Modular Genomic Variant Calling Workflow in Swift/T

Jacob Heldenbrand*, Azza Ahmed*, Yan Asmann, Faisal M. Fadlelmola, Daniel Katz, Katherine Kendig, Matthew C. Kendzior, Tiffany Li, Yingxue Ren, Elliott Rodriguez, Matthew R. Weber, Jennie Zermeno, Liudmila S. Mainzer

*These authors contributed equally to the project

NCSA Genomics, University of Illinois at Urbana-Champaign

Abstract

Genomic variant discovery is widely performed using the GATK's Variant Calling Best Practices pipeline, a complex workflow with multiple steps, fans/merges, and conditionals. Managing the workflow can be difficult on a computer cluster, especially when running in parallel on large batches of data. One potential solution is monolithic implementations that replace the multi-stage workflow with a single executable. While such implementations exist, they may not be sufficiently flexible to accommodate nuances of analysis particular to different species, types of sequencing, and research objectives. Here, we present a scalable GATK-based variant calling workflow written in the Swift/T parallel scripting language. Key built-in features include the flexibility to split by chromosome before variant calling, the option to continue the analysis when faulty samples are detected, and the ability to analyze multiple samples in parallel within each node. With its modular design, execution can easily be separated into multiple stages that request the resources optimal for each portion of the pipeline. Swift/T's ability to operate in multiple cluster scheduling environments (OGE, PBS Torque, SLURM, etc.) enables a workflow to be trivially portable across numerous clusters. With these features, users have an efficient and portable way to scale up their variant calling analyses to run in many traditional HPC architectures.

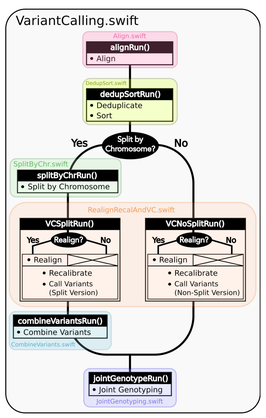
https://github.com/ncsa/Swift-T-Variant-Calling
http://swift-t-variant-calling.readthedocs.io/en/latest/

Key Design Principles

- Modularity – Independent sections linked together
- Option to split by chromosome
- Extensible function definitions
- Logical output structure
- Implicit parallelism driven by dataflow
- Compatible with many job schedulers (PBS Torque, SLURM, Cray, etc.)
- Real-time Job Monitoring

Modularity

Our Swift/T [1] implementation is comprised of a series of independent modules ("stages") that are chained together by the primary workflow script. At each stage, the user can set the workflow to generate the output files necessary for the next stage, or simply pass on the output generated from a previous run. With this architecture, users may restart the workflow at a failed stage without needing to recompute successful upstream calculations, or run a portion of the workflow, requesting only the resources optimal for each particular stage.



Modularity also ensures that the implementation of individual stages may be altered without breaking the workflow, as long as inputs and outputs remain consistent. Thus the workflow can be updated with new methodologies and tools as the field progresses. At the end of each stage, there is an implicit wait instruction that ensures all samples have completed that stage before each batch begins the next stage.

Step	Program Options
Alignment	BWA MEM or Novoalign
Sorting	Novosort
Marking Duplicates	Samblaster, Novosort or Picard
Indel Realignment	GATK
Base Recalibration	
Variant Calling	
Joint Genotyping	
Miscellaneous	Samtools

Table 1:
Tools available for the user to choose from in our implementation, for each stage of the workflow.

Extensible function definitions

Our implementation makes multiple tool choices available to the analyst at each stage of the workflow (Table 1). Although Indel realignment is not necessary past GATK version 3.6, it is included as an optional step to comply with legacy analyses, and to simplify the future introduction of other variant callers (i.e. UnifiedGenotyper, Samtools, or Platypus) that may require realignment. Additionally, the user is given the option to split aligned reads by chromosome before calling variants, to speed up analysis.

Extensible Design

The workflow was designed to be easily extensible as long as function inputs and outputs remain consistent. The choice of program to use for a given step is made in a function defined outside of the main logic of the workflow.

```
/* COMMAND-LINE PROGRAM CALLS */
@dispatch=WORKER
app (file bam) bwa_mem(X,Y,Z) { bwa mem X Y Z; }

@dispatch=WORKER
app (file bam) novoalign(X,Y,Z) { novoalign X Y Z; }

/* WRAPPER FOR ALIGNMENT PROGRAMS */
(file bam) performAlignment(string toolChoice, X,Y,Z) {
    if (toolChoice == BWAMEM) { bwa_mem(X,Y,Z); }
    /* potential for expansion */
    else { novoalign(X,Y,Z); }
}

/* WORKFLOW CODE: ALIGNMENT MODULE */
/*  ... prepare for alignment ... */
file outputBam = performAlignment(X,Y,Z);
/*  ... continue processing ... */
```

Real-time Job Monitoring

When analyzing many samples at once, especially in a production environment where the data flows continuously through the cluster, it pays to have a good system for logging and monitoring progress of the jobs. At any moment in time the analyst should be able to assess:

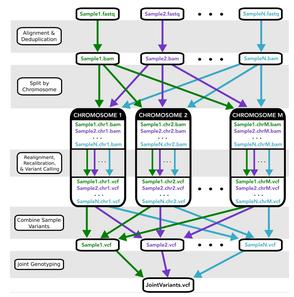
- Which stage of the workflow is running for every sample batch
- Which samples may have failed and why
- Which nodes the analyses are running on, and their health status.

The underlying MPI-based implementation of Swift/T logic makes it possible to leverage standard MPI logging libraries to collect such details. We used the Message Passing Environment (MPE) library [2] to log the usage of the MPI library itself and ADLB calls [3], and implemented visualization in Jumpshot viewer. To enable such logging requires installation of the MPE library in addition to the standard Swift/T components (C-utils, ADLB library, Turbine and STC).

Another approach to tracking the workflow run time execution is to manually implement Swift/T leaf functions such that the start and end timing of each function are logged and a timing graph is generated, showing the analysis steps across samples, chromosomes and specific applications. This approach permits one to view the patterns of pipeline execution even if it fails, and partial logs can similarly be viewed as the pipeline is running.

Split by Chromosome

The user is given the option to split aligned reads by chromosome before calling variants, as this is more efficient when analyzing WGS samples. However, when analyzing WES samples, the overhead of splitting by chromosome often outweighs the performance gain through increased parallelism.



Scheduling and Parallelization

Swift/T runs as an MPI program that uses Turbine [4] and Asynchronous Dynamic Load Balancing (ADLB) [2] libraries to manage and distribute the workflow execution on local compute resources (desktop/laptop), parallel computers (clusters/HPCs), and distributed systems (grid/cloud). Its built-in wrappers can launch jobs on many common resource schedulers, such as PBS Torque, Cobalt, Cray/APRUN, and SLURM [5].

The dataflow programming model of Swift/T implicitly allows for parallel execution of tasks. Statements are evaluated in parallel unless prohibited by a data dependency or resource constraints, without the user needing to explicitly code parallelism or synchronization. Within the variant calling workflow, implicit parallelism ensures that the number of samples processed in parallel is constrained only by the resources requested at runtime.

Dataflow Model

In most cases, workflow functions are pure, i.e. they have no side effects, such as modifying variables outside the function's scope. However, if a function has a side effect, one must add an explicit wait signal. This code excerpt presents an example where an explicit wait command is necessary.

```
/* FUNCTION DEFINITIONS */
(string bam) performAlignment() { ... }
(string sortedBam) performSort(string input) { ... }
(void) createIndex(A,B) { ... }
(file outFile) performDedup() { ... }

/* IMPLEMENTATION EXAMPLES */

/* Purely functional style */
file bam = performAlignment( A, B, C);
// This command will automatically wait because
//   of the data dependency
file sortedBam = performSort(bam);

/* Non-functional style */
void signal = createIndex(sortedBam);
/* Must have an explicit wait command
 *   (avoid writing in this style when possible)
 */
wait (signal) {
   (file outFile) performDedup(sortedBam) { ... }
}
```

Testing

We successfully tested our workflow on a variety of HPC systems with a range of job schedulers and test datasets (Table 2). Swift/T does deliver on its promise of portability and parallelization.

Table 2: Testing Information

	Resource Manager	Node Type	# Nodes/Run	Node Sharing	Test Data
iForge	PBS Torque	IvyBridge 20 cores 256 GB RAM	1;12	No	Soy NAM
Blue Waters	PBS Torque	AMD Bulldozer 32 integer cores 64 GB RAM	1;101	No	Synthetic chr 1 exome seq 50X ADSP WES 100 samples
XSEDE Stampede2	Slurm	KNL 68 cores 4 hardware threads/core 96 GB DDR4 16 GB MCDRAM	1	Yes	NA12878 sample, (GIAB)
Biocluster2	Slurm	Dell PowerEdge R620 24 cores 384 GB RAM	1;3	Yes	Synthetic WES 30X Synthetic WES 50X Synthetic WES 70X
Single server at CBSB. H3A Africa	N/A	HP Proliant dl380p gen 8 24 cores 125 GRAM	1	Yes	Synthetic chr1 exome seq 50X

Pros and Cons of Swift/T

- The greatest strength of Swift/T may be its portability: a workflow written in the language can be executed on a wide variety of compute infrastructures without changing the code, and the user does not need to know about the underlying scheduling environment on the cluster.
- While the implicit parallelism of Swift/T can increase the amount of simultaneous computation, it also increases the difficulty of debugging during development.
- The greatest drawback of Swift/T may be its inability to automatically shift work from one node to another after encountering a hardware failure. Because hardware failures become more likely as the number of nodes increases, this lack of resilience limits the scale of analysis that can be reliably performed with a Swift/T workflow.
- Swift/T has native support for restarting failing functions for any reason. The restarts will happen on the same node, because Swift/T is not hardware-failure resilient. However, this is useful when applications fail for nondeterministic reasons.
- The dataflow task parallelism framework has a substantial learning curve, although it offers familiar control flow statements and expressions in C-like syntax [6]. Interestingly, Swift/T does not support piping between applications, thus we must code each step individually.
- Swift/T abstracts away low-level concerns such as load balancing, inter-process communication and synchronization of tasks automatically through its compiler (stc) and runtime engine (Turbine), allowing the programmer to focus on the workflow design [7].

Conclusion

Swift/T language lends itself to creating highly portable, modular and implicitly parallel workflows. It is very powerful, especially when a workflow consists of raw code pieces written in C, C++, Fortran, etc. However, it may be overkill in bioinformatics, where workflows consist of pre-compiled executables glued together. Portability, the main advantage of Swift/T, could perhaps be accomplished in simpler ways. The lack of support for piping between applications is a major drawback for big-data bioinformatics, resulting in a proliferation of intermediary files.

References

1. Wozniak JM, Armstrong TG, Wilde M, Katz DS, Lusk E, Foster IT. Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. IEEE; 2013. p. 95–102. doi:10.1109/CCGrid.2013.99.
2. Lusk EL, Pieper SC, Butler RM. More scalability, less pain: A simple programming model and its implementation for extreme computing. SciDAC Review. 2010.
3. Wozniak JM, Chan A, Armstrong TG, et al. A model for tracing and debugging large-scale task-parallel programs with MPE. Proc LASH-C at .... 2013.
4. Wozniak JM, Armstrong TG, Maheshwari K, et al. Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications. Proceedings of the 1st .... 2012.
5. Wozniak JM. Swift/T Sites Guide. Swift/T Sites Guide. http://swift-lang.github.io/swift- t/sites.html. Accessed 20 Aug 2017.
6. Wozniak JM, Wilde M, Foster IT. Language Features for Scalable Distributed-Memory Dataflow Computing. In: Data-flow Execution Models for Extreme-scale Computing. 2014.
7. Armstrong TG, Wozniak JM, Wilde M, Foster IT. Compiler techniques for massively scalable implicit task parallelism. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE; 2014. p. 299–310. doi:10.1109/SC.2014.30.

Acknowledgements

We are grateful for the support of the Blue Waters team, NCSA Industry, and the Argonne/U. Chicago Swift/T developer team during the implementation, testing and scalability efforts in this project. Special thanks to Justin Wozniak.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the State of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

AA, FMF are H3ABioNet members supported by the National Institutes of Health Common Fund under grant number U41HG006941