XSEDE New Research Allocation Proposal:

Harnessing cloud computing and genomics to

facilitate adaptive species conservation of

non-model organisms

Code Performance and Benchmarking

Principal Investigator: Marlis R. Douglas

Co-Investigators: Michael E. Douglas

Contributor: Tyler K. Chafin

*University of Arkansas*

*Fayetteville, Arkansas*

In the Main Document for this XRAC Renewal request, Section 3.1 we list expected walltime and hardware (e.g. memory) limitations on our work, and in Section 3.2 detail how these cumulatively contribute to our requested SU usage across projects. We here 1) Expand on code enhancements made by our lab group to minimize SU consumption where possible and necessary, and 2) Further detail our benchmarking of several codes, and discuss how these codes scale.

1. **Optimizing MCMC-based code efficiency**

**Many of the application we use require long run times** due to the intrinsically sequential nature of the algorithm on which they are based, **Markov Chain Monte Carlo (MCMC)**. MCMC is used in Bayesian methods to explore the parameter landscape in order to approximate a posterior probability distribution (including tree topology). This is accomplished via randomly drawing parameter values from a prior distribution and assessing the effect of this change on the overall likelihood of the model - which, using some threshold, is used to either accept or reject a change. The expectation is that the Markov Chain should most densely ‘sample’ parameter space where likelihoods are maximized - thus by sampling the MCMC state, the expectation of a statistic under some complex model can be explored.

**One variation is the Metropolis-Coupled MCMC**, in which multiple Markov Chains have successively less stringent acceptance thresholds (‘heated chains’), more freely explore parameter space, and are less susceptible to ‘local optima’. In this case, at some exchange rate, proposed swaps between the primary (‘cold’) chain and the heated chains allow the cold chain to swap states if a ‘peak’ in parameter space is found that optimizes the model likelihood to a greater extent.

**We have created a parallel scalable version of one of these codes which has particularly exhorbitant runtimes, BUCKy**. For our ‘MPI-BUCKy’ (located at github.com/tkchafin/mpi-bucky), we use the **OpenMPI Message Passing Interface to parallelize the Metropolis-Coupled MCMC** **algorithm**. The program spreads Markov Chain state simulations across multiple threads, and uses MPI-facilitated message passing to communicate states between chains during proposed swaps. To minimize communication cost, rather than swapping chain ‘states’, ranks of these chains are swapped instead - thus which chain is ‘heated’ vs ‘cold’ changes - without necessitating the sending and receiving of all state information such as sampled topologies or parameter values.

The **relative speedup of this implementation** is shown in Table 1; fully linear speedup is not achieved due to the communication cost added by the MPI.

|  |  |
| --- | --- |
| ***n* threads** | **Relative speedup** |
| 2 | 2X |
| 4 | 3.5X |
| 8 | 6.5X |

**Table 1:** Relative speedup of the OpenMPI BUCKy version. Tested with 8 chains, and 1 million generations per chain.

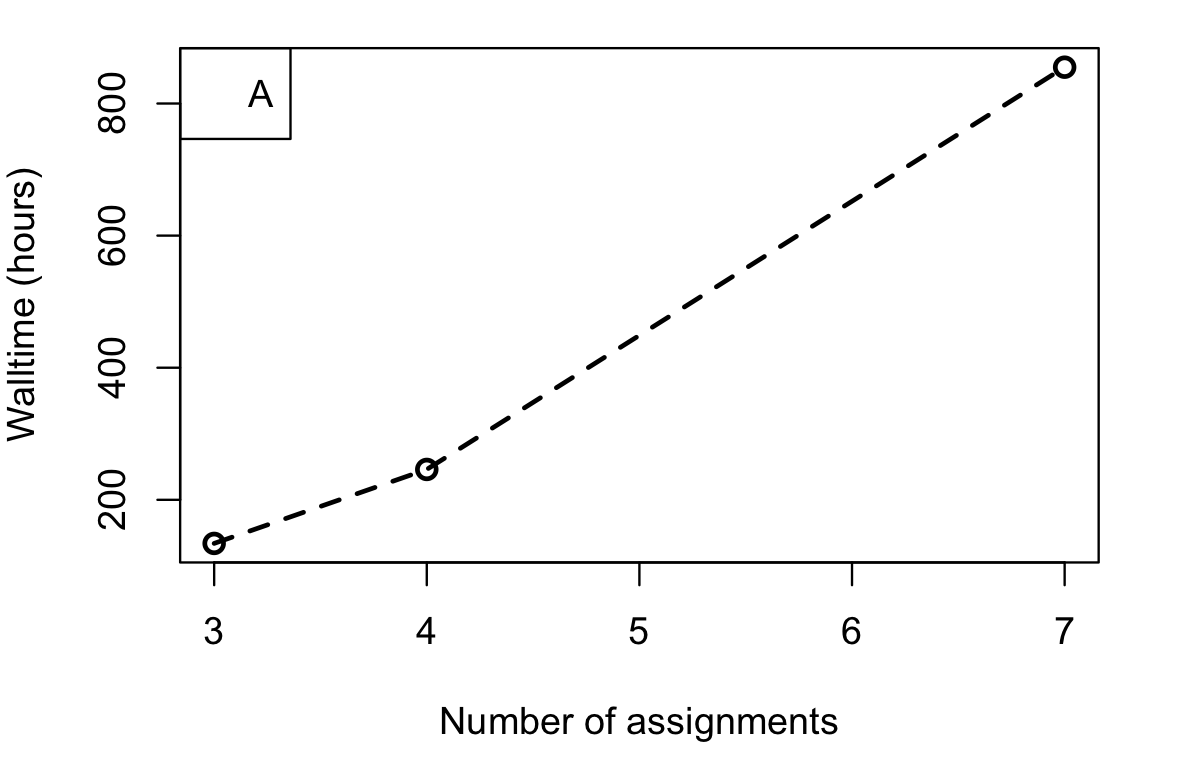
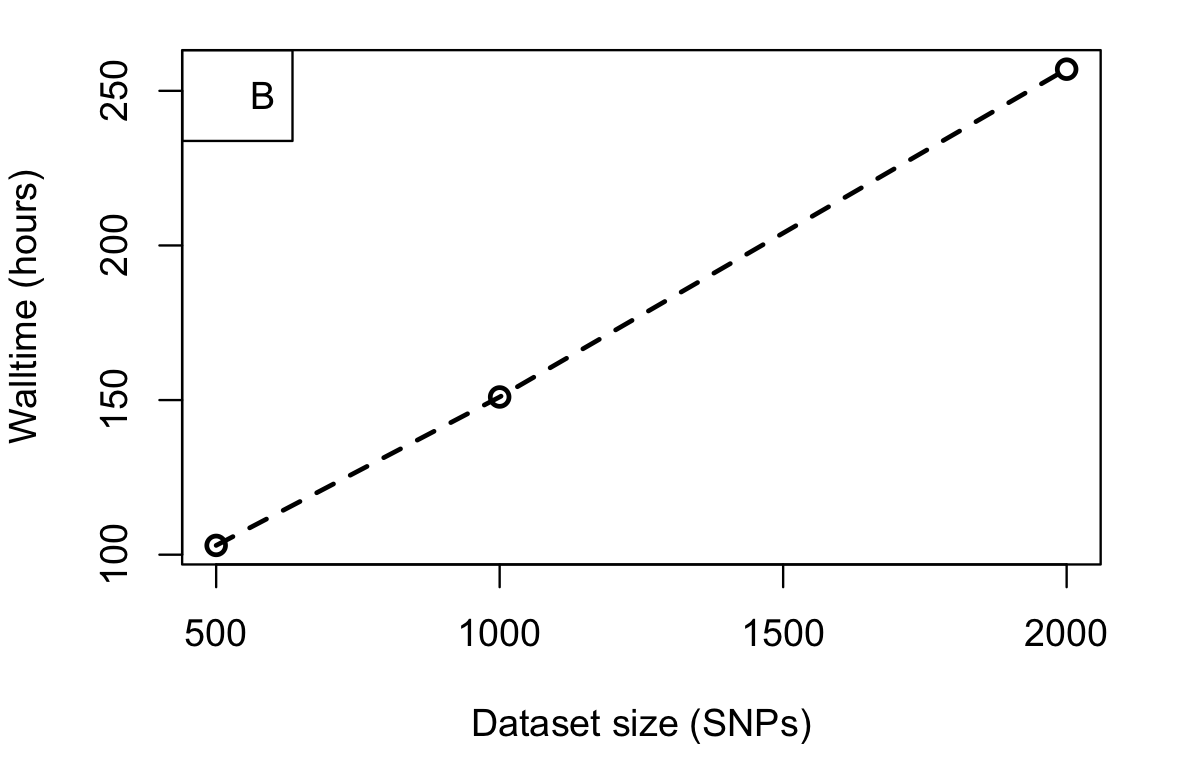
**This MPI implementation will allow us to more efficiently use allocated XSEDE resources**, **but requires substantially more memory**, as each thread must track the various parameter distributions, and contain local copies of all data structures. We understand the implications of receiving access to public HPC resources such as Jetstream and plan to try implementing similar ‘homegrown’ scaling schemes where possible and practical in order to use our allocation in the most efficient way possible.

1. **Benchmarking biological software**

We here provide detail benchmarking and discussion of scaling for many of the analyses we plan to use to accomplish the objectives listed in Section 2 of the main document, and in support of the total allocation requested per objective, as listed in Section 3.2 of the main document.

*2.1 Bayesian species delimitation by model comparison (supporting Objective 2.1)*

**Model based species delimitation requires multi-core, high memory resources, as well as substantially long run times**. Additionally, run time varies by complexity of the model, size of the dataset, and specification of hyperpriors (see Figure 1). Tracking memory usage empirically, memory requirement *per thread* varied from 8.9 to 10.1 GB. Running these models requires large memory resources such as the ‘XXL’ instance size offered by Jetstream (44 cores, 140 GB memory). **Figure 1A shows walltime usage for analysis of single models of varying complexity** - a standard delimitation analysis by BFD may run between 5 and 20 different delimitation models, each with a walltime in this range. In addition, numerous test runs are necessary to evaluate the effect of specification of priors - goal in any complex model is to avoid wasted computational effort by either sampling for unneeded hyperpriors, while also not biasing the analysis with miss-specified priors. In particular, **we are concerned about proper specification of the birthrate for the Yule process used by SNAPP (lambda) and plan to perform extensive testing to this end.**



**Figure 1:** Runtimes for BFD (Bayes Factor Delimitation) by A) number of species assignments (model complexity) and B) size of dataset (numbers of individuals and loci).

We have additionally investigated how the program scales as we change parameters and aspects of dataset size, and have come to the general conclusions that: 1) BFD\* runtime scales exponentially as the number of taxa partitions increases (Fig 1A); 2) BFD\* scales linearly with dataset total size (SNPs, individuals; Fig 1B); 3) BFD\* takes approximately half the runtime when using a fixed value for lineage birth rate as compared to sampling values from a hyperprior (Table 2); 4) Runtimes scale linearly as MCMC and path sampling lengths increase (Table 2).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Model | N Groups | N Taxa | N sites | MCMC | HyperprioRs | Walltime (h) |
| bhs\_m1 | 2 | 53 | 1742 | 600K/600K |  | 35.6 |
| bhs\_m1\_lamHy | 2 | 53 | 1742 | 600K/600K | lambda | 74.5 |
| bhs\_m2 | 3 | 53 | 1742 | 600K/600K |  | 101 |
| bhs\_m2\_lamHy | 3 | 53 | 1742 | 600K/600K | lambda | 144 |
| bhs\_m3 | 3 | 53 | 1742 | 600K/600K |  | 125 |
| bhs+m3\_lamHy | 3 | 53 | 1742 | 600K/600K | lambda | 230 |
| bhs\_m5 | 4 | 53 | 1742 | 600K/600K |  | 162 |
| bhs\_m6 | 4 | 53 | 1742 | 600K/600K |  | 297 |
| bhs\_m9 | 5 | 53 | 1742 | 600K/600K |  | 606 |
| gila\_mE | 5 | 43 | 500 | 600K/600K |  | 161 |
| gila\_mH | 4 | 43 | 500 | 600K/600K |  | 63.6 |
| gila\_mJ | 3 | 43 | 500 | 600K/600K |  | 33 |

**Table 2:** Runtimes across many BFD runs showing effect of N groups and hyperprior sampling

*2.2 Runtimes of phylogenetics packages (Objective 2.1 and 2.2)*

**Other Bayesian analyses** (BUCKy, BGC, many phylogenetics packages) needed to conduct the research outlined in Section 2 also utilize an inherently sequential MCMC approach, and **similarly will require the high memory and long walltimes** available on Jetstream. Memory requirements for one phylogenetics package, BUCKy, can be seen in Table 3. Timing estimates were calculated using an unrealistically short run time, and a small subset of genes compared to a final run. However walltime will scale approximately linearly with MCMC chain generation. We projected runtimes and memory usage assuming 10 million MCMC generations and 500 loci, with an approximate linear scaling as dataset size and chain length increase - an supposition congruent with the theoretical growth of the algorithm employed by our MPI implementation of BUCKy. In Table 4, we list runtimes for many runs of various phylogenetics packages, standardizes as much as possible, for MCMC lengths generally appropriate to achieve convergence (200K – 500K iterations) across RADseq datasets varying from 10,000 – 20,000 loci and 200-1000 individuals. Note that translation to estimates SU cost per analysis in Section 3.1 of the Main Document takes into account parallelization capability of each respective analysis (e.g. some use MPI, some ‘embarrassingly parallel’ using GNU Parallel, and some obligately sequential), as well as estimated memory usage forcing ‘larger’ (=higher SU cost per hour) VM sizes. Most analyses demand Jetstream XL or XXL instances due to memory requirements associated with genomic data.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of loci | Chains per thread | Approximate memory usage per thread (GB) | Projected memory usage per thread (GB) | Runtime  (short run; s) | Runtime  (projected; h) |
| 100 | 1 | 31 | 155 | 678 | 188 |
| 100 | 2 | 15 | 75 | 1478 | 410 |
| 100 | 4 | 7 | 35 | 2399 | 666 |
| 100 | 8 | 4 | 20 | 4670 | 1297 |

**Table 3:** Observed and predicted resource use of BUCKy. Observed memory usage and walltime were estimated using 100 loci and 1 x 104 MCMC generations, and projected assuming a more realistic analysis of 500 loci and 1 x 107 generations.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program** | **Method** | **Parallel?** | **Average observed runtime** | **Standard deviation** | **Maximum observed** |
| ASTRAL | Maximum likelihood | partial | ~30 days | not enough data | ~30 days |
| BEAST | Bayesian MC3 | partial | 7 days | 2 days | 12 days |
| ExaML | Maximum likelihood | yes | ~30 days | not enough data | ~30 days |
| ExaBayes | Bayesian | yes | ~30 days | not enough data | ~30 days |
| D-stats | N/A | no | 4 hours | 1 hour | 8 hours |
| IQTree (PM) | Maximum likelihood | partial | 14 days | 10 days | 45 days |
| RAxML | Maximum likelihood | bootstraps | 5 days | 3 days | 14 days |
| SNAPP | Bayesian | partial | >20 days | not enough data | >60 days |
| SVDQuartets | Quartet inference | bootstraps | 8 days | 7 days | 35 days |

**Table 4:** Benchmarking of phylogenetics packages. Note that some could not be run long enough for mean/sd calculations

*2.3 Bayesian assignment tests of population structure (Objectives 2.3 and 2.4)*

For estimating genetic structure of wild populations (see Section 2) using the program STRUCTURE (Pritchard *et al.* 2000), we have observed an **approximately exponential increase in total walltime as *K* number of clusters** (i.e., gene pools) in the model is increased (see Figure 2). For estimating timing, we used a dataset of 50 individuals and >10,000 loci. To capture variation among runs in practice, often >10 replicates of each *k* value are performed. These are sequential, single CPU algorithms and thus **are expected to consume 1 SU per vCPU hour per model.**

**We accomplish parallelization by running STRUCTURE runs in parallel via GNU Parallel**, and manually setting the seed for the random number generator for each independent run. Note that runtimes are also estimated to increase linearly with dataset size (not shown).

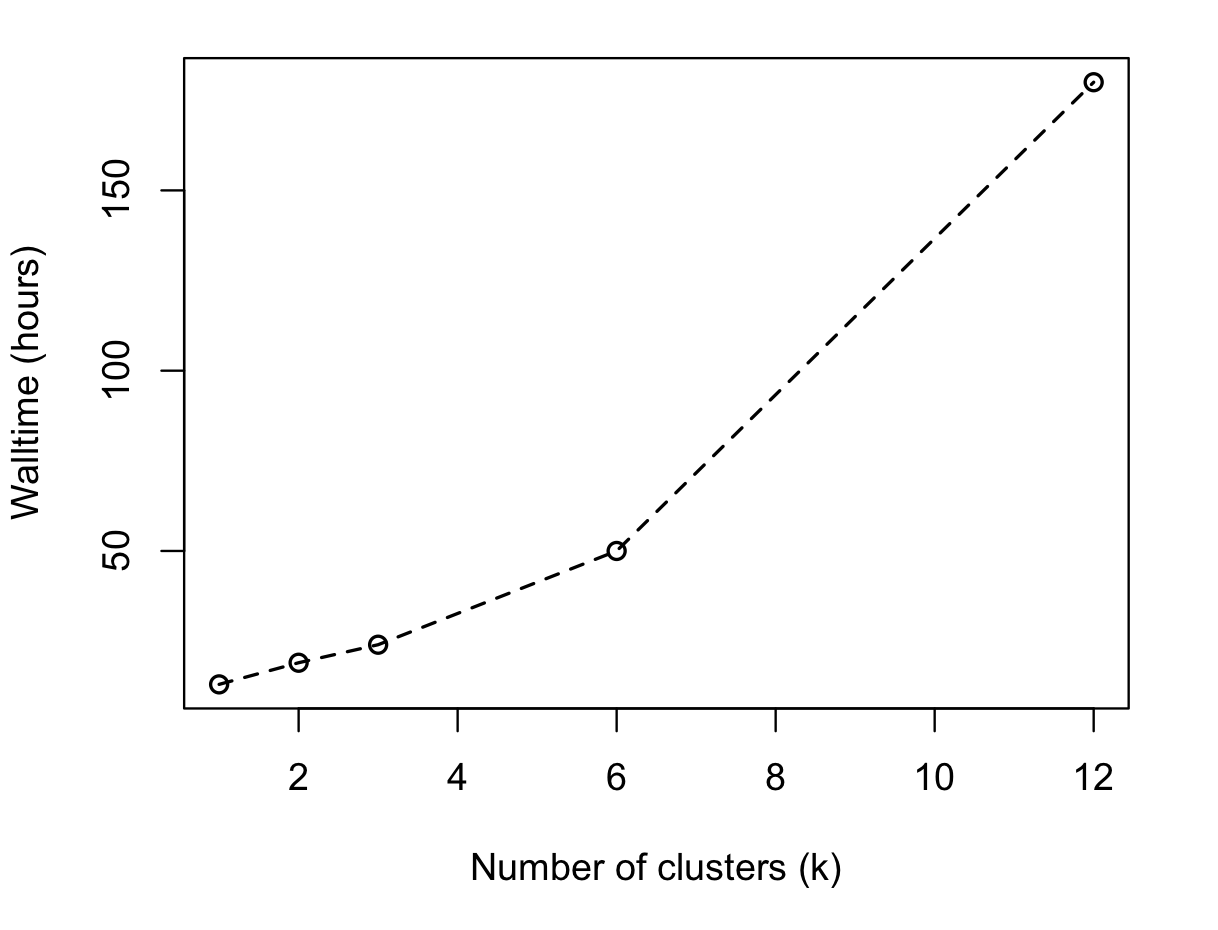


Figure 2: Walltime for STRUCTURE runs as *k* value (number of gene pools) increases.

*2.4 Bayesian modelling of contemporary migration rates (Objective 2.5)*

A critical step in establishing units within species for conservation prioritization is diagnosing ‘demographically independence’ among units, or essentially establishing populations for which the majority of recruitment within generations comes from within-population reproduction, or migration. This allows us to understand contemporary levels of dispersal of individuals across a species distribution, which we in turn translate to ‘Management Units’. An example of the utility of this distinction is in helping fisheries managers determine which hatchery broodstocks are genetically equivalent to populations within a species. **This is important because artificially mixing lines that are locally adapted to different conditions can lead to ‘outbreeding depression’, or lower fitness in the population**.

Because populations within species tend to exhibit allele frequency differences, and not fixed diagnostic differences, this is not easily accomplished simply by ‘genetic scoring’ at certain loci. Instead, our approach is to scan much variation throughout the genome and employ Bayesian models that consider both variation among population, and genotyping error, to assign migrant individuals probabilistically. For example Population A might have 70% frequency of ‘A’ at one SNP locus, and 80% G at another SNP locus, while Population B might have 30% ‘A’ and 25% G, respectively. An individual with genotype ‘A/G’ at these loci is more likely to originate from population A. In considering tens of thousands of polymorphic sites, while computationally expensive, allows us to assign individuals on the basis of this simple scheme. To accomplish this, we use the program BayesAss (Wilson and Rannala, 2003), which has been modified by lab member Steve Mussmann to be slightly more efficient, and to accommodate the large datasets we use with RADseq. We have estimated an average per thread memory usage of >5GB, meaning we expect to be able to run 20-24 runs per Jetstream XXL instance, thus 24 pairwise-population comparisons should take an average of 200 hours on one XXL instance (=8,800 SUs). We have found the program to scale linearly with dataset size (Figure 3).



Figure 3: Walltime in hours across 615 BayesAss runs using our 2017 XRAC Allocation on Jetstream. Note that “Individuals per group” does not scale as predictably as “Number of Loci”- likely a consequence of runtime by actual variation among the groups compared rather than simple number of individuals within those populations.

*2.6 De novo population genomics* *(Objective 2.6)*

A variety of software will be employed, many of which we have taken advantage of our 2016 XRAC Allocation to perform benchmarking and trial runs for our *Nippotaenia* genome project (see Main Document Section 2.6). We summarize various aspects of this pipeline in Table 5 below, with observed walltime and memory requirements for our exploratory analyses.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Program | Memory | Walltime (h) | SU cost per run |
| Quality trimming | BBDuk | 81GB | 0.5 | 22 |
| QC and Normalization | BBNorm | 43GB | 0.5 | 22 |
| Remove contaminants | Bbmap | 15.6GB | 0.5 | 22 |
| Assembly | SOAPdenovo | 96GB | 3 | 132 |
| Assembly | Ray | 7GB | 1 | 44 |
| Assembly | SPAdes | >100GB | 400 | 17,600 |
| ID redundant contigs | Redundans | 32.9GB | 18 | 792 |
| Scaffolding | LINKS | >100GB | 12 | 528 |
| Polishing | Pilon | 30.8GB | 4 | 176 |
| Quality assessment | Quast | 3.4GB | 4 | 176 |
| Population analysis | Popoolation2 | 30.1GB | 35 | 1540 |
| Initial gene models | MAKER | <10GB | 12 | 528 |
| ab initio gene models | MAKER | <10GB | 36 | 1584 |

Table 4: Observed runtimes for steps on our de novo assembly and analysis pipeline. Note that for final steps, we plan to investigate the new MAKER implementation WQ-MAKER available on Jetstream, which will shorten runtimes but increase SU cost per hour as it scales across multiple nodes.