Tools to Execute An Ensemble of Serial Jobs on a Cray

Abhinav Thota*, Scott Michael*, Sen Xu*, Thomas G. Doak*, Robert Henschel*
*Indiana University, Bloomington, IN 47403
{athota, scamicha, henschel}@iu.edu, {senxu, tdoak}@indiana.edu

***Abstract*—Traditionally, Cray supercomputers have been located at large supercomputing centers and were used to run highly parallel applications. The user base consisted mostly of researchers from the fields of physics, mathematics, astronomy and chemistry. But in recent times, Cray supercomputers have become available to a wider range of users from a variety of disciplines. Examples include the Kraken machine at the National Institute for Computational Sciences (NICS) [1], Hopper at the National Energy Research Scientific Computing Center (NERSC) [2], and Big Red II at Indiana University [3]. Predictably, as the diversity of end users has grown, the workload has expanded to include a variety of workflows containing serial and hybrid applications, as well as complex workflows involving pilot-jobs. Projects that employ a massive number of serial jobs–in an embarrassingly data-parallel manner–have not been targeted to run on Cray supercomputers. To accomplish such projects, it is usually necessary to bundle a large number of serial jobs into a much larger parallel job, *via* either a pilot job framework, an MPI wrapper, or custom scripting. In this article, we explore several of the current offerings for bundling serial jobs on a Cray supercomputer and discuss some of the benefits and shortcomings of each of the approaches. The approaches we evaluate include BigJob, PCP, and native aprun with scripts.**

I. Introduction

Over the past decade, supercomputing resources have become increasingly accessible to the individual researcher. As a result, the number of researchers and the breadth of scientific disciplines using supercomputers has steadily increased. As a natural result, there is now a tremendous amount of diversity in both the science domains and the workflows of the user base of supercomputers. In tandem with the increase of user numbers and diversity, the number of applications supported on supercomputing resources has grown. Though it is still primarily the case that the consumers of the largest amount of supercomputing resources come from the core fields of physics, mathematics, astronomy and chemistry, one can observe an increasing number of users from the fields of biology, bioinformatics, finance, geology, and psychology.

With this influx of new users in a more diverse range of scientific domains, the thinking on what kinds of applications are considered appropriate to run on a supercomputer has shifted. While parallel jobs are widely recognized as being most suitable to be run on a supercomputer, a growing percentage of the total workload is now serial jobs. Many users have hundreds to thousands of very similar jobs, sometimes referred to as parametric sweeps or data-parallel applications. These workloads are common in the fields of bioinformatics and biology. Other non-traditional serial jobs include data analysis applications with extremely large data volumes, which may require large amounts of memory.

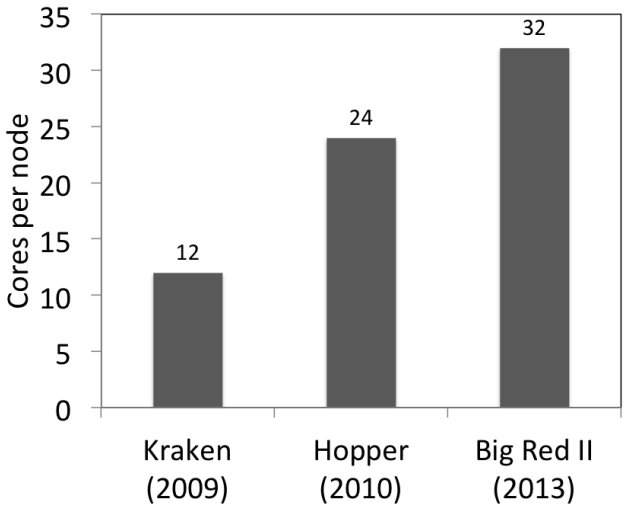

Fig. 1: Over the years, processors have become more and more parallel. The chart shows the number of cores per compute node on Kraken, Hopper and Big Red II and the year the machines were launched. Each of these machines have two processors per computer node, but the number of cores per node has been increasing.

Juxtaposed with the changing landscape of workloads is the fact that processors are becoming more and more parallel. For example, Kraken at NICS has 12 cores per node, Hopper at NERSC has 24 cores per node and Big Red II at Indiana University has 32 cores per node, as shown in Figure 1. The only way to ensure high levels of core utilization on a large machine, while welcoming a diverse set of users, is to put in place helpful tools and policies to enable both traditional users and massively serial users in making effective use of supercomputing resources.

Over the past several months our research group has had no small amount of experience with this very issue. For this project we investigated genomic data using a software package called mlRho. The mlRho software is a serial program that estimates mutation, recombination, and sequencing error rates from genome sequences [4]. To complete an analysis for a variety of different species and individuals, we estimated that the mlRho program would require ≈6 million core hours. mlRho jobs are extremely data parallel, in that the hundreds of thousands of iterations needed per genome in the linkage disequilbrium method used for analysis can all be performed independently. To complete the planned analysis, we routinely

ran hundreds to thousands of mlRho jobs in parallel. Generally speaking, there are two ways to run many serial jobs simultaneously: using the existing job submission scripts or using a specialized tool. In both cases, we wanted to make sure that we were fully utilizing all of the resources on a node.

The primary goal of this paper is to list the different tools which help serial application users run their jobs efficiently. Many of the tools that we will list here are not widely known to the community. When selecting a tool for managing many serial jobs, users must consider a variety of factors beyond the efficacy of the tool and its ease of use. For example, many computing centers limit the number of jobs that a user can submit to the batch system and the number of jobs that can be running concurrently, which can adversely impact the throughput of the simplest script based methods. To deal with this potential shortcoming, we present in this paper two tools which can be used to redesign job submissions to more closely suit a center's policies. We also test a simple script based job submission method and present the results.

To evaluate these different bundling options, we have worked with a group of biologists at Indiana University, who are using mlRho[4]. Given the non-trivial amount of computational resources required, we assisted in preparing an XSEDE[5] allocation request and conducted a survey of tools that could be used to bundle serial jobs on the Cray supercomputer Kraken at NICS. We present these findings for the mlRho code as a case study of the BigJob, PCP, and aprun serial job bundling approaches, and will describe ways in which users with hundreds of thousands of serial jobs can efficiently bundle and run their projects on large Cray machines.

Some of the other solutions that have been suggested previously include enabling shared nodes using cluster compatibility mode for serial jobs[6] and Swift[7], a parallel scripting language. While end users can take advantage of the shared node setup without making any significant changes to their workflow, they would definitely need to make some changes to take advantage of Swift and some of the tools we suggest in this paper.

The rest of the paper is organized as follows: in section II, we introduce the scientific problem we are working on. In section III, we introduce the tools that can be used to run serial jobs on a Cray. In section IV, we describe the experiments we have conducted on an XSEDE machine Kraken, compare the different tools and discuss the list of things to consider before choosing a particular tool. We conclude the paper in section V.

II. Scientific Background

The amazing biodiversity on our planet has fascinated humans for thousands of years. To understand how this diversity arises and is maintained, it is critical to determine fundamental ecological and genetic parameters (e.g., population sizes, recombination rates) for a range of species. These parameters play important roles in creating opportunities for increasing genetic diversity, population divergence, and speciation. The mlRho software is a package which uses next-generation genomic sequencing to generate a novel measure of linkage disequilibrium. Deploying mlRho on XSEDE resources has allowed us to study these important population-genetic parameters in a broad assembly of eukaryotic genomes.

Although there are several methods for determining recombination rates, they can be both time and resource intensive. The mlRho software employs a novel analytic approach that uses a new metric called the zygosity correlation coefficient, which is estimated using maximum likelihood (ML) methods. It only requires single individual genome sequences, but is extremely data intensive. Using mlRho and XSEDE computational resources, we have been able to examine the recombination rates of a plethora of species with accuracy that was previously unachievable.

A. Program Description

The underlying data consists of assembled sequencing reads obtained from a single diploid individual. Such data are collected, for example, for the 1000 human genome project. mlRho reads a profile consisting of the number of each nucleotide (`A`, `C`, `G`, and `T`) from a file at each sequenced position. Given a mutation and error rate, mlRho computes two probabilities for each profile: the probabilities of observing the profile given that the position is either mutated (heterozygous), or not (homozygous). These probabilities depend on the mutation and error rates. By varying these, mlRho finds the values that maximize the overall likelihood of the data.

While mutation and sequencing error affect individual genome positions, recombination uncouples the evolutionary history of pairs of positions. This is observable as a decorrelation of the zygosity states between pairs of positions. To estimate recombination, mlRho computes the probability of observing profile pairs separated by, say, 1000 nucleotides. This is a function of the recombination rate and the single position likelihoods.

III. Tools to run serial jobs on a Cray

Running serial jobs on a Cray is not a difficult task. The difficulty lies in running serial jobs in such a way that we are: (i) using all the cores on a node, and (ii) are not putting ourselves at a disadvantage by submitting a lot of single node requests when the scheduler is set up to prioritize large jobs or vice versa. The administrators at a supercomputer center usually configure the job scheduler in a way that best supports the mission of that particular supercomputer . There can be many types of queues that the users can submit to. Common configurations include queues for serial jobs, and queues for small, medium, and large jobs, where the specific size of small, medium, and large depend heavily on the size of the system.

Cray supercomputers are usually designed to run highly parallel applications on thousands of processors. The compute nodes on a Cray cannot be shared by multiple users. Therefore, it is highly inefficient to submit single core jobs to the scheduler, as only a single core per node will be utilized and 90% of the node is unused. At a minimum, the user has to bundle enough jobs to use all the cores on a node. This can be done by including enough job execution commands in a job submission script. But, even with this solution, the user is required to submit many single node requests to the scheduler. What is needed is a tool that allows one to bundle as many serial jobs as needed in to one large job of a size that makes

it appropriate for a particular machine. This tool should make it possible to use multiple compute nodes and all the cores on an individual node. We describe a few tools that can be used to do this in the following sections. We do not go into the implementation details, but rather present these tools from a user's point of view.

A. BigJob, a SAGA-based Pilot-Job

BigJob is a SAGA-based, general purpose pilot-job framework. It is maintained by the Research in Advanced Distributed Cyberinfrastructure and Applications Laboratory (RADICAL) at Rutgers University [8]. BigJob can be used in many ways – as a container job, to distribute jobs to multiple resources or to coordinate the launch and interaction of jobs within the container. But in this paper, we are using it as a container job. The pilot-job/container job takes in all the serial jobs and holds them. At the same time it also submits a job request big enough to contain all of the serial jobs to the scheduler and runs the serial jobs concurrently when the compute resources become available. The software is written in Python and a basic knowledge of Python is needed to work with the programming interface. More information about the framework can be found on the BigJob website [9].

We used BigJob to bundle our serial mlRho simulations; BigJob is available on many XSEDE resources, such as Kraken, Stampede, and Lonestar. Many researchers have successfully used BigJob [10] to bundle hundreds of smaller jobs into larger, more manageable groups of jobs [11, 12].

The BigJob tool is available for download from the Python Package Index[13]. The only prerequisite is Python 2.6 or higher. The architecture of BigJob is shown in Figure 2. Though we will not go into the architectural details of BigJob, as it is beyond the scope of this work, it is sufficient to say that the architecture may appear quite complex due to the fact that BigJob can be used for many different purposes. Briefly, to achieve the goal of running a very large number of serial jobs, a python script is created that the user edits and runs. This script submits the container job to the scheduler and stores the serial job descriptions in a database. As compute nodes become available, the next serial job in the database is run. A pilot-job size can be chosen to optimally match the job scheduler policies on a particular machine.

BigJob maintains the list of processors allocated after the job request becomes active. The user can design the BigJob to assign these processors to start and manage smaller jobs. The main benefit of doing this is that instead of submitting thousands of single core job requests to the queue, we can submit hundreds of large job requests ($\approx$ 500 to 5000 cores) to the queue. This reduces the overall number of job submissions to the queue and thereby, to some extent, time spent waiting in the queue. This job size is also more appropriate for many of the larger machines. We have tested this on the Cray XT5 machine Kraken.

An example BigJob script would include the following parts:

Information about the job request to be submitted to the scheduler (with comments explaining code inline):

```
pilot_compute_description =
```

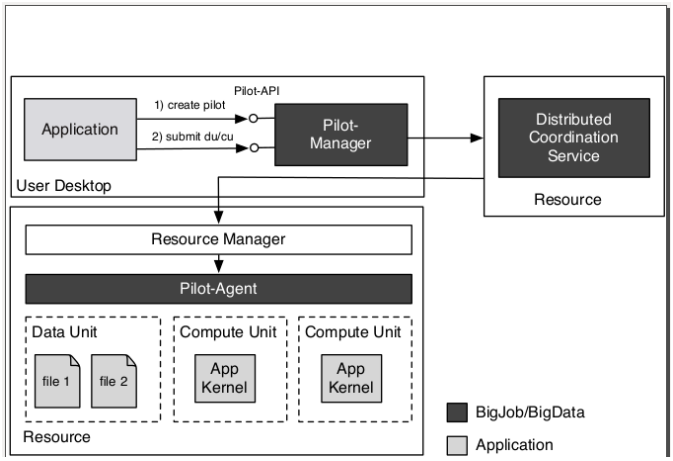

Fig. 2: BigJob Architecture: The Application is a python script that submits the container job to the scheduler and manages the whole workflow. The Pilot-Manager and Pilot-Agent are part of BigJob, which monitor and manage the jobs based on instructions from the Application. It is possible to manage both the compute and data parts of the workflow using BigJob. A Distributed Coordination Service (a database) is usually running on a separate resource and stores the job data in key/value pairs. (Source: https://github.com/saga-project/BigJob/wiki/BigJob-Architecture)

```
{ "service_url":
"xt5torque+gsissh://kraken.nics.xsede.org",
# specify queuing system of scheduler
"number_of_processes": 960,
#total number of cores requested
"allocation": "TG-123456",
"queue": "debug",
"working_directory": "/work/user/",
"walltime":120, #minutes
}
```

Job descriptions of serial jobs to be run once the scheduler allocates the resources. These can be specified in a for loop:

```
# submit compute units
for i in range(0, NUMBER_JOBS):
 compute_unit_description = {
"executable": "/work/user/mlRho",
"arguments":
[" -m "+ str(start) +" -M "+ str(end)+
" -n profileDb"],
"number_of_processes": 1,
#cores per executable
"spmd_variation":"single",
#MPI or serial
"working_directory":"/work/user/",
"output": "output"+str(i)+".out",
"error": "error"+str(i)+".err",
                }
```

It is beneficial for the end user to know and understand all these details, but it is not necessary. At the minimum, the user only needs to know what the software can do for them and

how to get the software to do it. The only interaction the user has with the BigJob software is via the python job submission script, which takes in similar details as a batch job submission script. A quick-start guide is available on the BigJob website [14].

B. Parallel Command Processor

The Parallel Command Processor (PCP) tool was first brought to our attention by the user support group at NICS. The original implementation of the tool was produced by the Ohio Supercomputer Center (OSC) [15] and the original version was ported by the NICS team to work on a Cray specific architecture [16]. The source code of PCP is available from NICS [17]. We have tested this code on multiple Cray machines and it has worked as expected.

The PCP binary expects a text file containing a list of commands to be run. We have used PCP to run hundreds of mlRho jobs concurrently. Basic scripting knowledge would be useful in creating text files with the jobs that need to be executed, however the barrier to entry for using PCP is very low compared to other similar tools.

On a Cray, the command "cc pcp.c" can be used to build the PCP binary. In the job submission script, hundreds of serial jobs can be run with this command:

```
aprun -n 512 ./pcp list.txt
```

where 512 is the number of processor cores and list.txt contains the list of jobs to be executed, for example, in the case of mlRho, 512 commands in total:

```
mlRho -m 1000 -M 1005 diatom.pro > out_1
mlRho -m 1006 -M 1010 diatom.pro > out_2
.
.
.
mlRho -m 2551 -M 2555 diatom.pro > out_511
mlRho -m 2556 -M 2560 diatom.pro > out_512
```

It should be noted that PCP has all of the advantages of a container job. We can adjust the number of serial jobs that are run in one group as needed. But that is where the similarities end, as we cannot control the execution workflow in more sophisticated ways with PCP.

C. aprun

Another approach to executing a large number of serial jobs is to generate a number of job submission scripts that each contain as many binary commands as there are cores in a single node. In order to conduct parameter sweeps, it is fairly straightforward to write a script that will generate the necessary job submission scripts for the range of parameters of interest. In the case of mlRho we wrote a python script to generate scripts of the form:

```
mlrho -m 1 -M 2500 input.pro > data1.out &
mlrho -m 2501 -M 5000 input.pro > data2.out &
.
.
.
mlrho -m 27501 -M 30000 input.pro > data12.out &
wait
```

where the number of lines in the script is equal to the number of cores per node. If this script is `run.sh` it can be invoked in the job submission script on an XT5 with 12 cores per node as:

```
aprun -n 1 -d 12 -cc none -a xt run.sh
```

which will run each of the 12 commands on 12 separate cores of a node.

IV. Experiments on Kraken

The purpose of these experiments is to satisfy two conditions: (i) as a proof of concept that these tools can be used to run hundreds of serial jobs on a Cray across multiple compute nodes, and (ii) an effort to check whether it is beneficial to bundle serial jobs in general into larger jobs to get better throughput. For these tests the metric of interest is total time to solution, that is, from the submission of the first job to the completion of the final job. We should note that the results are typically not useful to make broad generalizations, either with respect to Kraken or other large machines without a lot of qualifiers, and further studies are planned to support more general claims.

We ran the same workload with all three tools on Kraken and calculated the time to completion. Kraken is a 112,896 core Cray XT5 machine operated by NICS. Researchers from a wide variety of backgrounds use Kraken and a variety of queues are supported. We selected a job size of 960 cores, which is 80 compute nodes on Kraken. One instance of mlRho was run on each core and, in the runs where actual computations were done, ran 250 iterations on a zebra genome.

A. BigJob and PCP Experiments

With both BigJob and PCP, the processors are requested in a single job submission to the scheduler. Therefore, a single set of experiments suffice for BigJob and PCP. The experiments were repeated five times; however, because we had a limited number of core hours for testing on Kraken, the actual mlRho workload was only run on the first submission. The rest of the submissions were used to collect the queue wait times. We used PCP to submit the jobs and we believe that the run times would approximately be the same with BigJob too. We left a gap of multiple days between each job submission.

While the actual run time of the workload is about 4 hours, it took 52 hours on average for a job to complete. This is shown in Figure 3.

B. aprun Experiments

To compare the BigJob and PCP frameworks to a more basic job submission technique, we used a series of scripts following the template outlined in section III-C. We wrote a python script to generate 80 `run.sh` style scripts each of which had 12 mlRho commands, which each ran 250 iterations. The python script also generated 80 batch job submission scripts with an aprun line like that in section III-C. Finally, a master submission script that submitted all 80 of the batch job submission scripts at once was written.

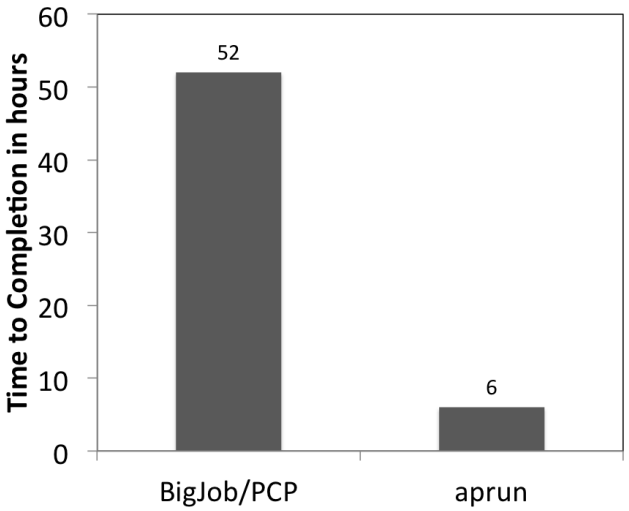

Fig. 3: For a BigJob/PCP style experiment, one job of size 80 nodes was submitted to the scheduler, while 80 separate jobs were submitted to the scheduler in the case of an aprun experiment on Kraken. BigJob/PCP jobs took 52 hours on average to complete, while the aprun jobs took 6 hours. The experiments were repeated five times.

The experiments were repeated five times; however, because we had a limited number of core hours for testing on Kraken, the actual mlRho workload was only run on the first submission. The assumption is that the run time of the mlRho program does not vary and is a constant offset. For the first run we measured the mlRho runtime to be about 4 hours. The other four sets of jobs were submitted to the queue but returned immediately once the jobs started running. The average queue wait time for the five sets of jobs was 2.2 hours. After adding in the four hour runtime, this gives an average overall time to completion of approximately six hours, as shown in Figure 3.

C. Analysis

The data from Figure 3 appears to suggest that submitting 80 separate single node job requests to the scheduler achieves better throughput than a single large job request of 80 nodes. However, it should be noted that this only applies to certain types of jobs. There are a variety of policies in place at different sites that may limit the overall throughput one can achieve with single job submissions. For example, NICS has a policy that allows a user to only have 25 simultaneously running jobs. This is shown in Table I as "run limit". Although the NICS website says that there is a run limit of 25 jobs per user on Kraken, it is unclear as to whether this policy is currently being enforced because we were able to run a total of 80 jobs in just over 10 hours of total wallclock time. However, if the policy were being enforced it would require a minimum of 16 hours to complete 80 jobs with an average runtime of 4 hours each.

Another factor that can potentially effect the total time to completion is the "queue limit", which is the maximum number of jobs a user can submit to the queue. On Kraken a user can have a total of 100 jobs in the queue at any given time. Had the parameter sweep we were modeling been much larger, we could easily have asked for 320 instead of 80 nodes with the BigJob/PCP as a single job request, but would not have been able to have more than 100 single node jobs in the queue simultaneously with aprun.

	Kraken	Hopper
Queued Limit	100	16
Run Limit	25	16

TABLE I: The tables shows queue limit, the maximum number of jobs a user can submit to the queue and run limit, the maximum number of jobs belonging to one user than can simultaneously be running in the queue. Hopper has a separate throughput queue, where the queued limit is 500 and run limit is 250, but a maximum of only 2 nodes can be requested per job in this queue.

Yet another factor that could be effecting the total time to completion reported in Figure 3 is backfilling. Backfilling is a scheduling feature used to maximize the resource utilization by running jobs out of order[18]. Typically, the backfilling algorithm attempts to find any unused nodes or "holes" in the schedule and fills them with appropriately sized jobs. Since these holes tend to be small, the backfilling algorithm generally benefits jobs that request fewer nodes. Another potential issue is that while there was a gap of multiple days between each BigJob/PCP type experiment and each of the five repetitions of aprun experiments, for a single set of aprun submissions, all 80 of the aprun jobs were submitted simultaneously. As four sets of these sets of jobs exited immediately so that we only measured the queue wait time, it is possible for several of the nodes to be reused for subsequent jobs in the set of 80. This in turn would result in a lower overall queue wait time for the entire set of jobs. To see whether our experiment was affected by this phenomenon, we recorded all the node numbers of the compute nodes that our jobs ran on. With the exception of one set of runs, the number of unique nodes used for the 80 jobs was in the 65 – 80 node range. One set ran on the same five nodes, however this set of runs did not have the smallest overall wait time, in fact, it had the third longest wait time in the set of five aprun submissions.

In the end, the user needs to consider various factors particular to the computational problem and local machine policies to drive the decision making. Other factors to consider include the priority that the scheduler assigns to jobs of various sizes, wall clock limits and any service unit discounts that may be available. For example, Hopper (NERSC) and Kraken (NICS) currently provide discounts to users submitting large jobs [19, 20]

V. Conclusion

As Cray machines become more widely available, the number of researchers using them to run non-traditional applications is on the rise. We believe that both Cray and non-traditional users are moving towards each other. Researchers are adapting to the machines that are the most widely available and can provide them with the largest number of compute

hours. Cray is providing the flexibility to run a variety of applications with the Cluster Compatibility Mode.

Although parametric sweeps are not new to the supercomputing field, they are a more recent phenomena on Cray supercomputers. Previous obstacles to running multiple binaries on the same compute node have now been overcome. Some challenges still remain, particularly in Extreme Scalability Mode, such as not being able to obtain a node list. But workarounds for these issues have been developed and deployed at many sites.

We discovered many solutions when we ran into the problem of running a parametric sweep application on Kraken. The brute force solution of just submitting separate single node job requests to the scheduler is straightforward and easy to implement. But the more elegant solutions of BigJob and PCP offer users the ability to submit much larger job requests, which can be advantageous, depending on a site's policies. Although the barriers to entry for using each of these tools are different, the user has to make an informed decision when choosing one of these tools. The decision will have to depend on many factors specific to the application, machine, scheduler policies and ease of use.

ACKNOWLEDGMENTS

We would like to thank the SAGA-BigJob team at Rutgers University for their help with BigJob on Kraken. We would like to thank the user support team at NICS for providing the source code and instructions for using PCP. This research was enabled by IU's advanced cyberinfrastructure, including the Big Red II supercomputer and the Data Capacitor II storage system, the implementation of which has been supported by the Lilly Endowment through their support for the IU Pervasive Technology Institute and the Indiana Metacyt initiative. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

REFERENCES

[1] "National Institute for Computational Sciences (NICS) at the University of Tennessee ," http://www.nics.tennessee.edu/.

[2] "National Energy Research Scientific Computing Center (NERSC)," http://www.nersc.gov/.

[3] "Big Red II at Indiana University," http://kb.iu.edu/data/bcqt.html.

[4] B. Haubold, P. Pfaffelhuber, and M. Lynch, "mlrho a program for estimating the population mutation and recombination rates from shotgun-sequenced diploid genomes," *Molecular Ecology*, vol. 19, pp. 277–284, 2010. [Online]. Available: http://dx.doi.org/10.1111/j.1365-294X.2009.04482.x

[5] "The Extreme Science and Engineering Discovery Environment (XSEDE)," https://www.xsede.org/.

[6] R. S. Canon, L. Ramakrishnan, and J. Srinivasan, "My Cray can do that? Supporting Diverse Workloads on the Cray XE-6," *Cray User Group*, 2012. [Online]. Available: https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap157.pdf

[7] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.05.005

[8] "The Research in Advanced Distributed Cyberinfrastructure and Applications Laboratory (RADICAL)," http://radical.rutgers.edu/.

[9] "SAGA BigJob," https://github.com/saga-project/BigJob.

[10] 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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CCGRID.2010.91

[11] A. Luckow, S. Jha, J. Kim, A. Merzky, and B. Schnor, "Adaptive Replica-Exchange Simulations," *Royal Society Philosophical Transactions A*, 2009.

[12] A. Thota, A. Luckow, and S. Jha, "Efficient large-scale replica-exchange simulations on production infrastructure," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 369, no. 1949, pp. 3318–3335, 2011. [Online]. Available: http://rsta.royalsocietypublishing.org/content/369/1949/3318.abstract

[13] "PyPI - the Python Package Index," https://pypi.python.org/pypi.

[14] "BigJob Tutorial," https://github.com/saga-project/BigJob/wiki/BigJob-Tutorial-Part-3:--Simple-Ensemble-Example.

[15] "Ohio Supercomputing Center," https://osc.edu/.

[16] "OSC and NICS Utilities," http://www.nics.tennessee.edu/~troy/pbstools/.

[17] "PCP (parallel-command-processor) Source Code," http://svn.nics.tennessee.edu/repos/pbstools/trunk/src/parallel-command-processor.c.

[18] "Kraken Scheduling Policy," http://www.nics.tennessee.edu/computing-resources/kraken/running-jobs#scheduling-policy.

[19] "Kraken Capability Jobs Discount Policy," http://www.nics.tennessee.edu/computing-resources/kraken/running-jobs#queues.

[20] "Hopper Queue Charge Factor," http://www.nersc.gov/users/computational-systems/hopper/running-jobs/queues-and-policies/.