PEACE: A Parallel Framework for Ab Initio Transcript-Clustering

Dhananjai M. Rao
Miami University
Computer Science and Software Engineering (CSE) Department
Oxford, Ohio 45056
raodm@miamiOH.edu

ABSTRACT

Clustering is used to partition the genomic data from Next Generation Sequencing (NGS) techniques into disjoint subsets that can be independently processed to ensure further analysis is computationally tractable. Performance is critical in clustering. Consequently, clustering software is typically developed as a tightly coupled monolithic system which hinders software reusability, extensibility and introduction of new algorithms as well as data structures. Having experienced similar issues in our own clustering software, we have developed a flexible and extensible parallel framework based on the Message Passing Interface (MPI) called PEACE. The objective of the framework is to ease design, implementation, and use of various clustering methods, including those developed by other investigators. This paper presents the PEACE framework, its software architecture, parallel infrastructure, and distributed data structures along with a case study of developing a clustering algorithm. Various filters, heuristics, and fragment comparison algorithms are also discussed to illustrate modularity and extensibility of PEACE which enables software reuse in unique ways that may not have been foreseen when individual components were developed. The paper also presents an empirical evaluation of the framework by comparing its performance to an earlier monolithic implementation of MST-based clustering. The results illustrate that the framework meets its objectives of modularity and extensibility without compromising performance.

KEYWORDS

Clustering; Expressed Sequence Tag (EST); Object-Oriented Design Patterns; distributed caches, Minimum Spanning Tree (MST); MPI; parallel framework

1 INTRODUCTION

Sequencing is the technique used for "reading" nucleotide bases constituting a biological sample. It has rapidly advanced to next-generation sequencing (NGS) technologies that are widely used to analyze gene expressions to obtain an organism's transcriptome, the set of (spliced) transcripts expressed by the genes of the organism [1]. NGS techniques achieve high throughput and lower cost by breaking transcripts into short fragments and "reading" multiple fragments from different transcripts in parallel [2]. The mix of transcript fragments generated by NGS technologies need to be suitably post-processed using bioinformatics tools to obtain the actual transcripts [1, 2]. Post-processing large number of fragments is a computationally demanding and a time consuming task even when distributed computing techniques are employed [2]. Therfore, reducing time, increasing quality, and improving efficiency of post-processing phase is critical for handling diverse and exponentially increasing volumes of genomic data.

A common approach used to ensure tractable post-processing analysis of NGS reads is called *clustering* or *binning* [3, 4]. In this context, clustering is the process of segregating the reads according to the transcript from which they were derived [1]. Clustering permits further processing and analysis to focus on a related subset of data and is often implicitly performed as part of many bioinformatics tools [4, 5]. However, clustering has to deal with complete set of sequencing output data and therefore it is a computationally challenging problem. It has a quadratic average time-complexity as determined by the length and number of fragments to be clustered [3].

Consequently, sophisticated parallel clustering algorithms and distributed software tools are necessary to efficiently cluster today's data sets. However, implementing such algorithms is a complex task and the modules constituting the software tend to be tightly coupled to extract maximum performance. Tight coupling combined with poor cohesion exacerbates extensibility and reusability of software modules. Seemingly straightforward tasks of mixing and matching (or plug-n-play) of modules from different software to construct alternative software pipelines can be a daunting task. Moreover, introducing even small changes to the software system is tedious. Such issues hinder reusing software modules in modalities that may not have been envisioned at the time tightly coupled systems are developed. These issues with clustering software are indeed akin to a similar situation with other bioinformatics software as summarized in [6] — "Historically, all these sequence assemblers – each representing many man-years of effort – appear to require complete and costly overhaul, with each introduction of a new short-read or long-range technology."

Unfortunately, the initial version of our clustering software system [7] also suffered from aforementioned shortcomings, hindering incorporation of new clustering algorithms, heuristics, and seamless use of different data structures. These issues motivated questions and investigations into easing design and implementation of reconfigurable clustering pipelines that are flexible, extensible, and reusable. Needless to emphasize, the objective was to gain these benefits without compromising overall performance of the software-pipelines.

Our continuing investigations into clustering have led to the development of a generic framework called PEACE. The PEACE framework provides software infrastructure to develop independent software modules that are suitably interconnected by the framework to establish a software-pipeline for clustering. It is important to note that the PEACE framework presented in this paper is not a specific

Extended version of short paper from ICSE'18, Gothenburg, Sweden
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

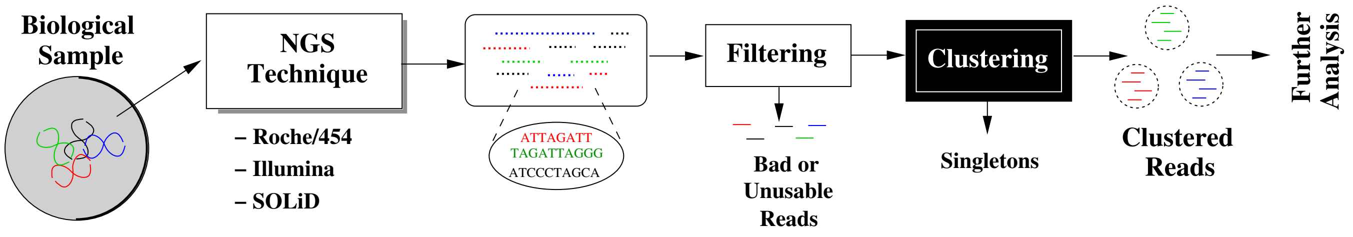

Figure 1: Overview of major steps leading up to clustering of reads from NGS technologies

clustering algorithm but a generic, framework to ease design, implementation, and customization of various clustering algorithms. In other words, the focus of this paper is on the design and effectiveness of the generic software infrastructure provided by PEACE and not on the specifics of a given clustering solution. However, for assessing the effectiveness of the framework, selected algorithms and heuristics published in the literature [7, 8] have been implemented using the framework. Consequently, the paper does not discuss clustering algorithms to contrast them against this research but to provide background on heuristics and clustering solutions used for empirical assessment of the framework.

The design of PEACE has been motivated by the need to address the shortcomings of the earlier tightly coupled, monolithic design. The revised design of PEACE has steadily evolved over a period of few years to addresses earlier shortcomings by providing a more flexible core framework that loosely couples various independent, but cooperating cohesive subsystems to provide an effective software framework for clustering. One of the primary goals of the new design is to provide flexibility and extensibility without compromising performance. Accordingly, suitable design strategies have been chosen from alternatives using performance profiling to provide best possible tradeoffs between flexibility and performance.

This paper presents the details on the new object-oriented framework of PEACE, its parallel infrastructure, and its distributed data structures and caches in Section 4 through Section 8. These sections are preceded by Section 2 that provides background information and Section 3 that presents a brief survey of prior, closely related research investigations. An experimental assessment of scalability and performance of PEACE is presented in Section 10. Section 11 concludes the paper by summarizing the outcomes of this research.

2 BACKGROUND

This section presents only a succinct and simplified summary on sequencing and the various operations leading to clustering. However, the field of sequencing, associated technologies, and challenges is vast and readers are referred to the literature for more detailed survey of this fast moving field [2]. An overview of the major steps involved in clustering sequences is shown in Figure 1 and each major step is briefly described in the following paragraphs.

Sequencing is the process of identifying the sequence of nucleotide bases, namely namely adenine (A), guanine (G), cytosine (C), and thymine (T), that constitute Ribonucleic acid (RNA) and Deoxyribonucleic acid (DNA) [2]. Sequencing of RNA, particularly messenger RNA (mRNA) that determines the protein produced due to cellular activity, is not easy. The RNA is typically reverse-transcribed into DNA called complementary DNA (cDNA) and cDNA is sequenced. Prior to NGS techniques, RNA sequencing was accomplished using Sanger sequencing which produced Expressed Sequence Tags (ESTs). Immaterial of the technology used, from a computing perspective, sequencing produces a string over the alphabet string A, G, C, T.

Sequencing techniques have rapidly advanced to next-generation sequencing (NGS) technologies such as Roche/454, Illumina, and SOLiD [2]. NGS techniques perform massively parallel sequencing of nucleotide bases by generating "polymerase colonies" or polonies. The sequences are typically short, ranging from 400-500 bases for Roche/454 down to just 50 bases for SOLiD. Furthermore, the reads from various sub-segments of the sample are delivered in a mixed batch and have to be segregated for further processing. The NGS technologies typically have few percentage points of error in the reads [2]. Sequencing errors manifest themselves in characteristic manners often leading to repeated patterns called low-complexity regions.

Reads with errors or low-complexity regions can negatively impact quality of clustering. Consequently, the reads are typically filtered to eliminate reads that may degrade quality of clustering [2]. Filtering is a complex operation that may or may not be included as part of clustering software. The filtered NGS reads are then clustered to suitably segregate reads obtained from the same sample. Depending on the NGS technology and the nature of the biological sample a variety of clustering strategies are used. Input fragments that could not be clustered are typically reported as singletons or outliers after clustering. In certain scenarios, singletons are useful for drawing biological or environmental observations. The clustered data is then subject to further analysis and processing based on the objective of the study. Clustering is a powerful tool to explore and study large-scale complex data. It is often implicitly performed as part of many bioinformatics tools and as a "pre-assembly" step. It is an important phase in processing fragments from NGS technologies and is an active area of research and development.

3 RELATED RESEARCH

Clustering continues to remain an active area of research and a number of clustering approaches have been proposed. However, most clustering software have been developed as tightly coupled, relatively inflexible systems and at the time of this work, very few generic frameworks such as PEACE have been proposed. Consequently, rather than contrasting the generic framework (which is not a clustering solution) against specific transcript clustering algorithms, this section surveys clustering solutions, including those used for empirical assessments, to highlight various aspects to be addressed to develop a framework. The readers are directed to the literature for a comprehensive survey of various clustering algorithms and tools [7, 8].

Hazelhurst *et al* [8] propose a parallel clustering tool called WCD that uses the d^2 algorithm along with the u/v and t/v huristics to improve performance. These algorithms divide a given NGS read into a series of 6 to 8 contiguous nucleotides called *words* and compare words to detect relationship. WCD forms clusters by merging reads with minimal dissimilarity into the same cluster. The d^2 algorithm along with u/v and t/v heuristics proposed by Hazelhurst have been implemented using the PEACE framework for experimental evaluation. In addition, enhanced versions of these algorithms developed as part of our earlier investigations [7] and are also used for empicial evaluation. Recently, Hazelhurst *et al* [3] proposed a heuristic based on multiple long exact matches that are sufficient spread across a pair of reads to identify candidates for clustering. Implemented using suffix arrays, Hazelhurst *et al* [3] show significant performance improvements. A similar parallel suffix tree based implementation involving the use of maximal common substring of pairs has been proposed by Kalyanaraman *et al* [5]. These clustering solutions are implemented in C language using MPI for parallelization and heavily use heuristics to improve performance.

Moretti *et al* [9] present a framework called SAND that interfaces with the Celera open-source assembly toolkit, to enable scalable assembly on clusters, clouds, and grids. The software uses a k-mer counting approach to identify candidates, a strategy sufficiently close to clustering using common substrings mentioned in the previous paragraph. Similar to the proposed effort, Moretti *et al* also strive to develop a modular system that in which the Celera assembler can be replaced with other backends. Furthermore, they also note that development of modular bioinformatics software components is a fast growing trend [9].

4 CORE PEACE FRAMEWORK

The primary objective of the PEACE framework is to provide a modular, extensible, and performant infrastructure that eases incorporating new algorithms and data structures in various phases of clustering. Modularity and extensibility has been realized by leveraging object-oriented design patterns [10] and architecting the framework as a collection of loosely coupled subsystems. The subsystems have been designed and developed to ease parallelization by adopting the Single Program Multiple Data (SPMD) [11] paradigm. PEACE uses the Message Passing Interface (MPI) [11] to enable parallel and distributed computing on Supercomputing clusters. The design also places emphasis on rigorous memory management to ensure minimal memory footprint as it is a critical resource when processing "big data". The proposed architecture has been has been implemented in C++ to meet the performance goals and expose full control on memory usage. Moreover, the design have been chosen from alternatives using performance profiling to provide best possible tradeoffs between flexibility and performance.

An architectural overview of the subsystems constituting the core of the framework is shown in Figure 2. The top-level `PEACE` class performs the core task of controlling and coordinating instances of four independent subsystems, namely: the INPUT subsystem, the FILTERING subsystem, the CLUSTERING subsystem, and the OUTPUT subsystem. Life cycle management of the subsystems are accomplished via the `SubSystem` class which exposes a polymorphic Application Programming Interface (API) and serves as the base class for the concrete subsystem implementations. In addition, the design facilitates introduction of additional subsystems into the framework. The `SubSystem` API is comprehensive to encompass the functionality of diverse subsystems. Consequently, some subsystems merely provide no-op implementations for some of the APIs in `SubSystem`. For example, only the input subsystem typically implements the `loadInputs` method while the output subsystem implements the `generateOutputs` method.

Subsystems in PEACE, have been designed using the façade pattern [10] to encapsulate and manage the collection of components constituting the subsystem. The `Component` class is the abstract leaf class of the subsystem hierarchy. It employs a composite pattern [10] to compose subsystems using hierarchies of components. In contrast to `SubSystem`, the `Component` defines only a minimal API for all components to enable further delegation of life cycle activities. Figure 2 illustrates the composition of the OUTPUT subsystem using a subset of components, namely the `StandardOutput` and `ClusterWriter` components, to illustrate an example. A more detailed discussion on the OUTPUT subsytem, other subsystems in PEACE is presented in Section 5 through Section 8.

A two-phase approach has been utilized in the main `PEACE` class to dynamically compose and configure a subsystem using information from command-line arguments. The first phase commences the life cycle activities by instantiating the subsystems. Subsystems establish default component hierarchies when constructed. Next, valid command-line parameters at the subsystem-level are gathered in an `ArgParser` object using the `addCommandLineArguments` API call. The `ArgParser` is used to parse command-line arguments, completing the first phase. The second phase involves using command-line arguments to instantiate and customize components and sub-components. This phase proceeds in a recursive, depth-first manner through invocation of the `initializeSubSystem` and `initializeComponents` API methods. During this phase, the framework permits additional command-line parameters to be added to the `ArgParser` by components and sub-components. The second phase ends after command-line arguments are re-parsed and recursive, depth-first calls to `initializeSubComponents` API method complete successfully. At the end of the two phases a complete software pipeline is established for performing clustering. Recollect that PEACE embodies the SPMD paradigm and consequently the same software pipeline is established on all parallel processes. The initial setup phases establish necessary cross-references to frequently accessed data to minimize subsequent overheads.

The independent subsystems are loosely data-coupled via a shared, singleton [10] object called the `RuntimeContext` shown in Figure 2. In addition to enabling loose coupling, the shared context also aids in minimizing memory footprint and eliminating unnecessary computation. The context object is populated during second phase of subsystem composition with references to shared data

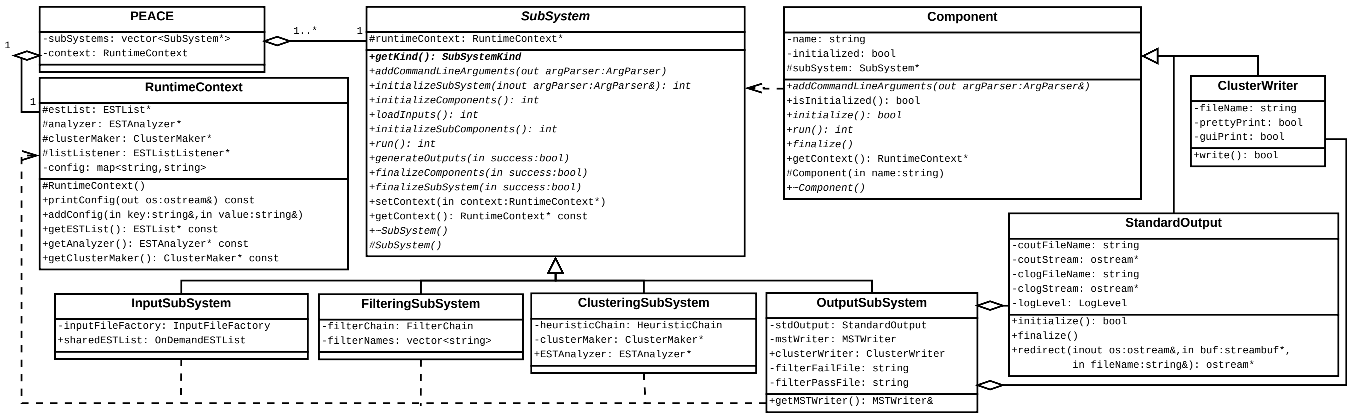
PEACE
-subSystems: vector<SubSystem*>
-context: RuntimeContext
1
1..*
1
SubSystem
#runtimeContext: RuntimeContext*
+getKind(): SubSystemKind
+addCommandLineArguments(out argParser:ArgParser)
+initializeSubSystem(inout argParser:ArgParser&): int
+initializeComponents(): int
+loadInputs(): int
+initializeSubComponents(): int
+run(): int
+generateOutputs(in success:bool)
+finalizeComponents(in success:bool)
+finalizeSubSystem(in success:bool)
+setContext(in context:RuntimeContext*)
+getContext(): RuntimeContext* const
+~SubSystem()
#SubSystem()
Component
-name: string
-initialized: bool
#subSystem: SubSystem*
+addCommandLineArguments(out argParser:ArgParser&)
+isInitialized(): bool
+initialize(): bool
+run(): int
+finalize()
+getContext(): RuntimeContext*
#Component(in name:string)
+~Component()
ClusterWriter
-fileName: string
-prettyPrint: bool
-guiPrint: bool
+write(): bool
RuntimeContext
#estList: ESTList*
#analyzer: ESTAnalyzer*
#clusterMaker: ClusterMaker*
#listListener: ESTListListener*
-config: map<string,string>
#RuntimeContext()
+printConfig(out os:ostream&) const
+addConfig(in key:string&,in value:string&)
+getESTList(): ESTList* const
+getAnalyzer(): ESTAnalyzer* const
+getClusterMaker(): ClusterMaker* const
StandardOutput
-coutFileName: string
-coutStream: ostream*
-clogFileName: string
-clogStream: ostream*
-logLevel: LogLevel
+initialize(): bool
+finalize()
+redirect(inout os:ostream&,in buf:streambuf*,
in fileName:string&): ostream*
InputSubSystem
-inputFileFactory: InputFileFactory
+sharedESTList: OnDemandESTList
FilteringSubSystem
-filterChain: FilterChain
-filterNames: vector<string>
ClusteringSubSystem
-heuristicChain: HeuristicChain
-clusterMaker: ClusterMaker*
+ESTAnalyzer: ESTAnalyzer*
OutputSubSystem
-stdOutput: StandardOutput
-mstWriter: MSTWriter
+clusterWriter: ClusterWriter
-filterFailFile: string
-filterPassFile: string
+getMSTWriter(): MSTWriter&

Figure 2: Architectural overview of core sub-systems constituting PEACE (diagram shows only the core attributes and methods)

structures and primary components managed by each subsystem. The components are exposed only via an abstract interface permitting the use of various concrete implementations at initialization. The `RuntimeContext` only exposes an immutable interface to shared components and data structures to enforce an implicit publish-subscribe style of interaction with a single publisher. The architecture also permits the components to be used in a ad hoc manner via the framework's interactive console.

PEACE has been designed from the ground-up using the SPMD paradigm to run on Supercomputing platforms via MPI. The framework provides lightweight wrappers for most of the MPI calls to streamline cross-platform development and increased portability. The MPI wrappers enable a single implementation to run in parallel when MPI is enabled or gracefully degrade and run serially when MPI is disabled. The wrappers utilize compiler macros to adapt their operation to the operating system and platform configuration. Moreover, the wrappers are designed to ease replacement of MPI with alternative message-passing libraries to foster diverse research investigations. A more detailed discussion of the operations and data structures managed in four subsystems are presented in the following sections.

5 THE INPUT SUBSYSTEM

The INPUT subsystem performs the task of loading NGS fragments or Expressed Sequence Tag (EST) entries from various input file formats. A general purpose cDNA class, also aliased to `EST` class, encapsulates all the information about a transcript fragment obtained via NGS techniques. It provides a storage-format independent unmarshalled version of fragment data including: the nucleotide `sequence`, phred quality score for each nucleotide, and general information about the fragment. The set of fragments to be processed are stored in an abstract `ESTList` which is the primary data structure that is exposed and shared by this subsystem via the `RuntimeContex` (see Figure 2). It logically represents the complete set of data to be clustered from one or more input data files. However, its concrete implementation does not load the complete set of fragments into memory in all parallel processes. Instead, PEACE evenly subdivides the total number of fragments across the various processes and only the pertinent subset of fragments are initially loaded. Other fragments are loaded on-demand. However, this behavior can be overridden in the framework. A prefetching call was introduced in the framework to cache a batch of fragments and bypass various checks performed for on-demand loading. The prefetching improved performance of u/v heuristic by almost 25% for large data sets.

An `InputFile` interface class facilitates loading fragments from commonly used data file formats such as: FASTA which is a text file format; and a binary Standard Flowgram Format (SFF). The concrete classes for reading various file formats are instantiated using the factory design pattern [10]. The `InputFile` class handles the task of removing masked bases from reads based on command-line arguments. Command-line options can also be used to substitute ambiguous nucleotides with suitable amino-acid entries. These operations were found to streamline various sub-steps in clustering operations [7]. The fragments loaded by this subsystem are filtered and clustered as discussed in the following subsections.

6 THE FILTERING SUBSYSTEM

Operations of the FILTERING subsystem are performed prior to clustering to suppress erroneous fragments (see Section 2 for sources of errors) from clustering as they are known to hinder clustering and degrade clustering quality. This subsystem is centered around a `FilterChain` component which composes the subsystem as an arbitrary series of filters. The order of filters and individual filter configuration enables different types of filtering to be accomplished. Concrete filters are indirectly instantiated via a `FilterFactory` which uses the factory design pattern. Each filter in this subsystem extends the `Filter` sub-component base class to provide specific filtering operations. For example the framework includes a `LCFilter` component that is used to eliminate reads with low-complexity regions. The `LCFilter` uses the analyzer component from the globally shared `RuntimeContext` to compare a repeated pattern of nucleotides representing low-complexity region against input fragments to identify and filter out problematic fragments. Multiple instances of the LCFilter with different patterns are used in a chain to filter out reads with different forms of low-complexity regions. Configuration of the filter chain and individual filters is accomplished with command-line parameters that are processed in two phases as discussed in Section 4.

The SPMD paradigm dictates that the same filter chain is established on all parallel MPI processes. However, the FILTERING subsystem evenly subdivides the global list of fragments in the shared `RuntimeContext` between multiple processes to perform filtering in parallel. The `FilterChain` short circuits filtering and does not permit a fragment to flow through the complete chain when it is rejected by a filter. Filtered fragments from each process is tracked using their unique identifiers. After all fragments are processed, the identifiers of filtered fragments are broadcast and each process and are ignored during clustering. Furthermore, dummy-clusters are created for each filter and rejected fragments are placed in corresponding dummy-clusters for review by the user at the end of clustering. The fragments that successfully pass filtering are clustered in parallel by the CLUSTERING subsystem as discussed in Section 7.

7 THE CLUSTERING SUBSYSTEM

The operations related to parallel clustering of NGS fragments are performed by the CLUSTERING subsystem. It is the most complex subsystem with multiple, independent but interacting, hierarchies of `Components`. The components constituting this subsystem are categorized into the following three main class-hierarchies as shown in Figure 3: `ClusterMakers`, `Analyzers`, and `Heuristics`. The task of a `ClusterMaker` is to construct clusters by placing related fragments in the same cluster. Identification of related fragments is accomplished by using a suitable analyzer that can compare two given fragments to yield a pseudo metric that is a measure of similarity or dissimilarity between them. Comparison of fragments is often computationally intensive and typically lightweight heuristics are used to avoid unnecessary comparisons. The `Heuristic` class-hierarchy provides provides various heuristics that can be used to improve overall performance of clustering.

The `ClusterMakerFactory` shown in Figure 3 is used to instantiate a concrete child component such as the `MSTClusterMaker` during the second phase of initialization after first round of command-line argument parsing has been completed. Second phase of initialization described in Section 4 is used to further customize the object and establish cross-references in the `RuntimeContext` and other `Components` in the subsystem. Due to diversity in clustering strategies, the `ClusterMaker` does not enforce any specific clustering semantics. Instead, specifics of parallel clustering are delegated to concrete implementation classes such as `MSTClusterMaker`. Section 9 provides a more detailed discussion on the operations of the `MSTClusterMaker` and illustrates detailed use of the framework.

The framework provides various utility methods to facilitate parallel clustering and managing distributed caches along with interfaces to streamline inter-component interactions. For instance, the framework strives to streamline processing by permitting a single fragment to be designated as the reference fragment which is to be compared with other fragments to be clustered. Setting a consistent reference fragment across the parallel system permits components in the analyzer and heuristic hierarchies to optimize their operations by caching and reusing previous values until the reference fragment is changed.

The `ESTAnalyzer` class hierarchy shown in Figure 3 provides the infrastructure for developing various algorithms that can be used to compare cDNA fragments to identify candidates for clustering. The framework API method `analyze` mandates analyzers generate a quantitative pseudo metric after comparison of a pair of fragments. Although represented as floating-point numbers by the framework, it is important to note that pseudo metrics need not be transitive, commutative, or associative. Consequently, the API includes additional polymorphic methods for comparisons and other operations involving pseudo metrics.

The analyzer hierarchy supports both similarity-based and distance-based metrics. The API methods such as `getInvalidMetric` and `compareMetrics` shown in Figure 3 provide a neutral interfaces that can be consistently used with both types of analyzers that can be distinguished via the `isDistanceMetric`. For example the `CLU` class shown in Figure 3 implements the CLU algorithm proposed by Ptitsyn *et al* [12] for generating similarity scores.

The classes `D2`, `TwoPassD2`, and `AdaptiveTwoPassD2` provide distance-based pseudo metrics that are based on the alignment free d^2 algorithm [8]. The `TwoPassD2` analyzer proposed in earlier research [7] determines the sub-fragments with maximum likelyhood of best d^2 sore in the first phase and then runs d^2 within the sub-fragments during the second phase [7]. Similarly, the `AdaptiveTwoPassD2` is a redesigned version of the distance measurement algorithm used in the previous version of PEACE. It extends `TwoPassD2` by using a set of precomputed parameters managed by the `ParameterSetManager` to customize the operation of d^2 algorithm based on the length of fragments being compared. Customization based on fragment length provides better quality while reducing average number of comparisons. A more detailed description and comparisons of these algorithms is available in the literature [7, 8]. The framework also includes a straightforward `MatrixFileAnalyzer` that obtains pseudo metrics from a text file that is organized as a matrix of values. The data values can either be similarity or distance pseudo metrics. Other algorithms have also been implemented to verify and demonstrate the flexibility of the PEACE framework.

The CLUSTERING subsystem includes a collection of lightweight heuristics that can be used to minimize the number of heavyweight comparisons performed by analyzers. The `Heuristic` component forms the base class of heuristics in the subsystem. The framework has been used to implement a redesigned version of the u/v and t/v heuristics reported in the literature [7]. Although these two heuristics are independent of each other, in the revised design, the heuristics have been implemented using the polymorphic API to maximize code reuse, demonstrate modularity, and effective coupling supported by the framework API.

Similar to filters, heuristics are organized into a flexible and dynamically composable chain managed by the `HeuristicChain` class. Configuration of heuristics constituting the chain is accomplished via the two-phase processing of command-line arguments as discussed in Section 4. The use of heuristics in the chain is short circuited when a heuristic rejects a given pair of fragments for further processing. Consequently, it is preferable to configure the chain to contain faster or more permissive heuristics prior to slower or more restrictive ones. Accordingly, by default the framework configures the chain to contain the u/v heuristic prior to the t/v

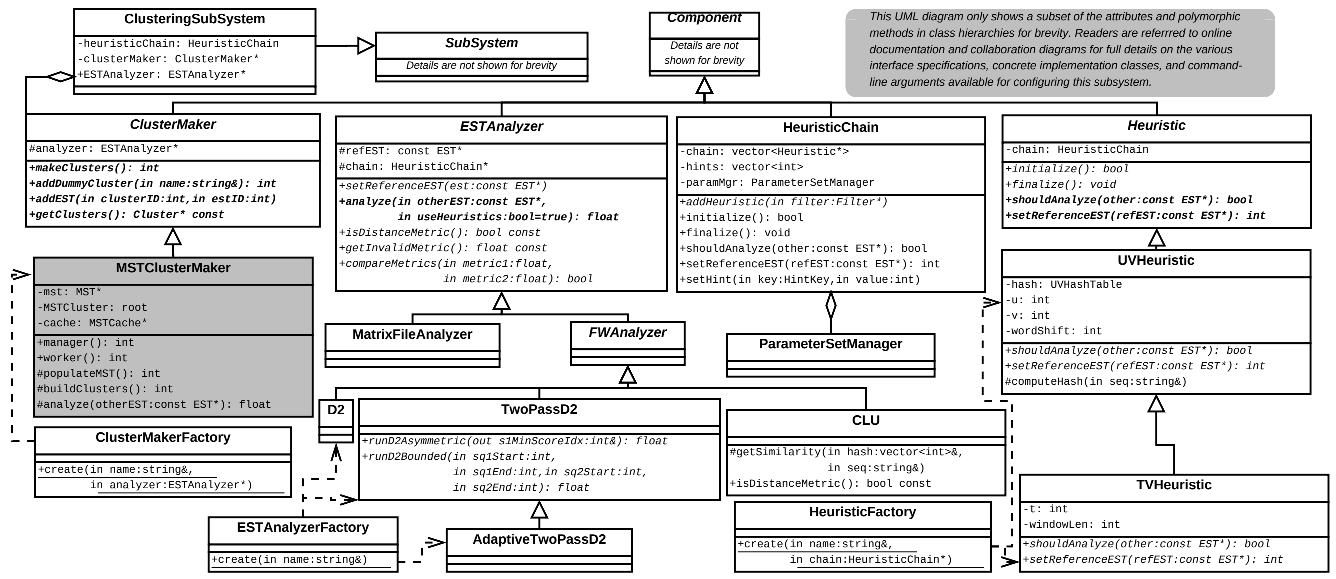
ClusteringSubSystem
-heuristicChain: HeuristicChain
-clusterMaker: ClusterMaker*
+ESTAnalyzer: ESTAnalyzer*
SubSystem
Details are not shown for brevity
Component
Details are not shown for brevity
This UML diagram only shows a subset of the attributes and polymorphic methods in class hierarchies for brevity. Readers are referrred to online documentation and collaboration diagrams for full details on the various interface specifications, concrete implementation classes, and command-line arguments available for configuring this subsystem.
ClusterMaker
#analyzer: ESTAnalyzer*
+makeClusters(): int
+addDummyCluster(in name:string&): int
+addEST(in clusterID:int,in estID:int)
+getClusters(): Cluster* const
ESTAnalyzer
#refEST: const EST*
#chain: HeuristicChain*
+setReferenceEST(est:const EST*)
+analyze(in otherEST:const EST*, in useHeuristics:bool=true): float
+isDistanceMetric(): bool const
+getInvalidMetric(): float const
+compareMetrics(in metric1:float, in metric2:float): bool
HeuristicChain
-chain: vector<Heuristic*>
-hints: vector<int>
-paramMgr: ParameterSetManager
+addHeuristic(in filter:Filter*)
+initialize(): bool
+finalize(): void
+shouldAnalyze(other:const EST*): bool
+setReferenceEST(refEST:const EST*): int
+setHint(in key:HintKey,in value:int)
Heuristic
-chain: HeuristicChain
+initialize(): bool
+finalize(): void
+shouldAnalyze(other:const EST*): bool
+setReferenceEST(refEST:const EST*): int
MSTClusterMaker
-mst: MST*
-MSTCluster: root
-cache: MSTCache*
+manager(): int
+worker(): int
#populateMST(): int
#buildClusters(): int
#analyze(otherEST:const EST*): float
MatrixFileAnalyzer
FWAnalyzer
ParameterSetManager
UVHeuristic
-hash: UVHashTable
-u: int
-v: int
-wordShift: int
+shouldAnalyze(other:const EST*): bool
+setReferenceEST(refEST:const EST*): int
#computeHash(in seq:string&)
D2
TwoPassD2
+runD2Asymmetric(out s1MinScoreIdx:int&): float
+runD2Bounded(in sq1Start:int, in sq1End:int,in sq2Start:int, in sq2End:int): float
CLU
#getSimilarity(in hash:vector<int>&, in seq:string&)
+isDistanceMetric(): bool const
ClusterMakerFactory
+create(in name:string&, in analyzer:ESTAnalyzer*)
TVHeuristic
-t: int
-windowLen: int
+shouldAnalyze(other:const EST*): bool
+setReferenceEST(refEST:const EST*): int
HeuristicFactory
+create(in name:string&, in chain:HeuristicChain*)
ESTAnalyzerFactory
+create(in name:string&)
AdaptiveTwoPassD2

Figure 3: Overview of the commonly used classes and class hierarchies in the CLUSTERING subsystem

heuristic as the former is faster and more permissive than the latter. The use of the CLUSTERING subsystem components, its distributed caching infrastructure, and the operations of the redesigned MST-based clustering algorithm are further discussed in Section 9.

8 THE OUTPUT SUBSYSTEM

The OUTPUT subsystem houses the framework components that deal with writing various outputs and data from the RuntimeContext in different formats. Some of the core components constituting this subsystem are shown in Figure 2. The framework already includes wrappers for standard output (std::cout), standard error (std::cerr), and standard log (std::clog) streams to facilitate redirection of the streams to files. In addition, wrappers for logging also provide a convenient mechanism to generate logs at various levels for selective logging. The framework also facilitates coordination between corresponding components when operating in parallel.

The existing components in this subsystem operate in three distinct modes supported by the framework. Components such as output stream wrappers operate asynchronously writing outputs on a per-process basis. Singleton components such as progress report generators run only on process with MPI-rank zero and periodically write status reports to a specified output file. Other output components work under the assumption of shared, networked file system (NFS) and write voluminous data to a given output file using barrier-synchronization and by taking turns in the order of their MPI-ranks. Individual components suitably handle specifics of file formats and other peculiarities associated with a given output format.

9 MST-BASED CLUSTERING

The framework described in Section 4 through Section 8 has been used to redesign the Minimum Spanning Tree (MST) based method for clustering from our earlier research [7]. The primary objective of the redesign was to conduct "apples-to-apples" experimental comparisons to establish effectiveness of the proposed framework. It is important to bear in mind that PEACE is a general purpose framework and is not a specific clustering approach. Consequently, any clustering solution can be implemented using the framework. However, a MST-based clustering propsed in earlier investigations was chosen as the reference test case due to availability and sufficient complexity. Readers are referred to the literature for detailed discussion on the biological significance of MST-based clustering and the quality of clusters produced by it [7]. Selected details are discussed in this section to highlight the role of the framework and provide basis for implementing other clustering algorithms.

A conceptual overview of the various framework components and distributed data structures in the new design is shown in Figure 4. The MSTClusterMaker introduced in Section 7 coordinates the tasks associated with construing the MST in parallel and generating clusters from the MST. The MST is built using a modified version of Prim's algorithm and single-linkage style of clustering is performed by suitably cutting edges of the MST [7]. It has been implemented by extending the abstract ClusterMaker base class in the framework and operates serially on a single process or in parallel when multiple processes are used. The unified implementation approach ensures that the operations in serial and parallel mode are not significantly different as the framework insulates the class from several details such as partitioning of reads to be clustered, filtering reads, and providing wrappers to streamline MPI-based operations. Partitioning of fragments also logically assigns ownership and processing of fragments to a given process. Furthermore, the conceptual structure and organization of the software pipeline is identical on all parallel processes in concordance with the SPMD paradigm. However, as shown in Figure 4, when running in parallel, the process with MPI-rank 0 implicitly operates as the manager to

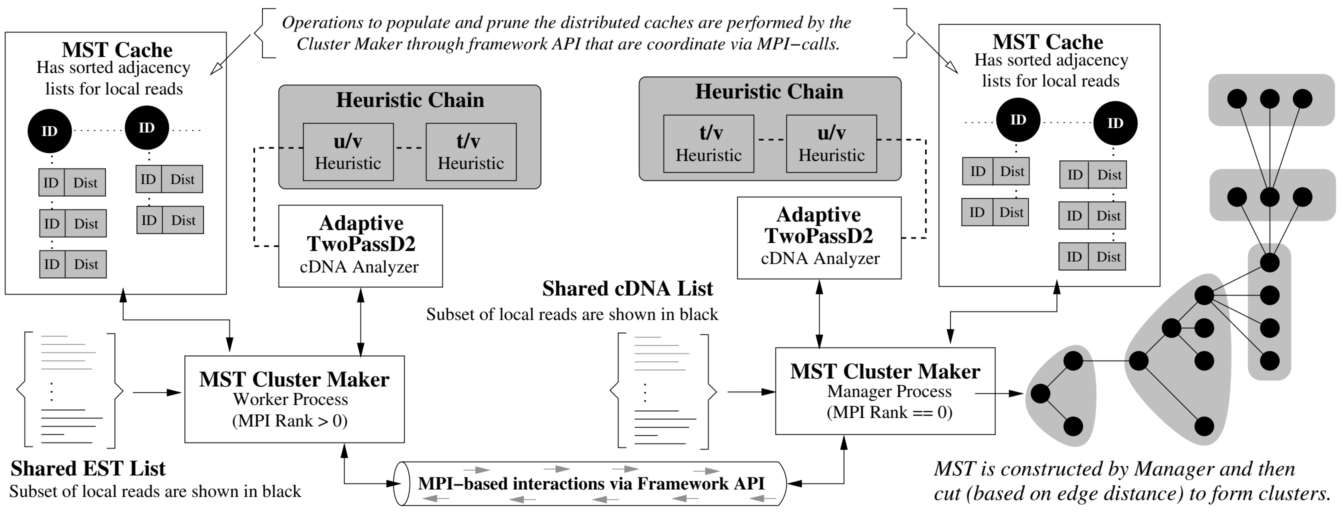
Operations to populate and prune the distributed caches are performed by the Cluster Maker through framework API that are coordinate via MPI−calls.
MST Cache
Has sorted adjacency lists for local reads
ID
ID Dist
Heuristic Chain
u/v Heuristic
t/v Heuristic
Adaptive TwoPassD2
cDNA Analyzer
Shared cDNA List
Subset of local reads are shown in black
MST Cluster Maker
Worker Process
(MPI Rank > 0)
Manager Process
(MPI Rank == 0)
Shared EST List
Subset of local reads are shown in black
MPI−based interactions via Framework API
MST is constructed by Manager and then cut (based on edge distance) to form clusters.

Figure 4: Conceptual overview of the runtime structure used for MST-based clustering

coordinate activities between itself and other parallel processes called `workers`.

One fragment is randomly chosen by the `manager` to serve as the new reference fragment to be added as a node to the MST being constructed. The `manager` broadcasts the newly added node to all processes. Every process searches its subset of unclustered fragments to identify reads that are closely related to the newly added node. Closely related reads are identified using the analyzer object set by the framework in the `RuntimeContext`. The pseudo-metric reported by the analyzer is also used as the edge weights in the MST. Furthermore, as illustrated in Figure 4 the analyzer can be configured to use a series of lightweight heuristics to avoid unnecessary calls to the heavyweight algorithms used by the fragment analyzer. The default configuration uses the adaptive two-pass d^2 analyzer along with the u/v and t/v heuristics that were discussed in Section 7.

Each process constructs a list containing candidate fragments along with edge-weights for clustering. The list is dispatched by the framework to the process that logically owns the reference fragment. The owning process merges the list dispatched by each process and sorts the entries to identify best candidates for clustering. The sorted list of information is cached at the owning process in the `MSTCache` as shown in Figure 4. The `MSTCache` is organized as lists-of-lists and contains best-neighbor information for the fragments that are logically assigned to it. Next, the `manager` broadcasts request to each `worker` to report its local best candidate to be added to the MST based on the information in the `MSTCache`. The `manager` uses its local cache and the information reported by the `worker` processes to identify the best fragment and suitably adds it to the MST. The newly added fragment is used as the reference in the next cycle of operations until all the fragments have been added to the MST. The framework also provides interfaces to prune caches as fragments are added to the MST to minimize memory footprint. Repeated pruning of caches may result in removal of all entries in lists associated with certain fragments. When lists become empty, the framework triggers repopulation of lists. Repopulation can be suppressed via command-line arguments to further optimize and control operations of the framework.

Once the MST has been constructed, the `manager` process computes the edge-cutting threshold for forming clusters. The edge-cutting threshold is computed as the mean of edge weights plus the standard deviation. Edges with weight larger than the edge-cutting threshold are removed to form clusters. Recollect that the pseudo-metrics reported by the `ESTAnalyzer` is used as the edge weights in the MST. Ergo, edges that exceed the edge-cutting threshold connect fragments that are not closely related and are removed to form clusters as shown in Figure 4. Once the clusters are formed and the `RuntimeContext` is updated, the framework suitably writes the cluster information via components in the OUTPUT subsystem.

10 EXPERIMENTS

The motivation for this research is to design a flexible and performant framework that can be used to design and implement various clustering algorithms (and not to propose a new clustering solution). Consequently, the primary objective of the experiments to empirically identify any runtime overheads introduced by the framework and not contrast various aspects of clustering against other clustering alogrithms. In this research study the monolithic MST-based clustering implementation along with various heuristics and distributed caches has been suitably incorporated into the newly designed framework as discussed in Section 9 to provide an "apples-to-apples" comparison. The objective of implementing a diverse set of algorithms is to demonstrate the flexibility and comprehensiveness of the API exposed by the various subsystems constituting the framework. The algorithms to be implemented were chosen based on their popularity and availability of software tools for comparison. The algorithms were suitably implemented and verified using a variety of data sets. Interoperability between various implementations was also tested. In addition, performance of our implementation was compared with original implementations of individual algorithms that were mostly in C-language. These experiments indicated that the object-oriented implementation using the

Table 1: Datasets used for experiments

Species Name	#Reads	Read Length Statistics		
		Avg±SD	Min	Max
R. Communis	57690	709±199	49	1361
A. Thaliana	76941	427±128	33	1552
C. Reinhardtii	189975	553±183	100	1541

PEACE framework provided comparable performance with less than 3% degradation. Detailed profiling using `valgrind` indicated that the degradation was primarily due to overhead of polymorphic calls used in the C++ implementation. The performance profiles indicate that the depth of polymorphism did not play a significant role. The slight degradation is the cost of achieving increased interoperability and extensibility of the object-oriented subsystems.

Having verified functionality and performance of individual subsystems, the next phase of experimentation places emphasis on system-level performance involving interactions between individual sub-systems. In order to provide effective empirical comparison, the MST-based clustering software reported in the literature [7] has been used as the base case. It must be noted that the clustering solutions generated in both cases were verified to be identical enabling undivided pursuit of the primary objective of identifying overheads introduced by the PEACE framework. An experimental comparison of the performance of the previous and new implementations using three different data sets shown in Table 1 is discussed below. The experiments focus on the objective of this research (which is to develop a general purpose framework and not a specific clustering approach) and readers are referred to the literature for other analysis involving clustering tools developed by other researchers [7].

The experiments conducted to compare the monolithic MST-based clustering against the new loosely coupled implementation commenced with validating that the clustering solution generated in both cases were identical. Having validated overall functionality, the two systems were used to cluster three different data sets shown in Table 1 using a varying number of MPI processes. The time for clustering and the peak memory usage is shown in Figure 5 and Figure 6 respectively. The experiments were conducted on our cluster which consists of 36 compute nodes interconnected by Myrinet and a shared file system of 15 TB. Each node has dual quad-core 2.26 GHz Intel Xeon E5520 CPUs and 24 GB of memory.

The experimental data in Figure 5 indicates that the new PEACE framework introduces about 5% degradation in performance when statistically compared to the earlier implementation. Similar to earlier profiler observation, overheads of polymorphic calls in the object-oriented framework was the primary factor contributing to the slight performance degradation. However, as the number of processes were increased the gap slowly decreased as the cost of polymorphism is distributed across more CPUs. In fact for the smaller data set with *R. Communis* gene expression data, the performance of the two systems on 32 processors were statistically indistinguishable when the 95% confidence intervals were taken into account.

The memory usage of the new framework is about 20% to 25% higher as shown in Figure 6 due to the following two reasons. The framework was configured to load all the reads on all processes to reflect the behavior of the base case implementation to avoid any influence on performance measurements. This causes a steady linear increase in overall memory usage for both systems as the number of MPI processes is increased. In addition, the framework also maintains placeholders for phred quality scores as discussed in Section 5. Consequently, each read occupies a slightly larger space in memory, contributing to the increased memory footprint.

The experiments also indicate that the framework did not introduce changes to the overall scalability or add overheads to the communication calls performed by the clustering algorithms. Consequently, the scalability and performance observed in Figure 5 is merely an artifact of the clustering approach chosen for experimental assessment. Ergo behaviors of other algorithms would depend on their charachteristics rather than those of the framework.

11 CONCLUSION

The accelerated growth in throughput of NGS techniques has made parallel clustering an indispensable step to analyze and further process the reads. Clustering systems typically tend to be developed as tightly coupled, monolithic systems which exacerbates their extensibility and reusability. This paper proposed a modular framework (and not a specific clustering algorithm) that loosely couples

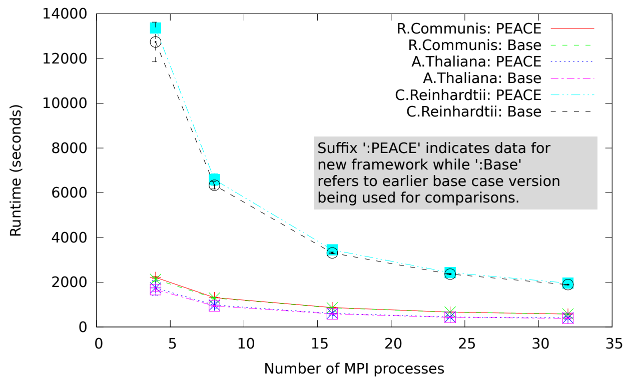

Figure 5: Runtime comparison

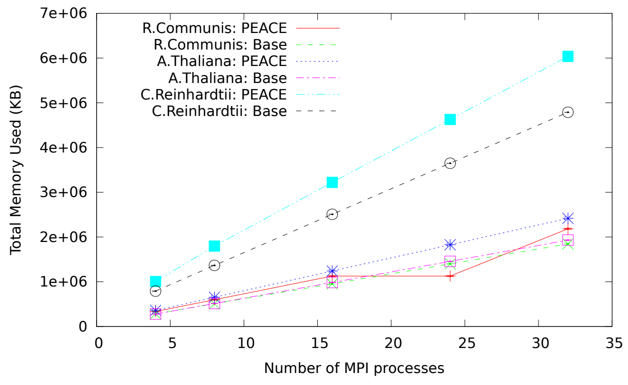

Figure 6: Total (sum on all processes) memory usage

components in various subsystems to construct a suitable software pipeline for clustering. The objective of the framework is to not only ease design and implementation of any parallel clustering algorithms but also facilitate interoperability and reuse. The paper presented the design rationale and patterns used in core framework, its four main subsystems, and the principal components constituting the subsystems.

The framework has been used to develop a variety of distance and similarity based fragment analyzers along with several heuristics used to improve overall clustering performance. The operation of filter chains, analyzers, and dynamic composition of heuristics into chains, was discussed. The diversity in analyzers and the flexibility of using filters and heuristics provides ample evidence to the comprehensive API provided by the framework. Furthermore, an existing MST-based clustering algorithm has been incorporated into the framework to compare and contrast its performance with is monolithic predecessor.

The comparative experiments performed using different real world data sets were presented to highlight that significant enhancements in overall quality of the software can be achieved without sacrificing much performance through effective object-oriented design. The experiments indicate that the performance degradation was less than 5% and the increase in memory footprint is about 25%. However, with memory getting cheaper, the increased memory footprint is offset by the benefits accrued. The performance losses primarily arose due to polymorphic calls involved in the API. The performance profiles indicate that the depth of polymorphism did not play a significant role. However, with newer C++ standards and compilers there is considerable promise that such issues will continue to diminish in magnitude.

The design of the PEACE framework has been developed over a period of few years. The design required several iterations involving many hours of meticulous profiling to ensure each API was efficient and it provided necessary features to be effectively used at a system level. In other words, the design and development of the API required considerable time and effort. Consequently, to enable effective sharing and reuse the framework has been released under the GNU General Public License (GPL). The framework and sample data sets are freely available for download from its website at http://removed.for.review. We envision PEACE to serve as a framework and testbed to ease study, design, implementation, interoperability, and use of clustering algorithms that further lead to innovations and discoveries in biology and medicine.

REFERENCES

[1] S. Nagaraj, R. Gasser, and S. Ranganathan, "A hitchhiker's guide to expressed sequence tag (EST) analysis," *Brief Bioinformatics*, Jan 2007. [Online]. Available: http://intl-bib.oxfordjournals.org/cgi/content/abstract/8/1/6

[2] E. R. Mardis, "A decadefis perspective on DNA sequencing technology," *Nature*, vol. 470, pp. 198–203, Feb. 2011.

[3] S. Hazelhurst and Z. Liptk, "Kaboom! a new suffix array based algorithm for clustering expression data," *Bioinformatics*, vol. 27, no. 24, pp. 3348–3355, 2011. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/27/24/3348.abstract

[4] Y. Wang, H. C. Leung, S. Yiu, and F. Y. Chin, "Metacluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample," *Bioinformatics*, vol. 28, no. 18, pp. i356–i362, 2012. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/28/18/i356.abstract

[5] A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel, "Efficient clustering of large EST data sets on parallel computers," *Nucleic Acids Res*, vol. 31, no. 11, pp. 2963–74, Jun 2003. [Online]. Available: http://nar.oxfordjournals.org/cgi/content/full/31/11/2963

[6] G. Narzisi and B. Mishra, "Comparing de novo genome assembly: The long and short of it," *PLoS ONE*, vol. 6, no. 4, p. e19175, 04 2011. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0019175

[7] Omitted for Review, "Omitted for Review," *Nucleic Acids Research*, vol. 38, no. 2, Jun.

[8] S. Hazelhurst, "Algorithms for clustering expressed sequence tags: the wcd tool," *South African Computer Journal*, vol. 40, pp. 51–62, May 2008. [Online]. Available: http://www.cs.wits.ac.za/~scott/papers/sacj527.pdf

[9] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain, "A framework for scalable genome assembly on clusters, clouds, and grids," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2189–2197, 2012.

[10] J. M. Smith, *Elemental Design Patterns*, 1st ed. Reading, MA: Addison-Wesley Professional, 2012.

[11] P. Pacheco, *An Introduction to Parallel Programming.* Reading, MA: Morgan Kaufmann, 2011.

[12] A. Ptitsyn and W. Hide, "CLU: a new algorithm for EST clustering," *BMC Bioinformatics*, vol. 6 Suppl 2, p. S3, Jul 2005, [PubMed:16026600] [PubMed Central:PMC1637039] [doi:10.1186/1471-2105-6-S2-S3].