P*: An Extensible Model of Pilot-Abstractions

Andre Luckow[1], Mark Santcroos[2,1], Sharath Maddineni[1], Andre Merzky[1], Ole Weidner[4,1], Shantenu Jha[3,1*]

[1]*Center for Computation & Technology, Louisiana State University, USA*

[2]*Bioinformatics Laboratory, Academic Medical Center, University of Amsterdam, The Netherlands*

[3] *Rutgers University, Piscataway, NJ 08854, USA*

[4] *School of Informatics, University of Edinburgh, UK*

[*]*Contact Author:* shantenu.jha@rutgers.edu

Abstract

Pilot-Jobs have been notable in their ability to support dynamic resource utilization – in the number of applications that use them, in the scope of usage, as well as the number of CI that support them. In spite of broad uptake, there does not exist however, a well defined, unifying conceptual framework for pilot-jobs which can be used to define, compare and contrast different implementations. This presents a barrier to extensibility and interoperability. This paper is an attempt to (i) provide a minimal but complete model (P) of pilot-jobs, (ii) extend the basic model (but not implementation) from compute to data, (iii) introduce TROY as an implementation of this model, (iv) establish the generality of the P* Model by mapping various existing and well known pilot-job frameworks such as Condor and DIANE to P*, (v) establish and validate the implementation of TROY by concurrently using multiple distinct pilot-jobs.*

I. Introduction and Overview

Applications that support dynamic execution have the ability to respond to a fluctuating resource pool, i. e., the set of resources utilized at time (T), $T = 0$ is not the same as $T > 0$. Distributed CI almost by definition is comprised of a set of resources that is fluctuating – growing, shrinking, changing in load and capability; this is in contrast to a static resource utilization model traditionally a characteristic of parallel and cluster computing.

The ability to utilize a dynamic resource pool is thus an important attribute of any application that needs to utilize distributed CI efficiently. However, even for traditional high-performance/parallel applications, the evolution or internal dynamics of an application may vary, thereby changing the resource requirements. For example, different solvers, adaptive algorithms and/or implementations, can also require applications to utilize different set/amounts of resources. Thus, the need to support dynamic execution is widespread for computational science applications.

Multiple approaches exist to support dynamic resource utilization, for example, advanced scheduling (without preemption) provides essentially a guarantee of resources at a sufficiently far out time window. However, distributed applications that are able to break the coupling between workload management and resource assignment/scheduling have been successful at efficiently utilizing distributed resources, without the policy-level complexity of implementing advanced reservations. A common approach for decoupling workload management and resource scheduling typically in user-space (i.e. application-level scheduling) are *pilot-jobs (PJ)*. The PJ abstraction is also a promising route to address additional, as well as specific requirements of distributed scientific applications [1], [2], [3].

In general, the uptake of distributed infrastructures by scientific applications has been limited by the availability of extensible, pervasive and simple-to-use abstractions at multiple levels – development, deployment and execution stages of scientific applications [4].

The general problem applies to the specific situation of pilot-jobs, i.e., there are many implementations of the PJ abstraction, and although they are all for the most parts functionally equivalent – they support the decoupling of workload submission from resource assignment – it is often impossible to use them interoperably or even just to compare them. Different projects and users have rolled-out their own PJ abstractions. The fact that users have voted with their feet for PJs reinforces that the Pilot-Job abstraction is both a useful and correct abstraction for distributed CI; the fact that it has become an "unregulated cottage industry" reaffirms the lack of common nomenclature, integration, interoperability and extension.

Our work is partly motivated by the status of the usage and availability of the pilot abstraction vis-à-vis the current landscape of distributed applications and CI. Our objective is to provide a minimal, but complete model for the pilot-job abstraction. This model called the P* Model, can be used to provide an analytical framework to compare and contrast different pilot-job implementations. This is, to the best of our knowledge, the first such attempt. We present the P* Model in §II.

In §III we introduce TROY – A Tiered Resource

OverlaY – as an implementation of the P* Model. TROY provides an API for the P* Model and exposes the semantics of PJ frameworks; TROY also has a runtime system that enables it to work with different middleware on heterogeneous distributed platforms.

Implementations of PJ frameworks, such as BigJob and DIANE, are integrated into TROY via an adaptor mechanism; we posit that the TROY framework and API can be used for most, if not all PJ frameworks. Before validating this claim with empirical evidence, in §IV we analyze and discuss several well-known PJ frameworks (DIANE, Swift-Coaster and Condor-G) using the analytical framework provided by the P* Model. TROY is also able to manage multiple, potentially different pilot-job instances concurrently. In §V, we characterize TROY via performance measurements and demonstrate interoperability – across middleware, platform and different PJ frameworks.

Implicit validation of this paper lies in the practical implications of our work: TROY is positioned to be used on every major national and international distributed CI, including NSF XSEDE, NSF/DOE Open Science Grid, EU EGI amongst others. In §V we validate the TROY framework by demonstrating how Condor-G [5] and DIANE [6] – an existing and widely used pilot-job framework, can be given a simple and API via the TROY framework that is consistent with other commonly pilot-job frameworks. To further substantiate the impact of TROY, we will demonstrate interoperability between different PJ frameworks – BigJob and DIANE – using TROY. We believe this is also the first demonstration of interoperation of different pilot-job implementations.

After validating the P* Model – as measured by extensibility and interoperability, in we investigate generalizations to the base P* Model in §VI. A natural and logical extension of the P* Model, arises from the need to extend it to include data in addition to computational tasks. This leads to analogous abstraction to the pilot-job: the *pilot-data (PD)* abstraction, and the introduction of the BigData concept. The potentially consistent treatment of data and compute, suggests symmetrical compute and data elements in the P* model; thus we refer to this model as the P* model ("P-star").

II. P* Model: A Conceptual Framework for Pilot-Abstractions for Dynamic Execution

In an attempt to provide a common analytical framework to understand most, if not all commonly used pilot-jobs, we present the P* model of pilot-abstractions. The P* model is derived from an analysis of many pilot-job implementations; based upon this analysis, we first present the common *elements* of the P* model, followed by a description of the properties that determine the interaction of these elements and the overall functioning of any pilot-job framework that implements the P* model. All pilot-job frameworks (whether they adhere rigorously to the P* model or not), have these *elements*, but differ in specific attributes and characteristics. Further, we will show that these elements and interactions can be used to describe a pilot-data model.

Before we proceed to discuss the P* Model, it is important to emphasize that there exist a set of commonly used terms — abstraction, model, framework, and implementation, that are overloaded and often used inconsistently; thus we establish their context and usage in this paper.

Terms and Usage: The *abstraction* of a pilot-job generalizes the reoccuring concept of utilizing a placeholder job as a container for a set of compute tasks; instances of that placeholder job are commonly referred to as *pilot-jobs* or *pilots*. The P* *model* provides a description of pilot-job abstractions based on a set of identified elements and their interactions. The P* model can be used as a *conceptual framework*, for analyzing different implementations of the pilot-job abstraction. It is important to distinguish that P* provides a conceptual framework from an implementation of the P* model; *TROY* is a specific implementation of the P* model. Finally, a *pilot-job framework* refers to a specific instance of a pilot-job implementation that provides the complete pilot-job functionality (e. g. Condor-G and DIANE).

A. Elements of the P* Model

This sub-section defines the elements of the P* model:

- **Pilot-Job (PJ) or Pilot:** The PJ (or 'pilot') is the entity that actually gets submitted and scheduled on a resource using the resource's RM system. The PJ provides application (user) level control and management of the set of allocated resources.
- **Work Unit (WU):** A WU encapsulates a self-contained piece of work (a task) specified by the application that is submitted to the pilot-job framework. There is no intrinsic notion of resource associated with a WU.
- **Scheduling Unit (SU):** SUs are the units of scheduling used internal to the P* model, i.e., an SU is not known by or visible to an application. An SU is created after the submission of a WU, i. e. once a WU has been passed into the control of the pilot-job framework.
- **Pilot-Manager (PM):** The PM is responsible for (i) orchestrating the interaction between the pilots as well as the different components of the P* model (WUs, SUs)and (ii) decisions related to internal resource assignment (once resources have been acquired by the pilot-job). For example, an SU can consists of one or more WUs. Further, WUs and SUs can be combined and aggregated; the PM determines how to group them, when SUs are scheduled and executed

on a resource via the pilot, as well as how many resources to assign to an SU.

The application itself is not strictly part of the core P* Model. An application kernel is the actual binary that gets executed. The application utilizes a PJ framework to execute multiple instances of an application kernel (an ensemble) or alternatively instances of multiple different application kernels (a workflow). To execute an application kernel, an application must define a WU specifying the application kernel as well as other parameters. This WU is then submitted to the pilot-manager (as an entry point to the pilot-framework), where it transitions to an SU. The PM is then responsible for scheduling the SU onto a pilot and then onto a physical resource. As we will see in §IV, the above elements can be mapped to specific entities in many pilot-jobs in existence and use.

B. Characteristics of P* Model:

We propose a set of fundamental properties/characteristics that describe the interactions between the elements, and thus aid in the description of P* model.

Coordination: The coordination characteristics describes how various elements of the P* Model, i. e. the PM, the pilot, the WUs and the SUs, interact. A common coordination pattern is master/worker (M/W): the PM represents the master process that controls a set of worker processes, the pilots. The point of decision making is the master process. In addition to the *centralized* M/W, M/W can also be deployed *hierarchically*. Alternatively, coordination between the elements, in particular the pilots, can be performed so as to be *decentralized*, i. e. without central decision making point.

Communication: The communication characteristics describes the mechanisms for data exchange between the elements of the P* Model: e. g. messages (point-to-point, all-to-all, one-to-all, all-to-one, or group-to-group), streams (potentially unicast or multicast), publish/subscribe messaging or shared data spaces.

Scheduling: The scheduling characteristics describes the process of mapping a SU to resources via a pilot and potential multiple levels of scheduling. Scheduling has a spatial component (which SU is executed on which pilot?) but also a temporal component (when to bind?). The different scheduling decisions that need to be made are representative of multi-level scheduling decisions that are often required in distributed environments. For example, when should a SU be bound to a pilot? An SU can be bound to a pilot either before the pilot has in turn been scheduled (*early* binding), whereas *late* binding occurs if the SU is bound after the pilot has been scheduled. In general, there are multiple-levels at which scheduling decisions, i.e., resource selection and binding, are made.

The term *agent*, although not a part of the P* Model, finds mention when discussing implementations. For the purposes of this paper, an agent refers to a "proxy process" that either inside the pilot-job framework that has some decision making capability, and could aid the implementation one or more of the characteristics of a P* Model – coordination, communication, scheduling, within a pilot-job framework. These agents can be used to enforce, a set of (user-defined) policies (e.g. resource capabilities, data/compute affinities, etc.) and heuristics.

Putting it all together: Figure 1 illustrates the interactions between the elements of the P* model. In the first step the application specifies the capabilities of the resources required using a pilot-job description (step 1). The PM then submits the necessary number of pilots in order to fulfill the resource requirements of the application (step 2). Each pilot is queued at the resource manager, which is responsible for starting the pilot (step 3). There can be variations of this flow: while in the described model, the application defines the required resources, the PM could also decide based on the submitted WU workload whether and when it submits new pilots.

The application can submit WUs to the PM at any time (step 4). A submitted WU becomes an SU, i. e. the PM is now in control of it. In the simplest case one WU corresponds to one SU; however, SUs can be combined and aggregated to optimize throughputs and response times. Commonly, a hierarchical M/W model is used internally: the PM uses M/W to coordinate a set of pilots, the pilot itself functions as manager for the execution of the assigned SUs.

In the first step the PM chooses the pilot on which an SU is executed (step 5). Once a SU has been scheduled to a pilot, the pilot decides when and on what physical resource the SU is scheduled. Further, the pilot also manages the subsequent execution of the SU (step 6). Again there can be variations of this flow. PJ frameworks with decentralized decision making often utilize autonomic agents that accepts respectively pull SUs according to a set of defined policies.

III. TROY: A Reference Implementation and API for the P* Model

TROY is an implementation of the P* Model (§II). It consists of the TROY API [7], a runtime system and a set of adaptors for various PJ frameworks (see figure 2). TROY is consistent with the SAGA [8], [9] API.

SAGA provides a simple, POSIX-style API to the most common grid functions at a sufficiently high-level of abstraction so as to be independent of the diverse and dynamic grid environments. The TROY API itself was designed to be similar to SAGA in appearance and philosophy: it re-uses many of the well defined and standardized semantics and syntax. A further design consideration of

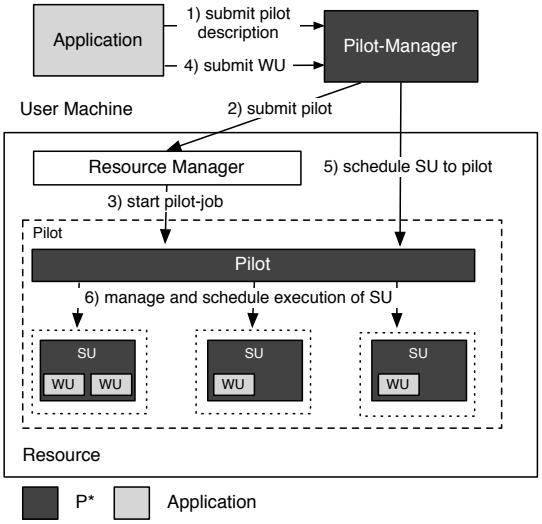

Fig. 1: **P* Model: Elements and Interactions:** The core of the model is the manager that is responsible for managing pilot-jobs and WUs. After a WU is submitted to the manager, it transitions to an SU, which is scheduled to a pilot by the PM. The pilot then schedules the SU to an available resource.

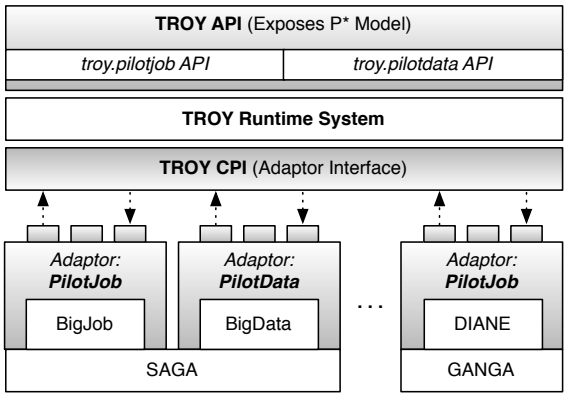

Fig. 2: **TROY – An API and Runtime System for the P* Model:** TROY provides an API for managing PJs and PDs. BigJob and BigData are realizations of the actual PJ and PD functionality. BJ and BD rely on SAGA for implementation of the PJ/PD.

TROY was a minimal but complete API.

The pilot-job capabilities in TROY are provided by different PJ frameworks that are integrated into the TROY runtime system via adaptors. For this purpose, TROY defines a Capability Provider Interface (CPI) that must be implemented by the adaptors of the underlying pilot-job frameworks. This architecture also enables the concurrent usage of multiple PJ frameworks.

Figure 3 displays the main classes of TROY. Figure 4 shows the interactions between the TROY elements. The TROY API decouples workload management and resource scheduling by exposing two separate services: The `PilotJobService` (PJS) and `WorkUnitService` (WUS). The PJS serves as factory for instantiating pilots. Also, the PJS can be used to query the PJS for currently active pilots. A `PilotJob` object is returned as result of the `create_pilotjob()` method of the PJS (step 1). As in SAGA, the instantiation of `PilotJob` object is done by using a `PilotJobDescription`. The description can be reused and has no state, while the `PilotJob` instance has state and is a reference for further usage. The actual creation of the PJ is done via the adaptor, which is responsible for calling the underlying PJ framework (step 2 and 3).

```
pjs = PilotJobService()
pj_desc = PilotJobDescription()
pj_desc.total_core_count = 8
pj = pjs.create_pilotjob('gram://queenbee',
                         pj_desc, 'bigjob')
```

Listing 1: **PilotJob Creation:** Instantiation of a Pilot Job Service using a Pilot Job Description.

The `PilotJob` object represents a pilot instance and allows the application to interact with a pilot, e. g. to query its state or to cancel it. The process of PJ creation is depicted in step 1-3 of figure 4 and in listing 1.

Listing 2 shows the creation of a `WorkUnitService`. Having instantiated a `WorkUnitService` object, PJS objects can be added and removed at any time. This enables applications to respond to dynamic resource requirements at runtime, i. e. if peak demands arise an application can request additional resources; if resources are not required anymore, they can be released.

```
wus = WorkUnitService()
wus.add(pjs)
```

Listing 2: **WorkUnitService Creation:** Instantiation of a `WorkUnitService` using a reference to the `PilotJobService`.

```
wud = WorkUnitDescription()
wud.executable = '/bin/bfast'
wud.arguments = ['match', '-t4', '/data/file1']
wud.total_core_count = 4
wu = wus.submit(wud)
```

Listing 3: **WorkUnit Submission:** Instantiation and submission of a `WorkUnitDescription`.

The `WorkUnitService` is responsible for managing the execution of WUs. Regardless of the state of the PJS, applications can submit WUs to a `WorkUnitService` at anytime (listing 3 and step 4 in figure 4). Once the `WorkUnitService` becomes responsible for a WU, the WU transitions to an SU. SUs are internally processed (they e. g. can be aggregated) and are then forwarded to the `Scheduler` (step 5), which selects an appropriate pilot. Having chosen an appropriate resource, the SU is dispatched using the respective adaptor (step 6 and 7). The PJ framework is responsible for the actual execution of the SU on a resource. Note that multiple levels of (hierarchical) scheduling can be present: commonly a SU is scheduled inside of TROY runtime system as well as in

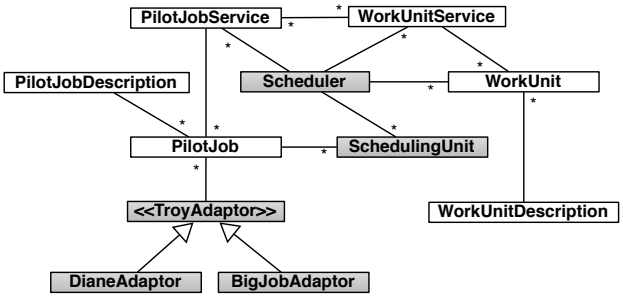

Fig. 3: **TROY Class Diagram:** The classes used in the TROY implementation. Classes that are not exposed to the application are depicted grey. The cardinality is shown between classes.

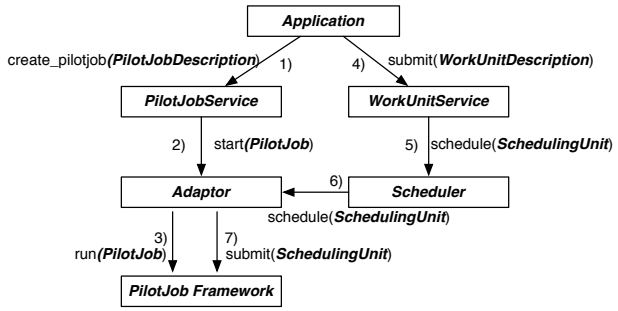

Fig. 4: **TROY Flow Diagram:** The functionality of TROY is exposed using two primary classes: The `PilotJobService` for the management of pilots, and the `WorkUnitService` for the management of WUs. The TROY runtime system processes application requests and forwards them via the respective adaptor to the PJ framework.

the underlying PJ framework.

Each `WorkUnit` and the `PilotJob` object is associated with a state. The state model is based on the SAGA job state model [9]. Applications can query the state using the `get_state()` method or they can subscribe to state update notifications.

The TROY API classes and interactions are designed to reflect the P* elements and characteristics. Table I summarizes the mapping of P* elements and TROY. As defined by P*, a WU represents a primary self-containing piece of work that is submitted to TROY. At the API boundary a WU transitions to an SU, which functions as internal unit of scheduling. The API follows object-oriented design principles and exposes the primary functionality of the Pilot-Manager using two classes: the `PilotJobService` for the management of pilots and the `WorkUnitService` for the management of WUs. Further, the framework utilizes a set of internal classes for implementing different P* characteristics, e. g. the `Scheduler` for scheduling and the adaptors implementations for providing the actual PJ capabilities.

TROY uses the M/W coordination model: The TROY runtime system functions as master for a set of PJ frameworks that are encapsulated within a TROY adaptor. Inside the PJ framework itself another coordination model can be used; most commonly this is also M/W. The TROY runtime system only utilizes local communication between the application, the runtime system and the adaptors. The PJ frameworks in turn, can rely on their own communication & coordination (c&c) model.

Pilot	PJ framework
WU	WU
SU	SU
Pilot-Manager	Pilot Job Service/Work Unit Service

TABLE I: Mapping of P* and TROY

IV. Pilot-Job Frameworks

As more applications take advantage of dynamic execution, the Pilot-Job concept has grown in popularity and has been extensively researched and implemented for different usage scenarios and infrastructure. There is a variety of PJ frameworks: Condor-G [5], SWIFT [10], DIANE [6], DIRAC [11], PanDA [12], ToPoS [13], Nimrod/G [14], Falkon [15] and MyCluster [16] to name a few. The aim of this section is to show that our P* Model can be used to explain/understand some of these PJ frameworks. In particular we focus on BigJob, DIANE and Condor-G.

A. BigJob: A SAGA-based Pilot-Job Implementation for TROY

BigJob (BJ) [17], [18] is a SAGA-based pilot-job implementation. BJ supports a wide range of application types, and is usable over a broad range of infrastructures, i. e. it is general-purpose and extensible. BJ is integrated into the TROY runtime environment via the described adaptor mechanism (see figure 6). In addition there are specific BJ flavors for cloud resources such as Amazon EC2 and Microsoft Azure that are capable of managing set of VMs, as well as a BJ with a Condor-G based backend.

Figure 5 illustrates the architecture of BJ. BJ utilizes a M/W coordination model: The BigJob-Manager is responsible for the orchestration of pilots, for the binding of WUs and for the scheduling of SUs. For submission of the pilots, SAGA relies on the SAGA Job API, and thus can be used in conjunction with different SAGA adaptors, e. g. the Globus, the PBS, the Condor and the Amazon Web Service adaptor. Each pilot initializes a so called BJ-agent. The agent is responsible for gathering local information and for executing tasks (SUs) on its local resource. The SAGA Advert Service API is used for communication between manager and agent. The Advert Service (AS) exposes a shared data space that can be accessed by manager and agent, which use the AS to realize a push/pull communication pattern, i. e. the manager pushes a SU to the AS while the agents periodically pull for new SUs. Results and state updates are similarly pushed back from

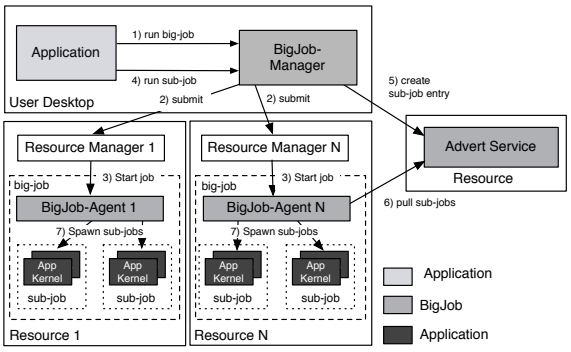

Fig. 5: **BigJob Architecture:** The core of the framework, the BigJob-Manager orchestrates a set of pilots. Pilots are started using the SAGA Job API. The application submits WUs, the so-called sub-jobs via the BigJob-Manager. Communication between the BJ-Manager and BJ-Agent is done via a shared data space, the Advert Service. The BJ-Agent is responsible for managing and monitoring sub-jobs.

the agent to the manager. Further, BJ provides a pluggable communication & coordination layer and also supports alternative c&c systems, e. g. Redis [19] and ZeroMQ [20].

BJ currently uses a simple binding mechanism: each WU submitted to the BigJob framework is mapped to one SU. Binding can take places at submission time (early binding) or delayed in case of multiple pilots (late binding). For scheduling, a simple FIFO scheduler is used (see also table III).

In many scenarios it is beneficial to utilize multiple resources, e. g. to accelerate the time-to-completion or to provide resilience to resource failures and/or unexpected delays. The TROY API allows for dynamic resource additions/removals as well as late binding. The support of this feature depends on the backend used. To support this feature on top of various BigJob implementations that are by default restricted to single resource use (e. g. BJ), the concept of a BigJob pool is introduced. A BigJob pool consists of multiple BJs (each BigJob managing one particular resource). An extensible scheduler is used for dispatching WUs to one of the BJs of the pool (late binding). By default a FIFO scheduler is provided. Other backends (such as DIANE and Condor) natively support elasticity, but can nevertheless be combined into a BJ pool.

B. DIANE

DIANE [6] is a task coordination framework, which was originally designed for implementing master/worker applications, but also provides PJ functionality for job-style executions. DIANE utilizes a single hierarchy of worker agents as well as a PJ manager referred to as `RunMaster`. For the spawning of PJs a separate script, the so-called submitter script, is required. For the access to the physical resources the GANGA framework [21] can be used. Once the worker agents are started they register themselves at the RunMaster. In contrast to TROY-BigJob, a worker agent generally manages only a single core and thus, by default is not able to run parallel applications (e. g. based on MPI). BJ utilizes the BJ-Agent that is able manage a set of local resources (e. g. a certain number of nodes and cores) and thus, is capable of running parallel applications. For communication between the RunMaster and worker agents point-to-point messaging based on CORBA [22] is used. CORBA is also used for file staging, which is not fully supported by BJ, yet.

DIANE is primarily designed with respect to HTC environments (such as EGI [23]), i. e. one PJ consists of a single worker agent with the size of 1 core. BJ in contrast is designed for HPC systems such as TG, where a job usually allocates multiple nodes and cores. To address this issue a so-called multinode submitter script can be used: the scripts starts a defined number of worker agents on a certain resource. However, WUs will be constrained to the specific number of cores managed by a worker agent. A flexible allocation of resource chunks as with BJ is not possible. By default a WU is mapped to a SU; application can however implement smarter allocation schemes, e. g. the clustering of multiple WUs into a SU.

DIANE includes a simple capability matcher and FIFO-based task scheduler. Plugins for other workloads, e. g. DAGs or for data-intensive application, exist or are under development. The framework is extensible: applications can implement a custom application-level scheduler.

DIANE is as BJ a single-user PJ, i. e. each PJ is executed with the privileges of the respective user. Also, only WUs of this respective user can be executed by DIANE. DIANE supports various middleware security mechanisms (e. g. GSI, X509 authentication). For this purpose it relies on GANGA. The implementation of GSI on TCP-level is possible, but currently not yet implemented. Further, DIANE supports fault tolerance: basic error detection and propagation mechanisms are in place. Further, an automatic re-execution of WUs is possible.

C. Condor-G

Condor-G pioneered the Pilot-Job concept [5]. The pilot is actually a complete Condor pool that is started using the Globus service of a resource. This mechanism is referred to as Condor Glide-In. Subsequently, jobs can be submitted to this Glide-In pool using the standard Condor tools and APIs. Condor utilizes a master/worker coordination model. The PJ manager is referred to as the Condor Central Manager. The functionality of the Central Manager is provided by several daemons: the condor_master that is generally responsible for managing all daemons on a machine, the condor_collector which collects resource information, the condor_negotiator that does the matchmaking and the condor_schedd that is responsible for managing the binding and scheduling process. Condor generally does not differentiate between workload, i. e. WU, and schedu-

P* Element	BigJob	DIANE	Condor-G	Swift-Coaster
Pilot-Manager	BigJob Manager	RunMaster	condor_master, condor_collector, condor_negotiator, condor_schedd	Coaster Service
Pilot-Job	BigJob Agent	Worker Agent	condor_master, condor_startd	Coaster Worker
Work Unit (WU)	Task	Task	Job	Application Interface Function (Swift Script)
Scheduling Unit (SU)	Sub-Job	Task	Job	Job

TABLE II: Table showing the mapping between the elements of the P* Model and different Pilot-Job Frameworks

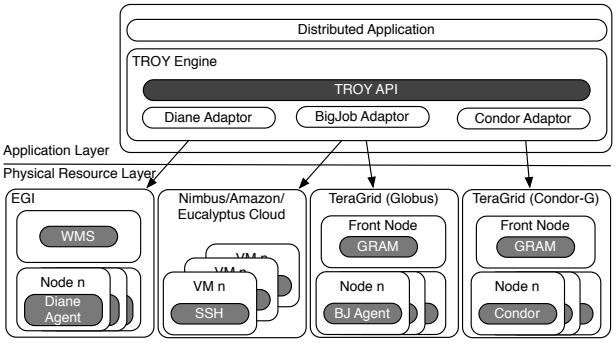

Fig. 6: **TROY and PJ frameworks:** TROY provides the ability to utilize the native Pilot-Job capabilities of different infrastructures, e. g. BigJob for TeraGrid/Cloud resources, DIANE for EGI and Condor for OSG resources. It also enables the concurrent usage of these infrastructures.

lable entity, i. e. SU. Both entities are referred to as job. However, it supports late binding, i. e. resources a job is submitted to must generally not be available at submission time. The scheduler matches the capabilities required by a WU to the available resources. This process is referred to as matchmaking. Further, a priority-based scheduler is used. For communication between the identified elements Condor utilizes point-to-point messaging using a binary protocol on top of TCP.

Different fault tolerance mechanisms, such as automatic retries, are supported. Further, Condor supports different security mechanisms: for authentication it integrates both with local account management systems (such as Kerberos) as well as grid authentication systems such as GIS. Communication traffic can be encrypted.

Discussion: P* provides a theoretical framework for describing and understanding of PJ frameworks. Table II summarizes how P* can be applied to BigJob, DIANE and Condor-G. Further, we show that P* is not limited to the described PJ frameworks, but can also be used with other PJ frameworks, e. g. SWIFT Costers [24]. The same applies to the P* characteristics that are summarized in table III. In addition to applying the theoretical P* framework, we integrated BigJob, DIANE and Condor into TROY, i. e. the TROY API can now be used as unified abstraction across multiple PJ frameworks (see figure 6). This validates (i) the TROY abstractions and (ii) the extensibility of the TROY framework.

V. Experiments and Results

In this section we analyze the performance and scalability of different PJ implementations in particular of BigJob and DIANE. Further, we investigate PJ framework interoperability by using TROY in conjunction with different TROY adaptors. For this purpose we deploy TROY with BigJob (with SAGA/PBS and SAGA/Condor) and DIANE in a genome sequencing application scenario. The experiments are conducted on different production (LONI, OSG) and research infrastructures (FutureGrid).

A. Understanding TROY Interoperability

To validate the abstractions developed, we conducted a series of experiments. We execute BFAST [25] – a genome sequencing application – using TROY in conjunction with different PJ frameworks and infrastructures. For this purpose we deploy TROY with various BigJob configurations and DIANE. TROY-BigJob is used with the SAGA-Globus adaptor to access LONI [26] resources and with SAGA-Condor adaptor to access OSG resources [27]. Further, we utilize TROY-DIANE on LONI. On LONI the Oliver machine and a total of 8 cores distributed across 2 nodes is used, on OSG a single machine was requested using the GRAM/Condor endpoint of the Renci gateway machine.

The investigated workload comprises of four WUs. Each WU executes a BFAST matching process, which is used to find potential sequence alignments. We run the experiment on four different setups: (i) TROY-BigJob/Globus only, (ii) TROY-BigJob/Condor only, (iii) TROY-DIANE only and (iv) TROY-BigJob and TROY-DIANE concurrently. The TROY Pilot Job Service is used to create pilots on the respective infrastructure(s). Each BFAST WU requires two cores; a total of four WUs requesting two cores each is specified and submitted to the Work Unit Service. In case (iv) on each backend two WUs are executed. Each WU is associated with a set of input files which are pre-staged in all cases except for case (ii) where the data is staged during the runtime on to the condor resources. After submission of the WU to the PJ manager, the TROY runtime system takes care of binding and scheduling the WUs to the pilots. The application can monitor the state of the PJs and WUs using the Work Unit Service of the TROY API.

P* Characteristic	SAGA BigJob	DIANE	Condor-G	SWIFT Coaster
End User Environment	API	API and Master/Worker Framework	CLI Tools	Swift script
Coordination	Master/Worker	Master/Worker	Master/Worker	Master/Worker
Communication	Advert Service	CORBA	TCP	GSI-enabled TCP
Binding	Early/Late	Late	Late	Late
Scheduling	FIFO, custom	FIFO, custom	Matchmaking, priority-based scheduler	Load-aware scheduler, WU grouping
Security	Multiple (GSI, Advert DB Login)	Multiple (GSI)	Multiple (GSI, Kerberos)	GSI
Resource Abstraction	SAGA	GANGA/SAGA	Globus	Resource Provider API/Globus CoG Kit
Agent Submission	API	GANGA Submission Script	Condor CLI	Resource Provider API
Fault Tolerance	Error propagation	Error propagation, Retries	Error propagation, Retries	Error propagation, retries, replication

TABLE III: P* Characteristics and Pilot-Job Frameworks

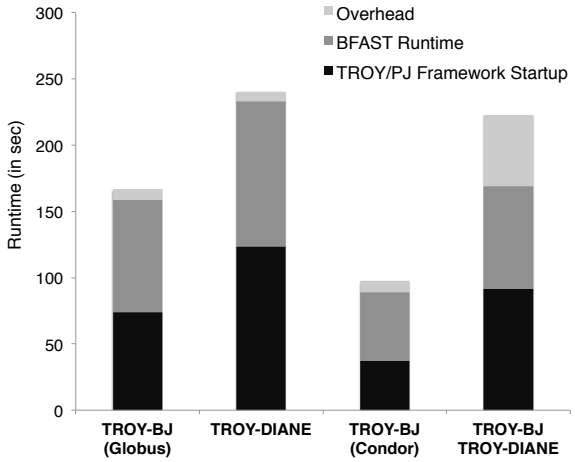

Fig. 7: **TROY Performance with BigJob/Globus, BigJob/Condor and DIANE:** Running BFAST on a 4 node cluster with a total of 16 cores. Each experiment is repeated at least 3 times. The runtime for TROY-DIANE is higher than for BJ mainly due to the higher startup time required and some light runtime overhead caused by the additional worker agents required on the resource.

Figure 7 shows the results of the experiments. In addition to the runtime we measured the startup time, i. e. the time required until the first WU changes its state to running. The overhead is defined as the difference between the actual runtime of the BFAST WU and the overall runtime using TROY (not including the startup time).

The runtime for TROY-DIANE is about 70 sec longer than for TROY-BigJob. The main contributor for this increased runtime is the deployment time required for DIANE. In a multi-node setup multiple work agents must be used (4 in this case). For each worker agent DIANE must be downloaded, installed and started separately. Further, in the used DIANE setup, the executable is staged. This is particularly different from the scenario investigated in section V-B, in which no file staging is used. In total the setup of DIANE requires about 110 sec of the overall 240 sec runtime, i. e. about 45 %. Further, each DIANE worker agent is queued as a separate job at the local resource manager – this contributes to the higher deviation in the measured runtimes. Despite the fact that BigJob has been also installed with each run, it shows a significant lower startup time of 73 sec, i. e. about 50 sec less than TROY-DIANE. Additionally, we also observed a runtime overhead of about 25 sec for the TROY-DIANE scenario. This overhead is likely caused by the additional agents required. While BigJob utilizes one BJ-Agent on the resource, DIANE currently requires the spawning of one worker agent per WU that must be executed in parallel. While the TROY API marshals these differences, i. e. while the API remains the same for both PJ frameworks, a light performance overhead remains observable.

The TROY with BigJob Condor shows the best performance of all scenarios. This can mainly be attributed to the better performance of the OSG resource, which is particularly visible in the BFAST runtimes, which are on average 40 % shorter than on LONI. Further, due to the lack of SAGA on OSG, BJ is deployed with the Redis c&c sub-system, which shows a significantly better performance than the Advert Service (see section V-B).

Finally scenario (iv) demonstrates that two PJ frameworks can be utilized concurrently using the TROY API and adaptor mechanism. The performance in this scenario is slightly better than in the DIANE only case, mainly due to the fact that only four DIANE worker agents need to be started. Also, only half of the WUs are executed on a DIANE node and thus, show a longer runtime. While there are some limitations in the current TROY-DIANE implementation, the aim of this experiment is to emphasizes the possibilities that the TROY API provides to dynamic applications. TROY enables applications to utilize a dynamic resource pool consisting of resources of different infrastructures, e. g. EGI and TG/XD resources, at large-scales hitherto unattainable. Dynamic applications

can utilize the elasticity of the TROY resource pool e. g. to improve the time-to-completion and/or to scale the accuracy of their computations.

B. Characterizing TROY-PJ Implementations

Each PJ implementation is associated with various degrees of freedom, in particular in the design of the communication & coordination (c&c) sub-system. The primary barrier for performance and scalability is not the WU submission, but the internal coordination of the elements of a P* implementation. There are many factors that influence the overall performance, e. g. the degree of distribution (local (LAN) vs. remote (WAN)), the communication pattern (1:n versus n:n) and the communication frequency. In the following we investigate the impact of different c&c related factors on the overall performance and scalability of the system.

The original design of BigJob is based on a shared, centralized data space, the SAGA Advert Service [28], which is essentially a PostgreSQL database. The communication between all components is done via this data space; this concept is also known as tuple space [29]. The data space decouples BJ-Manager and BJ-Agent very well and allows both entities to operate at their own pace optimizing the overall throughput. Depending on the setup this data space can be deployed locally, i. e. on the same resource, or remotely, e. g. during a distributed run. A particular issue during distributed runs is the latency between the application and the Advert Service. Another challenge is that this design introduces a potential single-point-of-failure and scalability bottleneck if the centralized data space is not carefully designed and operated.

BigJob also provides two alternative c&c sub-systems: Redis [19] and ZeroMQ [20]. Redis is a lightweight key/value store, which can be deployed in a distributed, fault tolerant way. Redis is used in a similar way as the Advert database, i. e. all communication between BJ-Agent and manager is channeled through it. Redis can be deployed locally and remotely. The ZeroMQ c&c sub-system in contrast utilizes a client-server architecture, which is similar to the CORBA-based [22] communication system of DIANE. In this architecture, the PM maintains the overall state. Clients connect to the PM to request new SUs or to report state updates. An advantage of this architecture is that it does not require a separate infrastructure for deployment of the Advert Service or the Redis database. While the BJ-Manager can be deployed remotely from the BJ-Agent, in most cases this will not be the case, i. e. the communication between BJ-Manager and BJ-Agent is mostly local communication. Both the data space and client-server c&c sub-system can be combined with a publish/subscribe mechanisms, i. e. instead of polling an agent or client can receive notifications when a new SU arrives.

We evaluate different BJ configurations and compare and contrast them with DIANE. For this purpose, we conducted several experiments on FutureGrid [30]. To evaluate the overall performance and throughput, we execute a different number of WUs on Alamo and Sierra utilizing up to 128 cores concurrently. Each experiment is repeated at least 10 times.

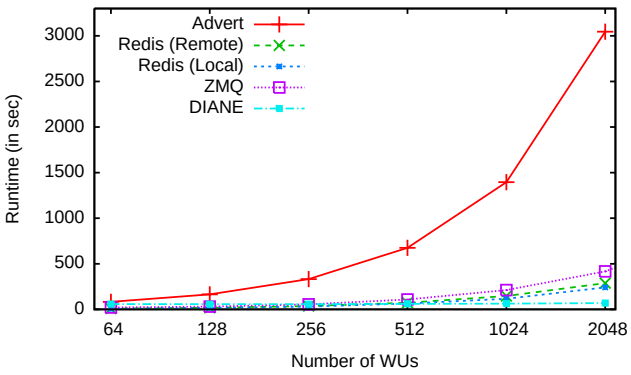

Fig. 8: **BigJob and DIANE Performance (1):** The time-to-completion for n WUs running `/bin/date` scales linearly in most cases, i. e. the coordination overhead imposed by the BJ is minimal.

Figure 8 illustrate the performance and scalability of the different BigJob configurations and DIANE with respect to the number of WUs. Clearly, the used c&c subsystem has a great impact on the overall performance. The Redis backend shows the best performance for small WU counts. The difference between local and remote coordination is moderate (about 20 %). While ZeroMQ is very fast and lightweight, it requires a careful implementation in particular concerning synchronization and throughput optimization. The overall performance is slightly worse than for Redis. The Advert Service currently has some limitations which will be discussed later. DIANE shows a higher startup overhead, which is particularly observable for smaller WU numbers. However, the runtime increases only slightly (about 10 sec) when going from 64 to 2048 WUs. A reason for this behavior is that DIANE aggregates SUs: for 2048 WUs only a single task description is created, which the framework then efficiently distribute to its worker agents. BigJob in contrast maintains for each WU a separate description.

Figure 9 illustrates the performance scalability with respect to the number of cores. In particular, the Redis (Local) configuration show an almost linear scalability up to 128 cores. The Redis remote setup again imposed some overhead (about 14 %). ZeroMQ performs very well with lower core counts; with larger core counts the runtimes increase indicating a potential scalability bottleneck. DIANE shows in particular for lower core counts a longer runtime

again due to the higher startup overhead. With higher core counts DIANE behaves similar to BigJob ZeroMQ showing a greater increase of the overall runtime. This increase can likely be attributed to the single central manager in the client-server architecture. As in the last experiment, the Advert c&c sub-system showed a significantly lower performance.

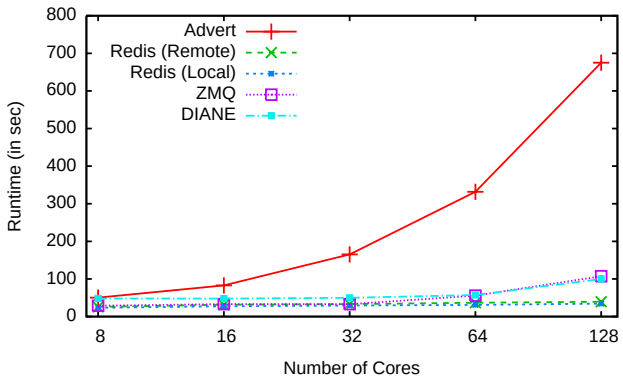

Fig. 9: **BigJob and DIANE Performance (2):** The runtime of a constant workload of 4 `/bin/date` jobs increases only slightly up to 128 cores. In particular, the Redis backend shows an almost linear scalability achieving a throughput of up to 4 WUs/sec.

The BigJob Advert implementation currently shows some performance limitations mainly caused by the prototypical nature of the Advert Service implementation. Further, it must be noted that the Advert Service was deployed remotely (mainly due to deployment constraints). However, even considering this aspect the discrepancy between Advert Service and Redis (Remote), which has been deployed on the same remote network, is significant. A reason for the worse performance is the used remote access protocol. The Advert Service is based on PostgreSQL; in the current architecture the client, i. e. the BJ-Manager and agent access the PostgreSQL database via the SOCI backend library. While this architecture provides a very flexible remote access to the database, running database access protocol over a WAN connection is not optimal. Transactions e. g. are very latency sensitive and require several roundtrips. Further, the API is based on a hierarchical namespace, which does not naturally map well to relational databases. In particular deeply nested namespaces exhibit an insufficient query and update performance. In contrast, both in the Redis and ZeroMQ scenario, data is stored in memory, which also explains to significant performance gains.

VI. P* Redux: Extension of P* to a Model for Pilot-Data

Dynamic execution is at least equally important for data-intensive applications: applications must cope with various challenging issues e. g. varying data sources (such as sensors and/or other application components), fluctuating data rates, optimizations for different queries, data-/compute co-location etc. Thus, having defined the P* model, we explore its extension to data. This will motivate an analogous abstraction that we call *pilot-data (PD)*. PD provides late-binding capabilities for data by separating the allocation of physical storage and application-level data units. Further, it provides an abstraction for expressing and managing relationships between data units and/or work units. These relationships are referred to as *affinities*.

Extension of P Model Elements to Data:* The elements defined by P* (in section II-B) can be extended by the following elements:

A. **Data Unit (DU):** DU is the base unit of data assigned by the application, e. g. a data file or chunk.
B. **Pilot-Data (PD):** PD allows the logical grouping of DUs and the expression of data-data affinities. This collection of files can be associated with an extensible set of properties. One of these properties is affinity.
C. **Pilot-Store (PS):** A PS functions as a placeholder object that reserves the space for pilot-data objects. By assigning a PD to a PS, the PD is bound to a physical resource. A PS facilitates the late-binding of data and resource and is equivalent to the pilot-job.
D. **Pilot-Data Manager** is analogous to the PJ-Manager responsible for managing DU, PD and PS elements. It implements the different characteristics of the P* model.

Note, each element can be mapped to an element in the P* Model by symmetry, e. g., a DU correspond to a WU in the original P* Model; a PS is a placeholder reserving a certain amount of storage on a physical resource and corresponds to a pilot in the P* Model.

Application of P Model Characteristics to Data:* While the extended P* Model introduces new elements, the characteristics however, remain the same to a great extent. The coordination characteristic describes how the elements of PD interact, e. g. utilizing the M/W model; the communication characteristic can be applied similarly. The scheduling characteristics must be extended to not only meet compute requirements, but also to support common data access patterns. The scheduling component particularly needs to consider affinities, i. e. user-defined relationships between WUs and/or DUs. Data-data affinities e. g. exist if different DUs must be present at the same compute element; data-compute affinities arise if data and compute must be co-located for a computation, but their current location is different. Data and compute placement decisions are made by the scheduler based on defined policies, affinities & dynamic resource information.

BigData: A SAGA-based Pilot-Data Prototype for TROY: BigData is the SAGA-based prototype of the Pilot-Data abstraction; in this paper it is referred to as simply

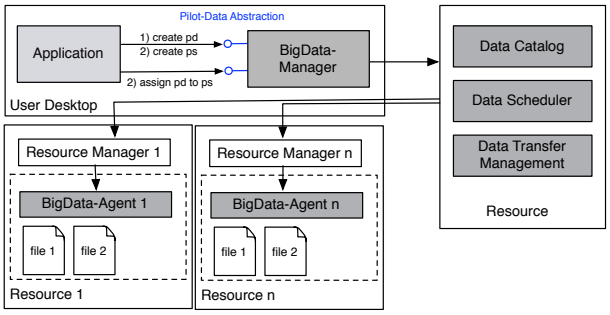

Fig. 10: **BigData Prototype Architecture:** The BD Manager exposes TROY's PD API. Application can create group of files and assign files to storage. The BD manager tracks file locations in the data catalog. The scheduler optimizes data-compute co-location. The transfer manager initiates and monitors data movements.

BigData (BD). Figure 10 gives an overview of the architecture. The system consists similarly to the BigJob architecture (see figure 5) of two components: the BD-Manager and the BD-Agents deployed on the physical resources. The coordination scheme used is again M/W with some intelligence that is located de-centrally at the BD-Agent. As communication mechanism the SAGA Advert Service is used, in a similar push/pull mode as for BJ.

The BD-Manager is responsible for (i) meta-data management, i. e. it keeps track of the pilot stores that a pilot data object is associated with, (ii) for scheduling data movements and data replications (taking into account the application requirements defined via affinities), and (iii) for managing data movements activities. For this purpose, it can rely on external service, e. g. Globus Online for data transfer management. Similar to BigJob, an agent on each resource is used to manage the physical storage on a resource.

A particular critical requirement for data-intensive application, is the management of affinity between DUs and also between WUs and DUs. The BD scheduler supports preliminary affinity-aware scheduling: both BigJob and BigData are tightly integrated to efficiently support compute- and data-related aspects of dynamic execution (see also § IV-A).

Both BigJob and BigData define similar elements that can be mapped to each other. Nevertheless, compute and data model sometimes require a different treatment. The extended TROY (an implementation of the extended P* Model) will optimize data- and computing according to a set of defined affinities and policies.

VII. Discussion and Future Work

The primary intellectual contribution of this work has been the development of the P* Model, its validation via TROY and the mapping of P* elements to PJ frameworks such as DIANE and Condor-G.

The P* Model provides a common framework for describing and characterizing Pilot-abstractions. TROY is an implementation of the P* model that captures the commonalities between the different PJ frameworks and provides a common access to them via a consistent API. We validate the P* Model by demonstrating that the most widely used PJ frameworks, viz., DIANE and Condor-G can be compared, contrasted and analyzed using this analytical framework.

Building on the P* Model, TROY's design enables the easy exchange of PJ implementations — both individual and the concurrent use of multiple PJ frameworks. TROY functions as common access layer for different PJ frameworks, and thus providing interoperability and extensibility.

There is potential for immense practical implications of our work, as these PJ frameworks collectively support millions of tasks yearly; however as alluded, these frameworks were not engineered with interoperability and extensibility as first-class considerations, and are thus faced with real barriers to scalability. Our work has the potential to ameliorate this situation. The TROY framework will be extended to support advanced autonomic resource management and selection strategies, e. g. by deploying more decentral decision logic into the agents, thus providing possible further improvements to scalability

Furthermore, our experience with fragile grid middleware and infrastructures, has taught us to appreciate the significance of details that come into play while deploying software on heterogeneous distributed infrastructures, and which in spite of theoretical and architectural advances have the potential to limit impact. However, the SAGA inspired approach to TROY's API design and the leverage of the design and deployment experiences of SAGA, e. g. by moving the responsibility of correctly deployed grid-middleware to the resource providers and not the end-users, should be an effective solution.

TROY provides significant future research, development & deployment extensions and opportunities. We discuss one opportunity along each of these axes: (i) Development and Extension: On the basis of successful validation of the P* Model, we proposed extensions to include data, i.e., PD and associated abstractions. However, only a prototype implementation of BigData exists; providing a unified implementation with BigJob and presenting these unified abstractions within TROY will support advanced dynamic execution modes and enhance the range of applications that will benefit from the pilot abstraction; (ii) Research Issues: These developments and extensions will also provide the basis to explore and reason on the relative roles of system versus application-level scheduling and heuristics for dynamic execution; (iii) Deployment: Currently, the widely used Condor-G framework is invoked via the Condor-SAGA adaptor; however, although the performance differ-

ence will be negligible, there will be significant ease of distribution if the Condor capabilities are directly integrated into TROY while retaining the advantage of a common consistent API to the PJ frameworks and functionality.

In summary, we will use existing and emerging capabilities of TROY to support the efficient and scalable solution of many scientific applications that involve multiple independent tasks.

Acknowledgements

This work is funded by Cybertools project (http://cybertools.loni.org; PI Jha) NSF/LEQSF (2007-10)-CyberRII-01, HPCOPS NSF-OCI 0710874 award, NSF-ExTENCI (OCI-1007115) and NIH Grant Number P20RR016456 from the NIH National Center For Research Resources. Important funding for SAGA has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII-UK). MS is sponsored by the program of BiG Grid, the Dutch e-Science Grid, which is financially supported by the Netherlands Organisation for Scientific Research, NWO. SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme. "Distributed Programming Abstractions" & 3DPAS. We thank J Kim (CCT) for assistance with the DNA models. SJ acknowledges useful related discussions with Jon Weissman (Minnesota) and Dan Katz (Chicago). This work has also been made possible thanks to computer resources provided by TeraGrid TRAC award TG-MCB090174 (Jha). This document was developed with support from the National Science Foundation (NSF) under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed".

References

[1] S.-H. Ko, N. Kim, J. Kim, A. Thota, and S. Jha, "Efficient runtime environment for coupled multi-physics simulations: Dynamic resource allocation and load-balancing," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 349–358. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.107

[2] J. Kim, W. Huang, S. Maddineni, F. Aboul-Ela, and S. Jha, "Exploring the RNA folding energy landscape using scalable distributed cyberinfrastructure," in *HPDC*, 2010, pp. 477–488.

[3] A. Luckow and S. Jha, "Abstractions for loosely-coupled and ensemble-based simulations on azure," *Cloud Computing Technology and Science, IEEE International Conference on*, pp. 550–556, 2010.

[4] S. Jha, D. S. Katz, M. Parashar, O. Rana, and J. Weissman, "Critical perspectives on large-scale distributed applications and production grids." Banff, Canada: IEEE Computer Society Press, 2009, pp. 1–8. [Online]. Available: http://www.cct.lsu.edu/~sjha/dpa_publications/dpa_grid2009.pdf

[5] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, July 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1015617019423

[6] J. Moscicki, "Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data," in *Nuclear Science Symposium Conference Record, 2003 IEEE*, vol. 3, 2003, pp. 1617 – 1620.

[7] TROY API, https://svn.cct.lsu.edu/repos/troy/trunk/doc/api.html, 2011.

[8] SAGA, http://saga.cct.lsu.edu, 2011.

[9] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith, "A Simple API for Grid Applications (SAGA)," OGF Document Series 90, http://www.ogf.org/documents/GFD.90.pdf, 2011.

[10] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000524

[11] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, and the Lhcb Dirac Team, "Dirac pilot framework and the dirac workload management system," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062049, 2010. [Online]. Available: http://stacks.iop.org/1742-6596/219/i=6/a=062049

[12] P.-H. Chiu and M. Potekhin, "Pilot factory – a condor-based system for scalable pilot job generation in the panda wms framework," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062041, 2010. [Online]. Available: http://stacks.iop.org/1742-6596/219/i=6/a=062041

[13] "Topos - a token pool server for pilot jobs," https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS, 2011.

[14] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," *International Conference on High-Performance Computing in the Asia-Pacific Region*, vol. 1, pp. 283–289, 2000.

[15] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A Fast and Light-Weight TasK ExecutiON Framework," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[16] E. Walker, J. Gardner, V. Litvin, and E. Turner, "Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment," in *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, 0-0 2006, pp. 95–103.

[17] "SAGA BigJob," http://faust.cct.lsu.edu/trac/bigjob, 2011.

[18] A. Luckow, L. Lacinski, and S. Jha, "SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2010, pp. 135–144.

[19] Redis, http://redis.io/, 2011.

[20] ZeroMQ, http://www.zeromq.org/, 2011.

[21] J. M. *et al*, "Ganga: A tool for computational-task management and easy access to grid resources," *Computer Physics Communications*, vol. 180, no. 11, pp. 2303 – 2316, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509001970

[22] *Common Object Request Broker Architecture: Core Specification*, Object Management Group, M 2004.

[23] EGI, http://www.egi.eu/, 2011.

[24] "Coasters," http://wiki.cogkit.org/wiki/Coasters, 2009.

[25] N. Homer, B. Merriman, and S. F. Nelson, "BFAST : An alignment tool for large scale genome resequencing," *PLoS One*, vol. 4, no. 11, p. e7767, 2009.

[26] LONI, http://www.loni.org, 2011.

[27] R. P. *et al*, "The open science grid," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012057, 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/i=1/a=012057

[28] A. Merzky, "SAGA API Extension: Advert API," OGF Document Series 177, http://www.gridforum.org/documents/GFD.177.pdf, 2011.

[29] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985. [Online]. Available: http://doi.acm.org/10.1145/2363.2433

[30] "FutureGrid: An Experimental, High-Performance Grid Test-bed," https://portal.futuregrid.org/, 2011.