided by SPM.Python.*

Leveraging OpenMPI applications using SPM.Python

As depicted in Figures 3 and 4, SPM.Python launches, tracks and monitors an OpenMPI application in four steps. It begins by (a) probing the OpenMPI library dependencies of the application, and (b) replacing the default OpenMPI environment with one provided by SPM.Python. Furthermore, the launching/tracking/monitoring would be done in a manner that results in the said application inheriting fault tolerance with robust support for timeouts and self-cleaning attributes without requiring any changes in the source code of the application.

Using the built-in package management sub-system described in the previous section, SPM.Python can easily replace default mpirun and orted executables (typically located under /usr/local) with the equivalent versions provided by SPM.Python.

(A) Invocation of mpirun at the Hub

SPM.Python's version of mpirun is invoked as a co-process at the Hub (step 1/Figure 3, and step A/Figure 4). Note that all co-processes in SPM.Python have a well-defined notion and support for timeouts and self-cleaning. Furthermore, mpirun is augmented so that:

- Requests for resources are redirected to the Hub (step 2/Figure 3).
- Requests to launch all orted processes are redirected to the respective Spokes by way of the Hub (step 3/Figure 3).

Having launched the co-process, the Hub enters an event loop which will only conclude when the co-process concludes (either naturally, or due to an induced timeout event). In the event loop at the Hub,

- Requests for resources are answered by probing SPM.Python's tracking utility for the list of resources that are available
- Requests for launching any orted processes are redirected to the respective Spokes (step 3/Figure 3).
- Resolution of final status is computed as described below (D).

(B) Invocation of OpenMPI application

At each Spoke (if requested by the Hub), SPM.Python's version of orted is invoked as a co-process, as depicted by (step 3/Figure 3, and step B/Figure 4). As in the case of Hub, note that all co-processes in SPM.Python have well-defined notion and support for timeouts and self-cleaning. Furthermore, all OpenMPI application nodes are invoked by either orted or mpirun by way of the wrapper in a manner so that:

- each OpenMPI application would establish connection with the parent (wrapper), as depicted by (step 6/Figure 3). Also, signal handles are initialized to facilitate the reporting of stack traces in the event of premature termination.
- each wrapper, in turn, would thereafter establish connection with the parent Spoke, as depicted by (step 7/Figure 3).

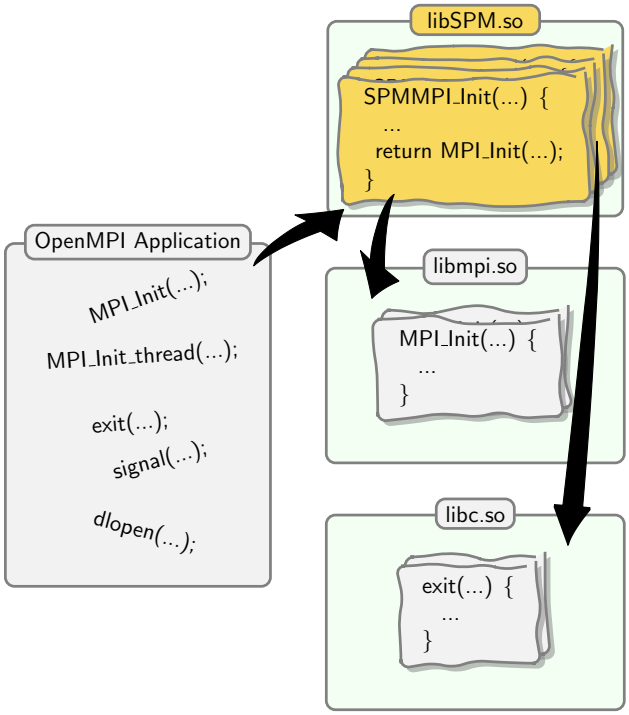

Figure 5: *In order to ensure that SPM.Python can track and monitor all OpenMPI application nodes, a connection must be established between each Spoke (of SPM.Python) and the corresponding application node by way of the wrapper. Given that we are using the original application without changing any code, the only way to establish this critical connection is to redirect critical system and OpenMPI shared library calls to* `libSPM.so`*. In* `libSPM.so`*, the redirected system calls are augmented as needed to establish connection, and provide any stack trace in the event of a premature termination.*

Meanwhile, in the event loop at the respective Spokes,

- A request to establish a connection with the wrapper is accepted,
- Termination of the shadow OpenMPI node and content (if any) of the establish connection is converted into the final status report of the node and reported as such to the Hub.

(B) On having wrapper launch OpenMPI application

Recall our goal of having the OpenMPI application inherit fault tolerance and support for timeout without requiring any changes in the source code. In other-words, we need SPM.Python to directly track and monitor all instances of the application (nodes) in the OpenMPI runtime environment.

This is achieved by having SPM.Python's wrapper executable launch the OpenMPI application in a manner so that critical system and OpenMPI shared library calls like `MPI_Init`, `MPI_Init_thread`, `exit`, `signal`, `dlopen` are redirected, as depicted in Figure 5. See ([ELF01], [ELF02]) for details on how this redirection is actually achieved.

Note that the redirection must be achieved without having to pay any penalty in terms of performance ... this is easily possible given that the system calls in question are typically called only a handful times per session. Furthermore, memory overhead is bounded and is proportional to the memory required to establish the socket connection with the wrapper.

See Appendix A for details on how the wrapper must launch a Python + OpenMPI application.

(C) Execution/Tracking of OpenMPI application

At this point, mpirun is shadowed by the Hub while all application nodes are shadowed by the respective/unique Spokes by way of the respective wrappers. Furthermore, the Hub and Spokes are executing event loops designed to only react to the events described. Henceforth, the OpenMPI application nodes can effectively consume CPU cycles of the respective resources without having to compete with SPM.Python.

(D) Resolution of final status

The resolution of the final status is triggered by the first co-process that concludes for any reason. However, special care is taken so that a successful conclusion is only possible when all co-processes (mpirun and OpenMPI nodes) conclude cleanly. In other-words, the Hub treats all resources as a single unit; the premature termination of mpirun or any node is treated as a premature termination of all co-processes and reported as such. See ([USP01], [SPM01]) for details on why resources must be treated as a single unit.

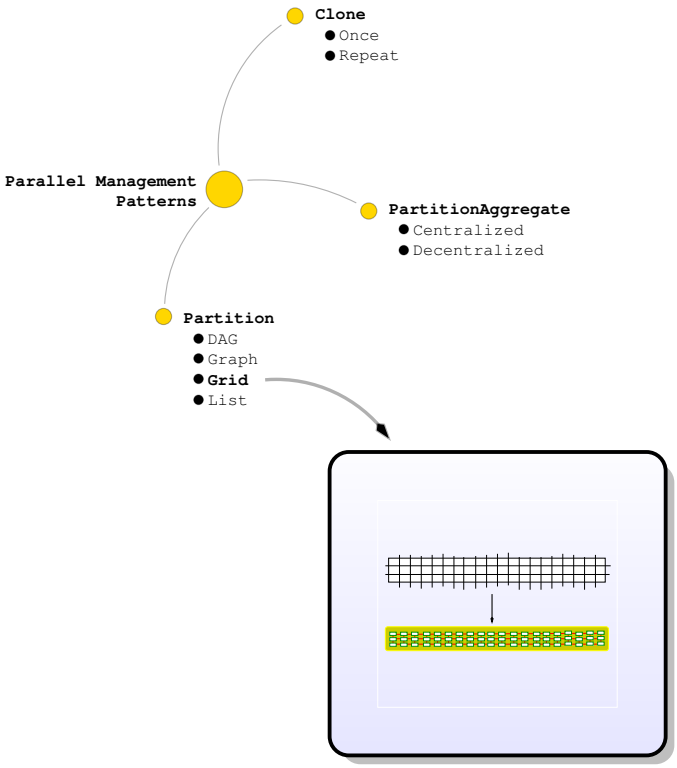

Figure 6: *Partition/Grid Parallel Management Pattern.*

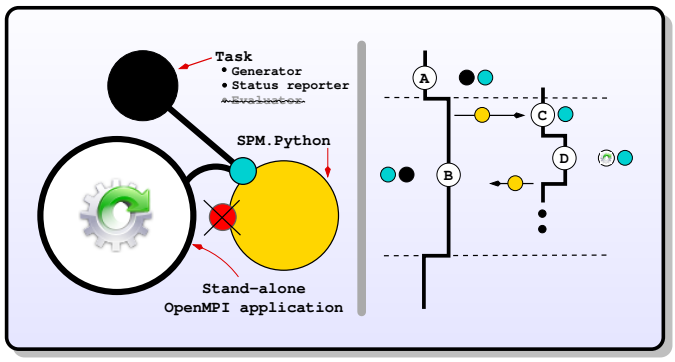

Figure 7: *The architectural and runtime perspectives of launching a stand-alone OpenMPI application in parallel using SPM.Python.*

Problem Decomposition with Parallel Management Patterns

Understanding the nature of any parallel problem is key to determining the appropriate solution. Parallel Management Patterns (PMPs) provide a framework for decomposing and authoring scalable, fault-tolerant parallel solutions. In other-words, if the end goal is some parallel application, PMPs enable us to classify the journey to the end goal in terms of the nature of parallelism to be exploited, while parallel closures provided by SPM.Python enable us to express the parallelism implied by any PMP.

For the purpose of illustration, we shall review an implementation of the Partition/Grid PMP, as depicted by Figure 6; this pattern captures the essence of how to execute a template task across many compute resources in a fault-tolerant manner.

Problem Statement

Our goal is to invoke SPM.Python's version of mpirun given an OpenMPI application. We shall capture the context (in the form of arguments needed, and the final result to be returned) of each execution by way of a template task. To that end, we shall augment the aforementioned parallel functionality by authoring a scalable, parallel, fault-tolerant Python wrapper script with the following components:

- declaration of a (task manager) closure at the Hub,
- definition of a template task, processing of status reports, and invocation of task manager at the Hub.

As an aside, note that the back-end of our closure will evaluate a copy of the template task on our behalf across all Spokes. Figure 7 illustrates the architectural and runtime perspectives of launching an OpenMPI application, which are described in detail next.

Ⓐ Task manager: Declaration and Definition

In order to create (declare and define) an instance of the task manager, we require the Hub to be off-line to in order to avoid various types of parallel race conditions. This invariant is captured by the decorator statements on lines 1 and 2.

A natural point in time to perform this initialization step would be when loading the module containing the statements prior to actual usage. In other words, initialization should occur when the file containing the `__init` method is imported by the Python interpreter.

The arguments for creating our instance bear highlighting. Each instance of any closure must be unique within a module, hence the unique string as argument 1. Furthermore, all instances of our closure are defined in terms of two stages. Of these, functionality for stage 1 is expected via a call-back, hence argument 2 (`__taskStat`).

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def __init():
4    return spm.pclosure.macro.papply.template.openMPI.\
5           policyA.defun(signature = 'signature::Hub',
6                         stage1Cb  = __taskStat,
7                        );
8
9  __pc = __init();
```

```
r"""
 task<template>     ::struct {
   # SPM component ...
   spm              ::struct {
     meta           ::struct {
       label        ::scalar<stringSnippet> = deferred;
       apiArgs      ::dict<string,mixed>    = deferred;
       timeout      ::scalar<timeout>       = deferred;
     };

     core           ::struct {
       relaunchPre  ::scalar<bool>          = None;
       relaunchPost ::scalar<bool>          = None;
       nameHost     ::scalar<auto>          = None;
       whoAmI       ::scalar<auto>          = None;
     };

     stat           ::struct {
       exception    ::scalar<auto>          = None;
       returnValue  ::scalar<record>        = None;
     };
   };
   # non-SPM component ...
 };
 """
```

Figure 8: *Typedef for the definition of a template task.*

```
Exception
 |
 +-- SPMError
       |
       +-- SPMTaskDropped
       +-- SPMTaskLoad
       |     |
       |     +-- SPMTaskLoadUncaught
       |     +-- SPMTaskLoadFatal
       |     +-- SPMTaskLoadKill
       |     +-- SPMTaskLoadTimeout
       |
       +-- SPMTaskEval
             |
             +-- SPMTaskEvalUncaught
             +-- SPMTaskEvalFatal
             +-- SPMTaskEvalKill
             +-- SPMTaskEvalTimeout
```

Figure 9: *Hierarchy of (parallel) SPM exceptions.*

Ⓐ Task manager: Population and Invocation

Our goal in the function **main** is to invoke the task manager (line 14). However, before doing so, we must populate it with the tasks to be executed. This is achieved by submitting our tasks by way of the API **stage0**, as shown in lines 8 through 12.

Once our task manager is invoked, the Hub transitions to the online state. The transition back to off-line does not occur until just prior to the conclusion of the invocation.

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def main(pool,
4           taskApiArgs,
5           taskTimeout):
6    # Initialize 'stage0'.
7    __pc.stage0.init.main(typedef = ...); # See Figure 8.
8    hdl = __pc.stage0.payload.tie();
9    # Populate the template task
10   hdl.spm.meta.label   = '***'; # Not interested.
11   hdl.spm.meta.apiArgs = taskApiArgs;
12   hdl.spm.meta.timeout = taskTimeout;
13   # Invoke the pmanager
14   __pc.stage0.event.manage(pool                  = pool,
15                            nSpokesMin            = ...
16                            nSpokesMax            = ...
17                            timeoutWaitForSpokes = ...
18                            timeoutExecution      = ...
19                            );
20   return;
```

Ⓑ Task manager: (Final) Status Reports

The method **__taskStat** (used when declaring and defining our closure) is automatically invoked by the task manager to process the status report of any task. Note that this method is invoked while the Hub is in the online state. This invariant is captured by the decorator statements on lines 1 and 2.

```
1  @spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
2  @spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
3  def __taskStat(pc):
4    try:
5      hdl         = pc.stage1.payload.tie();
6      returnValue = hdl.spm.stat.returnValue;
7      if (returnValue.Has(attr = 'stdOut')):
8        print("\tstdOut   : %s", returnValue.stdOut);
9      if (returnValue.Has(attr = 'stdErr')):
10       print("\tstdErr   : %s", returnValue.stdErr);
11     if (returnValue.Has(attr = 'stdOutErr')):
12       print("\tstdOutErr: %s", returnValue.stdOutErr);
13   except (SPMTaskDropped,
14           SPMTaskLoad,
15           SPMTaskEval,
16           ), (hdl,):
17     pass;
18
19   return (pc.stage1.event.done(),
20           None,
21           )[-1];
```

Ⓒ Task manager: Preloading of Python modules

Ⓓ Task manager: Task Evaluation

As each task involves the invocation of one of the built-in spm co-process methods, we do not need to define any method to accept and evaluate any task. Instead, our task manager will automatically evaluate our tasks on the Spokes, and return the respective status reports to the Hub. The automatic evaluation of our tasks is aided by the typedef used when initializing **stage0** (at the Hub).

```
GNU/Linux [] spm.3.110602.trial.A.python

●  >>> import pool
   >>> import demo
   >>> import os;
   >>> taskApiArgs = dict(app        = os.getcwd() + '/hello_world',
                          appOptions = "-prefix='app'",
                         );
   >>> taskTimeout = spm.util.timeout.after(seconds = 10);
✓  >>> demo.main(pool        = pool.intraAll(),
                  taskApiArgs = taskApiArgs,
                  taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
           app => 0
           app => 1
     #: MetaStatus (hub): Tasks   - EvalDone
✓  >>> demo.main(pool        = pool.intraOnePerServer(),
                  taskApiArgs = taskApiArgs,
                  taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
     #: MetaStatus (hub): Tasks   - EvalDone
✈  >>> demo.main(pool        = pool.inter(),
                  taskApiArgs = taskApiArgs,
                  taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
           app => 0
           app => 1
           app => 2
     #: MetaStatus (hub): Tasks   - EvalDone
✈  >>> demo.main(pool        = pool.interOnePerServer(),
                  taskApiArgs = taskApiArgs,
                  taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
     #: MetaStatus (hub): Tasks   - EvalDone
   >>> exit()
   GNU/Linux []
```

Figure 10: *A typical parallel session of SPM.Python.*

SPM.Python Session

Having reviewed our parallel application, we will conclude by describing an actual SPM.Python session, as depicted in Figure 10. We start off by importing the `pool` module (●). Next we import our parallel application `demo`, and run our application four times before exiting, as illustrated by ✓ and ✈.

The first two times (marked ✓), we limited ourselves to cores from the server running the Hub. `intraOnePerServer` refers to one unique core on the server.

The second two times (marked ✈), we limited our selves to cores from potentially different servers. `interOnePerServer` refers to one unique core from each server.

As expected, the openMPI application remains unchanged despite having selected four different sets of resources.

Note that notwithstanding our rather small script, our solution is not only fault-tolerant (thanks to closures), self-cleaning (thanks to timeout support), but also robust (thanks to the efficient manner by which parallel invariants are enforced). Once we have tested our solution in a serial-like environment, we can be sure our solution can be deployed on any cluster. See [PMP02] for a comprehensive list of problem decompositions using other PMPs, each including self contained and equally powerful examples.

Conclusion

In this paper, we reviewed the technical anatomy of how OpenMPI applications (legacy or otherwise) can inherit fault tolerance using SPM.Python. We began with a prologue presenting the acid test for general purpose parallel computing. Next, we described the solution to two technical problems, namely the built-in package management sub-system and the design and architecture of parallel closures that enable any OpenMPI application to inherit fault tolerance, with robust support for timeouts and self-cleaning by way of SPM.Python. We concluded by illustrating how a parallel problem, once classified in terms of a Parallel Management Pattern (PMP), can be decomposed and easily expressed in terms of SPM.Python's parallel closures.

References

[MPI01] William Gropp and Ewing Lusk, *Fault Tolerance in MPI Programs*,

[MPI02] George Bosilca at el, *MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes*,

[MPI03] Grapham E. Fagg and Jack J. Dongarra, *FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world*,

[MPI04] Morris Jette and Mark Grondona, *SLURM: Simple Linux Utility for Resource Management*,

[USP01] Minesh B. Amin, *Resource Tracking Method and Apparatus*, United States Patent #: 7,926,058 B2, April 12, 2011.

[SPM01] Minesh B. Amin, *A Technical Anatomy of SPM.Python*, SciPy 2011

[ELF01] Silvio Cesare, *Shared library call redirection using ELF PLT infection*

[ELF02] Quake2th, *PLT redirection through shared object injection into a running process*

[PMP02] Parallel Management Patterns, www.mbasciences.com/pmp.html

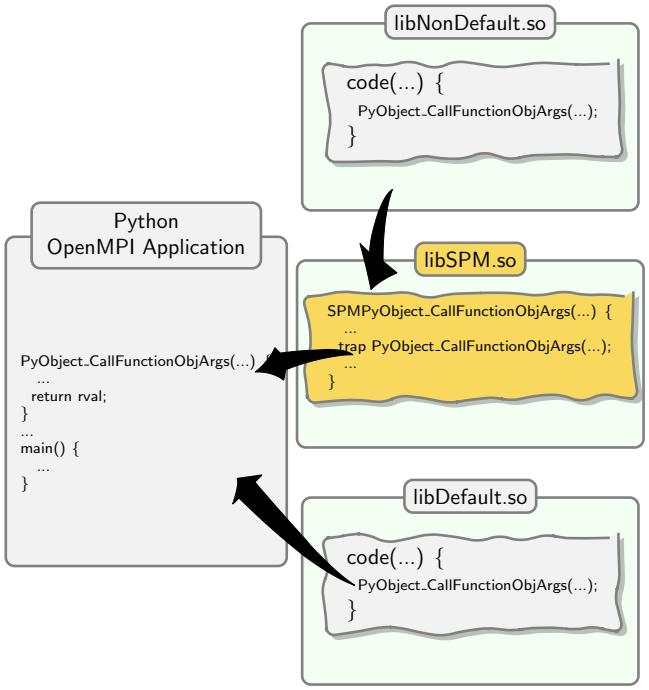

Figure 11: *Trapping invalid results from C APIs. Unlike the redirection of shared libraries described on page 5, here, the function calls of interest are in the Python executable itself. We would like to trap all invocations (from non-default shared libraries) that throw exceptions. To achieve this result, non-default shared libraries are loaded in a manner so that the symbols in question resolve to* `libSPM.so`*, while default shared libraries are loaded in a manner so that the symbols in question resolve to the ones in the application itself.*

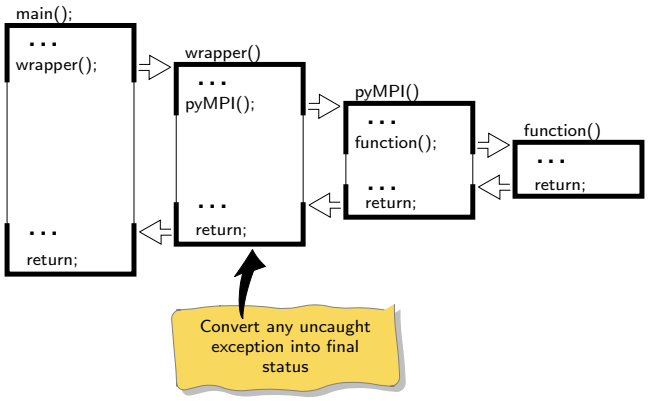

Figure 12: *Trapping all uncaught exceptions in Python. Here, we can simply introduce a wrapper written in Python whose sole purpose would be to convert any uncaught exception into the final status of the application, and thereby inducing a premature termination.*

Appendix A: On launching Python + OpenMPI application

When running a Python + OpenMPI application, there are two main sources of premature termination or undefined behavior:

- Uncaught exception via C API
  Examples for such situations include C implementations that assume that any call to (for example)

  `PyObject_CallFunctionObjArgs`

  will never throw an exception (or return a NULL value). Typically, non-default shared libraries tend to make this assumption as they are more likely to be under active development.
  Figure 11 illustrates how our wrapper must launch the application so that only (for example)

  `PyObject_CallFunctionObjArgs`

  calls made by non-default shared libraries are trapped by `libSPM.so`.
- Uncaught exception in Python script
  Such exceptions can be trapped by invoking the actual Python script through one level of indirection; i.e. having a Python wrapper script invoke the actual Python script, as depicted in Figure 12.

In summary, our goal is ensure that any uncaught exception is caught, and converted into the final status report. The report is sent to the respective Spoke by way of the wrapper, and the application terminated. The global ramifications of such a termination is left to SPM.Python's Hub. Typically, any such termination would result in the forcible termination of all remaining application nodes.

Note that another reason the application may be forcibly terminated is if the wrapper decides the application has not concluded within the timeout period.