

Case Studies in Designing Elastic Applications

Dinesh Rajan, Andrew Thrasher, Badi' Abdul-Wahid, Jesus A Izaguirre, Scott Emrich, and Douglas Thain

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, Indiana 46556

Email:{dpandiar, athrash1, cabdulwa, izaguirr, semrich, dthain}@nd.edu

Abstract—Clusters, clouds, and grids offer access to large scale computational resources at low cost. This is especially appealing to scientific applications that require a very large scale to compete in the research space. However, the resources available across these platforms differ significantly in their availability, hardware, environment, performance, cost of use, and more. This requires the use of *elastic applications* that can adapt to the resources available at run-time, transparently handling heterogeneity and failures. In this paper, we present case studies of several elastic applications built using the Work Queue programming framework. From this experience, we offer six general guidelines for the design and implementation of elastic applications that run on thousands of processors.

I. INTRODUCTION

Clusters, clouds, and grids have grown in popularity by offering access to large arrays of computational and storage resources at low costs. As the demand for such resources continues to increase, the number of providers of clusters, clouds, and grids, and the pool of resources offered through them continues to expand. This trend of increasing accessibility to low-cost resources is especially appealing to scientific applications with large computing needs.

However, this computing environment is very challenging, because it consists of heterogeneous resources with rapidly changing availability and a high probability of failure. For example, the Spot-Pricing [1] service offered by Amazon provides virtual machines whenever the market price falls below the user's threshold. Someone using this service must be prepared for the addition and removal of resources at a moment's notice. (In this way, it is not unlike using a cycle-stealing system such as Condor [2] or BOINC [3].)

Traditional scientific applications are not prepared for this environment, because they are usually built around the assumption of fixed, homogeneous resources at a single location. For example, message-passing applications are usually designed to run on a fixed number of processors – usually a power of two – that cannot change during runtime. A multi-threaded program is usually designed to run on a fixed number of cores selected at startup; changing the number of cores at runtime results in a serious performance penalty.

In contrast, an **elastic application** is designed to operate in the dynamic environment of clusters, clouds, and grids. An elastic application is a distributed program that can dynamically adapt to the resources available at any given moment while accommodating heterogeneity and fault tolerance. This

paper presents our experience in designing and implementing a selection of elastic applications on clusters, clouds, and grids: Elastic MAKER (E-MAKER), Elastic Replica Exchange (REPEX), and Folding At Work (FAW). Their case studies are performed using the Work Queue [4] framework, but the principles apply to many other programming environments.

While there is much guidance in the literature for how to design traditional parallel applications (e.g. [5]), there is little guidance for elastic applications. To that end, we articulate a set of principles based on our experience: (1) *Abolish shared writes*, (2) *Keep your software close and your dependencies closer*, (3) *Synchronize two, you make company; synchronize three, you make a crowd*, (4) *Make tasks of a feather flock together*, (5) *Seek simplicity, and gain power*, and (6) *Build a model before scaling new heights*.

The rest of the paper is organized as follows: Section II describes the characteristics of elastic applications and their architecture. In Section III, three different elastic applications are described. It also studies the factors affecting their performance in heterogeneous and dynamically changing environments and presents techniques applied to overcome or limit their influence. Section IV presents the guidelines derived and established from our experiences in building these applications. Section V describes the related work.

II. ELASTIC APPLICATIONS

Elastic applications harness dynamic distributed resources without imposing limitations on their size, type, platform, and environment. In order to achieve such functionality, elastic applications must exhibit the following characteristics:

- **Adaptability.** They must adapt to resource availability during run time. That is, they must dynamically expand their resource consumption to include resources that become available during execution. At the same time, they must also adapt to resources being lost or terminated during execution.
- **Fault-tolerance.** They must continue execution in the presence of run-time failures. They must isolate failures to individual executions or resources and dynamically recover by re-running the failed executions or by migrating them to successfully operating resources.
- **Portability.** They must be portable across different platforms and environments with limited effort from users. They must be able to execute on any platform using the

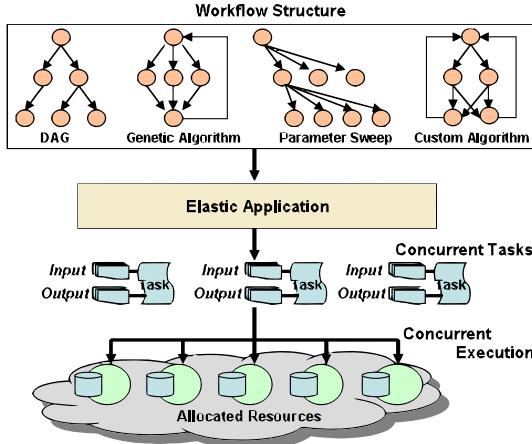


Fig. 1. Architecture of Elastic Applications.

software components specified by users for that platform without having to be re-engineered or rewritten.

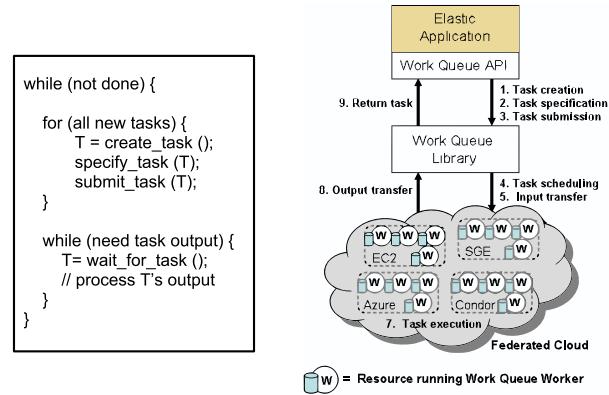
- **Versatility.** By leveraging their quick portability, elastic applications must be able to simultaneously harness resources with diverse operating environments. In other words, users must be able to run them using resources federated from any cluster, cloud, or grid platform as long as the software compatibility with the different operating environments is established.

These characteristics further enable elastic applications to achieve **scalability** and **reproducibility** without requiring dedicated and sophisticated hardware.

Elastic applications are typically implemented by constructing a long-running **coordinator** that submits a large number of short-running **tasks**. The coordinator is responsible for observing available resources, decomposing the workload into tasks of appropriate size, submitting and monitoring tasks, handling fault-tolerance, and interacting with users. The individual tasks are usually self-contained executable programs, along with their expected input and output files. The individual tasks may make use of local physical parallelism in the form of multi-core machines or accelerated hardware such as GPUs, while the coordinator operates at a parallelism of anywhere from one hundred to ten thousand tasks running simultaneously.

As shown in Figure 1, the logical structure of the tasks in an elastic application may vary widely. It could be a directed acyclic graph or *workflow* in which the tasks and their relationship are fully elaborated in advance. It could be a data decomposition in which a large dataset is broken into pieces and processed independently. It could be an iterative algorithm in which a set of tasks is dispatched, evaluated, and then dispatched again until an end condition is met. These structures can describe a variety of applications, such as Monte-carlo simulations [6], protein folding [7], and bioinformatics [8].

The development of elastic applications requires the use of a programming framework that allows the tasks in the application to be specified, dispatched, and executed across the allocated resources. In this paper, we use the Work Queue framework [4] for constructing each of the case studies. However, the principles discussed here are generally applicable



(a) Outline of a Work Queue application. (b) Working of a Work Queue based elastic application.

Fig. 2. The Work Queue Framework.

to applications written using tools such as SAGA [9], Pegasus [10], Taverna [11], Hadoop [12].

The Work Queue framework consists of the following components shown in Figure 2:

- The **Work Queue Library** implements and provides the functionality for coordinating the application execution across workers. It schedules tasks on workers, transfers the inputs and outputs of tasks, and reschedules failed tasks. The library also provides data management capabilities, such as caching and scheduling policies that favor workers with pre-existing data.
- The **Work Queue API** provides the interfaces (in C, Python, and Perl) to the Work Queue library. The API is used to create master (coordinator) programs that create, describe, and submit tasks for execution, and retrieve their outputs upon execution.
- The **Work Queue Worker** is a lightweight execution engine that is run on the allocated resources. It connects to an available or specified Work Queue master and executes the dispatched tasks.

Work Queue dispatches tasks to workers as they establish connection with the master and reschedules tasks running on terminated workers. This allows applications to aggregate resources during their run-time. Work Queue reschedules failed tasks and allows the application to examine completed tasks and resubmit tasks with erroneous results, thereby providing recovery from errors and failures. It requires developers to explicitly specify the inputs and outputs for each task so they can be transferred to the workers. In addition, it distinguishes workers based on their operating environments and transfers the version of inputs specified for a certain environment.

III. CASE STUDIES ON ELASTIC APPLICATIONS

In this section, we discuss, profile, and evaluate three different elastic applications built using Work Queue. Table I summarizes the properties of these applications.

A. Elastic MAKER (E-MAKER)

Genomic annotation is the process of identifying various cellular entities, such as genes, exons, mRNA, in the genome

Application	Function	Input	Computation Kernel	Work Queue Code Size	Number of Tasks	Logical Structure
E-MAKER	Annotate genome sequences	<i>Anopheles gambiae</i> genome	MAKER	~ 1150 lines	~ 10000	
REPEX	Sample conformational space of proteins	<i>WW protein domain</i>	ProtoMol	~ 700 lines	~ 20000	
FAW	Study of protein dynamics	<i>Alanine Dipeptide</i> molecule	Gromacs	~ 600 lines	~ 20000	

TABLE I
PROFILE AND SUMMARY OF THE ELASTIC SCIENTIFIC APPLICATIONS STUDIED IN THIS WORK.

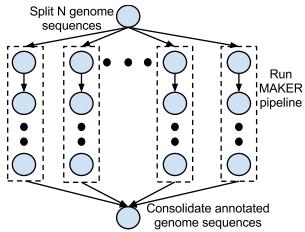


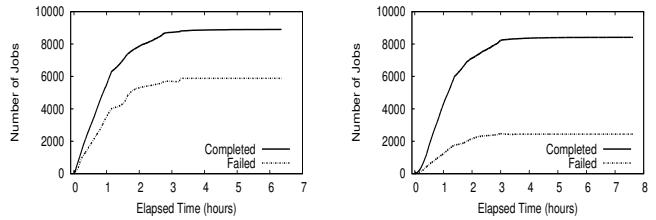
Fig. 3. Logical Structure of E-MAKER

of an organism. Additionally, genomic annotation seeks to assign functional information to these components by assessing similarity to known genomic components of other organisms. MAKER [13] is a commonly used toolchain for genomic annotation. The genomes processed by MAKER are comprised of *contigs*, or large contiguous sequences in a genome.

We previously described the conversion of the native MAKER implementation to a Work Queue based implementation in [14]. We refer to the Work Queue based implementation as Elastic MAKER or E-MAKER for the remainder of the paper. Figure 3 describes the logical structure of E-MAKER. In this work, we identify the factors affecting its performance, and the modifications we made to improve its performance.

E-MAKER inherits most of the techniques and mechanisms in the native MAKER implementation. However, the native MAKER implementation used MPI to operate in parallel. As a result, its design and development was heavily influenced by the execution environment in parallel computing systems that is homogeneous and consistently available across multiple resources. For instance, MAKER writes the outputs of its tasks to a common shared file. This allowed outputs to be continuously aggregated in a single file. However, this setup incurs the overheads of file locking mechanisms that enable concurrent and consistent write accesses. Further, MAKER requires a shared filesystem to store and manage its inputs and outputs. Such techniques and requirements limited the ability and performance of E-MAKER on heterogeneous resources.

To improve the performance of E-MAKER, we made the following modifications. First, we modified the tasks to write their outputs in their local execution environment. On completion of the tasks, we gathered these outputs from their execution sites to produce the final output. Second, we explicitly specify the inputs and outputs for tasks to be transferred to and from the resources chosen for task executions. This elim-



(a) Failures of E-MAKER using the shared file system. (b) Failures of E-MAKER using distributed & dedicated accesses.

Fig. 4. Comparison of failures in the E-MAKER versions using shared file system and dedicated data access.

inated the assumption or requirement of a shared filesystem spanning the allocated resources, which is impractical when resources are derived from multiple platforms. In addition, such operating environments include resources that fail or are terminated during run-time. Hence, dedicated and distributed write accesses avoid scenarios where resource failures corrupt the data stored in a shared filesystem. Figure 4 compares the failures observed in the E-MAKER implementations using shared file system and dedicated accesses when running on the Condor grid at Notre Dame. The failures in Figure 4(a) are due to (1) failures in write accesses due to locks and (2) failures due to Condor terminating jobs. The dedicated access version of E-MAKER eliminates the failures in write accesses thereby lowering overall failures and improving stability.

Figure 4(b) shows the number of failures to be lower during run time. We also notice that, despite the lower failure overheads, the overall completion time is longer compared to Figure 4(a). From our investigation, we attribute this to two factors: (1) overheads from explicitly transferring input and output data, and (2) use of shared and heterogeneous resources. In studying the transfer overheads, we found E-MAKER to operate by simply creating a task for each contig in the input set. This manner of uninformed decomposition often resulted in high transfer overheads leading to sub-optimal run-time performance. To overcome this limitation, we first formulated the running time of E-MAKER as follows:

$$T(k, m) = \alpha(k) + \frac{\beta}{k} + \Delta(k, m) \quad (1)$$

In Equation 1, α and β represent the sequential and parallel execution components of E-MAKER respectively. β is

dependent on the number of contigs, n , to annotate. β along with the number of tasks created for annotation, denoted by k , determine the parallel execution run times. The sequential component, α , includes the time to partition β and process the results of k tasks, and therefore is dependent on k . $\Delta(k, m)$ represents the transfer overheads of the k tasks and is dictated by the average length of the contigs m . Note that this model differs from Amdahl's law in that the sequential work done by coordinator (master) increases with the degree of parallelism.

We acquire information on number of contigs (n), their average length (m), and the time taken to annotate a contig of average length (t) using the inputs specified for E-MAKER. These can either be provided by the user or derived by a simple benchmark or profiling script run on the input set. We evaluate β as the product of the number of contigs and the average time to annotate a contig in the input set. We profile the execution of E-MAKER to measure the sequential execution overheads along with partitioning and processing overheads incurred with respect to the number of tasks. For simplicity in modeling, we assume that the allocated resources exhibit similar performance characteristics.

Variable	Value	Comments
$\alpha(k)$	$(a * k) + b$	a, b are empirically determined as 0.08 and 65 respectively
β	$t * n$	t is obtained from input
$\Delta(k, m)$	$(c * n) + (d * k)$	c is obtained from input; d is empirically determined as 10

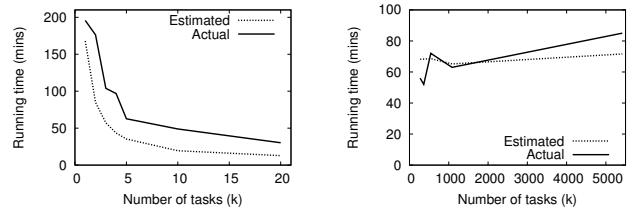
TABLE II
VALUES USED TO ESTIMATE $T(k, m)$ IN E-MAKER.

Table II describes values for the simple model used in computing $T(k, m)$ for E-MAKER. Using this model, we estimate the run times for an input set of 20 sequences each containing 20,000 contigs. For this set, we estimate each annotation to take about 500 seconds on the Condor grid at Notre Dame (i.e., $t=500$ seconds). These sequences together incur less than 1MB of data transferred as inputs and outputs of tasks. We use a conservative network bandwidth of 1 MB/s and therefore derive $c * n$ to be 1. Figure 5(a) shows the estimated running time using Equation 1 and the actual running time for annotation of this input set on the Condor grid at Notre Dame.

Equation 1 assumes that the resources allocated for execution is equivalent to the number of parallel tasks k . So let us consider the case where the number of allocated resources r is smaller than k . Here, only r tasks can be executed in parallel at any given time. As a result, the execution of k tasks will be prolonged by a factor of $\lceil k/r \rceil$. Equation 1 now becomes:

$$T(k, m, r) = \alpha(k) + \frac{\beta}{k} * \lceil \frac{k}{r} \rceil + \Delta(k, m, r) \quad (2)$$

The values in Table II also apply in this setup except for $\Delta(k, m, r)$. In E-MAKER, we cache the software components and other run-time dependencies after the initial transfer. In Equation 1, these transfer overheads were incurred for each task and modeled as $d * k$. In Equation 2 these overheads are incurred for each resource (since they are transferred once and cached), and therefore are modeled as $d * r$. Further, we



(a) 20 sequences of 20,000 contigs (b) 5418 sequences of 2000 contigs

Fig. 5. Illustration of the estimated running time from Equation 1 and the actual running time of E-MAKER with two different input sets.

ignore the transfer overheads specific to each task since they are negligible. $\Delta(k, m, r)$ is computed here as $(c * n) + (d * r)$.

Using Equation 2, we estimate the running times for an input set of 5418 sequences each containing 2,000 contigs. For this set, we estimate t to be 50 seconds on the Notre Dame Condor pool. The transfer overhead for these sequences is estimated around 6 seconds with a 1 MB/s bandwidth. Figure 5(b) shows the estimated running time using Equation 2 and the actual running time when r is 100 resources.

We have several observations from Figure 5. First, we notice from Figure 5(a) and 5(b) that increasing the number of tasks have contrasting effects on the running time. This shows that the run-time performance of E-MAKER varies based on the input set and size of allocated resources. Second, this model can guide the user in determining the size of resources to allocate. For instance, in the experiment in Figure 5(a), the user can benefit from increasing the size of allocated resources to 20. Third, when the resource allocation is fixed due to user constraints, it becomes important to adapt task decomposition to the resource allocation. In Figure 5(b), decomposing the parallelism into 1000 tasks or less is beneficial compared to larger task sizes. Our ongoing work is focussed on improving the accuracy of the model and incorporating it in E-MAKER to adapt its task decomposition and guide resource allocation.

Finally, the native implementation of MAKER utilized a global logging and failure recovery mechanism. This mechanism restarted MAKER at the last successfully logged global state on encountering a failure. We modified E-MAKER to use the failure recovery mechanisms offered in Work Queue. This allowed E-MAKER to isolate failures to individual tasks or resources, migrate tasks from failed resources, validate task outputs, and resubmit tasks with erroneous outputs. With this modification, E-MAKER eliminates the overheads and costs of global logging and recovery in the presence of failures.

We also noticed that the software executables of the tasks in E-MAKER required libraries, such as BioPerl, for their execution. Since the operating environments of the allocated resources are diverse and are not guaranteed to include these software dependencies, we explicitly specify the executables along with their required libraries as the inputs of each task. This allowed the operating environment for each task to be transferred and correctly setup at the allocated resources. This modification enabled E-MAKER to (1) harness resources irrespective of the suitability of their native operating environment to task executions, and (2) handle heterogeneous

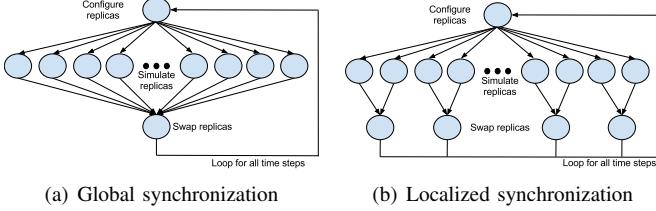


Fig. 6. Logical Structure of REPEX.

operating environments by transferring the version of the software components compatible with those environments.

Summary. We made the following modifications and improvements to E-MAKER: First, E-MAKER transfers required files and data to the resources running the tasks and avoids reliance on shared filesystems. This simplifies the run-time environment and avoids the bottleneck in using file locking mechanisms. Second, E-MAKER transfers the software components required for execution as input files to the resources. This enables the execution environment to be setup correctly on heterogeneous resources and avoid imposing limitations on the operating environments that can be used for task executions. Third, E-MAKER uses the failure isolation and recovery mechanisms in Work Queue to eliminate the overheads of a global logging and recovery system. Finally, we formulate a model of the run-time performance of E-MAKER to determine task decomposition strategies and guide users on the size of resources to allocate.

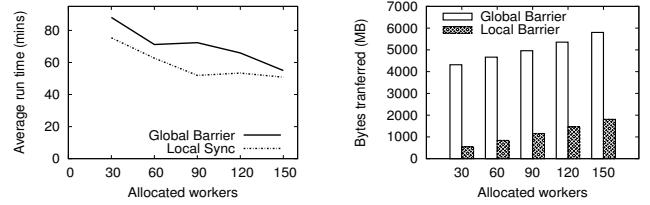
B. Replica Exchange simulations (REPEX)

Replica Exchange is a sampling method applied in studying protein molecules and their movements across different geometric boundaries in their conformational spaces. It samples the protein molecule in different geometric spaces and configurations by creating multiple replicas of the molecule.

Replica exchange simulations (REPEX) work by creating multiple replicas of a protein molecule at different temperature configurations and simulating their dynamics over multiple time steps. These replicas are independent of each other and therefore, can be simulated concurrently. At the completion of each step, a pair of neighboring replicas are randomly selected and an exchange of their configuration is attempted.

Our earlier work [15] described the construction of the replica exchange simulations as an elastic application using Work Queue. Similar to the native MPI implementation, elastic replica exchange was implemented using global synchronization barriers. That is, the entire set of simulated replicas were synchronized at the end of each time step when an exchange is attempted. Each time step, therefore, served as a global barrier. These barriers were used to ensure the two random neighboring replicas chosen to attempt an exchange were at the same time step in the simulation. Figure 6(a) illustrates the logical structure of replica exchange with global barriers.

These global barriers worked effectively in ensuring correctness of the replica exchange in parallel computing environments that typically consisted of dedicated homogeneous resources. Due to the homogeneity in these environments,



(a) Comparison of the average run times from 10 runs simulating 150 replicas over 150 time steps.

(b) Comparison of the transfer overheads from a single run simulating 150 replicas over 150 time steps.

Fig. 7. Comparison of the running time and transfer overheads of the global and localized barrier version of REPEX. These are plotted for different sizes of allocated workers.

the performance impact of using each time step as a global barrier was minimal and often overlooked. However, in the heterogeneous operating environments of elastic applications, global barriers introduce delays and overheads which adversely impact the overall performance of the application.

Therefore, to run replica exchange simulations efficiently as elastic applications, the presence of global barriers must be removed. In this work, we replace the global barrier at each time step with barriers spanning two neighboring replicas. We pre-compute the pairs of replicas used for an exchange attempt at each step, and require only those pairs to synchronize at their exchange step. That is, the barriers span a replica pair and occur at the exchange step of that pair. Figure 6(b) illustrates the replica exchange logical structure with localized barriers.

The use of local barriers resulted in improvements in the run-time performance over the globally synchronized version as illustrated in Figure 7. Figure 7(a) plots the running time of experiments involving 150 replicas averaged over 10 runs. It shows the average running time as the number of workers allocated on the Condor grid at Notre Dame is varied. We observe that the running times of the local barrier version were faster in the presence of shared resources, heterogeneous hardware, resource failures (these factors are also the reasons behind the uneven run times observed for both versions).

The use of localized barriers also enabled the workload to be partitioned such that the tasks within a partition share data and generate outputs that serve as inputs for other tasks in that partition. This avoids the transfer of input and output files at every time step and makes better use of the available storage capacities at allocated resources. Figure 7(b) illustrates the lower transfer overheads of the localized barrier version compared to the global barrier version.

In this work, we use ProtoMol [16] to run simulations on the replicas in REPEX. Similar to E-MAKER, we explicitly specify and transfer the software components of ProtoMol, such as its executable, to the allocated resources.

Summary. We replaced the global synchronization barriers in REPEX with localized barriers spanning two replicas. This resulted in two improvements: (1) lower time to completion and (2) lower data transfer overheads.

C. Folding@Work (FAW)

A number of biologically significant events in proteins happen at very small timescales (micro-seconds to femto-

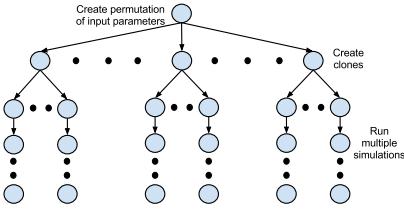


Fig. 8. Logical Structure of FAW.

seconds). To capture these events at such small resolutions, the sampling of these systems has to happen at unprecedented speeds and scales. This requires the use of sophisticated or supercomputing hardware and therefore is constrained by costs and access to these resources. As a result of this limitation, biomolecular scientists have developed alternative approaches that run on commodity hardware. These approaches leverage parallelism in the analysis and simulations techniques of such systems. Specifically, these approaches decompose long trajectories in the protein folding studies into smaller and parallel trajectories that are close approximations.

The Folding@home project is one such framework that applies parallelization to capture the protein folding phenomenon by running on idle commodity hardware [17]. We build on the Folding@home project to create a framework, called Folding@Work or FAW, that is customizable and flexible to suit the needs of the users and their studies. In FAW, we allow users to specify and customize the simulation environments and the nature of their analysis. We also allow resources from multiple sources and platforms such as Amazon EC2, Microsoft Azure, Condor, etc., to be federated by users to achieve the scale and performance they desire in their experiments.

The FAW framework is built using Work Queue to run as an elastic application. This framework is invoked with the specification of several experimental parameters such as molecular structure, temperature, etc. FAW applies these specifications to construct and decompose its workflow into tasks. Our work in [4] describes in detail the construction of FAW using Work Queue. In this paper, we focus on studying and improving the performance of the FAW framework.

We begin by observing that a typical FAW workflow involves a collection of clones called *trajectories* that are simulated in parallel. Each trajectory represents the path taken by a protein molecule in achieving a folded state. A fully formed trajectory contains the path taken by a protein molecule to reach folded state. The goal of such workflows is to aggregate and study as many fully formed trajectories as possible. As a result, the number of fully formed trajectories dictates the throughput of an experimental run in FAW.

We now describe the technique used to improve the performance of FAW in terms of its throughput. FAW decomposes its workflow into a set of tasks corresponding to each trajectory in the simulations. It uses a round-robin approach in creating and submitting tasks for each trajectory. By replacing the round-robin approach with a mechanism that clusters and prioritizes tasks corresponding to a trajectory closer to achieving fully folded state, the throughput of FAW can be enhanced. Using this insight, we modified the design of FAW to cluster tasks

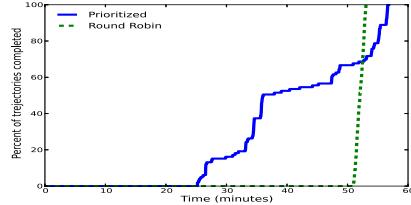


Fig. 9. Plot comparing the throughput of the prioritized and round-robin approaches in FAW.

such that trajectories closer to completion are prioritized during execution. The benefit of this approach is illustrated in Figure 9 where we compare the throughput, which is the percentage of completed trajectories, with and without prioritization. The experiments in this figure involved 100 clones each running 20 simulations using 50 workers on the Notre Dame SGE cluster. We observe that the clustering and prioritization approach yields fully folded trajectories throughout its run time. This provides opportunities to analyze and gather scientific data much earlier in the runs. This also implies that users can quickly achieve scientific output in running FAW, even with smaller or shorter resource allocations.

For the experiments in this work, we use Gromacs [18] to simulate the protein trajectories and we explicitly specify and transfer the components and environment required for the execution of Gromacs to the allocated resources.

Summary. We modified the construction of FAW to cluster and prioritize the tasks corresponding to the trajectories closer to achieving completion. This clustering and prioritization technique improved the throughput of the FAW framework.

IV. GUIDELINES FOR ELASTIC APPLICATION DESIGN

From our experiences in building the three elastic applications in the previous section, we observe an absence of general principles to guide their design and construction. Developers currently devise and apply application-specific strategies using their knowledge of the application or by profiling execution to improve performance. However, such techniques are error-prone, time-consuming, or ineffective when applied across a selection of elastic applications.

In this section, we derive general guidelines for the design and development of elastic applications using the lessons learned in building such applications and improving their performance. While these guidelines are not exhaustive, we believe they are a necessary and useful first step in helping developers build efficient elastic applications.

We first identify and define the metrics used in this work to describe the performance and efficiency of elastic applications:

- (1) *Time to completion*: This represents the overall execution time of the application. A lower time to completion presents benefits to users such as lower costs incurred in maintaining resources for execution.
- (2) *Scientific throughput*: This measures the scientific output achieved per time unit per allocated resource. Applications with good throughput allow users to quickly gather useful data and results, even with smaller resource allocations.

- (3) *Transfer overheads*: This tracks the overheads in transferring data required for execution on allocated resources. Lower transfer overheads result in lower network latencies and costs.
- (4) *Cost of failure*: This represents the costs of encountering, handling, and recovering from failures. These costs negatively impact the time to completion and throughput.
- (5) *Ease of deployment*: This evaluates the effort expended by users in deploying the application on allocated resources. It includes the effort in configuring resources to provide the operating environment required by the application.

These metrics determine the costs, overheads, and effort incurred by users, and the achievable scale in running elastic applications. We use these metrics to illustrate the benefits of the following guidelines for the design of elastic applications.

1) ***Abolish shared writes***: The use of a shared file to write and aggregate the outputs of tasks during execution are prone to locking overheads and failures as shown in Section III-A. Elastic applications must therefore implement dedicated and distributed write accesses where files are created and written locally at the site of the task execution. These files can then be transferred from the execution sites (allocated resources) and aggregated at the controller (master). This also allows the storage capabilities at the allocated resources to be utilized effectively. Further, distributed write accesses isolate the performance characteristics and failures of the individual resources thereby minimizing their impact on the overall performance of the application. We showed the benefits of dedicated and distributed write accesses in lowering failures in E-MAKER in Section III-A. In addition, such accesses simplify the execution environment by avoiding the requirement of a shared file system, and hence improve the ease of deployment.

2) ***Keep your software close and your dependencies closer***: Elastic application are often deployed on resources with diverse operating environments. To effectively utilize these allocated resources irrespective of their operating environments, elastic applications must transfer and setup the execution environment of each task on the allocated resources. That is, the software components and dependencies of each task, such as executables and libraries, must be encapsulated in the task inputs transferred to its execution site. We applied this technique in the elastic applications described in Section III to successfully harness resources without imposing any assumptions or requirements on their operating environments. Further, this results in high ease of deployment since the user can deploy and run the application on resources or environments of his choice without any additional effort.

3) ***Synchronize two, you make company; synchronize three, you make a crowd***: Elastic applications with dependencies between iterations or sets of tasks require synchronization mechanisms to maintain these dependencies. One such mechanism is the global synchronization barrier that spans the entire set of concurrent tasks in the application. However, such global barriers introduce inefficiencies in the presence of heterogeneous resources with diverse performance characteristics and adversely impact the time to completion.

Therefore, elastic applications must diligently isolate the synchronization requirements to the smallest feasible set of tasks. The use of this technique and its effects in lowering the time to completion were described in Section III-B. We also observed from the evaluations in Section III-B that removing the global barrier yields other benefits, such as lower transfer overheads.

4) ***Make tasks of a feather flock together***: Ensemble workflows, where a set of independent simulations or computations are run and aggregated as part of a scientific study, can be implemented as elastic applications. In such instances, the outputs of each task or a cluster of tasks contribute directly to a scientific result or output. Elastic applications of ensemble workflows must therefore cluster and prioritize the execution of sub-workflows or tasks that immediately contribute to the scientific output expected during their execution. We showed the application of this technique in the FAW framework (in Section III-C) to enhance its scientific throughput. Another benefit of improving the scientific throughput through such techniques is that it allows useful scientific output to be obtained quickly even with resource allocations of smaller sizes or shorter durations.

5) ***Seek simplicity, and gain power***: The choice of the programming abstraction for elastic applications plays a significant role in achieving scale and good performance. In our construction of the elastic applications in Section III, we employed Work Queue, that offers a simple and essential set of interfaces to implement and run programs in a master-worker framework. The simplistic and minimalist design of Work Queue requires applications to explicitly (i) decompose workflows into tasks, (ii) specify the inputs to be transferred to the workers for each task, and (iii) aggregate the outputs of completed tasks. However, these explicit requirements allowed applications to harness heterogeneous operating environments, manage and cache data across the allocated resources, and isolate failures. In other words, the sophistication in fault-tolerance, elasticity, handling heterogeneity, and data management directly follows the use of a simple and minimalist interface.

6) ***Build a model before scaling new heights***: Elastic applications run large computations by decomposing them into tasks. The decomposition of tasks allows concurrent execution but incurs transfer overheads. This decomposition also dictates the size of resources that achieve optimal running time. Therefore, it becomes imperative to formulate a model that captures the effects of task decomposition on the run-time performance of the application as shown in Section III-A. This model must be incorporated in the application and used to (i) drive the decomposition of the workflow into tasks, (ii) inform the user of the estimated performance for a given input, and (iii) guide the user in allocating resources for execution with a given input. Such models are especially useful when the applications are designed to achieve scale and are deployed on resources that incur monetary costs to the user.

V. RELATED WORK

Several efforts have studied the design and engineering of applications for heterogeneous computing platforms. Wolski et

al., were the first to describe the characteristics for programs that run on grids and present a framework for running such programs [19]. More recently, [20] demonstrates a framework that transparently runs applications across resources in both cloud and grid platforms.

There have been numerous efforts in studying scientific workflows, and identifying guidelines for their execution on distributed systems. The authors in [21] provide a characterization of scientific workflows in terms of their size, complexity, composition, and computation requirements. The authors in [22] study the evolution of distributed infrastructure such as grids, and profile a set of applications deployed on such infrastructure. They use this study to offer critical insights on the challenges facing the developers of distributed computing systems (such as grids) and the distributed applications deployed on these systems. The work in [23] describes the characteristics of workflow execution systems and identifies design guidelines to improve their usability and adoption rates. The work in [24] identifies the incorrect assumptions and practices observed among designers and developers of applications for cloud environments.

The work in [6] presents experiences in running Monte-Carlo based scientific applications using a master-worker framework on the Condor grid. The authors in [25] report and evaluate their experiences in running scientific applications in a virtualized cloud environment. They show that the scheduling and communication overheads need to be carefully considered in evaluating the benefits of running scientific applications on a cloud platform. Lu et al, in [26], describe their experiences in running a bioinformatics application on Microsoft's Windows Azure and present best practices for handling parallelism and overheads in such platforms. Our work differs from these efforts by presenting a systematic study of the construction of different scientific applications. We then use this study to establish guidelines to follow in their construction to achieve scale and efficient performance when running on inexpensive, dynamically changing, and multiple heterogeneous operating environments and platforms.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants OCI-1148330, CNS-0643229, and CNS-855047.

REFERENCES

- [1] Amazon EC2 Spot Instances, <http://aws.amazon.com/ec2/spot-instances>, 2012.
- [2] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Eighth International Conference of Distributed Computing Systems*, June 1988.
- [3] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.
- [4] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.
- [5] K. Hwang and Z. Xu, *Scalable parallel computing: technology, architecture, programming*, ser. Computer engineering series. WCB/McGraw-Hill, 1998.
- [6] J. Basney, R. Raman, and M. Livny, "High throughput monte carlo," in *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
- [7] B. Abdul-Wahid, L. Yu, D. Rajan, H. Feng, E. Darve, D. Thain, and J. A. Izaguirre, "Folding Proteins at 500 ns/hour with Work Queue," in *8th IEEE International Conference on eScience (eScience 2012)*, 2012.
- [8] A. Thrasher, Z. Musgrave, D. Thain, and S. Emrich, "Shifting the Bioinformatics Computing Paradigm: A Case Study in Parallelizing Genome Annotation Using Maker and Work Queue," in *IEEE International Conference on Computational Advances in Bio and Medical Sciences*, 2012.
- [9] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf, "SAGA: A simple API for grid applications. high-level application programming on the grid," in *Computational Methods in Science and Technology*.
- [10] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berrieman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.
- [11] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1067–1100, Aug. 2006.
- [12] Hadoop, <http://hadoop.apache.org/>, 2007.
- [13] C. Holt and M. Yandell, "MAKER2: an annotation pipeline and genome-database management tool for second-generation genome projects," *BMC Bioinformatics*, no. 12, p. 491, 2011.
- [14] A. Thrasher, D. Thain, S. Emrich, and Z. Musgrave, "Shifting the bioinformatics computing paradigm: A case study in parallelizing genome annotation using maker and work queue," *Computational Advances in Bio and Medical Sciences, IEEE International Conference on*, vol. 0, pp. 1–6, 2012.
- [15] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain, "Converting a high performance application to an elastic cloud application," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, Nov. 2011, pp. 383–390.
- [16] T. Matthey and et al., "Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics," *ACM Transactions on Mathematical Software*, vol. 30, pp. 237–265, September 2004.
- [17] M. Shirts and V. S. Pande, "Screen savers of the world unite!" *Science*, vol. 290, no. 5498, pp. 1903–1904, Dec. 2000.
- [18] E. Lindahl, B. Hess, and D. van der Spoel, "Gromacs 3.0: a package for molecular simulation and trajectory analysis," *Journal of Molecular Modeling*, vol. 7, pp. 306–317, 2001.
- [19] R. Wolski, J. Brevik, G. Obertelli, N. Spring, and A. Su, "Writing programs that run EveryWare on the computational grid," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, 2001.
- [20] A. Merzky, K. Stamou, and S. Jha, "Application level interoperability between clouds and grids," in *Workshops at the Grid and Pervasive Computing Conference, 2009*. IEEE, May 2009, pp. 143–150.
- [21] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*. IEEE, Nov. 2008, pp. 1–10.
- [22] S. Jha, D. S. Katz, M. Parashar, O. Rana, and J. Weissman, "Critical perspectives on large-scale distributed applications and production grids," in *Grid Computing, 2009 10th IEEE/ACM International Conference on*. IEEE, Oct. 2009, pp. 1–8.
- [23] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, "Scientific workflow design for mere mortals," *Future Gener. Comput. Syst.*, vol. 25, no. 5, pp. 541–551, May 2009.
- [24] D. G. Malte Schwarzkopf, "The seven deadly sins of cloud computing research," in *4th Usenix Workshop on Hot Topics in Cloud Computing*, 2012.
- [25] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berrieman, and J. Good, "On the use of cloud computing for scientific workflows," in *2008 IEEE Fourth International Conference on eScience*. Washington, DC, USA: IEEE, Dec. 2008, pp. 640–645.
- [26] W. Lu, J. Jackson, and R. Barga, "AzureBlast: a case study of developing science applications on the cloud," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 413–420.