A Technical Anatomy of SPM.Python
(A Scalable, Parallel Version of Python)

Minesh B. Amin
mamin@mbasciences.com
http://www.mbasciences.com

SciPy 2011 - Python and Core Technologies
Austin, Texas
Jul 13, 2011
Our story starts with a very simple observation ... on the left, we have a typical serial session made up of multiple invocations of serial modules. We would like to do the same thing in the parallel session, i.e. invoke multiple parallel modules, each potentially using the same hardware resources in very different ways.

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.
For a parallel language to be useful, the entire solution surrounding the parallel language needs to address three sources of friction as experienced by software architects, software developers, and IT teams.

Software architects need a scalable vocabulary to better capture the essence of their parallel problem. So, the typical approach of describing everything in terms of either send/recv or MapReduce is simply not rich enough.

Meanwhile, software developers need to be able to perform rapid prototyping. However, this ability to prototype is only possible if the semantics of the parallel language has a well-defined and built-in notion of fault-tolerance and the ability to self-clean.

Finally, IT teams should not need to be certified in order for programs developed in the parallel language to be executed on some cluster. After all, our goal is to be able to use the same resources in completely different ways within the same session. Therefore, once the software architects define an architecture and software developers implement a parallel solution, IT teams should limit themselves to managing and monitoring resources independent of how the said resources are utilized.
For a parallel language to be useful, the entire solution surrounding the parallel language needs to address three sources of friction as experienced by software architects, software developers, and IT teams.

Software architects need a scalable vocabulary to better capture the essence of their parallel problem. So, the typical approach of describing everything in terms of either send/recv or MapReduce is simply not rich enough.

Meanwhile, software developers need to be able to perform rapid prototyping. However, this ability to prototype is only possible if the semantics of the parallel language has a well-defined and built-in notion of fault-tolerance and the ability to self-clean.

Finally, IT teams should not need to be certified in order for programs developed in the parallel language to be executed on some cluster. After all, our goal is to be able to use the same resources in completely different ways within the same session. Therefore, once the software architects define an architecture and software developers implement a parallel solution, IT teams should limit themselves to managing and monitoring resources independent of how the said resources are utilized.
For a parallel language to be useful, the entire solution surrounding the parallel language needs to address three sources of friction as experienced by software architects, software developers, and IT teams.

Software architects need a scalable vocabulary to better capture the essence of their parallel problem. So, the typical approach of describing everything in terms of either send/recv or MapReduce is simply not rich enough.

Meanwhile, software developers need to be able to perform rapid prototyping. However, this ability to prototype is only possible if the semantics of the parallel language has a well-defined and built-in notion of fault-tolerance and the ability to self-clean.

Finally, IT teams should not need to be certified in order for programs developed in the parallel language to be executed on some cluster. After all, our goal is to be able to use the same resources in completely different ways within the same session. Therefore, once the software architects define an architecture and software developers implement a parallel solution, IT teams should limit themselves to managing and monitoring resources independent of how the said resources are utilized.
Software architects need to be able to classify their problem in terms of one of the Parallel Management Patterns (PMPs). Typically, this process should not take more than 5 minutes.

Armed with the PMP, the software developers should be able to make the transition from concept to initial (fault-tolerant) implementation within minutes. Next, thanks to the parallel semantics of SPM.Python, the developer can build on the initial implementation by rapidly prototyping within the constraints established by the initial implementation.

Finally, the parallel solution may be deployed on any cluster in a scalable, fault-tolerant manner without requiring the configuration of hardware resources or software packages.
Software architects need to be able to classify their problem in terms of one of the Parallel Management Patterns (PMPs). Typically, this process should not take more than 5 minutes.

Armed with the PMP, the software developers should be able to make the transition from concept to initial (fault-tolerant) implementation within minutes. Next, thanks to the parallel semantics of SPM.Python, the developer can build on the initial implementation by rapidly prototyping within the constraints established by the initial implementation.

Finally, the parallel solution may be deployed on any cluster in a scalable, fault-tolerant manner without requiring the configuration of hardware resources or software packages.
Software architects need to be able to classify their problem in terms of one of the Parallel Management Patterns (PMPs). Typically, this process should not take more than 5 minutes.

Armed with the PMP, the software developers should be able to make the transition from concept to initial (fault-tolerant) implementation within minutes. Next, thanks to the parallel semantics of SPM.Python, the developer can build on the initial implementation by rapidly prototyping within the constraints established by the initial implementation.

Finally, the parallel solution may be deployed on any cluster in a scalable, fault-tolerant manner without requiring the configuration of hardware resources or software packages.
Software architects need to be able to classify their problem in terms of one of the Parallel Management Patterns (PMPs). Typically, this process should not take more than 5 minutes.

Armed with the PMP, the software developers should be able to make the transition from concept to initial (fault-tolerant) implementation within minutes. Next, thanks to the parallel semantics of SPM.Python, the developer can build on the initial implementation by rapidly prototyping within the constraints established by the initial implementation.

Finally, the parallel solution may be deployed on any cluster in a scalable, fault-tolerant manner without requiring the configuration of hardware resources or software packages.
In short, our goal with SPM.Python is to enable software architects and developers to express parallelism in terms of a robust and powerful suite of parallel primitives ... without placing any SPM.Python-specific demands on the IT team.

To use an analogy, software architects and developers should be able to drive a car without knowing the details how of the engine works ... not because the engine is unimportant, but because it frees the architects and developers to focus on solving their problem and create value-added applications while leaving the non-differentiating heavy lifting on the parallel side to SPM.Python parallel primitives.
Before we dive into the anatomy of SPM.Python, a few words on basic terminology. Here we are in 2011, and notwithstanding all the buzz around cloud and parallel computing, there is no consensus on what the software industry or academia mean by the term “parallelism”.

We believe that parallelism entails nothing more than the management of a collection of serial tasks. Here, “management” is a fairly loaded term, and includes policies by which tasks are scheduled, while “serial tasks” come in two flavors depending on whether they may communicate or not.

One particular aspect of “management” bears highlighting, namely the ability for parallel managers to enable and disable communication primitives. Our conjecture is:

How tasks are managed has a direct bearing on what types of communication primitives the said tasks may leverage. Conversely, the usage of a particular type of communication primitive has a direct bearing on how the respective tasks must be managed.

Therefore, to avoid the vast majority of parallel deadlocks, managers must enable only compatible communication primitives.

© 2011 MBA Sciences, Inc. www.mbasciences.com
Recall that our goal is to be able to express parallelism in terms of parallel primitives that are baked into SPM.Python. The ability to express parallelism is predicated on the ability to safely declare and define instances of parallel primitives.

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.

**Offline**: A state when serial functionality cannot communicate

**Online**: A state when serial functionality may communicate
Anatomy: Tracker

Need a way to monitor resources independent of any task manager ...

On to the anatomy of SPM.Python ... consider a situation where we are waiting at the prompt of the Hub:

```python
>>> 
```

And, in the meantime, say, a resource/Spoke attempts to connect with the Hub. What should SPM.Python do? Clearly, the Python interpreter at the Hub cannot get involved as it is blocked at the prompt. But, it should also be clear that this attempt to connect must be somehow processed in real-time.

A similar situation can arise when a resource/Spoke disconnects from the Hub for any reason while the Hub is waiting at the prompt.

Thus, the need for a "tracker", a module designed to be active at all times independent of the Python interpreter and any task manager, and is in charge of nothing but tracking resources.
Recall that our goal is to be able to express parallelism in terms of parallel primitives that are baked into SPM.Python. The ability to express parallelism is, thus, predicated on the ability to safely declare and define instances of parallel primitives.

In other words, exploiting parallelism is anchored around the asynchronous declaration and definition of parallel primitives across all resources (Hub and Spokes). On the Hub, this is depicted by (A). On the Spokes, this is only possible prior to the evaluation of a task, as depicted by (B).

Furthermore, note that 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 (C), 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 (D) for the task; the transition back to offline does not occur until just after the callback concludes.
An instance of a coarse grain list task manager

```python
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline) # serial stat -> Am offline
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub) # parallel stat -> Am Hub
def __init__():
    return spm.pclosure.macro.papply.list.grainCoarse.policyA.defun(
        signature = 'signature::mainHub', # Something unique to module.
        stage1Cb = __taskStat,
    );
__pc = __init__();
```

An instance of a coarse grain template task manager

```python
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline) # serial stat -> Am offline
@spm.util.dassert(predicateCb = spm.sys.pstat.amSelf) # parallel stat -> Am Hub or Spoke
def __init__():
    return spm.pclosure.micro.aggregateRank.policyA.defun(
        signature = 'signature::_util', # Something unique to module.
        stage2Cb = __recvSignature,
        stage5Cb = __recvPayload);
__pc = __init__();
```

An instance of a communication primitive

```python
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline) # serial stat -> Am offline
@spm.util.dassert(predicateCb = spm.sys.pstat.amSelf) # parallel stat -> Am Hub or Spoke
def __init__():
    return spm.pclosure.micro.aggregateRank.policyA.defun(
        signature = 'signature::_util', # Something unique to module.
        stage2Cb = __recvSignature,
        stage5Cb = __recvPayload);
__pc = __init__();
```
Anatomy: Task Management P closures

How can one parallel language possibly provide a suite of:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management Pcloseres

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management P closures

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management Pclosures

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning
- robust

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management P closures

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning
- robust
- fundamentally different

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management Pclosures

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning
- robust
- fundamentally different
- very powerful, and yet easy-to-relate-to

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
Anatomy: Task Management Pclosures

How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning
- robust
- fundamentally different
- very powerful, and yet easy-to-relate-to

task managers?

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
How can one parallel language possibly provide a suite of:

- fault-tolerant
- self-cleaning
- robust
- fundamentally different
- very powerful, and yet easy-to-relate-to

task managers?

Because all types of task managers have one fundamental property ...

To recap, we reviewed the tracker and the logistics for declaring and defining parallel primitives, because we want software developers to think in terms of parallel primitives.

So, here we are facing the most critical challenge. How can SPM.Python single-handedly, without any external dependencies, packages, utilities, or support from IT, provide a suite of primitives that are:

- fault-tolerant - from day one
- self-cleaning - so that software developers and IT teams do not have to dedicate resources to remove runtime artifacts left behind in the event of any premature or self-induced terminations (due to timeouts)
- robust - to ensure that once a problem is classified in terms of a specific PMP, and implemented using appropriate primitives, that any and all parallel invariants are tracked and enforced
- fundamentally different - DAG/template/list forms of both fine and coarse grain parallelism
- powerful, and yet easy-to-relate-to - 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).

For this talk, we shall focus on task managers; the same set of requirements apply to communication primitives.

Our goal is to leverage a powerful parallel enabling technology expressed naturally using a parallel language, not a collection of frameworks. Furthermore, even SPM.Python cannot, behind the scenes, treat each type of task manager as a stand-alone framework if for no other reason than the prohibitive cost of testing, validating, verifying and maintaining highly non-deterministic parallel sub-components of the parallel primitives.
All types of task managers must be able to recognize the following events ...
On the Hub, we must recognize the following events:

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
Anatomy: Task Management P closures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Hub, we must recognize the following events:

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
All types of task managers must be able to recognize the following events ...

On the Hub, we must recognize the following events:

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
All types of task managers must be able to **recognize** the following events ...

- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
Anatomy: Task Management Pcloseures - Cont’d

All types of task managers must be able to recognize the following events ...

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
Anatomy: Task Management Closures - Cont’d

All types of task managers must be able to **recognize** the following events ...

On the Hub, we must recognize the following events:

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
All types of task managers must be able to recognize the following events:

- the declaration and definition of a task manager
- the act of populating a task manager. For example, a DAG task manager must be populated with a DAG of tasks, while a list task manager must be populated with a list of tasks
- the act of invoking a task manager, transitioning to the online state, and enabling compatible communication primitives
- once online, a task manager must commence with the scheduling of tasks
- the invocation of the callback to process an incoming status report of some task
- at the conclusion of the invocation of the callback, if possible, the act of scheduling additional pending tasks
- the transition back to offline just prior to conclusion
Anatomy: Task Management Plosures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:

- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
Anatomy: Task Management P closures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:

- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
Anatomy: Task Management Pcloseures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:

- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
Anatomy: Task Management P closures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:

- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
Anatomy: Task Management Pclosures - Cont’d

All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:

- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
All types of task managers must be able to recognize the following events ...

On the Spoke, we must recognize the following events:
- the act of accepting a task on behalf of the ultimate task evaluator
- the act of preloading any Python modules prior to task evaluation
- the act of transitioning to online, enabling compatible communication primitives and invoking a task evaluator
- the act of leveraging any enabled communication primitive
- the transition back to offline just after the conclusion of the task evaluator, and reporting of the final status report of the task to the Hub

Finally, all situations where an unexpected event may occur are depicted in red. Such events include any premature or self-induced termination, uncaught exception, and violation of any parallel invariant.

On the Hub, these events include any uncaught exceptions thrown by the callback that processes task reports. Additionally, any or all forms of premature termination detected while scheduling tasks need to be properly accounted for.

On the Spoke, the red events include any uncaught exceptions thrown during the preloading of any Python modules prior to invocation of the task evaluator. Additionally, any and all forms of uncaught exceptions thrown during the evaluation of a task need to be properly accounted for.

 поверхностные

Ok, so, what accounts for the clear differences in functionality among all types of task managers if they all have to recognize the same set of events?
All types of task managers must be able to recognize the following events ...

How they interpret these events is what differentiates one type of task manager from another (!)

Note that the act of interpreting events includes the processing of any and all side-effects of the respective events. For example, for some types of task managers, an event indicating premature termination of some task may trigger side-effects that include the forcible termination of all other active tasks. On the other hand, the same event may trigger no side-effects for other types of task managers.

Nevertheless, thanks to this particular decomposition, SPM.Python can safely centralize the logistics of how events are to recognized. Furthermore, each type of task manager may now safely inherit the said logistics, while defining and implementing a customized response to each event.

The end result of this decomposition is the flexibility to introduce a suite of new and very powerful task managers within the constraints established by the mechanism by which all events are recognized.
Stated another way, given a set of (both good and bad) events, different interpretations would give rise to different forms of parallelism.

For example, a task manager designed to express the parallelism implied by the Partition/List Parallel Management Pattern (PMP) may process the set of events to execute a list of tasks in a fault-tolerant, self-cleaning and robust manner across a collection of compute resources.

Alternatively, a task manager designed to express the parallelism implied by the Partition/DAG Parallel Management Pattern (PMP) may process the same set of events to execute a DAG of tasks in a fault-tolerant, self-cleaning and robust manner across a collection of compute resources.

For a comprehensive list of PMPs, please refer to: www.mbasciences.com/pmp.html
def main(pool,
    taskApi = spm.util.coprocess.shell.policyC,
    taskApiArgs = [   # SPM component ...
        {'cmd': "echo 'hostname' -@- 'uptime'",
         'timeout': spm.util.timeout.after(seconds = 2), # Api should finish within 2 seconds
        },
        {'cmd': "echo 'hostname' -@- 'uptime'",
         'timeout': spm.util.timeout.after(seconds = 2), # Api should finish within 2 seconds
        },
    ],
    taskTimeout = spm.util.timeout.after(seconds = 10)); # Task should finish within 10 seconds
    # Enforce invariants ...
    assert(taskApi in {spm.util.coprocess.shell.policyA,
                       spm.util.coprocess.shell.policyB,
                       spm.util.coprocess.shell.policyC},
    )
    # Initialize 'stage0'.
    __pc.stage0.init.main(typedef = r""
        task<list> {
           # SPM component ...
           struct spm {
              struct meta {
                 scalar<stringSnippet> label = deferred;
                 scalar<ApiMethod> api = deferred;
                 dict<string,mixed> apiArgs = deferred;
                 scalar<timeout> timeout = deferred;
              };
              struct core {
                 scalar<bool> relaunchPre = None;
                 scalar<bool> relaunchPost = None;
                 scalar<auto> nameHost = None;
                 scalar<auto> whoAmI = None;
              };
              struct stat {
                 scalar<auto> exception = None;
                 scalar<record> returnValue = None;
              };
           } # non-SPM component ...
        }""
    )
    hdl = __pc.stage0.payload.tie(); # Handle to the payload.
    # Create a list of tasks
    for entry in taskApiArgs:
        hdl.spm.meta.label = ' ***'; # Not interested, so any string (length < 35) will do.
        hdl.spm.meta.api = taskApi;
        hdl.spm.meta.apiArgs = entry;
        hdl.spm.meta.timeout = taskTimeout; # Builtin method.
        hdl.Push();
    # Invoke the pmanager
    __pc.stage0.event.manage(pool = pool,
        nSpokesMin = spm.env.const.default, # Minimum degree of parallelism
        nSpokesMax = spm.env.const.default, # Maximum degree of parallelism
        timeoutWaitForSpokes = spm.util.timeout.after(seconds = 2),
        timeoutExecution = spm.util.timeout.after(seconds = 300),
    );
    return;
def main(pool):
    taskApi = spm.util.coprocess.shell.policyC,
    taskApiArgs = {'cmd': 'echo 'hostname' -@- 'uptime''}
    # Api should finish within 2 seconds
    taskTimeout = spm.util.timeout.after(seconds = 2)
    # Task should finish within 10 seconds

    # Enforce invariants ...
    assert taskApi in (spm.util.coprocess.shell.policyA,
                        spm.util.coprocess.shell.policyB,
                        spm.util.coprocess.shell.policyC,
                        )

    # Initialize 'stage0'.
    __pc.stage0.init.main(typedef = r'''
    task<template> {
        # SPM component ...
        struct spm {
            struct meta {
                scalar<stringSnippet> label = deferred;
                scalar<ApiMethod> api = deferred;
                dict<string,mixed> apiArgs = deferred;
                scalar<timeout> timeout = deferred;
            }
            struct core {
                scalar<bool> relaunchPre = None;
                scalar<bool> relaunchPost = None;
                scalar<auto> nameHost = None;
                scalar<auto> whoAmI = None;
            }
            struct stat {
                scalar<auto> exception = None;
                scalar<record> returnValue = None;
            }
        }
        # non-SPM component ...
    }
    ''');

    hdl = __pc.stage0.payload.tie();
    # Handle to the payload.

    # Create a template task
    hdl.spm.meta.label = '***';
    hdl.spm.meta.api = taskApi;
    hdl.spm.meta.apiArgs = taskApiArgs;
    hdl.spm.meta.timeout = taskTimeout;
    # Invoke the manager
    __pc.stage0.event.manage(pool = pool,
        nSpokesMin = spm.env.const.default, # Minimum degree of parallelism
        nSpokesMax = spm.env.const.default, # Maximum degree of parallelism
        timeoutWaitForSpokes = spm.util.timeout.after(seconds = 2),
        timeoutExecution = spm.util.timeout.after(seconds = 300),
    );

    return;
Recall that our goal with SPM.Python is to enable the software architects and developers to express parallelism in terms of a robust and powerful suite of parallel primitives ... without placing any SPM.Python specific demands on the IT teams.

Let's conclude by tying together the event types introduced when reviewing the software developer’s perspective.

Consider the phrase all developers should be agonizingly familiar with:

```
some command/module is “very fragile”
```

Well, what does the phrase “very fragile” mean?

The source of the fragility can be traced back to the inability of the parallel solution to recognize and process unexpected events and conditions. Stated another way, it means that the author of the task manager completely punted on the recognition and interpretation of most, if not all, red events; thus, leading to the deeply frustrating behavior of the parallel solution in question.

After all, why should the software behave rationally if any event outside the norm is completely missed, mis-diagnosed, or skipped outright?
Recall that our goal with SPM.Python is to enable the software architects and developers to express parallelism in terms of a robust and powerful suite of parallel primitives ... without placing any SPM.Python specific demands on the IT teams.

Let’s conclude by tying together the event types introduced when reviewing the software developer’s perspective.

Consider the phrase all developers should be agonizingly familiar with: some command/module is “very fragile”

Well, what does the phrase “very fragile” mean?

The source of the fragility can be traced back to the inability of the parallel solution to recognize and process unexpected events and conditions. Stated another way, it means that the author of the task manager completely punted on the recognition and interpretation of most, if not all, red events; thus, leading to the deeply frustrating behavior of the parallel solution in question.

After all, why should the software behave rationally if any event outside the norm is completely missed, mis-diagnosed, or skipped outright?
Consider another phrase all developers should be agonizing familiar with:

```
some command/module "recovers from some errors"
```

Well, what does the phrase “recovers from some errors” mean?

Recall our observation that not all red events are equally easy to recognize, and therefore interpret. The easiest subset of red events are the ones that occur on the Spoke while the toughest subset of red events are the ones that occur on the Hub.

In this case, it would appear that the author of the task manager completely punted on the recognition and interpretation of the toughest set of red events - typically those that occur on the Hub; thus, leading to the frustrating behavior of the parallel solution in question.

Again, why should the software behave rationally if any abnormal event on the Hub is completely missed, mis-diagnosed, or skipped outright?
Consider another phrase all developers should be agonizing familiar with:

```
some command/module "recovers from some errors"
```

Well, what does the phrase “recovers from some errors” mean?

Recall our observation that not all red events are equally easy to recognize, and therefore interpret. The easiest subset of red events are the ones that occur on the Spoke while the toughest subset of red events are the ones that occur on the Hub.

In this case, it would appear that the author of the task manager completely punted on the recognition and interpretation of the toughest set of red events - typically those that occur on the Hub; thus, leading to the frustrating behavior of the parallel solution in question.

Again, why should the software behave rationally if any abnormal event on the Hub is completely missed, mis-diagnosed, or skipped outright?
Consider yet another phrase only a select few developers should be happily familiar with:

some command/module “works most of the time”

Well, what does the phrase “works most of the time” mean?

It means that the author of the task manager managed to recognize and interpret all the bad events except for the toughest of the tough events ... typically, those that have a rather complicated set of side-effects.
Consider yet another phrase only a select few developers should be happily familiar with:

some command/module “works most of the time”

Well, what does the phrase “works most of the time” mean?

It means that the author of the task manager managed to recognize and interpret all the bad events except for the toughest of the tough events ... typically, those that have a rather complicated set of side-effects.
And, finally, consider the ultimate, and rather rare phrase only an amazingly tiny number of developers should be happily familiar with:

\texttt{some command/module \textquoteleft is fault-tolerant\textquoteright}

This is pure nirvana because the solution would never hang, always conclude, and never leave zombie processes behind.

This state of nirvana is only possible when the author of the task manager recognizes and interprets any and all red events across both the Hub and all the Spokes.

This is our value proposition. The ability to provide a robust suite of very powerful parallel primitives across the breadth and depth of the parallel landscape.
And, finally, consider the ultimate, and rather rare phrase only an amazingly tiny number of developers should be happily familiar with:

```
some command/module "is fault-tolerant"
```

This is pure nirvana because the solution would never hang, always conclude, and never leave zombie processes behind.

This state of nirvana is only possible when the author of the task manager recognizes and interprets any and all red events across both the Hub and all the Spokes.

This is our value proposition. The ability to provide a robust suite of very powerful parallel primitives across the breadth and depth of the parallel landscape.
And, finally, consider the ultimate, and rather rare phrase only an amazingly tiny number of developers should be happily familiar with:

```
some command/module "is fault-tolerant"
```

This is pure nirvana because the solution would never hang, always conclude, and never leave zombie processes behind.

This state of nirvana is only possible when the author of the task manager recognizes and interprets any and all red events across both the Hub and all the Spokes.

This is our value proposition. The ability to provide a robust suite of very powerful parallel primitives across the breadth and depth of the parallel landscape.
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.

For more details, please refer to:

- [www.mbasciences.com](http://www.mbasciences.com)
- [www.mbasciences.com/pmp.html](http://www.mbasciences.com/pmp.html)