A Technical Anatomy of SPM.Python, a Scalable, Parallel Version of Python

Minesh B. Amin[‡][*]

Abstract—SPM.Python is a scalable, parallel fault-tolerant version of the serial Python language, and can be deployed to create parallel capabilities 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 a fault-tolerant manner taking into account hard deadlines.

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.

* *Corresponding author: mamin@mbasciences.com*
‡ *MBA Sciences, Inc*

Introduction

In this paper, we shall review (patented) SPM technology, and the methodology behind it, both 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 augments the serial Python language to include a suite of parallel primitives, henceforth referred to as parallel closures. These closures represent the sole means by which to express any parallelism when leveraging SPM.Python. Their APIs are designed to be as close to the developer's intent as possible, and therefore easy to relate to. Furthermore, the API of all closures represent the boundary that delineates the serial component (authored and maintained by the developer) from the parallel component (authored and embedded within SPM.Python).

Specifically, the context for and solutions to four formerly open technical problems will be reviewed:

a) decoupling tracking of resources from management of resources,
b) declaration and definition of parallel closures, the building blocks of all parallel constructs,
c) design and architecture of parallel closures in a way so that serial components are delineated from parallel components, and
d) extensions to the general exception handling infrastructure to account for exceptions across many compute resources.

We will illustrate key concepts by reviewing a simple, scalable, fault-tolerant, self-cleaning 60-line Python script that can be used to launch any stand-alone (x86 or GPU) applications in parallel. Appendix A will provide another self-contained Python script that calculates the total number of prime numbers within a given range;

thus, illustrating how any Python module may be parallelized using one of SPM.Python's several built-in parallel closures.

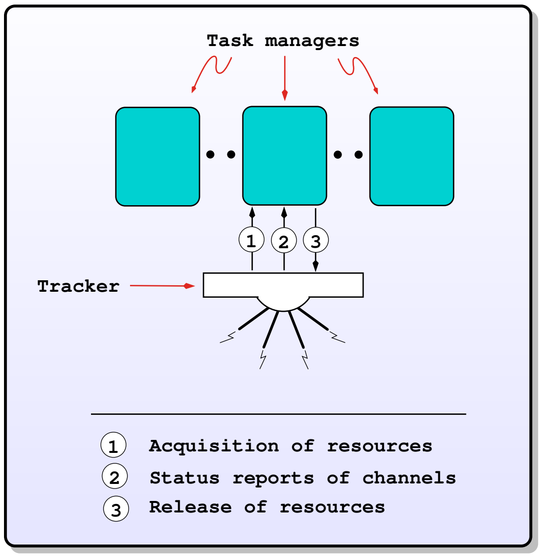


Fig. 1: In order to facilitate the exploitation of multiple, potentially different, forms of parallelism within a single session of SPM.Python, tracking of resources is decoupled from the management of resources. Therefore, while the tracker is always online, at any moment in time, at most one task manager may be online.

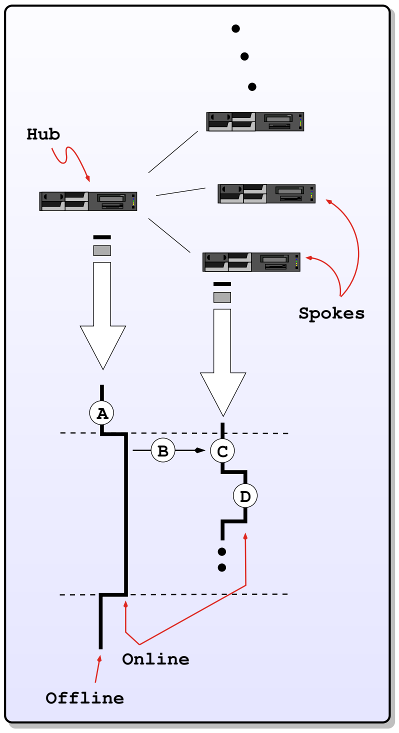


Fig. 2: Parallel sequence points in terms of online and offline states of the Hub and Spokes. On the Hub, transition to online occurs when a task manager is invoked; transition back to offline occurs when the said manager concludes. On the Spoke, transition to online occurs when a task evaluator is invoked; transition back to offline occurs when the said evaluator concludes.

Related Work

Traditionally, most parallel solutions in Python have taken the form of: (a) distributed task queues like Celery[1], Parallel Python[2], (b) distributed frameworks like Disco (MapReduce)[3], PaPy (parallel pipelines)[4], or (c) low-level wrappers around HPC libraries like MPI[5], PVM[6]. In sharp contrast, SPM.Python is a single runtime environment that provides access to multiple different, distinct forms of parallelism by way of parallel primitives called parallel closures.. Furthermore, these closures are architectured to be as close to the developer's intent as possible – in terms of, say, either coarse or fine-grain DAG/templates/hybrid flows, and lists – while de-emphasizing low-level error-prone concepts like locks, threads, pipes, mutexes and semaphores.

Tracking of Resources

In SPM.Python, compute resources are tracked independently of any task manager. In operation, any task manager may come online and request resources from the tracker. The task manager would then manage the execution of tasks using the acquired resources, and when done, go offline (i.e. release the resources back to the tracker). Another task manager may subsequently come online, obtain the same or different resources used by a previous task manager, and utilize those resources in a completely different way. In other words, the task managers can be implemented more simply because each manager would have a more narrowly focused discrete policy. Furthermore, a tight coupling can be established between a task manager and the communication closures, thus preventing a whole class of deadlocks from occurring. More details can be found at [7].

Declaration and Definition of Parallel Closures

In SPM.Python, parallel closures are the building blocks of all parallel constructs, and provide the sole means by which one may express how serial components interact with parallel components. The interactions may take place in one of two contexts (a) when creating, submitting, and evaluating tasks, and (b) when creating and processing messages. However, any usage of a parallel closure within any resource is predicated on a successful, safe, asynchronous and race-free declaration and definition across many compute resources. We solve this problem by augmenting the traditional concept of serial sequence points by introducing the notion of *offline* and *online* states. The declaration and definition of parallel closures is only permitted when the resource in question is in the *offline* state – a state when SPM.Python guarantees that the serial component of the resource may not communicate with the outside world and vice versa. So, all resources start off *offline* (Ⓐ, Ⓒ).

On

the Hub, the transition to the *online* state occurs when a parallel (task manager) closure is invoked; the transition back to the *offline* state does not occur until just before the closure concludes. On the Spoke, SPM.Python receives a task from the Hub while *offline* (Ⓒ), and at which point any preloading of Python modules is performed. One side effect of this preloading may be the declaration and definition of parallel closures. Next, the transition to *online* is made before SPM.Python invokes the callback (Ⓓ) for the task; the transition back to *offline* does not occur until just after the callback concludes.

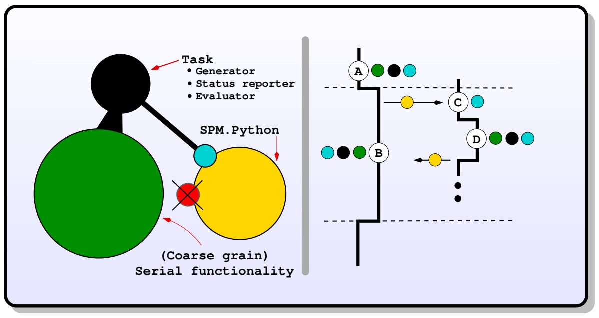


Fig. 3: The architectural and runtime perspectives of coarse grain task manager closures. Note that such closures do not permit tasks to communicate prior to conclusion.

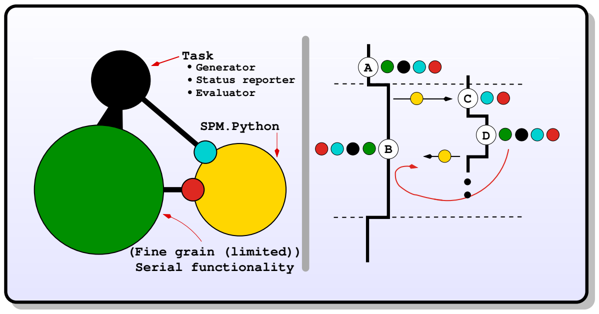


Fig. 4: The architectural and runtime perspective of fine grain (limited) task manager closures. Note that such closures permit tasks to communicate only with the Hub.

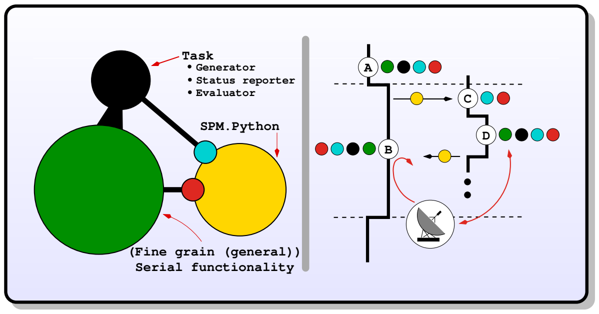


Fig. 5: The architectural and runtime perspective of fine grain (general) task manager closures. Note that such closures permit communication among Spokes and, if appropriate, with the Hub.

Types of Fault-Tolerant Parallel Closures

A key tenet of the serial software ecosystem is the asymptotic parity between the serial compute resources available to the developers and the end-users, which makes possible the reporting, reproduction, and resolution of bugs.

With parallel software, this most fundamental of tenets is violated; software engineers need to be able to produce high-quality parallel software in what is an essentially serial environment, yet be able to deploy the said software in a parallel environment.

SPM.Python addresses this dichotomy by offering a suite of easy to relate to parallel closures. These closures enable the prototyping, validation, and testing of parallel solutions in an essentially serial-like development environment, yet are scalable when exercised in any parallel environment.

Coarse grain

Exploiting coarse grain parallelism is anchored around the asynchronous declaration and definition of a parallel (task manager) closure (●) across all resources (Hub and Spokes). On the Hub, this is depicted by (Ⓐ). On the Spokes, this is only possible prior to the evaluation of a task, as depicted by (Ⓒ), when the modules may be preloaded.

Next,

existing serial functionality (●) may be parallelized by having it be augmented with serial code (●) to:

- generate and submit tasks to the parallel task manager, and handle status reports/exceptions from tasks, as depicted by (Ⓑ)
- evaluate tasks, as depicted by (Ⓓ)

Finally, actual parallelism can commence by invoking the task manager on the Hub with a collection of tasks, and a handle to a pool of resources (Ⓑ). The backend of the task manager would ensure the concurrent scheduling and evaluation of tasks across all Spokes. Note that coarse grain task manager closures do not permit the usage of any form of communication closures (⊗).

Fine grain (limited)

Fine grain (limited) parallelism augments the coarse grain parallelism by allowing tasks to communicate with the Hub prior to their conclusion. The closures (●) that would permit such communication must be declared and defined following the steps reviewed for parallel task manager closures (●).

However,

in order to avoid the vast majority of deadlocks, the communication closures must be designed in a way so that all communication is initiated by the Spokes; the Hub must be restricted to processing incoming messages from the Spokes, and, if appropriate, replying to them.

Fine grain (general)

Fine grain (general) parallelism augments the fine grain (limited) parallelism by permitting communication among Spokes.

However,

in order to avoid the vast majority of deadlocks, the fine grain (general) task manager closures must treat all Spokes under their control as a single unit; the premature termination of any Spoke must be treated as a premature termination of all Spokes.

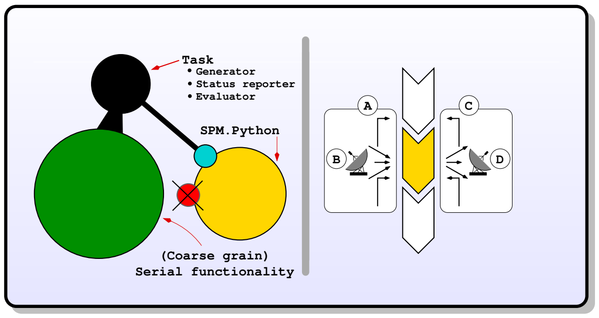


Fig. 6: The architectural and runtime perspectives of coarse grain parallel exceptions.

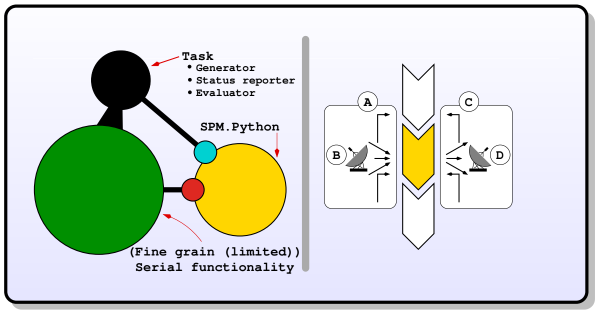


Fig. 7: The architectural and runtime perspectives of fine grain (limited) parallel exceptions.

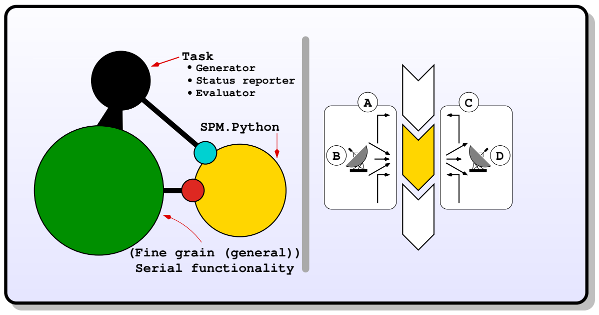


Fig. 8: The architectural and runtime perspectives of fine grain (general) parallel exceptions.

Types of Fault-Tolerant Parallel Exceptions

To quote Wikipedia, "exception handling is a construct designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution".

The ability to throw and catch exceptions forms the bedrock of the serial Python language. We will review details of how we extended the basic serial exception infrastructure to account for exceptions that may occur across many compute resources.

Our solution is predicated on the notion that parallel task managers must take ownership of how serial exceptions are handled across all resources under their control. Therefore, unlike in the serial world, the parallel exception handling infrastructure must be customized for each type of parallel task manager.

Coarse grain

Exception handling, as traditionally defined in the serial context, is designed to handle the change in the normal flow of program execution ... a rather straightforward concept given that there is only one call-stack.

However,

when exploiting parallelism, the normal flow of program execution involves multiple resources and, therefore, multiple call-stacks need to be processed in a fault-tolerant manner. Furthermore, in order to enforce various forms of parallel invariants, we need an ability to throw exceptions at any resource, but which may only be caught by the Hub.

Stated another way, in order to make our problem tractable in the context of coarse grain parallelism:

- on a Spoke, any uncaught/uncatchable exception must be treated and reported as final status of the task. Therefore, an exception free execution on the Hub would result in the normal unrolling of the call-stack at the Hub, as depicted by (Ⓐ, Ⓑ).
- on the Hub, any uncaught exception from any callbacks invoked by the task manager must result in the forcible termination and, if appropriate, relaunching of Spokes, as depicted by (Ⓒ, Ⓓ).

Fine grain (limited)

The exception handling infrastructure in the context of fine grain (limited) parallelism may be identical to that for coarse grain parallelism provided stale replies generated by the Hub and meant for some Spoke can be filtered out at the Hub itself.

Fine grain (general)

Given that fine grain task manager closures treat all Spokes as a single unit:

- on a Spoke, any uncaught/uncatchable exception must be treated and reported as final status of all the Spokes. Therefore, an exception free execution on the Hub and all Spokes would result in the normal unrolling of the call-stack at the Hub, as depicted by (Ⓐ, Ⓑ).
- any uncaught/uncatchable exception from any callbacks invoked by the task manager or by any Spoke should result in the forcible termination and, if appropriate, relaunching of Spokes, as depicted by (Ⓒ, Ⓓ).

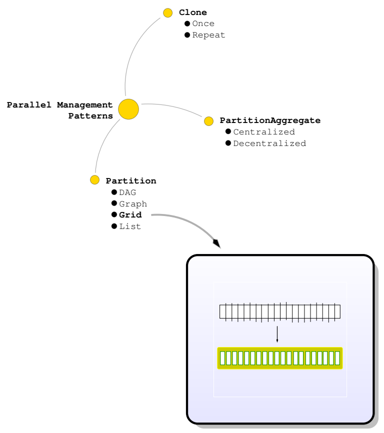


Fig. 9: *Partition/List Parallel Management Pattern.*

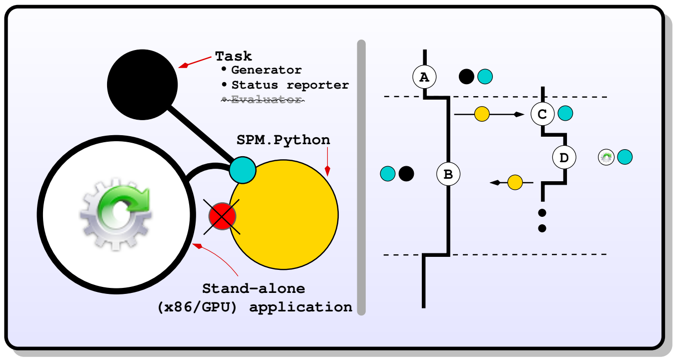


Fig. 10: *The architectural and runtime perspective of launching stand-alone applications in parallel using SPM.Python.*

Problem Decomposition

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/List PMP, a pattern that captures the essence of how to execute a list of tasks across many compute resources in a fault-tolerant manner.

Problem Statement

Our goal is to invoke the SPM coprocess API:

```
spm.util.coprocess.shell.policyA(cmd     = ...,
                                 timeout = ...,
                                )
```

across multiple resources. We shall capture the context - in the form of arguments needed, and the final result to be returned - of each execution by way of tasks. To that end, we shall augment the aforementioned serial functionality by authoring a scalable, parallel, fault-tolerant Python script made up of the following components:

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

As an aside, note that the backend of our closure will evaluate the task on our behalf ... a process that is rather straightforward given that we would be invoking a built-in method (`shell.policyA`).

Ⓐ Task manager: Declaration and Definition

In order to create (declare and define) an instance of the task manager, we require the Hub to be offline 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 `__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 callback; 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.list.grainCoarse.\
5            policyA.defun(signature = 'signature::Hub',
6                          stage1Cb  = __taskStat,
7                         );
8
9   __pc = __init();
```

```
r"""
 task<list>          :: struct {
   # SPM component ...
   spm               :: struct {
     meta            :: struct {
       label         :: scalar<stringSnippet> = deferred;
       api           :: scalar<ApiMethod>     = 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 ...
 };
 """
```

Fig. 11: *Typedef for the definition of list of tasks.*

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

Fig. 12: *Hierarchy of (parallel) SPM exceptions.*

Ⓐ Task manager: Population and Invocation

Our goal in the function `main` is to be able to invoke the task manager (line 18). 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 11 through 16.

Once our task manager is invoked, the Hub transitions to the online state. The transition back to offline 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           taskApi,
 5           taskApiArgs,
 6           taskTimeout):
 7    # Initialize 'stage0'.
 8    __pc.stage0.init.main(typedef = ...); # See Figure 11.
 9    hdl = __pc.stage0.payload.tie();
10    # Create a list of tasks
11    for entry in taskApiArgs:
12      hdl.spm.meta.label    = '***'; # Not interested.
13      hdl.spm.meta.api      = taskApi;
14      hdl.spm.meta.apiArgs = entry;
15      hdl.spm.meta.timeout = taskTimeout;
16      hdl.Push();
17    # Invoke the pmanager
18    __pc.stage0.event.manage(pool                  = pool,
19                             nSpokesMin            = ...
20                             nSpokesMax            = ...
21                             timeoutWaitForSpokes = ...
22                             timeoutExecution      = ...
23                             );
24    return;
```

Ⓑ Task manager: (Final) Status Reports

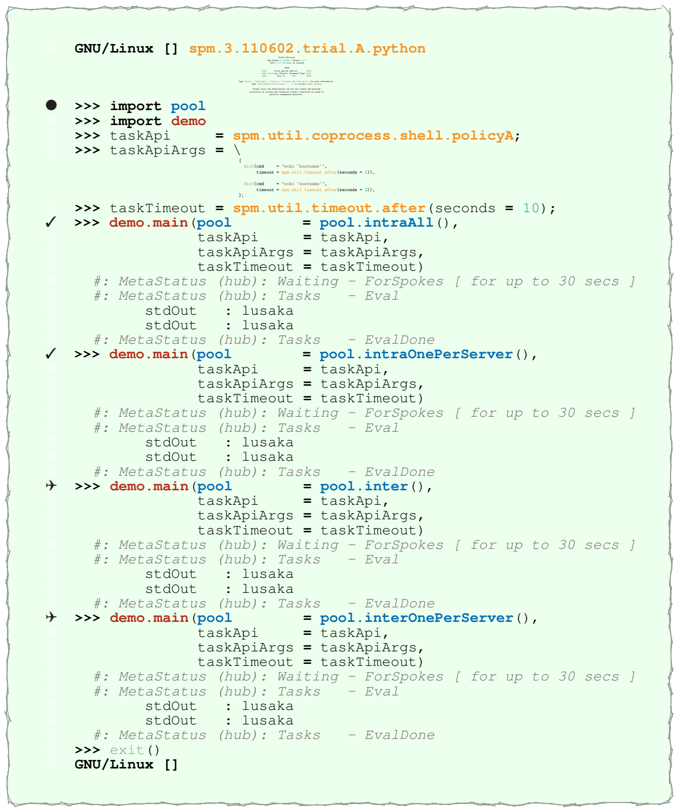
The method `__taskStat` (used when declaring and defining our closure) is automatically invoked by the task manager to process the status report of any task. Note that this method is invoked while the Hub is in the online state. This invariant is captured by the decorator statements on lines 1 and 2.

```
 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 coprocess 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.



```
   GNU/Linux [] spm.3.110602.trial.A.python
                              (Trial Edition)
                    Spm.Python 3.110602 / Python 2.6.6
                      [GCC 4.4.3 (64 bit) on linux2]
                                   NOTE
                 >>>>       Trial period ends at          <<<<
                 >>>> 24:00 hrs (Pacific Standard Time)   <<<<
                 >>>>          July 28,     2011          <<<<
       Type "help", "copyright", "credits", "license" or "spm.Api()" for more information.
               Type "spm.DemoExtract(dirname = ...)" to extract demo scripts.
            Please visit www.mbasciences.com for the latest and growing
           collection of scripts and technical briefs classified in terms of
                            parallel management patterns.

●  >>> import pool
   >>> import demo
   >>> taskApi     = spm.util.coprocess.shell.policyA;
   >>> taskApiArgs = \
                     [
                      dict(cmd     = "echo `hostname`",
                           timeout = spm.util.timeout.after(seconds = 1)),

                      dict(cmd     = "echo `hostname`",
                           timeout = spm.util.timeout.after(seconds = 2)),
                     ];
   >>> taskTimeout = spm.util.timeout.after(seconds = 10);
✓  >>> demo.main(pool        = pool.intraAll(),
                 taskApi     = taskApi,
                 taskApiArgs = taskApiArgs,
                 taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut   : lusaka
          stdOut   : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✓  >>> demo.main(pool        = pool.intraOnePerServer(),
                 taskApi     = taskApi,
                 taskApiArgs = taskApiArgs,
                 taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut   : lusaka
          stdOut   : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✈  >>> demo.main(pool        = pool.inter(),
                 taskApi     = taskApi,
                 taskApiArgs = taskApiArgs,
                 taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut   : lusaka
          stdOut   : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
✈  >>> demo.main(pool        = pool.interOnePerServer(),
                 taskApi     = taskApi,
                 taskApiArgs = taskApiArgs,
                 taskTimeout = taskTimeout)
     #: MetaStatus (hub): Waiting - ForSpokes [ for up to 30 secs ]
     #: MetaStatus (hub): Tasks   - Eval
          stdOut   : lusaka
          stdOut   : lusaka
     #: MetaStatus (hub): Tasks   - EvalDone
   >>> exit()
   GNU/Linux []
```

Fig. 13: *A typical parallel session of SPM.Python.*

The automatic evaluation of our tasks is aided by the typedef used when initializing `stage0` (at the Hub). Specifically, all Spokes end up executing the pseudo-code:

```
try:
  task.spm.stat.returnValue = apply(task.spm.meta.api,
                                     (),
                                     task.spm.meta.apiArgs);
except e:
  task.spm.stat.exception   = str(e);

return task;
```

SPM.Python Session

Having reviewed our parallel application, we will conclude by describing an actual SPM.Python session. 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.

The

fact that the results produced are identical should not be a surprise since our code is a function of a handle to a pool, and not its content. In other words, user code 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 robust timeout support), but also robust (thanks to the efficient manner by which parallel invariants are enforced). So, once we have tested our solution in a serial-like environment, we can be sure our solution can be deployed on any cluster. See [8] for a comprehensive list of problem decomposition using other PMPs including self contained and equally powerful examples.

Conclusion

In this paper, we reviewed the technical anatomy of SPM.Python, a scalable parallel version of the serial Python language. We began with a prologue presenting the acid test for general purpose parallel computing. Next, we described the solution to four formerly open technical problems, namely the decoupling of tracking of resources from management of resources; the declaration and definition of parallel closures; the design and architecture of parallel closures that delineate serial and parallel components; and fault-tolerant parallel exception handling. 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

[1] Celery, celeryproject.org
[2] Parallel Python, www.parallelpython.com
[3] Disco, discoproject.org
[4] PaPy, code.google.com/p/papy
[5] PyMPI, mpi4py.sourceforge.net
[6] PyPVM, pypvm.sourceforge.net
[7] Minesh B. Amin,. *Resource Tracking Method and Apparatus*, United States Patent #: 7,926,058 B2, April 12, 2011.
[8] Parallel Management Patterns, www.mbasciences.com/pmp.html

Appendix A

Figures 14 through 16 highlight the manner by which any module can be parallelized using SPM.Python. Specifically, a serial module that computes number of prime numbers within a given range (Figure 14) is parallelized by introducing two wrappers as depicted by Figure 15 (for Spoke), and Figure 16 (for Hub). Recall that SPM.Python has built-in support for multiple different and distinct forms of parallelism. However, for our purpose, we are only interested in the closure that executes a list of tasks in parallel.

```
#
# Serial module to compute prime numbers
#
def am(n):
    #
    # Came across this algo on the internet.
    #
    import math
    n = abs(n)
    i = 2
    while i <= math.sqrt(n):
        if n % i == 0:
            return False
        i += 1
    return True

def ctRange(nMin, nMax):
    if ((nMin % 2) == 0):
        nMin = nMin + 1; # Focus on odd numbers (!)

    nprimes = 0;
    while (nMax > nMin):
        if (am(nMin)):
            nprimes += 1;

        nMin += 2;

    return nprimes;
```

Fig. 14: *Spoke: Original 'serial' module that computes the number of prime numbers given a range.*

```
#
# Compute the number of primes between 3 and 502347 ...
#

@spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amSpoke)
def taskEval(pc):
    from serial import ctRange as ctRange;

    hdl                    = pc.stage2.payload.tie();
    hdl.spm.stat.returnValue = ctRange(nMin = hdl.nMin,
                                       nMax = hdl.nMax,
                                       );

    return (pc.stage2.event.done(),
            None,
           )[-1];
```

Fig. 15: *Spoke: Wrapper around serial functionality. The wrapper is automatically invoked by SPM.Python based on the content of the task's 'spm' sub-structure.*

```
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __init():
    # Create parallel closure (task manager) of the type
    # we are interested in (coarse grain parallel list manager) ...
    return spm.pclosure.macro.pinterp.list.grainCoarse.policyA.defun \
               (signature = 'is_prime::main', # Something unique to module.
                stage1Cb  = __taskStat,
               );

__pc      = __init();
__nprimes = 0;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __taskStat(pc):
    # Callback for incoming status reports ...
    try:
        global __nprimes;

        hdl        = pc.stage1.payload.tie();
        __nprimes += hdl.spm.stat.returnValue;
        print('  --> Rolling count of (# of prime numbers) :: %d' \
              % (__nprimes,));
    except (SPMTaskDropped,
            SPMTaskLoad,
            SPMTaskEval,
            ), (hdl,):
        pass;

    return (pc.stage1.event.done(), # Explicitly let the backend
                                    # know we are done;
            None,
            )[-1];

#
# Compute the number of primes between 3 and 502347
# by dividing the range into 'nBuckets' ...
#
import os;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def main(pool,
         nBuckets = 10,
         ):
    # Initialize 'stage0'.
    global __nprimes;

    assert(nBuckets >= 1);
    __pc.stage0.init.main(typedef = \
        r"""
         task<list>::struct {
           #
           # SPM component ...
           #
           spm::struct {
             meta::struct {
               label        ::scalar<stringSnippet> = deferred;
               path         ::tuple<string>         = deferred;
               modulePreload::tuple<string>         = deferred;
               module       ::scalar<stringSnippet> = deferred;
               timeout      ::scalar<timeout>       = deferred;
             };

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

             stat::struct {
               exception    ::scalar<auto>          = None;
               returnValue  ::scalar<auto>          = None;
             };
           };

           #
           # non-SPM component ...
           #
           nMin             ::scalar<auto>          = deferred;
           nMax             ::scalar<auto>          = deferred;
         };
         """);
    __nprimes = 0;                         # Always reset counter.
    hdl       = __pc.stage0.payload.tie(); # Handle to the payload.
    nMin      = 2;
    for ct in range(0, nBuckets):
        # Initialize 'spm' component so that Spokes know what to
        # preload ...
        hdl.spm.meta.label         = '***';
        hdl.spm.meta.path          = \
             (os.path.dirname(__pc.meta.module.srcDir),);
        hdl.spm.meta.modulePreload = ('is_prime',);
        hdl.spm.meta.module        =  'is_prime';
        hdl.spm.meta.timeout       = \
             spm.util.timeout.after(seconds = 10);
        hdl.nMin                   = nMin; nMin += ((502347) / nBuckets);

        if (ct == (nBuckets - 1)):
            hdl.nMax = 502347;
        else:
            hdl.nMax = nMin;

        hdl.Push();

    #
    # Invoke the pmanager ...
    #
    __pc.stage0.event.manage \         (pool                 = pool,
         nSpokesMin           = spm.env.const.default,
         nSpokesMax           = spm.env.const.default,
         timeoutWaitForSpokes = spm.util.timeout.after(seconds = 2),
         timeoutExecution     = spm.util.timeout.after(seconds = 300),
        );

    return;
```

Fig. 16: *Hub: Creation/population/invocation of parallel (task manager) closure. The backend of the closure, once invoked, would execute as many tasks in parallel as possible using resources within the* **pool***.*