Adapting Bioinformatics Applications for Heterogeneous Systems: A Case Study

Irena Lanc
University of Notre Dame

At Present

- Growth in size of biological datasets driving the need for greater processing power
- Greater numbers of research facilities relying on clouds and grids
- Bioinformatics software incorporates MPI or MapReduce

  → leverages multi-core and distributed computing resources

Motivation

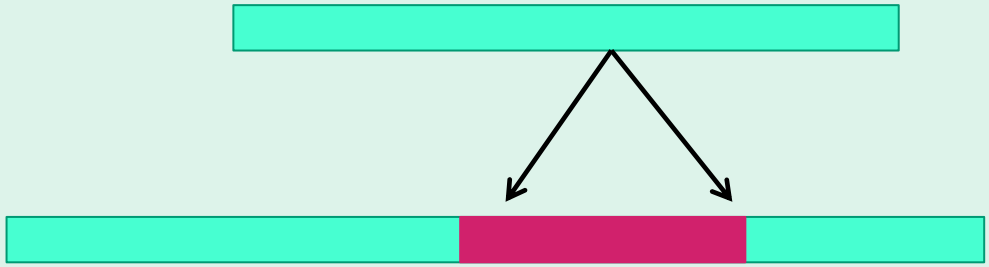
- Proliferation of small-scale, specialized bioinformatics programs, designed with particular project or even data set in mind
- Programs often serial, or tied to a particular distributed system
- Burdens end users

The Case of PEMer

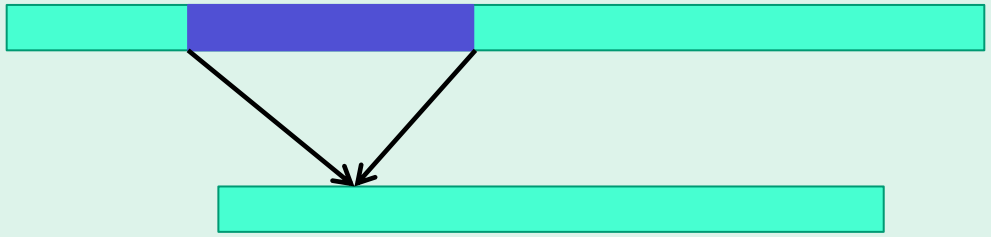
- PEMer is a structural variation (SV) detection pipeline, written in Python
- SVs including indels, inversions, and duplications, are an important contributor to genetic variation
- PEMer provides a 5-step workflow to extract structural variation from given data gene sequence

Korbel J, Abyzov A, Mu XJ, Carriero N, Cayting P, Zhang Z, Snyder M, Gerstein M: PEMer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome Biology* 2009, 10:R23.

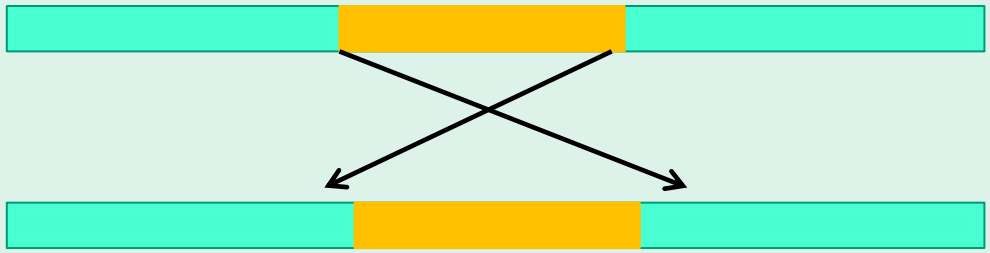
A Brief Introduction to Structural Variations



Insertion – Addition of DNA into gene sequence

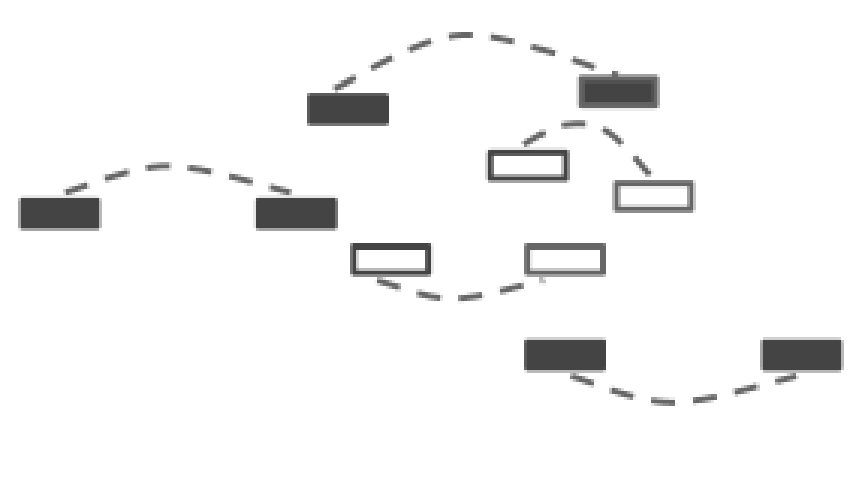


Deletion – Removal of DNA from sequence

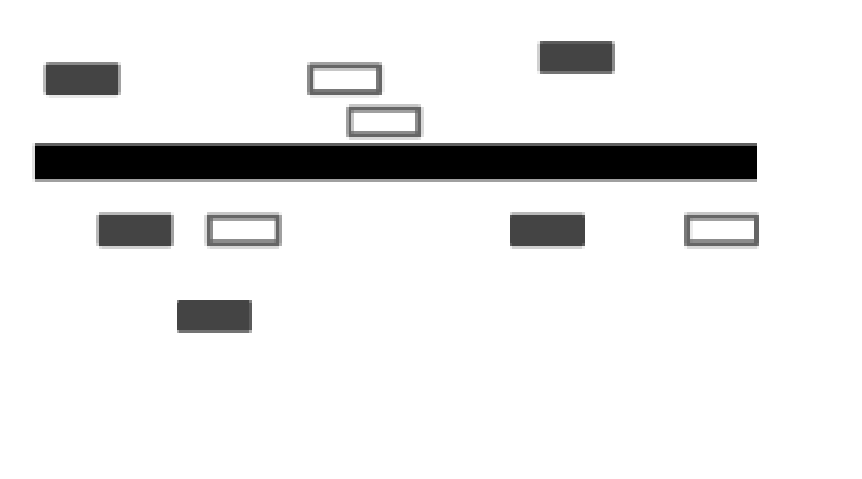


Inversion – reversal of portion of DNA

The PEMer SV Pipeline

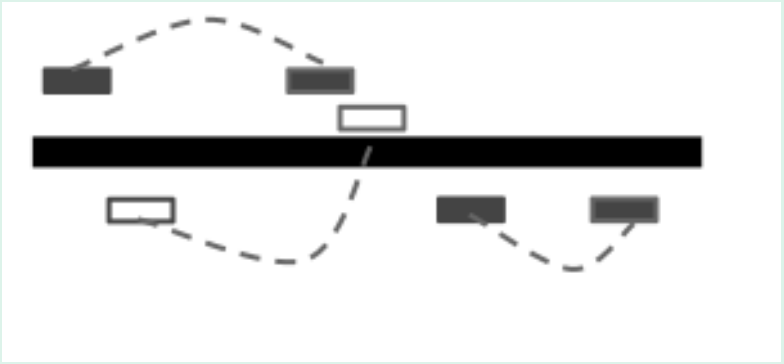


1. Preprocessing to put PEM data in proper format

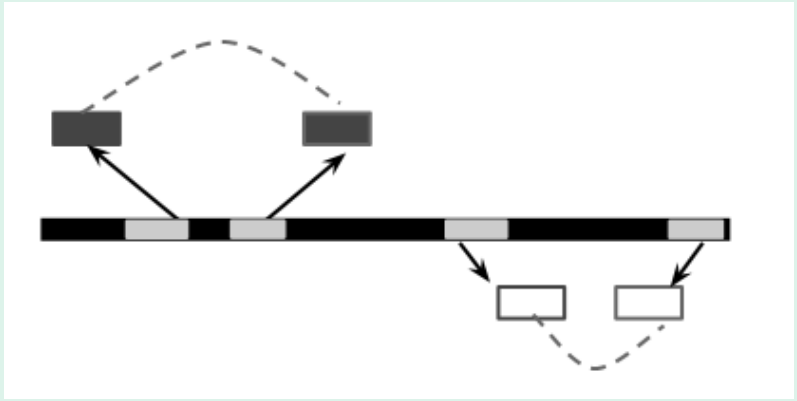


2. Mate-pair ends independently aligned to reference using MAQ or Megablast

The PEMer SV Pipeline

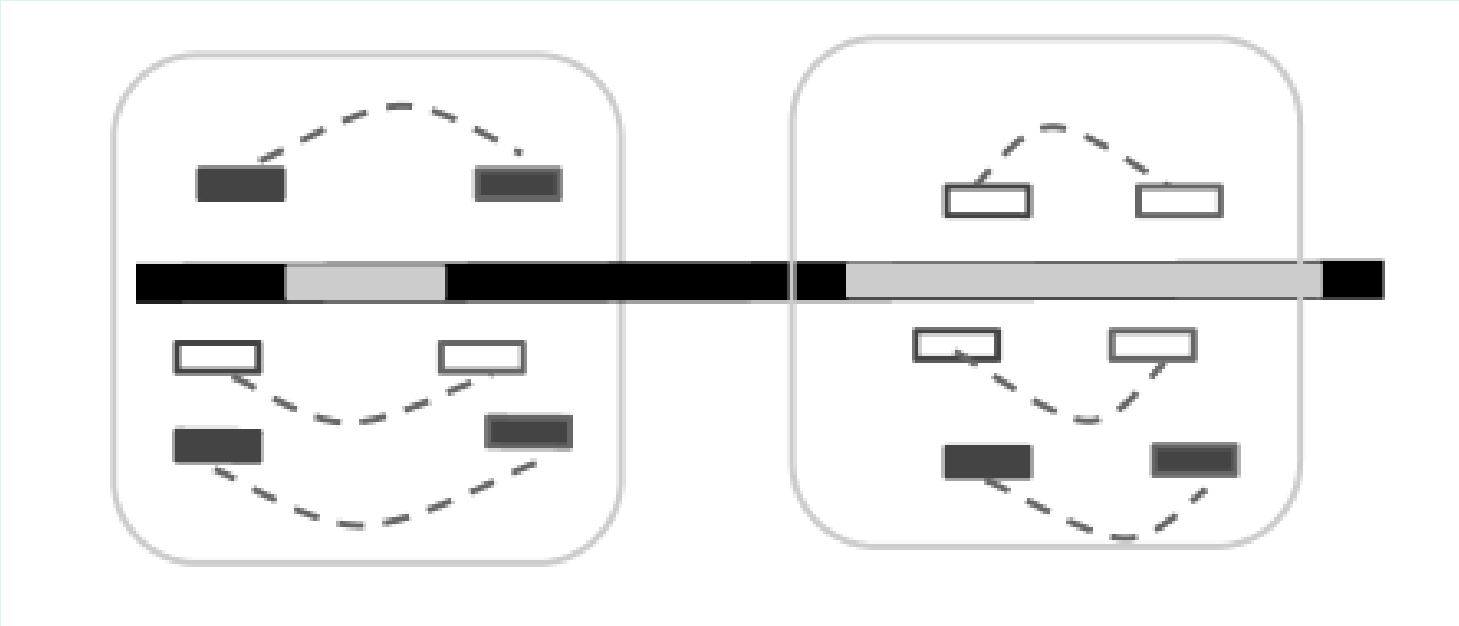


3. Optimal placement of mate-pair reads according to algorithm that seeks to minimize outliers



4. Mate pairs identified using experimentally defined cutoff span value

The PEMer SV Pipeline



5. Outliers classified into unique SVs. Clusters indicating the same SV are merged together

The PEMer SV Pipeline

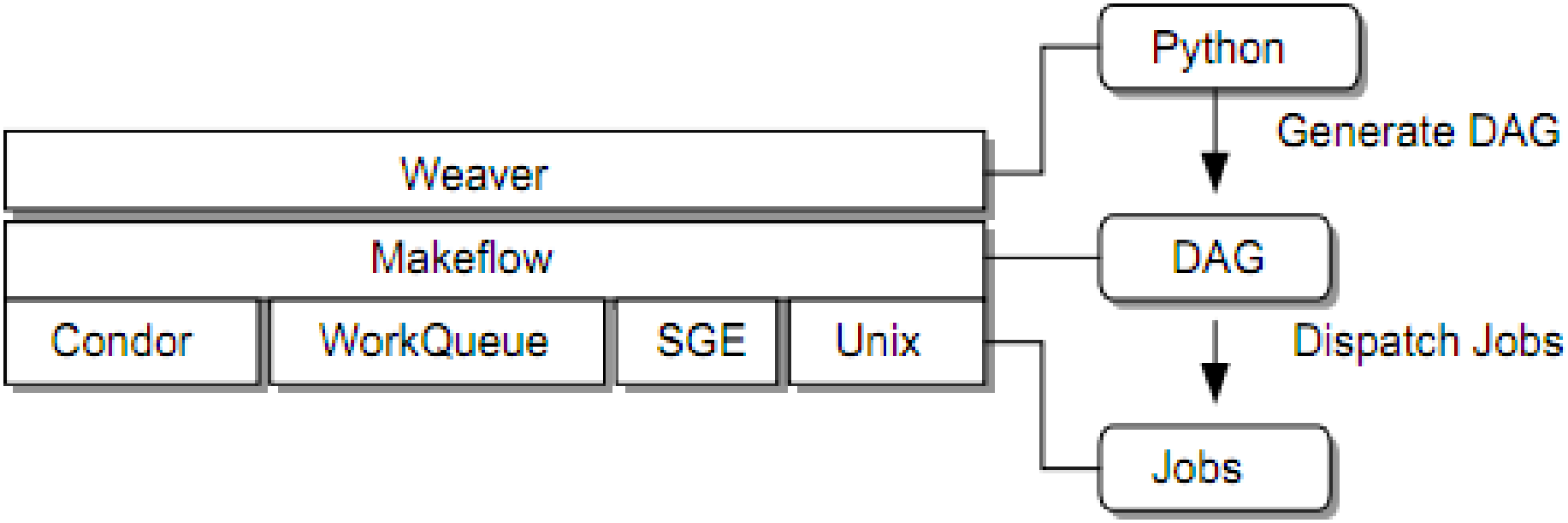
- A distributed-system version of PEMer came bundled, but it was restricted to shared-memory batch systems
- What was missing: a flexible, modular adaptation of the pipeline for heterogeneous systems and ad-hoc clouds

The PEMer SV Pipeline

- We refactored the pipeline using the Weaver/Starch/Makeflow stack (ND CCL) to allow for execution on multiple systems
- Scripts and higher level programs are practical solution for managing parallelization
- Several key lessons from this process can be applied to adapting other bioinformatics applications

Anatomy of the Weaver/Starch/Makeflow Stack

Weaver – Python-based framework for organizing/executing large-scale bioinformatics workflows



Anatomy of the Weaver/Starch/Makeflow Stack

- Datasets → collection of data objects with metadata accessible by query functions
- Functions → define interface to executables
- Abstractions → higher-order functions that are applied in specific patterns (i.e. Map, AllPairs, WaveFront)

Anatomy of the Weaver/Starch/Makeflow Stack

Advantages of using Python-based Weaver:

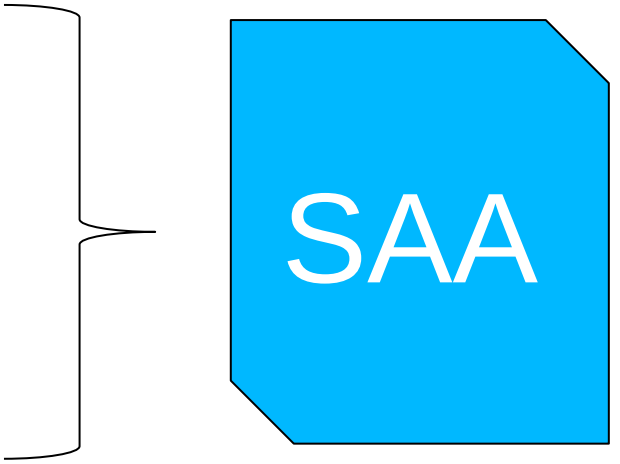
- Familiar syntax
- Easily deployable
- Extensible

Anatomy of the Weaver/Starch/Makeflow Stack

Starch

Application for encapsulating program dependencies in the form of **“Standalone Application Archives” (SAAs)**

- Complicated sets of dependencies
- Environment variables
- Input files
- User-specified commands



Anatomy of the Weaver/Starch/Makeflow Stack

Starch, cont.

All elements are compressed into a tarball, which is appended to a template shell script wrapper.

- Wrapper script automatically extracts the archive, configures the environment, and executes the provided commands.

Weaver + Starch – enable the easy generation of Makeflows and the packaging of dependencies

Anatomy of the Weaver/Starch/Makeflow Stack

Makeflow

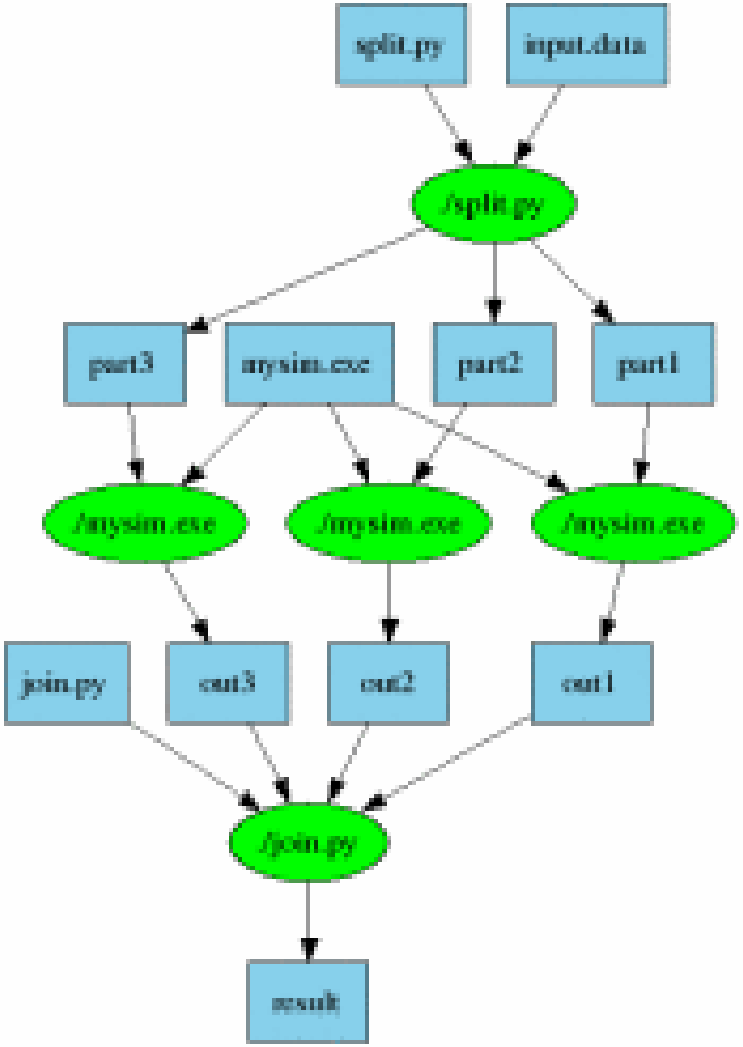
- Workflow engine designed for execution on clusters, grids and clouds
- Takes in a specified workflow, and parallelizes it
- Workflows are similar to Unix **make** files, and take the following format:

```
target(s): source input(s)
    command(s)
```

Anatomy of the Weaver/Starch/Makeflow Stack

Makeflow

- Workflow takes the form of a DAG
- Fault-tolerant. If the workflow stops or fails, Makeflow initiates resumption from failure point



Application of the Stack

- Refactoring begins with identifying data parallel portions of PEMer
  - → Luckily, all of the major steps can be executed in parallel
- Each step of the pipeline re-written as Weaver function, which in turn generates the corresponding Makeflow rules
  - → Made use of the Map abstraction
- All appropriate dependencies were packaged using Starch

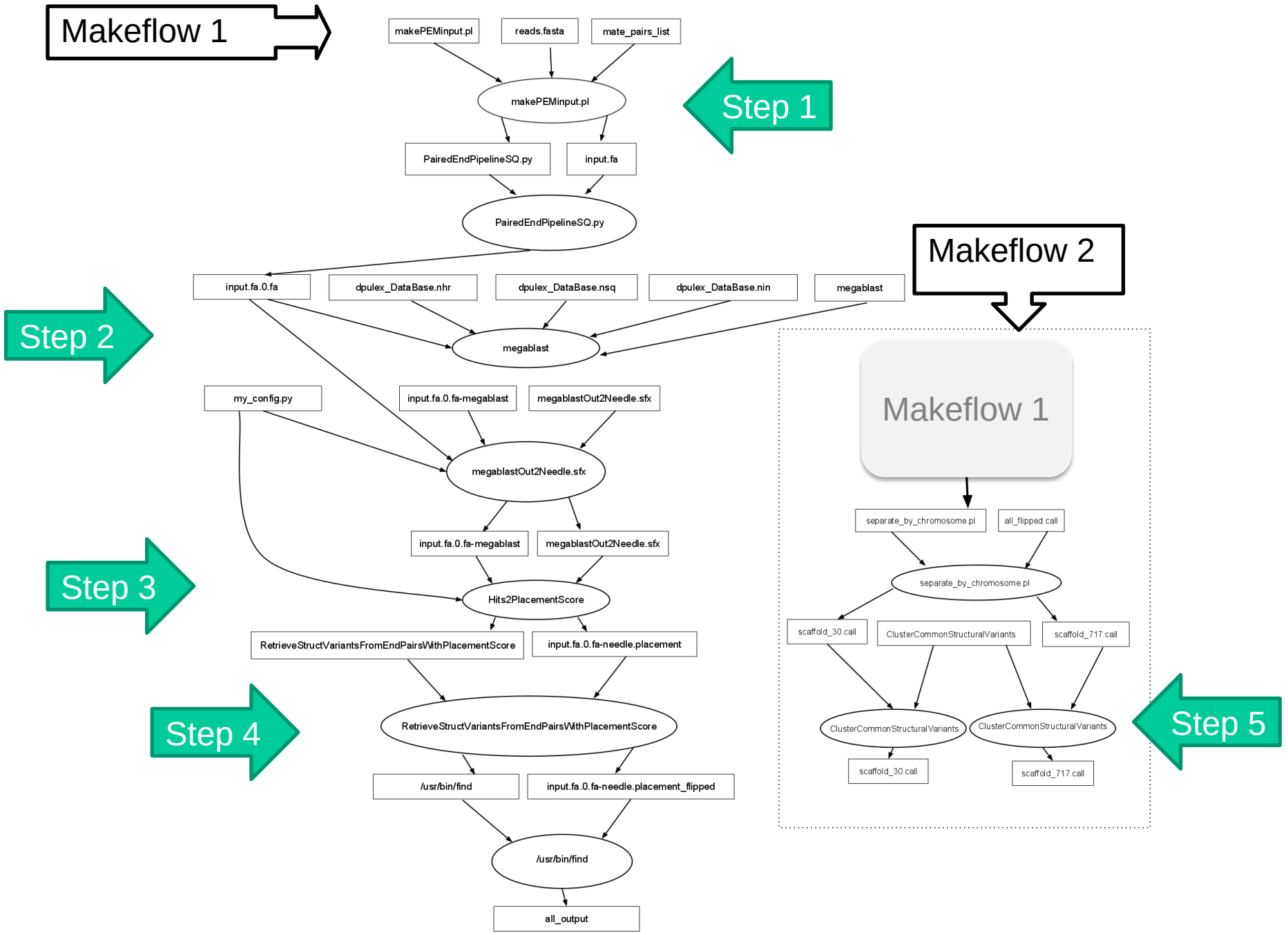
Data Used

PEMer pipeline applied to set of data from *Daphnia pulex*, an aquatic crustacean known for its extreme phenotypic plasticity

We provide PEMer with the following files

- File containing mate pair reads – 2.0 GB
- List of mate pairs
- Reference genome 222 MB

- First step of PEMer created 231 MB formatted file for subsequent distributed steps


Makeflow 1
makePEMinput.pl
reads.fasta
mate_pairs_list
makePEMinput.pl
Step 1
PairedEndPipelineSQ.py
input.fa
PairedEndPipelineSQ.py
Makeflow 2
input.fa.0.fa
dpulex_DataBase.nhr
dpulex_DataBase.nsq
dpulex_DataBase.nin
megablast
Step 2
megablast
Makeflow 1
my_config.py
input.fa.0.fa-megablast
megablastOut2Needle.sfx
megablastOut2Needle.sfx
separate_by_chromosome.pl
all_flipped.call
input.fa.0.fa-megablast
megablastOut2Needle.sfx
Step 3
separate_by_chromosome.pl
Hits2PlacementScore
scaffold_30.call
ClusterCommonStructuralVariants
scaffold_717.call
RetrieveStructVariantsFromEndPairsWithPlacementScore
input.fa.0.fa-needle.placement
Step 4
Step 5
RetrieveStructVariantsFromEndPairsWithPlacementScore
ClusterCommonStructuralVariants
ClusterCommonStructuralVariants
/usr/bin/find
input.fa.0.fa-needle.placement_flipped
scaffold_30.call
scaffold_717.call
/usr/bin/find
all_output

Deployment

Makeflow framework used to execute workflow, using 3 different frameworks

- **Condor** – heterogeneous, highly contentious environment
- **SGE** – more homogeneous, less pre-emption, shared FS
- **Work Queue** – lightweight, distributable through different batch systems or manually deployed

→ **Work Queue** executed using **Condor** and **SGE**

Deployment

- Submissions to **Condor** performed from 12-core machine with 12 GB of memory
- Submissions to **SGE** performed from an 8-core machine with 32 GB of memory
- Both machines were accessible to students across campus
- Frequently had to contend with multiple users sharing the machine

Results

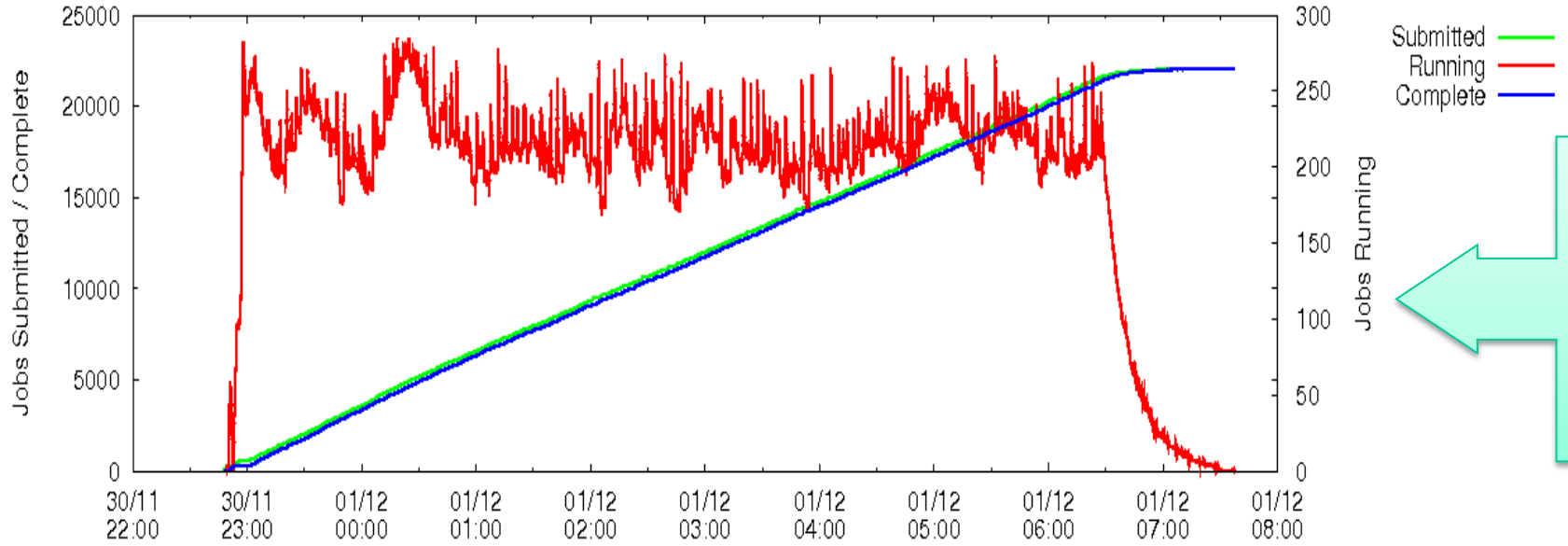
Implementation	Wall Clock Time	CPU Time	Speedup
Sequential	> 2 weeks	N/A	N/A
SGE original (100)	0 days 1:16:33	5 days 2:9:37	95.7
Condor (100)	0 days 19:15:32	73 days 10:29:00	91.5
Work Queue (100) *Condor*	0 days 23:44:24	84 days 17:21:57	85
Work Queue (100) *SGE*	0 days 18:31:21	73 days 12:8:54	95.2
Condor (300)	0 days 08:49:57	71 days 12:43:27	194.36
Work Queue (300) *Condor*	0 days 11:5:47	78 days 9:39:27	169
Work Queue (scaled) *Condor*	0 days 10:10:49	73 days 15:37:24	173

Comparison to Existing Batch Executable

Attribute	Provided Batch Scrip	Makeflow
Requires Shared File System	Yes	No
Code Encapsulation	A single script that handles the four core programs at once	A pipeline consisting of discrete steps executed consecutively
Deployment Environment	Shared file system/batch system, e.g. **SGE**	Any batch system, e.g. **Condor**, **SGE**, **Work Queue**
Logging	Start/stop times Program log captured using stderr and stdout	Detailed execution log Batch system log Optional debugging output

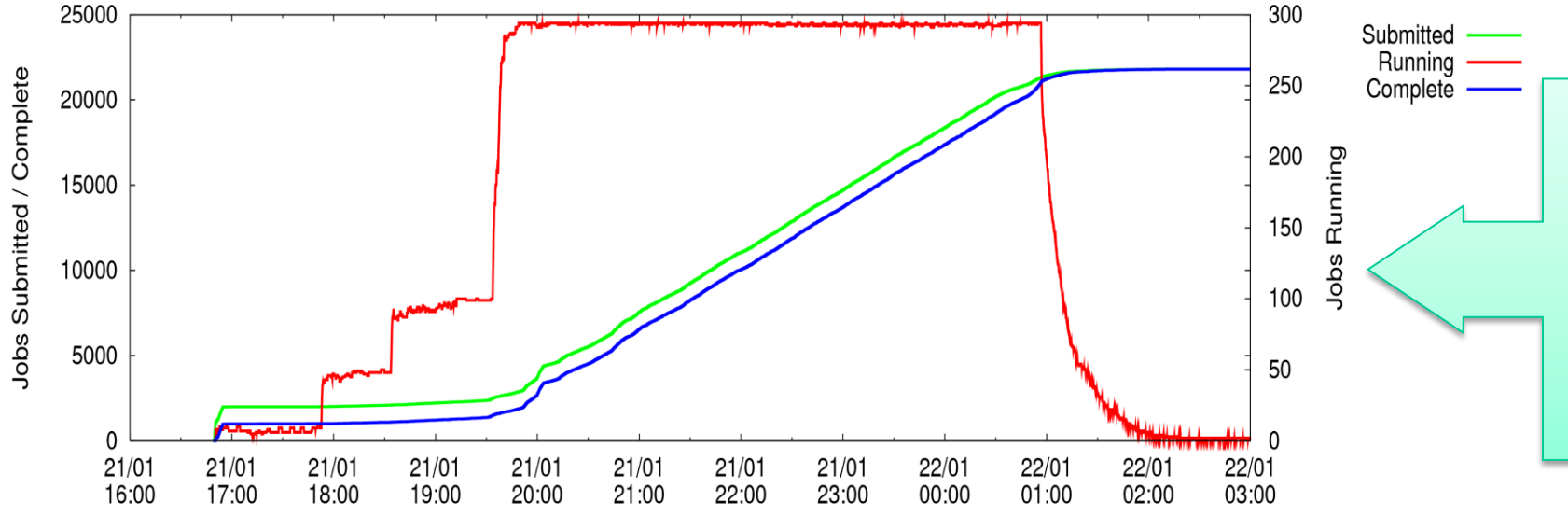
Results

Condor - 300 jobs



- Overhead from separate matchmaking, data caching for each new job

Work Queue- scaled



- Work Queue caches data , workers run continuously, avoid startup overhead on Condor

Lessons Learned

I. Determine optimal granularity

- Strike a good balance between the size of jobs and the number of jobs
- Small jobs can overwhelm the workflow engine ability to dispatch jobs effectively
- Large jobs susceptible to eviction, preemption

Lessons Learned

II. Understand remote path conventions

- Batch systems can have idiosyncratic interpretation of paths on remote machines
- A closer look at the required format can reveal unexpected requirements, even in established systems

Weaver/Makeflow– soft links not accepted, full path required underscores rather than backslashes

Lessons Learned

III. Be aware of scalability of native OS utilities

- Native functions such as `cat` and `rm` have limits on number of arguments
- Make sure these are not being overloaded by using utilities such as `find` with -`exec` to avoid
- Folder file limits can also be problematic, so consider this when choosing granularity

Lessons Learned

IV. Identify semantic differences between batch system and local programs

- Goals of batch system and pipeline can differ
- Batch system “success” = a returned file
- Pipeline “success” = a correctly processed file
- Try to align the goals of the two systems

PEMer/Makeflow – Jobs ran **`stat`** *to check size of returned file and return appropriate job status*

Lessons Learned

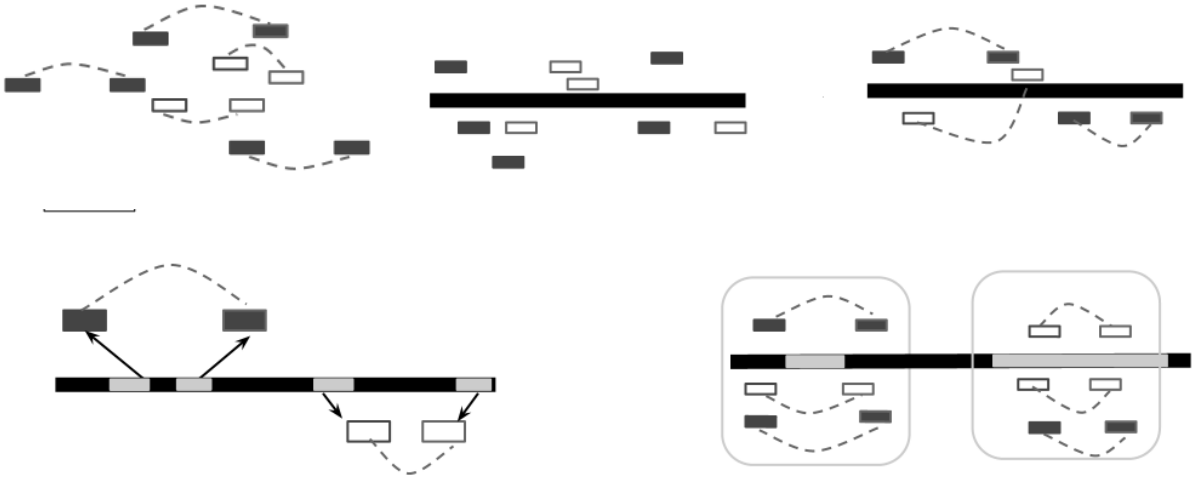
V. Establish execution patterns of program pipeline

- Recognize opportunities to apply abstractions
- Determine granularity
- Analyze data flow
- Problems arise if input for some steps not known *a priori*

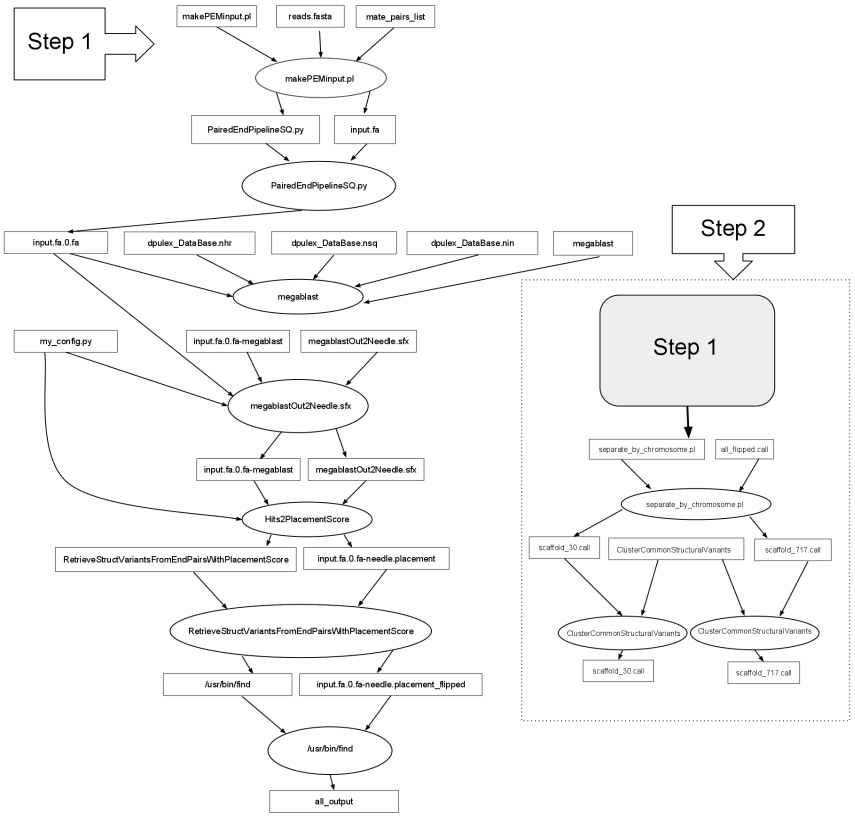
PEMer/Weaver – Lack of dynamic compilation necessitates multiple sequential Makeflows

Conclusions

- Refactoring was a success
- Weaver/Starch/Makeflow stack allowed for clean, intuitive adaptation of the program for distribution
- Execution on multiple heterogeneous systems now possible
- Scaled well, with good speedup
- Various obstacles provided excellent learning experience



Questions?


Step 1
makePEMinput.pl
reads.fasta
mate_pairs_list
makePEMinput.pl
PairedEndPipelineSQ.py
input.fa
PairedEndPipelineSQ.py
input.fa.0.fa
dpulex_DataBase.nhr
dpulex_DataBase.nsq
dpulex_DataBase.nin
megablast
megablast
my_config.py
input.fa.0.fa-megablast
megablastOut2Needle.sfx
megablastOut2Needle.sfx
input.fa.0.fa-megablast
megablastOut2Needle.sfx
Hits2PlacementScore
RetrieveStructVariantsFromEndPairsWithPlacementScore
input.fa.0.fa-needle.placement
RetrieveStructVariantsFromEndPairsWithPlacementScore
/usr/bin/find
input.fa.0.fa-needle.placement_flipped
/usr/bin/find
all_output
Step 2
Step 1
separate_by_chromosome.pl
all_flipped.call
separate_by_chromosome.pl
scaffold_30.call
ClusterCommonStructuralVariants
scaffold_717.call
ClusterCommonStructuralVariants
ClusterCommonStructuralVariants
scaffold_30.call
scaffold_717.call