GSPC19 Architecture

1 Introduction

This document describes the new architecture for GSPC19, focusing on scheduling and on the interaction between components.

2 Architecture

The main components shown in Figure 1 are described as follows, together with a set of definitions that will be used in this document.

Table 1: Terms used in this document

Res	Resource; each resource in the system is associated with a unique ID
RClass	Resource class; type of resources (e.g., core, socket, node, GPU)
Task	GPISpace activity; includes an ID, state, description (e.g., ports, modules), associated resource class
DAG	DAG of Resources (hierarchical representation of resources, describing the dependencies between existing resource IDs, based on their class; e.g., a socket includes multiple cores)
DAG_report	DAG Report; connection information for each resource in a given DAG
Preference	Resource preference; represents a list of specific resource IDs, one of which should be allocated by the scheduler for a given task
Requirement	Requirement; represents a list of tuples (resource class, list of preferences) to be used for co-allocation scheduling: the task requires one resource for each element of the list, defined by the given class and preferences
JobServ	Job Server
RTS	Run-Time System
IML	Independent Memory Layer

2.1 RTS – The Run-Time System

The *Run-Time System* is in charge of initializing RIF daemons on each compute node, as well as starting (or connecting to an existing) Resource Manager. The *Run-Time System* also keeps track of all the resources associated with its physical machines and of the mapping between resources, hostnames, and corresponding RIF processes.

It interacts with the following components:

RIF: The RTS can start/stop multiple RIF daemons

RM: The RTS works in conjunction with one RM; whereas the RTS manages the connections to physical resources, the RM keeps track of usage and allocations for the same set of resources

User: The user is the only component in charge (and capable) of initializing and configuring the RTS

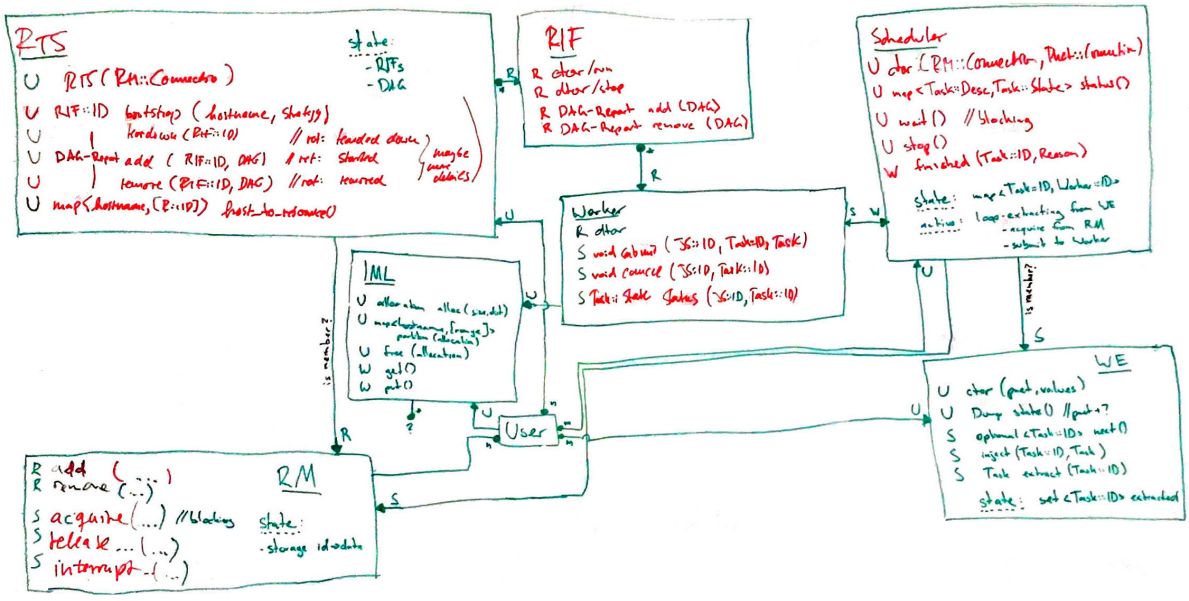

Figure 1: GSPC19 Architecture

Table 2: Run-Time System

Caller	Method
User	RTS (RM::Connection)
User	RIF::ID bootstrap(hostname, strategy)
User	RIF::ID teardown(RIF::ID) // ret: toredown
User	DAG_report add(RIF::ID, DAG) // ret: started
User	DAG_report remove(RIF::ID, DAG) // ret: removed
User	map<hostname, [Res::ID]> host_to_resource()
	Data members
	RIFs
	DAGs

2.2 RIF

The RIF daemon is a process that runs on each compute node managed by GPISpace and manages the GPISpace resources defined for that particular machine. Each resource ID is associated with a worker that can execute tasks defined for that type of resource. Upon receiving a resource description (DAG describing the resource hierarchy for the node), the RIF daemon is in charge of starting a worker for each resource in the DAG, and return the connection information for each worker back to the user. The workers can be implemented either as processes or as threads. The decision here should consider the fact that thread failures in task computations can affect other workers, which may need to be canceled.

The RIF should provide transparent access to workers to the RTS, that is it should make connection information available for each resource it manages. The RIF could then act as a proxy and forward tasks to the right worker or allow external processes to connect to workers directly.

It interacts with the following components:

Worker: The RIF can start/stop multiple Workers (either as processes or as threads); it can potentially handle pinning to cores for specific workers (e.g., socket workers)

RTS: The RTS manages all the communication with all the RIF daemons in the system, and forwards the worker information to the user

IML: Currently, the RIF initializes the IML on the node it is responsible for

Table 3: RIF

Caller	Method
RTS	ctor/run
RTS	dtor/stop
RTS	DAG_report add(DAG)
RTS	DAG_report remove(DAG)
RTS	start_iml() // current implementation

2.3 Worker

The Worker is the component that executes workflow tasks on compute nodes. A Worker is a process/thread associated with a specific resource ID and can only execute tasks requiring that type of resource.

Instead of executing a single task at a given moment, a worker may be implemented to overlap communication (incoming data transfers, outgoing data transfers) with the computation phase of another task. Thus it needs to keep track of the three tasks that may run concurrently, and to synchronize their phases. Furthermore, the failure of one of these running tasks should trigger the failure/cancellation of the others.

Workers interact with the following components:

Scheduler: The Scheduler starts/cancels tasks on any worker, and can as well interrogate the status of a given task

RTS: The RTS manages the creation of workers and keeps track of connection information for each Worker

IML: The Worker needs access to the global memory in order to perform memory transfers

Table 4: Worker

Caller	Method
RTS	ctor
RTS	dtor
Scheduler	void submit(JobServ::ID, Task::ID, Task)
Scheduler	void cancel(JobServ::ID, Task::ID)
Scheduler	Task::State status(JobServ::ID, Task::ID)

2.4 RM – Resource Manager

The Resource Manager is the component that keeps track of all existing resources in the system, together with their state (available or in use).

Interactions:

RTS: The RTS can add and remove resource hierarchies (DAGs) from the

Scheduler: The scheduler is the only component that can acquire/release resources (but the system should allow for multiple schedulers to use the same RM, for instance to have multiple applications running on the same physical machines). The provides a means to mark resources that are in use, while taking into account resource hierarchies; this is done in a transparent manner relative to the scheduler, which only needs to keep track of the class of resource it needs, without any knowledge of dependent resources

In the case of scheduling with preferences, we define the following types:

- Preference:= [Res::ID]
- Requirement:= (RClass, [Preference])

Table 5: Resource Manager

Caller	Method
RTS	add(DAG <Res::ID >)
RTS	remove(DAG <Res::ID >)
Scheduler	acquire(JobServ::ID, RClass) // blocking, return Res::ID
Scheduler	release(JobServ::ID, Res::ID)
Scheduler	release(JobServ::ID, [Res::ID])
Scheduler	interrupt(JobServ::ID)
Scheduler	acquire(JobServ::ID, [RClass]) // blocking, co-allocation, return [Res::ID]
Scheduler	acquire(JobServ::ID, [Requirement]) // blocking, scheduling with preferences, return [Res::ID]
Scheduler	acquire(JobServ::ID, RClass, [memsize]) // blocking, return (Res::ID,[Mem::ID, offset])
	Data members
	Storage id -> data

2.4.1 Dealing with Resource Hierarchies

The RM finds out about resources in the system only when a *resource description* is provided by the RTS through an *add* operation. The received resource DAG is then recorded and each resource Res::ID in the graph is added to a class-specific FIFO queue of available resources. Any incoming *acquire* call only interacts with a specific resource queue by requesting a resource class (and possibly even a set of resource IDs of that type).

When no resource hierarchy is defined (i.e., all resources represent independent cores), the RM becomes a round-robin scheduler that serves resources from a single queue. In the case of a hierarchy defined by nodes, sockets and cores, the RM will maintain three resource queues, one for each resource class.

When a core is *acquired*, it is removed from its queue, and the parent socket and its parent node are also removed from their respective queues. If there are other cores in the core queue (or other sockets) belonging to the same node, they are unaffected and can be assigned to other workers. When a node is acquired, all the depending resources are also removed from their queues, i.e., all the node's sockets and cores.

A *release* operation on a core will add it back to the available cores queue, as well as its parent socket and node to their respective queues.

Resources can be removed from the system at the request of the RTS through the *remove* call. The RTS is triggered either by the user (add/remove) or by the RIF (remove after crash). At the RM level, the removal will affect the queues in the same way as an acquire would. Depending on the policy, a *remove* operation can ignore the resources that ar missing from the queues (i.e., this means they are already acquired), and rely on the scheduler to detect that the associated tasks have failed and re-schedule them on different nodes.

2.4.2 Possible Issues

Possible issues with the acquire/release blocking interface of the Resource Manager:

Deadlock: Can occur when acquiring a set of resources in a single call

Livelock: Similar to the deadlock situation, in the case of requiring multiple resources within the same blocking acquire

Class conflict: Starvation may occur in the following cases Resources from different classes are required by concurrent *acquire* calls, e.g., nodes and cores are requested in the same time. Fulfilling the request for *cores* may result in never having a *node* available and thus starving the *node*-level requests.

Another resource class issue might be due to the blocking nature of the *acquire* call. Example: An application with two types of tasks, IO and computational (Calc) tasks that run on separate classes of resources. Let us assume the workflow engine generates many IO tasks followed by corresponding Calc tasks, each of which can be executed as soon as one IO task is finished. However, since the number of available IO workers is limited, the scheduler may block in an IO *acquire* call and only start processing the Calc tasks when all the IO tasks are done. Such behavior may be avoided by implementing one or both of the following mechanisms:

- class-aware task generation at the workflow engine level

- class-aware scheduling, for example implementing one *acquire* thread per resource class, instead of one thread per task in the scheduler

Queue fragmentation with mixed-class acquires: This issue can occur in the following scenario:

Assuming we have first n requests for cores and serve them by allocating n cores from n (all) different sockets. Now n sockets are used even though just n/C are needed (C being the number of cores per socket). An additional *acquire* for a socket will be blocked until a core task is done.

Such a scenario can be mitigated by relying on a "well-behaved" client, which creates complete resource descriptions, i.e., adds entire nodes (with all their sockets and cores) to the system at once, thus inserting related cores in order into the queues. This would mean that consecutive *acquire* calls for a core will most likely obtain cores from the same socket/node.

Releasing the resources at different times will still cause fragmentation and potential delays for jobs that use mixed-class tasks.

2.5 Scheduler

The Scheduler is the component that provides flexibility to GPISpace, in particular by allowing for the implementation of multiple scheduling policies to match application needs. A possible basic implementation is described in the following section, along with several policies targeted at specific use cases.

Interactions:

User: The scheduler is started upon its creation by the user; it provides the user with a method to wait for the end of the workflow execution, as well as a stop function to cancel running tasks

Worker: The Worker executing a task has to be able to notify the Scheduler that the execution has finished, either successfully or due to a failure/cancel.

Table 6: Scheduler

Caller	Method
User	ctor(RM::Connection, Pnet::Connection)
User	map<Task::Desc, Task::State> status()
User	wait() // blocking
User	stop()
Worker	finished(Task::ID, Reason)
	Data members
	map<Task::ID, Worker::ID >

2.6 WE – Workflow Engine

The workflow engine is a simplified version of the current workflow engine of GPISpace, which only needs to be able to handle one workflow (specified by the user at the WE creation) instead of an arbitrary number of concurrent workflows. Its role is to extract available tasks and inject results back into the workflow. It also has to provide a method that informs the user of the state of the workflow, regardless of whether its execution has finished or not.

The most significant change with respect to the current implementation is that the WE does not need to support a *cancel* operation, as stopping the workflow execution can simply be seen as ceasing the calls to *extract* at the scheduler level.

Interactions:

Scheduler: The scheduler drives the extraction of available tasks; it is also responsible for injecting the results of finished tasks back into the workflow

User: The User initializes the WE based on the application workflow; it can at any moment discover the state of the workflow by directly interrogating the WE (regardless of whether the scheduler has finished the execution of the workflow)

Table 7: Workflow Engine

Caller	Method
User	ctor(pnet, values)
User	dump state()
Scheduler	optional<Task::ID > next()
Scheduler	inject(Task::ID, Task)
Scheduler	Task extract(Task::ID)
	Data members
	set<Task::ID > extracted

2.7 IML – Independent Memory Layer

The IML handles the distributed global memory that spans over multiple physical machines. Currently, the memory layer is not completely independent, as its initialization is handled by the RIF daemons.

Interactions:

User: The user is in charge of allocating memory ranges in the IML; it can also require information about data partitioning, that is the mapping between allocated memory and physical machine hostnames where the data is actually located

Worker: Workers need access to the IML to trigger incoming/outgoing data transfers

Table 8: Independent Memory Layer (IML)

Caller	Method
User	allocation alloc(size, data)
User	map<hostname,[range]> partition(allocation)
User	void free(allocation)
Worker	get()
Worker	put()
	Data members
	set<Task::ID > extracted

3 Scheduling Policies

The proposed architecture enables different scheduling policies to be plugged in depending on the application requirements.

3.1 Base Scheduler

The implementation should provide a basic scheduler class with the following simple functionality:

```
1 task_id = WE.next()      // Extract new task from the Workflow Engine
2 res_id = RM.acquire(JobServ , task_id::class)     // Acquire resources for the class
    that the task belongs to
3
4 worker = res_id_to_worker(res_id)    // retrieve assigned worker(s)
5 worker.submit(WE.extract(task_id))
6 Repeat until no more running tasks and no new tasks can be extracted from the
    WE.
7
```

```
// in the finished callback (called by the workers)
if (reason == Finished)
    WE.inject(task_id, workflow_response)
endif
```

3.2 Rescheduling

Failed tasks can be handled through a rescheduling mechanism that performs multiple attempts to acquire resources for a task and execute it upon failure, e.g. in the case of a worker becoming unavailable.

3.3 Look-Ahead Scheduler

Such a scheduler can be implemented for a more fine grained resource management than what the simple *acquire/release* interface can provide. A look-ahead scheduler should be able to extract multiple (or all) available tasks and update this list dynamically when other tasks become available.

For instance, such a look-ahead mechanism is essential to avoid *class conflict* issues by extracting multiple tasks and acquiring resources based on their classes. Additionally, resource pre-assignment and work stealing can be implemented by maintaining task queues for each resource, acquiring resources only once and dynamically distributing extracted tasks to the queues, instead of the default acquire-submit loop.

The *look-ahead* mechanism is not expensive to implement. It requires a list of task queues, in which each element corresponds to the resource class that the task requires. The types of queues do not need to be known in advance, they can be dynamically added to the list when a new resource type is generated by the workflow. A look-ahead thread will generate task ids and classify them by the type of resources they need by executing the following loop:

```
t = WE.next()
if (!class_queues.contains(resource_class(t)))
    class_queues[resource_class(t)] = queue()
endif
class_queues[resource_class(t)].push(t)
```

Finally, the basic scheduling loop in Section 3.1 is applied for each entry in the `class_queues` list (from separated threads or by using coroutines and yielding on *acquire* and `class_queues.get()`).

3.4 Co-allocation Scheduler

This type of scheduler addresses the problem of having tasks that require multiple resources in the same or in different classes. At the scheduler level, such a requirement can be solved for instance by acquiring the needed resources sequentially. Such a solution, however, gives rise to several issues:

- Starvation in the case that all needed resources never become available simultaneously
- Possible performance issues when waiting for the needed resources to become available
- Backfilling not possible without a *look-ahead mechanism*.

3.5 Location/Multiple Resource Managers

When an application needs resources that belong to different physical clusters (possibly geographically distributed), multiple Resource Managers can be used to handle resources for each individual location. In this case, the Scheduler can be similar to the Base scheduler, with the additional capability to inspect each task and select the appropriate Resource Manager from which to acquire resources.

3.6 Performance Model

If a performance model is available for the task run-times, the scheduler can select the (possible) multiple module implementations/resource type based on-the-fly before calling the *acquire* method. Alternatively, it can specify a list of preferences, which the Resource Manager will try to fulfill in order.

A dynamic performance model, i.e., a performance model that is capable to make use of tracing to update the performance estimations at run-time, can also be devised with tracing support.

3.7 Transfer Costs Scheduler

If an Independent Memory Layer (IML) is available, a special scheduler that takes into account the cost of memory transfers can be devised. Thus, the scheduler needs access to the list of hostnames associated with the available resources, as well as to the mapping between memory ranges in the IML and the hostnames where they are stored. This information (resource hostnames and memory range hostnames) is obtained beforehand by the user from the RTS and the IML, respectively, and forwarded to the scheduler.

When requesting resources for a given task, the scheduler can provide the RM with a preference list including only resources located on the same physical machines as the needed memory ranges.

4 Application Helper Code

The implementations outlined below can be used as scoped objects when writing the application startup binary, or as standalone binaries.

4.1 RTS.exe

Initializes the Run-Time System and publishes connection information so that multiple users/application can access it simultaneously. Implementation:

```
1 rm = Create RM ()
2 rts = Create RTS (rm)
3 rpc = Create RPC server()
4 Publish connection infoRM ()
5 Publish connection infoRTS ()
6 Publish connection info(RPC)
```

4.2 IML.exe

Initializes the memory layer and makes it available for the applications that need to store data in the global memory. Implementation:

```
1 Create IML
2 Publish connection info for IML
```

4.3 JobServer.exe

The JobServer implements the main application execution steps, such as defining and initializing the GPISpace components required by the application, reading command line parameters and executing the application workflow.

Whereas GPISpace can provide a set of template JobServers customized for different purposes, they can be further extended to fit various application need. For example, the JobServer can implement application-specific RPC calls.

Based on the application type, the JobServer may use an IML, one or more Resource Managers or different scheduler implementations. Below we give an overview of the most common JobServer patterns.

4.3.1 Simple Job Server

A basic implementation of the JobServer needs to instantiate an existing Scheduler, to start a workflow engine, to execute the user-supplied workflow (potentially after applying some transformations), and collect results. The main steps the JobServer has to perform can be summarized as follows:

```
1 Get connection information for RM
2 Get the user-provided app_workflow
3 (Optional) Apply automatic transformations to app_workflow (e.g., termination
    transitions) before submitting it to the workflow engine
4 WE (app_workflow) // Create workflow engine with the user provided workflow
5 s = Scheduler,(RM, WE ) // Initialize scheduler
6 (Optional) Create and publish application-specific RPC server
```

```
7 s.wait()     // Wait for the scheduler to execute the workflow
```

4.3.2 Cloud Job Server

We call a *Cloud JobServer* a JobServer that is able to take advantage of multiple Resource Managers, each of which being in charge of a different set of resources (possibly geograpically distributed to different locations). The only changes that the Simple Job Server requires in this case are to store a mapping between locations and Resource Managers, and to instantiate a type of scheduler that can handle them, such as the scheduler described in Section 3.5.

4.3.3 Job Server with Data Transfer Costs

When the application uses the global memory (IML) to store data, the run-time performance can be improved by executing tasks close to the nodes that store the data they access. To this end, a scheduler that can take into account the cost of data transfers is required. Such a JobServer is outlined below:

```
1 Get connection information for RM
2 Get connection information for IML
3 Get connection information for RTS
4 a = IML.alloc(size, data)     // Allocate global memory
5 htr = RTS.host_to_resource()    // Retrieve resource locations
6 s = SWT(RM, WE, htr, IML.partition(a)) // Initialize data transfer-aware scheduler
7 s.wait()     // Wait for the workflow execution
```

5 Implementation plan

Following are the first steps towards switching to the proposed architecture in GPISpace.

- Simplify the current Workflow Engine to only handle one workflow.
  - the main change would be to abandon the *cancel* operation, and simplify the interaction between the *agent* and WE towards the simple interface in Sec. 2.6.
- Simplify current scheduler
  - remove cost computation
  - remove work stealing/backfilling
  - separate functionality from that of the *worker manager*, which should be converted into the RM in the new architecture
- Discuss/address potential issues such as races, cancellation, starvation
- Separate the functionalities of the JobServer, RTS and RM within the existing *agent*; try to switch to the new API for each of them.