USING PILOT-JOBS FOR DEVELOPING *E*THREAD, A META-THREADING PIPELINE

BY ANJANIBHARGAVI RAGOTHAMAN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

DR. SHANTENU JHA

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2014

ABSTRACT OF THE THESIS

Using Pilot-Jobs for Developing *e*Thread, a Meta-threading Pipeline

by Anjanibhargavi Ragothaman

Thesis Director: DR. SHANTENU JHA

The genome revolution has produced vast amount of sequence information, but the functional annotation of most of the gene products are yet to be explored in depth. Functional inference of low sequence identity is brought about by the structure based template methods. To model and understand these proteome scale functions, state-of-the-art algorithms like *e*Thread is used. They are compute intensive and demand efficient and optimal use of the underlying resources. Combination of large scale data and complex workload raises the need for pilot based approaches. *e*Thread is a meta-threading protein structure modeling algorithm which is supported by ten independent single-threading algorithms whose computational complexity also depends on the number and size of the input sequences. In this thesis, *e*Thread pipeline is developed on an extensible, scalable and interoperable pilot-job based framework and it supports concurrent tasks execution and data-parallelization on heterogeneous resources deployed on Amazon EC2 with S3 as data repository. This study aims to understand the dominant factors which influence the performance of *e*Thread on EC2. This analysis suggests an optimized solution based on execution time and cost of implementation. It primarily achieves better utilization of resources by scaling workload on multiple resources. Further ideas on increasing resource capacity and discussions on the importance of dynamic

execution of tasks are also laid out.

Acknowledgements

I express my sincere gratitude to my advisor, Dr. Shantenu Jha, for providing me an opportunity to work with him. His continuous guidance, support, encouragement and patience were very valuable throughout the course of my research. I thank Dr. Nayong Kim, Dr. Joohyun Kim, Dr. Michal Brylinski and other LSU colleagues for their excellent support and discussions during every phase of the project. I thank the members of my committee, Dr. Laleh Najafizadeh and Dr. Saman Zonouz for their valuable inputs and time. I would like to extend my gratitude to all the members of Radical-Research team who were major support during the complete cycle of development and execution. I'm grateful to Dr. Wei P Feinstein for providing the Amazon computing resources that enabled this work. Last but not the least; I would like to thank my family and friends for their support and encouragement during the process of writing my thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Systems Biology aims to understand the components in the living systems, how they interact and how diseases are manifested due to its malfunctions. Genome revolution was a major point which created a huge amount of sequence information for the community. At the point of writing this dissertation, the Protein Data Bank had 101741 structures [1]. Nevertheless, molecular functions of most of these products still remain unknown. Standard homology-based tools over-predict the molecular function eventually leading to high level of mis-annotation [2]. Structure-based approaches to annotate provide a promising solution for this issue. Protein structure modeling plays an essential role in Functional Genomics by helping to decipher structural information which subsequently gets utilized for protein function inference [3].

Presently, the most accurate and widely used computational protein structure prediction methods build on information from related proteins and many assessment methods like CASP (Critical Assessment of Protein Structure Prediction) use threading and template based methods for tertiary structure prediction [4, 5]. A number of techniques search for low-sequence identity templates called as the twilight zone sequence similarity structures [6] to construct the structural model and infer its molecular function to avoid the issue of mis-annotation. These structure based approaches are powerful tools in speeding up the genome-wide protein annotation and help to overcome the limitation of traditional sequence based approaches.

Specifically, there is a development of meta-threading techniques for protein structure prediction in the recent time. These methods consider output from different individual threading algorithms and construct target-to-template alignments. They stand

a better chance in accurate predictions when compared to single threading methodologies. There are existing successful meta-threading predictors like LOMETS [7] and neural-network based predictor Pcons [8]. Along this line is *e*Thread, a highly accurate meta-threading model to identify templates for the template-based modelling of protein structures [4]. It combines ten state-of-the-art threading algorithms and uses machine learning to optimally select structural templates and provide functional modeling.

However, one of the critical scientific challenges is to optimally combine the output of the individual threading algorithms to provide an increased overall accuracy over single threading methods. But, it comes with a caveat. With the increase in complexity of the algorithms, there is an equivalent increase in the demand for more powerful and intelligent computational resources. As these meta-threading procedures have multiple algorithms, it is highly important to consider the implementation details and optimal utilization of the computing resources. These meta-threading pipelines pose computational challenges due to their heterogeneous collection of the individual threading algorithms and differ in terms of input files, time to completion, memory usage and I/O operations. Moreover, they involve data intensive computations. Hence, a thorough profiling of the resource and the algorithm is necessary to execute the complete implementation in an optimal way.

In previous work by Brylinski et al, a comprehensive study of resource profiling for *e*Thread in terms of time to solution and memory footprint was performed on dedicated HPC clusters [6]. Though this approach of using individual machine is widely used in many scenarios, there is a developing need for distributed computing due to various reasons. First of all, *e*Thread comprises of diverse set of algorithms each differing in their computational needs. Typically HPC clusters are homogeneous and consist of identical nodes which may not be necessary for all the threading algorithms. They typical involve large amount of data sets which demands for more storage space.

On this account, cloud infrastructures offer wide and flexible range of instance types in different combinations of CPU, memory, storage and networking capacity on demand. The user gets the freedom to customize the environment for specific needs. Also, cloud infrastructure is aptly suited for applications which are loosely coupled [9]. *e*Thread is

a classic example for such an application as the ten threading algorithms are not dependent on each other. Thus, it provides an opportunity to optimally use the infrastructure and on efficient implementation methods, it avoids over or under utilization of the resources. Apart from resource profiling it is also essential to consider the cost-to-solution as different instance types come with different hourly rates.

Nevertheless, for an application like *e*Thread, the design and development on any particular infrastructure requires knowledge about that platform, its programming system and functioning. In this case, if the application development is independent of the runtime environment it will be portable to any kind of infrastructure. With such an application, it becomes very flexible to deploy it in diverse infrastructure types like HPC, Grids or clouds. Those applications that are flexible, extensible and easily deployable are not only at an advantage to adapt to any kind of infrastructure and but are also readily scalable. Hence the application can support the testing of both simple genomic scale and complex proteome scale simulations.

There are key considerations to make while designing the *e*Thread application as there are varied options available with the cloud infrastructure. It is important to decompose the application, design the workflow for the individual threading algorithm and schedule appropriate workloads to instances for optimal utilization along with cost in consideration. Interestingly, distributed infrastructure [10, 11] and pilot-job abstractions [9, 12, 13] have been effectively used in many large scale bioinformatics applications which share these afore mentioned concerns.

BigJob [14] is a SAGA based Pilot-Job framework which is very helpful in flexible and scalable implementation, coordination and management of the infrastructure and help in efficient utilization of the resource. The pilot-job system acts as a container for the number of sub-jobs in the workflow and helps in concurrent execution of the chain and domain libraries in the protein threading process of each single-threading tool. They also help in submitting the tasks to heterogeneous virtual machines and in parallel execution of all the ten protein threading tools there by making it very optimal. This dissertation handles the pilot-job based implementation of *e*Thread meta-threading pipeline on cloud infrastructure to tackle an efficient execution of the workflow.

1.1 Related Work

Due to the increase in the improvements of available computational resources, the last two decades has seen lot of work on using HPDC resources for scientific workflows. For instance, Montage [15] is an image processing application which takes multiple images of the sky from different telescopes and builds a mosaic equivalent to a single image. It follows a complex set of workflow represented by a DAG and is fed to Pegasus workflow planner [16]. Montage is computationally intensive and the jobs are executed across distributed resources which help in better utilization of the idle computational resources and in optimizing the time-to-solution. Replica Exchange molecular dynamics simulations [17] are classic examples of compute and data intensive bio-molecular applications. They are used to understand problems ranging from protein folding dynamics to binding affinity calculations [18, 19]. Naturally, there are multiple implementations of RE simulations to fulfill the need of efficient and scalable mechanisms to execute in the available resources [20, 21, 22, 23].

These diverse examples show case few non-trivial applications which are similar in resource requirements, which are mostly parallel and distributed resources, and demand an efficient model for optimized time-to-solution, data transfer and overall performance. There are also other upcoming programming models and algorithms like Hadoop and MapReduce to facilitate large scale distributed data processing [11, 24].

In direct relation to this thesis, the initial work on *e*Thread was performed by Brylinskis group to understand the behaviour of the application on a dedicated resource [6]. The implementation methods followed a simplified version of the Portable Batch System (PBS) prescribed by preliminary set of rules. The resource profiling helped to derive a fundamental understanding of the nature of single-threading tools. It helped in getting an insight that the behaviour of each protein threading tool varies widely in terms of time-to-solution and memory utilization. Though standalone systems have their own advantages, they are limited by nature. They do not meet increasing computing demands and the user would meet storage constraints with increase in data set size. An application which uses heterogeneous set of algorithms is bound to have varied resource

demands and the individual resource will not be able to meet the expectations in an efficient manner. This paved the way to think about alternate solutions to implement *e*Thread. Cloud computing fits the requirement as they provide abundant immediately available resources in varied ranges and provide both scale-up and scale-out options.

1.2 Structure of the Thesis

This thesis contains five chapters. Chapter 1 provided the motivation behind this dissertation. It introduced meta-threading in genomics, discussed the concepts underlying *e*Thread pipeline and the motivation behind its need. Chapter 2 provides the required theory and background of *e*Thread, pilot-jobs , cloud infrastructure and the benchmark dataset used. Chapter 3 introduces the implementation of *e*Thread pipeline. It explains the details of the python program and the deployment and execution details. Experimental benchmark results are presented with the analysis in Chapter 4 . Chapter 5 provides the concluding remarks with directions for future work. Finally, Appendix is included explaining the steps required for any user to replicate the experiment to their environment.

Chapter 2

Background

This chapter deals with the background required to develop the *e*Thread pipeline on the Amazon EC2 cloud infrastructure using the pilot framework. It provides an overview about *e*Thread, the infrastructure used and explains the concept of pilot-jobs and discusses the specific implementation (BigJob) of the concept including the control flow and data flow within BigJob.

2.1 Overview of *e*Thread

Protein threading helps in predicting the protein structure and further in functional annotation. Meta-threading techniques are creating headway in protein structure prediction. These methods identify template structures and construct target-to-template alignments by analyzing outputs from different threading algorithms. These combined predictions have a higher chance to be accurate than those produced by single threading algorithms. Also, previous work has shown that meta-threading supported by machine-learning outperforms single-threading approaches in functional template selection [25]. It effectively identifies many facets of protein molecular function even in a low sequence identity regime. Additional advantage of the meta-predictors is the improved estimation of the reliability of the predictions [25]. *e*Thread employs the meta-threading analysis method with the support of machine-learning making it one of the robust meta-threading analysis tools.

2.2 Method Overview

The meta-threading flowchart for the *e*Thread algorithm is shown in figure 2.1. For an amino acid input sequence, the algorithm applies the meta-threading to search for

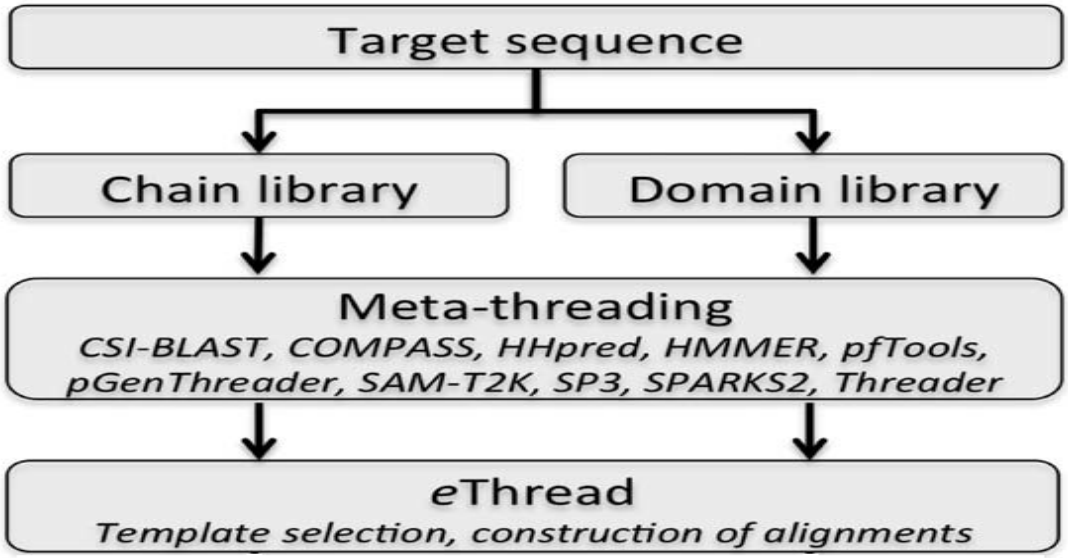

Figure 2.1: Flowchart of meta-threading pipeline of *e*Thread. Pipeline includes template identification by individual threading algorithms followed by target-to-template alignment using *e*Thread. Partial representation of the complete flowchart from [4]

structurally similar templates in two libraries, which consists of full protein chains and individual domains. In threading individual domains are included to improve the recognition of the templates that may only partially cover a multiple-domain target [26]. Also, if a full chain template is found, it provides information about the mutual orientation of domains [4]. The identified templates are subsequently filtered by *e*Thread and the corresponding target-to-template alignments are constructed. The optimization of *e*Thread pipeline using pilot framework is implemented till the meta-threading process. Later stages of the alignment, followed by inter-residue contact prediction, 3D structure modeling and model ranking are not the focal point in this dissertation.

2.3 Threading Component Methods

*e*Thread is a meta-threading procedure which combines prediction from ten state-of-the-art single-threading algorithms: CS/CSI-BLAST [27], HHpred [28], HMMER [29], pfTools [30], pGenThreader [31], COMPASS [32], SAM-T2K [33], SPARKS [26], SP3 [26], and Threader [34]. It also uses PSIPRED [35] for secondary structure prediction and NCBI BLAST [36] for sequence profile construction. Table 2.1 provides the function

Table 2.1: Components of *e*Thread pipeline for protein threading and sequence analysis

Threading Tool	**Purpose**
COMPASS	Protein threading/fold recognition
CS/CSI-BLAST	Protein threading/fold recognition
HHpred	Protein threading/fold recognition
HMMER	Protein threading/fold recognition
NCBI BLAST	Sequence alignment
pfTools	Protein threading/fold recognition and motif recognition
pGenThreader	Protein threading/fold recognition
PSIPRED	Secondary structure prediction
SAM-T2K	Protein threading/fold recognition
SPARKS/SP3	Protein threading/fold recognition
Threader	Protein threading/fold recognition

of each of the programs used in *e*Thread. Each individual threading/fold recognition algorithm assesses structures in the template library using a scoring system. For example, SP3, SPARKS and Threader assign Z-scores using the entire template library as a background. COMPASS, CSI-BLAST, HMMER and SAM-T2K employ analytically estimated E-values whereas HHpred uses calibrated probabilities for true relationship between proteins. For template selection, *e*Thread was constructed using Support Vector Machines for classification problems (SVC) to assess whether a specific template is structurally related to the target with a TM-score of > 0.4. It is observed that the template structures above this value contain sufficient information to enable the full-length reconstruction of the target structure [37].The accuracy of the template selected is evaluated using 2-fold cross validation excluding those templates whose sequence identity to target is $> 40\%$.

2.4 Computational Requirements of Threading Tools

The meta-threading pipeline consists of heterogeneous set of algorithms and it poses significant challenges for implementation and optimized utilization of the resources. The computational load would vary for each algorithm in terms of time to completion, memory, I/O operations and network bandwidth utilization. Some tools do not share the common input files or access the external data libraries. The threading algorithms quite often employ complicated dependencies between the individual tasks. Present

solutions to these meta-threading tools provide pseudo-gateways such as web interfaces that query several public servers.

An attempt to use a dedicated HPC machine with local installation of all the packages and software has been tried earlier [6]. Though it provides a reliable solution, not many are equipped with powerful resources to implement it in full scale. An alternative approach was taken to implement the meta-threading pipeline on a cloud platform. It provides a flexible and cost effective infrastructure solution. Further, the resource-on-demand option by the cloud providers avoids the queue wait time which is faced while using public shared scientific computing resources. These factors made cloud technology a viable option for *e*Thread application.

2.5 Experiment Testing Infrastructure

*e*Thread pipeline was implemented on cloud infrastructure. Cloud computing environment offers infrastructure service as pay per use model. It provides abundant resources immediately on demand. It also helps to scale up or down to accommodate changing computing requirements. Cloud infrastructure provides diverse types of instances for different computing needs. It is very advantageous to run heterogeneous tools on various tailored platforms rather on a single dedicated hardware.

Specifically, Amazon EC2 cloud infrastructure was used for our experimentation. It offers Amazon machine Images (AMIs) which are the pre-configured templates for the instances. It enables faster creation of instances and in creation of images with the required tools.

Table 2.2: EC2 Instance Types Chosen for the experimentation and their specifications

Instance	**Type**	**Number of Cores**	**Memory(GB)**
t1.micro	economic	1	0.613
m1.small	General Purpose	1	1.7
m1.medium	General Purpose	1	3.7
m1.large	Memory optimized	2	7.5
c1.medium	Compute optimized	2	1.7
c1.xlarge	Compute optimized	8	7.0
hi1.4xlarge	Storage optimized	16	60.5

Amazon EC2 provides varied instance types which fit different purposes. The instance types are classified based on varying CPU, memory, storage and networking capability. Users are free to choose any instance combinations for their needs. For our experimentation five types of VMs were chosen based on the results from previous study [6] of characterization of the threading algorithms. Economic, general purpose, memory optimized, compute optimized and storage optimized were the instance types chosen based on memory and storage requirements of the threading tools. Economic type of instance used was t1.micro. This is very low cost instances which provides burst of CPU performance for a short period of time. The general purpose instances chosen were m1.small and m1.medium. Memory optimized instance chosen was m1.large. C1.medium and c1.xlarge were the compute optimized instances. Finally, the storage optimized instance chosen was hi1.4xlarge. The specification of each instance chosen is listed in the table 2.2.

2.6 Pilot-Jobs

Pilot-Job is a kind of multi-level scheduling mechanism which manages the workload submission to a resource. The structure for a pilot-job framework was proposed in P* model [38]. According to P* Model, pilot-jobs enable utilization of a placeholder job as a container for a dynamically determined set of compute tasks. They are used in distributed computing for scheduling tasks at multiple levels possibly to heterogeneous systems. They provide an efficient abstraction for dynamic execution and utilization of a dynamic resource pool. They are very helpful for decoupling task submission from resource assignment. These tasks could be a single task, set of independent sub-tasks forming a bag of tasks, or a set of dependent tasks forming an ensemble. Thus, they effectively reduce the queue wait time in distributed HPC machines.

Pilot-Jobs also relax the user from the challenge of mapping specific tasks to specific resource in a heterogeneous environment. Some of the existing pilot-job frameworks are BigJob [14], Condor-G/Glide-in [39], Swift [40], DIANE [41], DIRAC [42], Falkon[43], PanDA [44], ToPoS [45], Nimrod/G [46] and MyCluster [47]. Utilization of

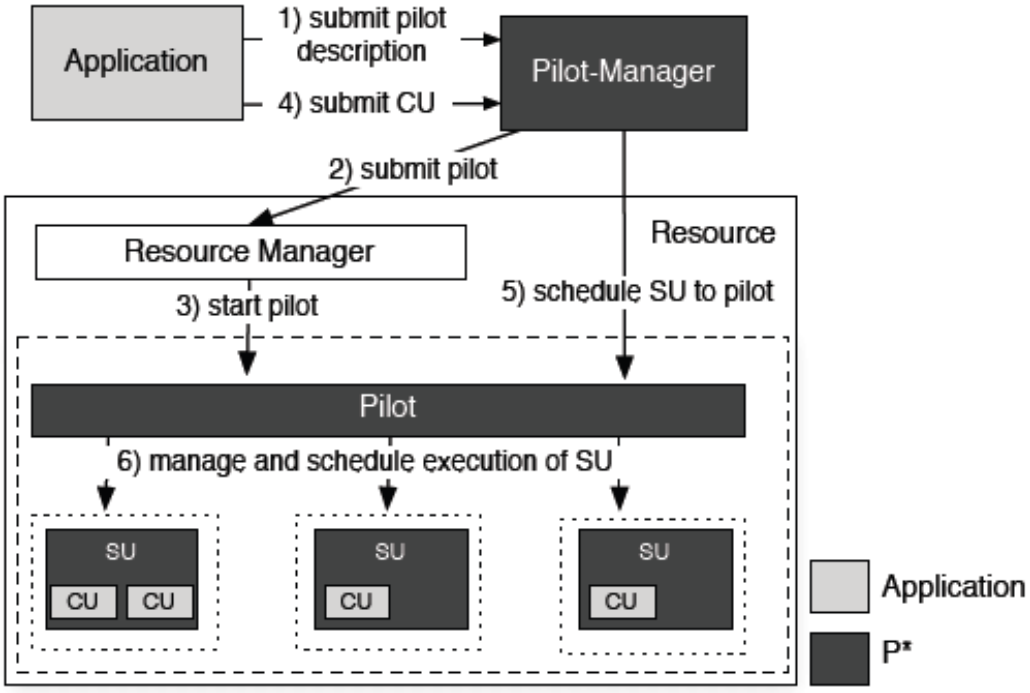

Figure 2.2: Elements, Characteristics and Interactions of P* model. Pilot Manager manages Pilots and execution of CUs. CU submitted to PM becomes SU, which is scheduled to a Pilot by the PM [38]

distributed cyber-infrastructure in an efficient manner is essential in a distributed application such as *e*Thread pipeline. Pilot abstractions aid in effectively decoupling the compute oriented tasks and associated data management. This alleviates the burden of the application to confine to a particular resource for scheduling compute and data units.

In order to proceed further with the understanding of the pilot-jobs and their use in applications like *e*Thread, it is helpful to understand few standard terminologies that will be dealt in later chapters. The P* model provides a detailed description of the pilot-job abstractions which can be used as a conceptual model for different pilot-job frameworks. The elements and characteristics of the P* model are stated below.

- **Pilot (Pilot-Compute)** is the actual unit that gets submitted or scheduled on the target resource. It enables the user to control and manage the allocated resources.

- **Compute Unit (CU)** encapsulates the compute task defined by the user which gets submitted to the framework. There is no view of the allocated resource at the CU.

- **Scheduling Unit (SU)** Once the CU is submitted to the pilot-job framework, it is assigned to the SU which is internal to the P* model. It is not visible to the application or user.

- **Pilot Manager (PM)** is the master entity that is responsible for the management and coordination of the different components in the P* model. PM handles the decisions related to resource management between CUs once assigned to the pilot; it manages the number of resources assigned to SU and their internal grouping. It is responsible for scheduling the SU on to a pilot and then onto a target physical resource.

The interactions between the elements of the P* model are determined by Coordination characteristics. The properties of affinity, i.e. early and late binding between the SU and pilot are determined by the Scheduling characteristics.

2.7 Pilot-Data

In distributed systems it is essential to manage the data movement and interaction within the application. Pilot-Data [48] is an extension of the Pilot-Job abstraction which helps in the management of the data movement in conjunction with the CUs. The pilot-data of the P* model addresses the data placement issues and the complexity that arise due to the heterogeneous nature of the existing storage and file-system types. Pilot-Data abstractions are analogous to that of Pilot-Job. The elements of the Pilot-Data are:

- **Pilot (Pilot-Data)** acts like a place holder object for the data units. It is similar to the pilot in the compute model managing the data placement in resource.

- **Data Unit (DU)** It is the actual data that interacts with the application.

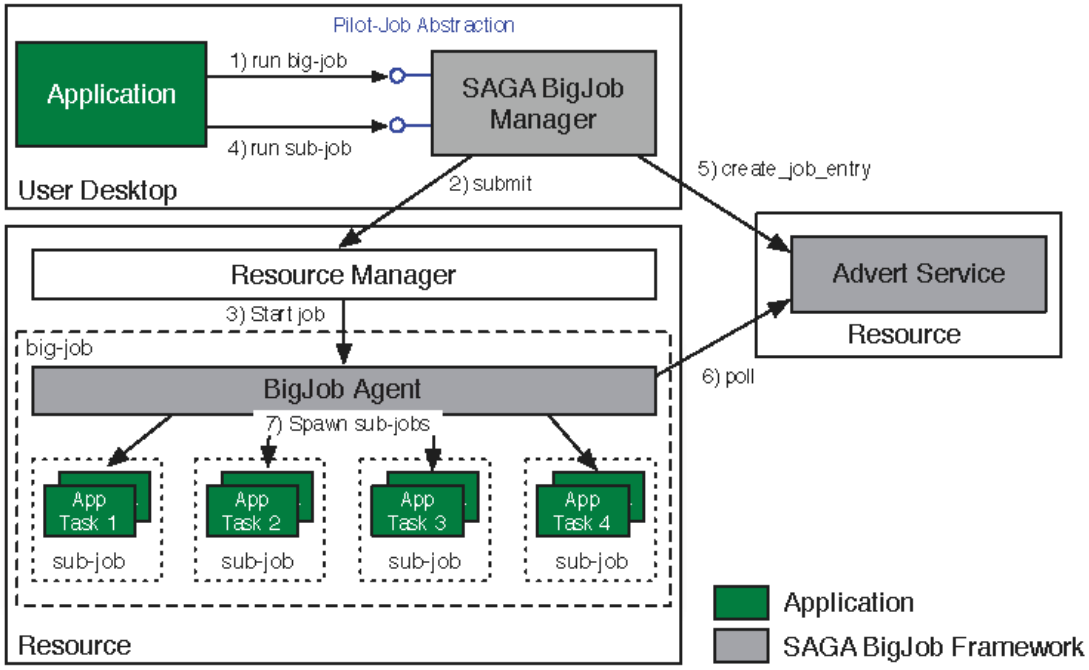

Figure 2.3: Architecture of BigJob. BigJob Manager manages the sub-jobs via BigJob Agent with the help of SAGA job and file API. BigJob Agent monitors and manages the sub-jobs [14]

- **Scheduling Unit (SU)** is the internal unit managing data unit scheduling to one or more SUs.
- **Pilot Manager (PM)** is the central entity similar to the compute model managing the DUs and SUs. Once the DU gets attached to the framework, PM handles the movement of the DUs from SU to the physical storage resource.

The coordination and scheduling characteristics in P* compute model and the Pilot-Data model remain the same. The scheduling characteristics are crucial in terms of affined placement of data units to correspond to specific CUs. The complete illustration of the P* model is shown in figure 2.2.

2.8 BigJob

BigJob [14] is a SAGA (Simple API for Grid Applications) based classic pilot-job framework which follows a Master-Worker coordination model. It is a high-level and

easy-to use API for accessing distributed resources and manages job submission, monitor, and more. BigJob works independent of the underlying resources which makes it easy to work on heterogeneous platform. It has been demonstrated for efficient executions of loosely coupled and embarrassingly parallel applications on distributed cyber-infrastructure (DCI). This flexibility and provision to execute complicated workflows makes it a suitable candidate for *e*Thread. Figure 2.3 shows an overview of the SAGA BigJob implementation. The three major components of the BigJob framework reflect the components of the P* model.

- **BigJob-Manager** is responsible for the coordination and management of pilots (Pilot-Compute and Pilot-Data) which run on remote resources to run assigned tasks. BigJob-Manager maps a data unit to a compute unit. BigJob is built upon SAGA Job API which invokes SAGA adaptors for submitting jobs to the resources while hiding all details to BigJob level API.

- **BigJob-Agent** is responsible for gathering local information and for executing the compute unit(s) and placing the data units appropriately on the resource where the tasks are submitted.

- **Advert-Service** employing a redis server, helps in coordination and communication to facilitate the control flow and data exchange between BigJob-Manager and BigJob-Agent.

2.8.1 Cloud BigJob

BigJob differs in terms of execution with respect to cloud infrastructure. The user level job contains the container with the description of the target and not the description of the workload. So, the physical resources are provisioned to the container. The Cloud BigJob ensures basic fault tolerance methods. It waits till the confirmation of allocation of resource (requested instance start up) is received and then sets up the SSH keys. Once the preparation steps are successfully completely BigJob starts executing the task.

2.9 Workload Management

Application workload management is provided by Pilot-API as follows. Pilot-API comprises Compute-Unit and Data-Unit classes as primary abstraction. Using these, a distributed application can specify a computational task along with required input and output files. Once compute-units and data-units are submitted, they are queued at the redis-based coordination service which is processed recurrently by a scheduler. Importantly, BigJob-Manger's asynchronous interface allows the application to respond instantaneously without waiting for BigJob to complete the placement of Compute/Data Unit, which is critical for dealing with a large number of tasks. Unprocessed tasks are stored in a FIFO queue. Once the target resource is made available, the sub-tasks are assigned. For parallel tasks, BigJob-Manager uses a node-file and spawns the sub-tasks using SAGA job API and SSH adaptor.

Current BigJob implementation supports data management between tasks provisioned by Pilot. It utilizes S3 as data repository as default. Any task once completed deposits pre-defined output into S3 storage and the subsequent tasks locate the output if specified as its input.

The complete set of main classes exposed by the compute part of the API is described in the BigJob user manual [49]. The essential classes and attributes used for this experimentation are mentioned here. Pilot-Job and Pilot-Data classes are symmetric and are described adjacently.

- **PilotCompute (PC):** A pilot-job, which can execute the compute workload (ComputeUnit). There can be any number of PilotComputes based on the number of resources available or required.
- **PilotComputeDescription (PCD):** Description for specifying the requirements of a PilotCompute. The python dictionary mandates following parameters:
  - Service_url: Specifies the SAGA-Python job adaptor
  - Number_of_processes: The number of cores that need to be allocated to run the jobs.

The parameters mentioned below are Amazon EC2 cloud specific attributes:

- Vm_id: Template Instance ID from which the VM has to be created
- Vm_ssh_username: username for the VM Authentication
- Vm_ssh_keyname: Authentication word stored at Amazon account
- Vm_ssh_keyfile: Location of the certificate authentication file (could be SSH or pem file)
- Vm_type: The instance type to be created
- Region: region for the AWS instance to be located
- Access_key_id: The username for Amazon AWS compliant instances.
- Secret_access_key: The password for Amazon AWS compliant instances.

- **PilotComputeService (PCS):** A factory for creating PilotComputes. It takes Coordination URL as an argument.
- **PilotDataService (PDS):** A factory (service) which can create PilotData according to the specification. It takes coordination url as an argument.
- **PilotData (PD):** A pilot that manages data workload (DataUnit)
- **PilotDataDescription (PDD):** An abstract description of the requirements of the PD. It requests resources required to run all sub-jobs. The python dictionary requires the following parameters to access the cloud storage resource:
  - Service_url: Specifies the file adaptor and target resource hostname on which a Pilot-Data will be created
  - Access_key_id: The username for Amazon AWS instance.
  - Secret_access_key: The password for Amazon AWS instance.

The actual job is represented by ComputeUnits and DataUnits:

- **ComputeUnit (CU):** A work item executed on a PilotCompute.
- **DataUnit (DU):** A data item managed by a PilotData

Compute and Data Units are specified using an abstract description object:

- **ComputeUnitDescription (CUD):** abstract description of a ComputeUnit. The python dictionary requires following arguments. The arguments can take direct input or as environment variables.
  - Executable: Specifies the path to the executable that will be run
  - Working_directory: The working directory for the executable
  - Arguments: Specifies any arguments that the executable needs
  - Number_of_processes: Defines how many CPU cores are reserved for the application process
  - Input_data: Specifies the input data flow for a ComputeUnit. This is used in conjunction with PilotData
  - Output_data: Specifies the output data flow for a ComputeUnit. This is used in conjunction with PilotData
  - Output: Specifies the name of the file who captures the output from <stdout>
  - Error: Specifies the name of the file who captures the output from <stderr>
- **DataUnitDescription (DUD):** Abstract description of a DataUnit. The data unit description defines the different files with their location to be moved around as a python dictionary.

2.10 Benchmark Dataset

For the benchmarking experiments, 20 protein gene sequences whose length range from 50-600 amino acids were used. The information on sequence length and number of sequences used in the range is specified in table 2.3. These 20 sequences are the sub-set of the 110 benchmark sequences used in previous work by Brylinski et all [6]. The 20 sequences were used for the analysis of ten threading tools and two tools PSIPRED and BLAST and meta-analysis.

All these data sets were placed in Amazon S3 and the required sequences were placed in the experiment environment during runtime by PilotData. This enabled

Table 2.3: Benchmark dataset: Test Sequence length range and number of sequences

Sequence length	**Number of sequences**
50 - 100	2
100 - 150	2
150 - 200	2
200 - 250	2
250 - 300	2
300 - 350	2
350 - 400	2
400 - 450	2
450 - 500	1
500 - 550	1
550 - 600	2

better storage utilization as the time for the data movement was negligible compared to the runtime completion.

Chapter 3

Pilot based *e*Thread Implementation

In chapter 2 we discussed the theory behind the tools required to build the *e*Thread application. In this chapter we provide the details of the BigJob based *e*Thread python program followed by the high-level design and workflow of the *e*Thread module.

3.1 BigJob-*e*Thread Python Module

At the heart of the implementation of the BigJob based *e*Thread module is a python program which enables the execution of the workflow. Since the individual protein threading tools are not dependent on each other, each protein threading module can be concurrently executed. So, the python program is multi-threaded to enable concurrent execution of the ten protein threading tools. There are ten threads launching each individual protein threading tool to the respective Amazon EC2 instance. The python *e*Thread program works in conjunction with a JSON interface to fetch the user input. The necessary installations required for the execution of the *e*Thread program is explained in appendix A.

3.1.1 JSON Configuration Input

The input to the program is passed through a JSON Configuration file. This enables to change the required parameters easily. The parameters that are set through the Config file are mentioned as follows:

- programs: A list of names of the threading tools that are present
- Programs_to_omit: A list of names of the threading tools not to be executed. It provides flexibility to the user to choose to run only the required threading tools,

if necessary.

- Coordination_url: The redis server location co-ordination URL to be used by BigJob
- Pcd_common: A dictionary of the pilot compute description parameters which are common across pilot for each threading tool
  - Service_url
  - Number_of_processes
  - Vm_ssh_username
  - Vm_ssh_keyname
  - Vm_ssh_keyfile
  - Region
  - Access_key_id
  - Secret_access_key
- Vm_id: A dictionary of the AMI ID to create the instance for each of the individual threading tool
- Vm_type: A dictionary of the instance type to create for each of the individual threading tool
- Input_file_path: A list of the paths where the input files are placed for pilot data to collect it and place it in the working directory of the task
- Dataset: A list of amino acid sequences used for the experiment
- Output_file_path: Path where the required output files need to be stored in a local location
- Cud_common: A dictionary of the Compute Unit Description parameters that are common across tasks

  Working_directory: Location on the remote machine where the tasks are to be executed

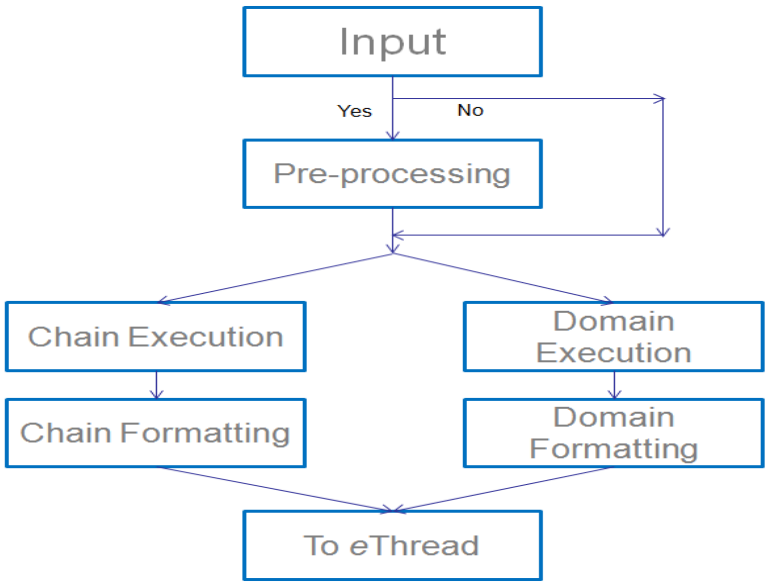

Figure 3.1: Generic Workflow of individual threading tools. Input sequence undergoes pre-processing step if necessary. It is followed by independent chain and domain execution and formatting.

3.2 *e*Thread Workflow

One of the significant outcomes of this dissertation is the pilot-job based *e*Thread python program to submit the jobs to Amazon EC2 cloud. *e*Thread has ten protein threading tools that are not dependent on each other and can be run in parallel. The python program developed is a multi-threaded program which launches all the ten individual protein threading/fold recognition tools in parallel. The inputs given are the amino acid sequences. The individual threading algorithms have a pipeline of sub-processes. First is the pre-processing for certain protein threading tool, then the main processing or the execution phase and the post processing or the formatting phase. The generic workflow of the individual threading tools in the *e*Thread pipeline is shown in figure 3.1 and the overall experiment workflow DAG is shown in the figure 3.2.

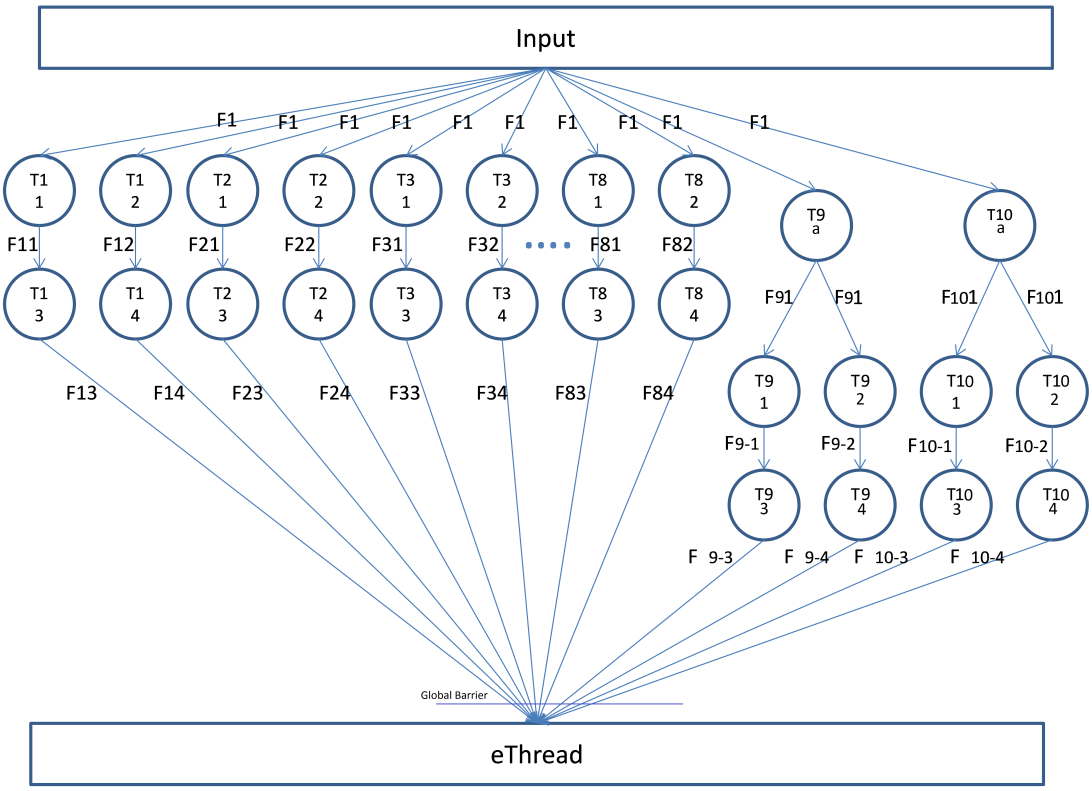

Figure 3.2: Workflow DAG of *e*Thread meta-threading pipeline. The vertices in the graph are the tasks in the workflow which are represented by 'Txy' and edges represent the data dependencies between the tasks in the workflow represented by 'Fxy'.x-'N'th VM, y-chain/domain execution/formatting task

3.3 High-level Design of *e*Thread Module

The *e*Thread program is simple and straight forward. There are twelve modules for threading and sequence alignment programs. Csblast(), pftools(), hmmer(), pgenthreader(), threader(), compass(), hhpred(), sparks(), sp3() and samt2k() are the protein threading tool modules. Psipred() is a prerequisite to run threader() module and blast() is prerequisite to run samt2k() module. These two set of tools are executed in different VMs as the computational requirements of the pre-requisite modules and the protein threading modules are different which is discussed later in chapter 4. But, they are run in sequence in single thread. Ten threads are executed in parallel. Each thread launches a pilot for executing a protein threading tool. Each pilot creates a new instance (created by Cloud BigJob object) specific to the protein threading tool. After the completion of the execution, each module's results are used as input for *e*Thread()

```
pilot_compute_description = {
                            "service_url": str(json_parse['service_url']),
                            "number_of_processes": int(json_parse['number_of_processes']),
                            "vm_id": str(self._cfg_data['vm_id']['csblast']),
                            "vm_ssh_username":str(json_parse['vm_ssh_username']),
                            "vm_ssh_keyname":str(json_parse['vm_ssh_keyname']),
                            "vm_ssh_keyfile":str(json_parse['vm_ssh_keyfile']),
                            "vm_type":str(self._cfg_data['vm_type']['csblast']),
                            "region" :str(json_parse['region']),
                            "access_key_id":ACCESS_KEY_ID,
                            "secret_access_key":SECRET_ACCESS_KEY
                           }
  pilotjob = self.pilot_compute_service.create_pilot(pilot_compute_description=pilot_compute_description)
```

Figure 3.3: Pilot compute creation: Instantiation of pilot compute using pilot_compute_description.

module which performs template selection and construction of alignment.

3.4 Implementation of Protein Threading Module using BigJob

In each of the protein threading tool, a pilot is launched using Cloud BigJob object. Once the pilot is launched, it creates an instance using the specifications from the *pilot_compute_description* provided by the user in the JSON file. A typical request for an instance looks like the one shown in figure 3.3. The information can be a direct string or environment variable and it can also accept variables. Authentication key information required for a creation of VM is provided here. The resource details are provided only to the pilot. The compute unit is not exposed to the resource information except for the number of cores the compute unit requires.

In this way the infrastructure details are isolated from the tasks to be executed. It makes the user easily concentrate on executing the tasks and also makes the portability of the application from one resource to the other much easier.

Once the pilot-job is instantiated, many compute units (where actual tasks are performed) can be created in the container job. For each step in the workflow of the protein threading tool shown in figure 3.1, one *compute_unit* is created. The creation of the *compute_unit* is shown in figure 3.4.

The *ComputeUnitDescription*(CUD) has the information about the executable, number of cores required to execute the task (if the application is MPI based, it can use multi-cores), link to the S3 bucket location where the required input files are placed and a link to the S3 bucket location where the required output files need to be placed.

```
compute_unit_description = {
    "working_directory": str(self._cfg_data['cud_common']['working_directory']),
    "executable": "/home/ethread/apps/csblast-2.1.0-linux64/csblast_static",
    "arguments": [" -i "+dataset+".fasta -d /home/ethread/libs/ethread-lib-latest/blast/chain -D
    /home/ethread/apps/csblast-2.1.0-linux64/K4000.lib -j 5 -h 0.002 -e 100.0 --blast-path
    /home/ethread/apps/blast-2.2.25/bin -o "+dataset+"-chain.csblast"],
    "number_of_processes": 1,
    "input_data": [self.input_data_unit.get_url()],
    "output_data": [
                {
                        output_du_chain[i].get_url():

                        ["stdout_"+dataset+"_chain_blast.txt","stderr_"+dataset+"_chain_blast.txt",dataset+"
                        -chain.csblast"]
                }
    ],
    "output": "stdout_"+dataset+"_chain_blast.txt",
    "error": "stderr_"+dataset+"_chain_blast.txt"
}
task = pilotjob.submit_compute_unit(compute_unit_description)
```

Figure 3.4: ComputeUnit creation using ComputeUnitDescription

Here, the CUDs are appended in a list for the total number of amino acid sequences to be analyzed. Equivalent number of *compute_units* are created for the list of *compute_unit_description*. The chain and domain tasks are not dependent on each other. So, the chain execution step and domain execution step are executed concurrently.

Once the execution step is finished successfully, the output files are transported to the compute units of the respective chain and domain formatting step by pilot-data. Again, chain and domain formatting are executed in parallel. After successful completion of the formatting, the result files are placed in a specific S3 bucket, to be used by *e*Thread() module for template alignment. This process is similar for all protein threading tools, except for few tools which have an additional pre-processing step before the chain/domain execution steps. In that case, the pre-processing step is executed first followed by the chain and domain processing.

After completion of all single-threading algorithms, another pilot is created to execute the *e*Thread() module. It acquires the output files from all the previously executed protein threading tools and uses it as input to perform the alignment. This summarizes the implementation of the *e*Thread pipeline using BigJob.

Chapter 4

Results and Discussion

In chapter 3 we presented the implementation details of the *e*Thread program and the high-level workflow. In this chapter we discuss the set of experiments performed and analyze their results.

Some characteristics of the applications require usage of Distributed cyber-infrastructure which are designed to support peak utilization under varying load conditions. It is essential to understand the behavior of the application in order to utilize the resources optimally. Otherwise, it will be time consuming to execute the tools and will not be economic, thus making it impractical to use.

There are three modes of implementations handled for building the task-resource mapping for *e*Thread pipeline that are discussed further. The experiments were performed on Amazon EC2 cloud infrastructure. Use of different instance types were studied according to the computational requirements of each protein threading tool. Tasks were submitted to the cloud VMs using BigJob. The benchmark was performed for all the ten threading tools and two standalone tools.

4.1 Task-Resource Mapping

There are three combinations of tasks to resource mapping that have been implemented. They can help towards the best mode of usage of the allocated resources. They help in increasing the scale of operation, easily extending the functionality to a new module and reuse the patterns and abstractions for any new infrastructure. The usage modes that are implemented are

i. Homogeneous tasks on homogeneous VMs

ii. Heterogeneous tasks on homogeneous VMs

iii. Heterogeneous tasks on heterogeneous VMs

Category (i) is a simple bag of tasks assigned to one instance type. Here, the same task is executed multiple times with different input files. This has been implemented as a first step in *e*Thread module. Each sub-job in a pipeline of tasks has been executed for multiple input sequences.

Category (ii) is a again a bag of tasks, but proceeding a step further, has different set of homogeneous tasks working together to form a workflow. It is implemented in *e*Thread as chain and domain execution tasks working in parallel for set of amino acid sequences. These parallel sub-jobs are followed by a pipeline of tasks forming a classic heterogeneous set of tasks. This kind of heterogeneous set of tasks is recurring across the different protein threading tools and each protein threading tool is submitted to similar kind of instances. Also, as a part of experimentation, sub-jobs of each threading tool are simultaneously submitted to multiple homogeneous instances to understand if distributing the tasks across instances is beneficial.

Finally, category (i) and (ii) lead towards category (iii), where the heterogeneous tasks forming the protein threading tool are executed in different types of instances simultaneously. Thus is completes the *e*Thread meta-threading pipeline module. It also includes few special set of tasks like in Threader and SAM-T2K, where a pre-requisite step is implemented in a different instance and the main execution is implemented in another instance.

4.2 Profiling of Meta-Threading Components for Different EC2 Instance Types

There is a significant difference between each individual protein threading tools of the *e*Thread pipeline in terms of TTC and memory utilization. It is important to gain insight on relative computing loads across all the protein threading tools against all the instances chosen. TTC varies considerably between the VM types due to the difference in the configurations mentioned in table 2.2.

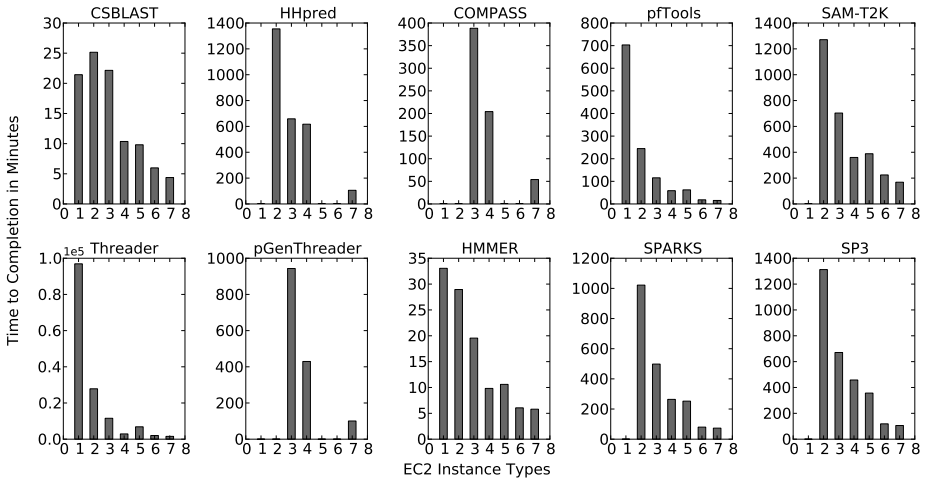

Figure 4.1: Average TTC (in minutes) of the ten individual threading tools for 20 amino acid sequences in different instance types. It is seen that Threader takes the maximum TTC. Along x-axis of each subplot, 1:t1.micro, 2:m1.small, 3:m1.medium, 4:m1.large, 5:c1.medium, 6:c1.xlarge, 7:hi1.4xlarge

The average TTC and memory footprints are shown in figure 4.1 and 4.2. The input sequences are 20 amino acid sequences. Figure 4.1 shows the average TTC for each individual threading component in each VM type. An expected speed-up in instances that use multi-core is observed across all the threading tools. It is observed that the average TTC of CSBLAST and HMMER across the seven instance types is much smaller when compared to other threading tools. Threader tool takes the maximum TTC. In few tools like pfTools, SAM-T2K, Threader and HMMER TTC is larger in c1.medium than m1.large though both have two cores. This behaviour is observed because, c1.medium instance is compute optimized and have only 1.7 GB RAM memory. Whereas m1.large is memory optimized and has 7.5 GB RAM memory. Few other tools like HHpred, COMPASS and pGenThreader were highly memory intensive and were chosen not to be executed on c1.medium and other smaller memory instance types.

A note about the behaviour of CSBLAST on t1.micro and m1.small instance has to be considered. t1.micro is small machine with 0.613 GB RAM and are suited for small throughput applications and m1.small is memory optimized with 1.7 GB RAM.In Figure 4.1, when comparing the TTC of t1.micro and m1.small of CSBLAST, it is seen that t1.micro completes faster than m1.small. Upon investigation, it is seen that

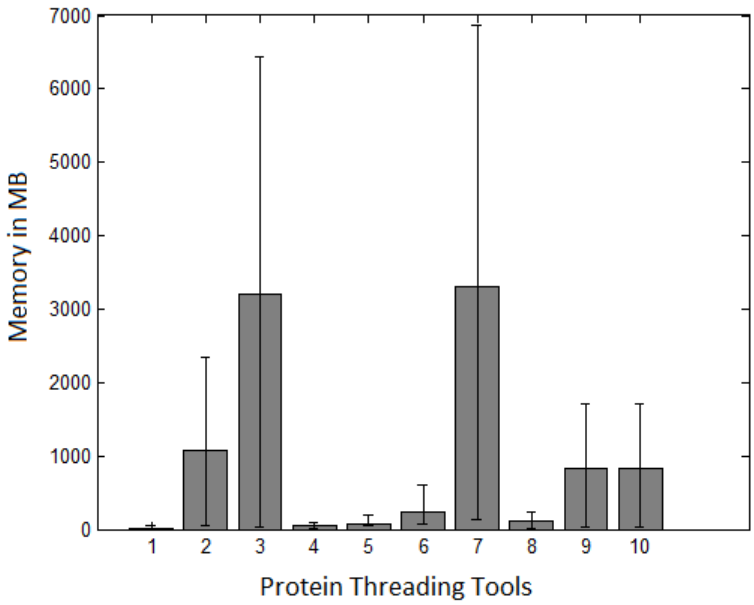

Figure 4.2: Average Memory Consumption of different threading tools in m1.large instance using 20 amino acid sequences. The error bar shows the maximum and minimum values for the largest and smallest length sequences. Along x-axis, the different threading tools represented are 1:CSBLAST, 2:HHpred, 3:COMPASS, 4:pfTools, 5:SAM-T2k, 6:Threader, 7:pGenThreader, 8:HMMER, 9:SPARKS, 10:SP3

t1.micro instances are allowed to operate at up to two EC2 Compute Units (ECU) (one ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor) whereas m1.small instances get a constant one ECU at all times [50]. T1.micro instances are designed to support tens of requests per minute from the application, but the actual performance depends on the amount of CPU resource used for each request. When there is a spike in activity for a short duration and the application requires more CPU resources, the instance utilizes up to two ECUs. If the CPU resource utilization continues to occur for longer time duration than desired (the duration is not known exactly) the instance is limited and it runs on a low CPU level. Figure 4.3 shows the average CPU utilization of both t1.micro and m1.small. As seen, t1.micro completely utilizes the CPU resources for a short duration of time. So, even though m1.small has more memory than t1.micro, as CSBLAST is not a memory intensive task and due to the short TTC, t1.micro is observed to complete faster.

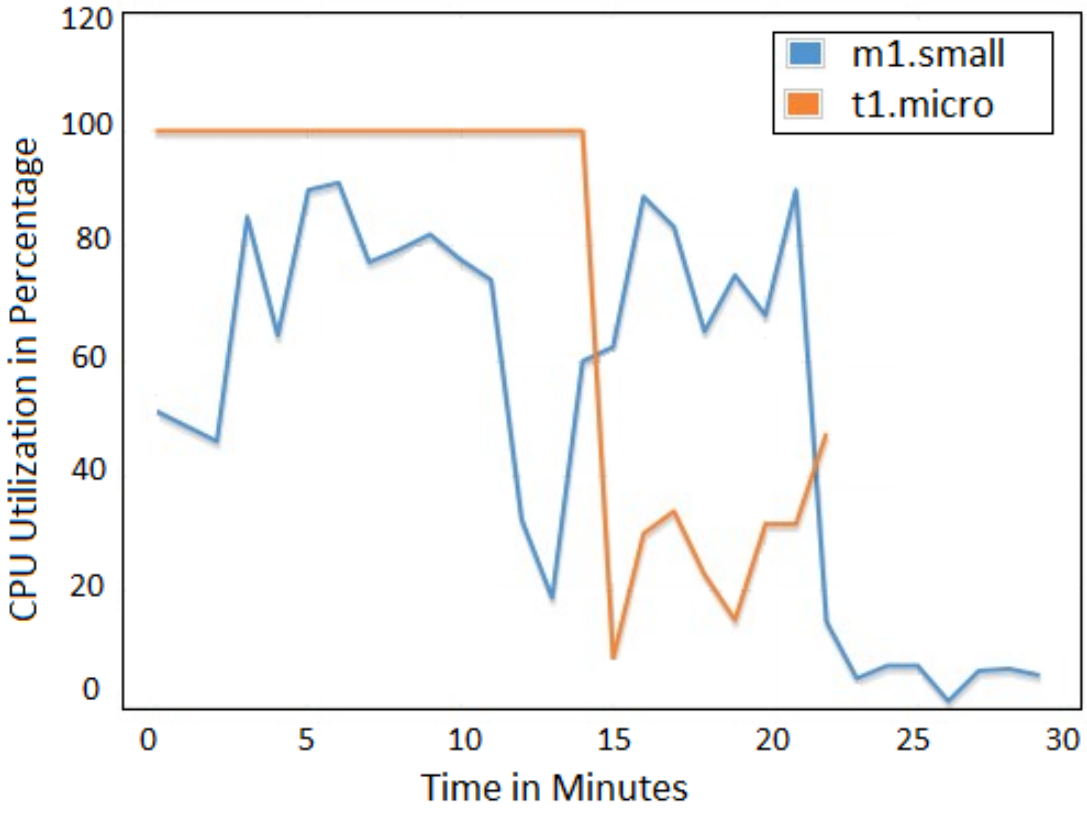

Figure 4.3: Average CPU utilization of t1.micro (orange) and m1.small (blue) while running CSBLAST for 20 sequences. X-axis shows the TTC and y-axis represents CPU utilization. Chart obtained from Amazon Performance measurement toolkit during execution of tasks.

Individual threading components also differ with respect to memory utilization. Figure 4.2 shows the memory utilization of the threading algorithms in m1.large instance. COMPASS and pGenThreader are memory intensive and require a minimum of 3 GB RAM memory. HHpred, SPARKS and SP3 are also considerably heavy on memory usage as they require more than 1 GB RAM. HHpred, COMPASS and pGenThreader use PSI-BLAST for constructing sequence profiles and hence require more RAM memory. CSBLAST, pfTools, HMMER and Threader are very low on memory consumption. Memory utilization remains the same across different instance types for each threading tool. Hence the data for other instance types are not shown in figure 4.2.

When compared to previous study [6], SAM-T2k is observed to consume very low memory. SAM-T2k uses BLASTP for sequence alignment which loads a large sequence

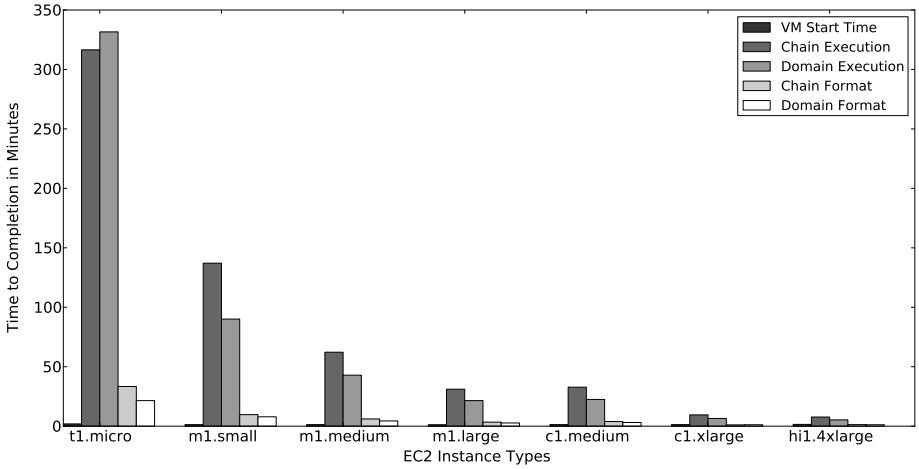

Figure 4.4: Time to Completion of the subtasks in the *e*Thread meta-threading pipeline for pfTools using 20 amino acid sequences in different instance types.

library into memory. The actual threading process requires only about 1% of the memory required by BLASTP. For the test sequences shown in table 2.3, BLASTP consumes around 5589 MB whereas SAM-T2K as such consumes 68 MB. Hence, in the cloud implementation, sequence alignment using BLASTP was isolated and was executed in a different VM and it was configured to use two cores during the experimentation. This enabled SAM-T2K to be run on instance types of smaller memory capacity. Similarly, Threader requires PSIPRED as a pre-requisite which has been modified to run PSIPRED in a separate instance and then run the main Threader execution, which occupies very low memory. This hints towards overall optimization with the task level decomposition.

Along with this, the main processing steps involve execution of chain and domain libraries in each of the threading tool. They are independent of each other and they can be run in parallel. The time taken to complete chain and domain tasks were measured. It was observed that most of the tools had chain and domain tasks having 60% and 40% ratio of completion time. Figure 4.4 shows the TTC of the subtasks for pfTools for all the instance types. However, there was one anomaly to this pattern, where pfTools domain library took more time to complete than chain library execution in t1.micro instance.

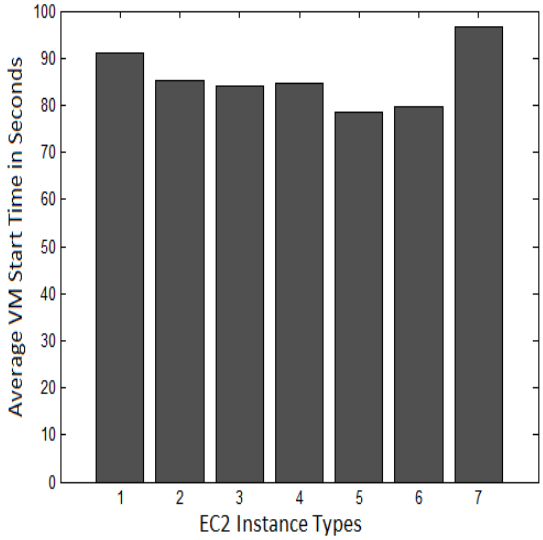

Figure 4.5: Average VM start time of different types of Amazon EC2 cloud instances. Along x-axis the different VM types are 1:t1.micro, 2:m1.small, 3:m1.medium, 4:m1.large, 5:c1.medium, 6:c1.xlarge, 7:hi1.4xlarge

4.3 Cloud-BigJob Performance Overload

Table 4.1: Comparing VM launching time using BigJob and TTC for all the jobs submitted on m1.large instance through BigJob using for 20 amino acid sequences. Time is measured in minutes

Protein Threading Tool	**VM Start time**	TTC for all jobs
CS/CSI-BLAST	1.75	10.37
HHpred	2.7	617.08
COMPASS	2.62	204.15
pfTools	1.24	58.61
SAM-T2K	1.74	359.83
Threader	2.6	2897.2
pGenThreader	1.24	429.48
HMMER	1.23	9.81
SPARKS	1.25	263.9
SP3	2.64	458

Main performance overload while using BigJob with cloud environment is the VM creation time [14]. Figure 4.5 shows average VM start time for different instance types. Starting up a VM has higher overhead than spawning a job in already running machine. It involves creation of an instance from the template, wait for the booting of the machine and all the required processes to start and then wait for BigJob to communicate with

the machine and then submit the job and start execution. But as shown in table 4.1, when compared to the Total Time to Completion (TTC) of the jobs submitted by *e*Thread, the VM start up time was significantly small. This offsets the concerns about the overload caused in using cloud infrastructure. Also, in production applications, typically large number of sequences are processed which will make the overload due to VM start time insignificant even for threading tools like CSBLAST, HMMER and pftools which executes relatively fast.

Another overhead is the data-transfer that happens between the local resource and the remote machine where the tasks are executed. Also, the data-transfer that happens between the intermediate tasks of the protein threading acts as an overhead. In figure 3.1, The arrows show the data transfer happening between each stage. The intermediate data are exported by Pilot-Data to S3 external storage to save the memory in the working instance. They are again imported to the working directory when required. However, the data transfer is not a significant overload as it takes less than 1% of TTC to transfer the data.

4.4 Executing Single Threading Tool on Multiple Instances

An essential factor to consider when using large number of sequences is the natural increase in TTC. Hence, distributing the tasks on multiple homogeneous or heterogeneous resources will help to provide an optimal solution. This is the implementation of category (ii) and (iii) mode of usage discussed in section 4.1 , where mapping heterogeneous tasks to homogeneous/heterogeneous resources was presented.

4.4.1 Homogeneous Instances

In this parallelization experimentation, two instances were launched simultaneously using the pilot-job for each protein threading tool and 20 amino acid sequences were analysed. As the *e*Thread pipeline is independent for each sequence, multiple tasks can be executed in parallel for multiple sequences. In our experiments, based on the instance type, for each core in the launched instances, one task was executed. Figure

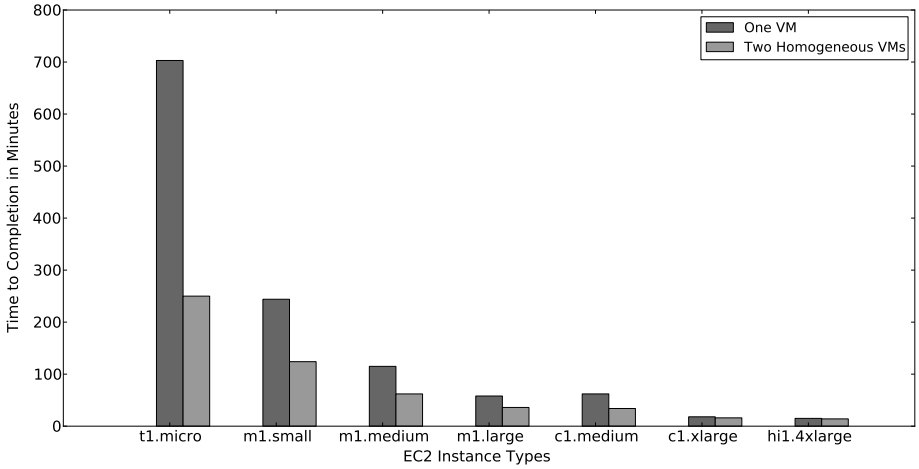

Figure 4.6: Time to completion for pfTools for 20 amino acid sequences using Single instance and two homogeneous instances

4.6 shows the comparison of TTC of the pftools using two instances and one instance for 20 sequences. PfTools was chosen as an example among the threading tools to present the analysis. There is no significant improvement observed for large instance types like c1.xlarge and hi1.4xlarge while using two instances as only 20 input sequences were used for test purpose. With more number of data sets, this multiple instances implementation will be very effective even for large instances to reduce the Total Time to Completion.

4.4.2 Heterogeneous Instances

Along with the homogeneous instances experimentation, usage of heterogeneous instance type was also analysed. Different combinations of t1.micro, m1.small, m1.medium, m1.large and c1.xlarge instances were employed to understand the variations which are presented in figure 4.7 for pftools in comparison with single and homogeneous multiple instances. This experiment tried to combine a single core instance with a multi-core instance and study different heterogeneous combinations. It is seen that pfTools takes the longest TTC using one t1.micro instance. When t1.micro is combined with m1.large, the heterogeneous combination is 82% faster when compared to using two t1.micro instances and 24% faster than using a single m1.large instance suggesting to be effective

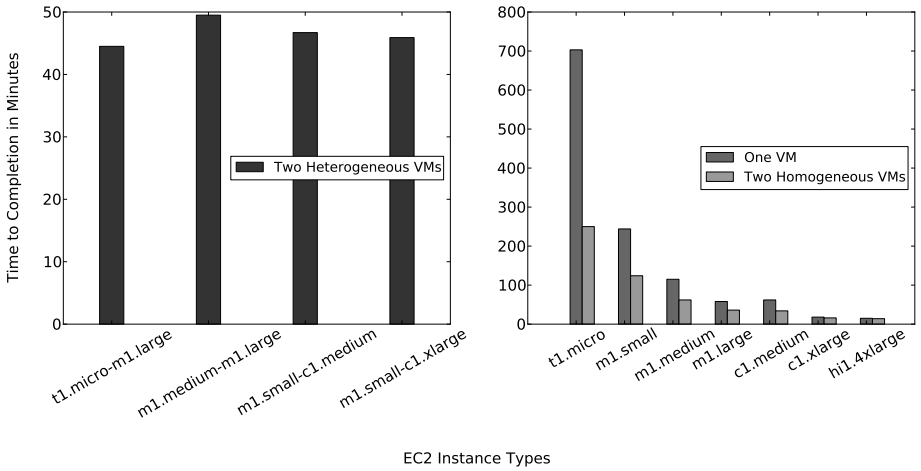

Figure 4.7: The graph on the left is Total Time to Completion of pfTools for 20 sequences using heterogeneous mix of two instances. On the right, the graph is a comparison of single instance and two homogeneous instances

in terms of time and cost. Yet, the choice of combination has to be carefully undertaken. For example, when c1.xlarge is combined with m1.small, it takes more time to complete than using a single c1.xlarge instance. Though additional core was used in the heterogeneous combination, sharing few tasks with another core in another physical machine has shown to be detrimental.

4.5 Cost of Running *e*Thread on Cloud

There are multiple ways the research community has developed funding and usage models for their computational and storage needs. Well funded projects have their own clusters while other look for options like campus/national grids, compute time allocations to shared clusters (TeraGrid) and some share their resources with others in the community (OSG). Each of these options has its own pros and cons. It might be very expensive to own a local cluster which might not be used optimally at time, or there could be situations where required resources are not allocation in a shared cluster due to heavy demand. A new addition to these, the cloud computing models are coming to the limelight. Amazon Web Services is among the first to provide commercial computational and storage cloud resources on a pay per usage basis. They provide resources on-demand

and enable customization of the environment based on the applications needs. In turn, the vendors charge according to usage based on a fee structure. This model can be very attractive and affordable as it eliminates the initial huge investments, operating and maintenance cost involved in owning a computing resource and eliminates the huge wait time typically incurred in using shared resources.

Table 4.2: Pricing of AWS EC2 instances per hour as of writing the thesis

Instance Type	**Cost($/hr)**
t1.micro	0.02
m1.small	0.06
m1.medium	0.12
m1.large	0.24
c1.medium	0.145
c1.xlarge	0.58
hi1.4xlarge	3.1

In this section, the computing cost of using Amazon EC2 for *e*Thread meta-threading pipeline will be examined. The storage and communication cost are not explored in this study for two reasons. The data exchange between the cloud resources, EC2 and S3, are free and the data stored in S3 is not long term. During the period of experimentation, the cost of different instance types per hour is stated in table 4.2. These values were used to estimate the cost-for-solution for the benchmark experiments conducted.

Figure 4.9 presents the prorated cost-for-solution of ten protein threading tools on each instance type used for it. The number of amino acid sequences used was 20. 'N'cores available in each instance was utilized by executing N tasks in parallel. In terms of cost effective utilization, though t1.micro could be thought of as most viable option due to its cheapest pricing, the actual affordable instance in terms of optimized TTC and cost-for-solution happened to be c1.medium in most of the threading tools. C1.medium instance type is compute optimized and offers two cores. That makes it much faster when compared to other small instance types and it is not as costly as hi1.4xlarge instance thus striking a balance between time and cost. Time-to-completion figure has been included here for relative comparison. Nevertheless, this method of optimization has to be improved, as high computing resources like hi1.4xlagre machine could easily fall prey to being underutilized if the number of tasks is not in multiples of the number

Table 4.3: Comparison of TTC using pilot based *e*Thread pipeline and ideal limit derived by executing tasks without pilot and dividing the TTC by number of cores. The tasks were executed using 20 sequences. Unit of time is in minutes

Tool	**m1.small**		**c1.xlarge**		**hi1.4xlarge**	
	pilot	ideal	pilot	ideal	pilot	ideal
SAM-T2K	1270.97	1771.25	224.45	65.58	168.28	35.66
SP3	1312.25	1124.44	118.58	68.09	105.71	32.94
CSBLAST	25.15	15.44	5.98	1.23	4.38	0.47
HMMER	28.95	16.03	6.07	1.04	5.80	0.59
pfTools	244.77	225.64	18.27	12.75	15.48	9.24
Threader	27842.19	23744.35	2019.32	1487.98	1552.20	1090.41
SPARKS	1021.77	1037.67	79.97	54.28	73.27	41.79

of cores.

4.5.1 Alternate Scheduling Strategy

There is another approach that could be handled for cost and time optimization. It is known that the *e*Thread module has to wait for all the ten threading tools to complete the meta-threading and provide the results to start the alignment process. On observation, Threader tool takes the maximum TTC and cost-for-solution. The shortest time taken to complete Threader tool task by hi1.4xlarge (1552 minutes) is still larger when compared to all other tools longest TTC. It would be intuitive to choose m1.large kind of instance for Threader which is optimized for both TTC and Cost-for-Solution, and for the rest of the threading tools, choosing the cost effective solution. However, it should be noted that the price observed in the graph is not the actual cost as the pricing used for the calculations are prorated cost and also the pricing scheme are subject to change by Amazon.

Though BigJob offers parallel execution of multiple tasks, without an effective task planning and scheduling, the resources cannot be utilized to the maximum benefit. Table 4.3 compares the TTC of the current pilot-job based pipeline and a theoretical ideal limit. The ideal limit was derived by executing the tasks without pilot mechanism and dividing the TTC by the number of cores available to the instance. From the table it is evident that the existing pipeline has lot of room for improvement more so with resources of higher number of cores. It calls for better methods and algorithms for

scheduling tasks to the resources.

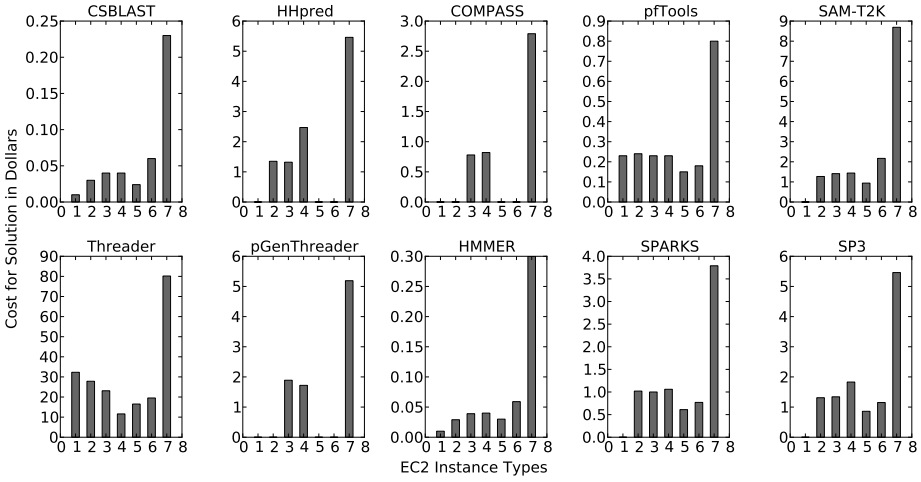

Figure 4.8: Prorated Cost-for-Solution in USD,Pricing is used from table 4.2

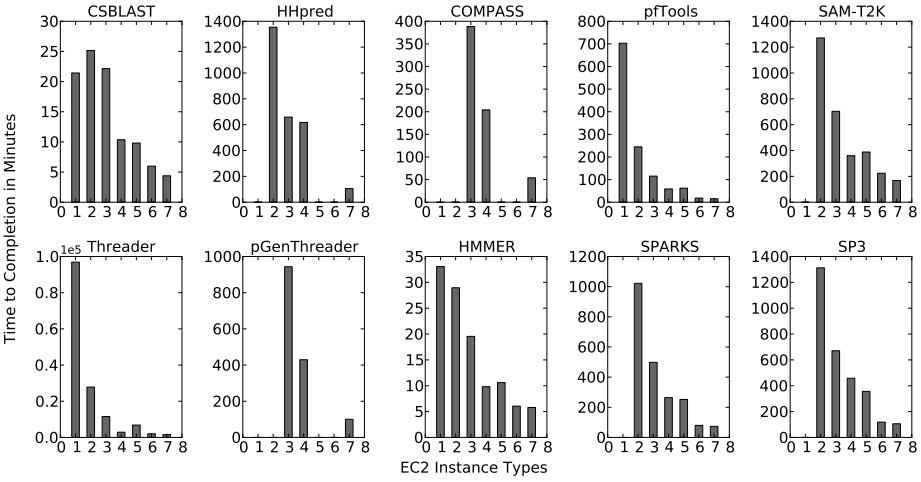

Figure 4.9: Average TTC (in minutes) of the ten individual threading tools for 20 amino acid sequences in different instance types

Chapter 5

Conclusion

Since *e*Thread combines heterogeneous algorithms, the optimal utilization of resources requires critical attention. In this communication, Pilot based *e*Thread pipeline was developed and an extensive profiling was performed on Amazon EC2 instances in conjunction with S3 as the data repository. In continuation to earlier work [6], in this study, multiple data-level and task-level parallelization experiments using pilot-jobs were performed to understand the optimal resource utilization for various kinds of task-resource mapping. A detailed time and cost based analysis was performed to understand the behavior of the *e*Thread pipeline.

From the results in figures 4.1 and 4.2 it is seen that the individual threading tools in the *e*Thread pipeline varies widely from each other in terms of Total Time to Completion and memory utilization. In this scenario utilization of heterogeneous resources has enabled better utilization of resources. Also, from the figure 4.7 it is seen that using right combination of heterogeneous resources for a single threading tool helps to reduce the overall TTC. Since the most of the individual protein threading tools have longer time to completion, an economic approach to the implementation of pipeline was essential. From figure 4.1 and 4.9 it is seen that using a time and cost optimized instance for Threader has to be implemented.

Yet, from table 4.3 it is understood that the current implementation requires lot more improvisation and alternate strategies to dynamically allocate tasks to resources in order to improve the performance.

5.1 Proposed Dynamic Scheduling of *e*Thread Pipeline

There are many aspects associated with the current execution of *e*Thread pipeline which requires more exploration, understanding and optimization. From figures 4.1 and 4.4, it is seen that the behaviour of certain threading tools on t1.micro is still not completely understood as they deviate from the expected results. Also, table 4.3 shows the need for effective optimization models for multi-core instances. These reasons suggest a need for a better approach in the task scheduling mechanism and choice of resources for different threading tools.

One of the strategies to improve the utilization of the resources will be to understand the diversity in the length of the input sequences and the computational load of the threading tools. Whenever parallelization is possible at the threading tool algorithm level, it should be explored and implemented. It is also possible to estimate the computation time for different tools based on the length of the input sequence.

Here, a dynamic scheduling mechanism is proposed, where different tasks are scheduled using the pilot framework. The scheduling mechanism can be trained to compare the incoming task with the parameters including the input sequence length and the threading tool to be executed. The tasks can be classified as long executing tasks and short executing tasks by comparing the TTC for the given sequence length and the threading tool with the existing results. Now, the longer executing tasks and threading tools which require more computation resources (like Threader, COMPASS and pGenThreader) can be scheduled to execute using high performing compute instances. The shorter executing tasks and threading tools which do not require large computing resources (like CSBlast, HMMER and pfTools) can be scheduled to execute in smaller instance types. As soon as one task is finished next task is submitted immediately and thus made sure that the computing resources are effectively utilized.

5.2 Future Work

Along with the proposed dynamic scheduling methods, task-level parallelization techniques using advanced accelerated techniques such as GPGPU with parallel programming techniques like CUDA or MPI based methods can be used to reduce the execution time. Especially, PSI-Blast is used as a pre-requisite step in many threading tools. A GPU based implementation of this algorithm would significantly improve the performance. In conjunction to this, using sophisticated machine learning techniques to classify the incoming tasks based on time and cost profiling can help in optimal and economic utilization of the resources. On the other hand, it will also be equally interesting to experiment a hybrid mix of distributed cyber-infrastructure and cloud.

Wide range of computational tools and techniques are being developed for high throughput protein annotation and structure-based functional annotation of the twilight zone of sequence identity. Template based approaches are gaining popularity and especially, protein meta-threading is of particular interest as they combine heterogeneous algorithms to improve the accuracy of the predictions. In those lines, *e*Thread which combines ten single protein threading algorithms offers a robust method to identify protein structure and function.

Pilot-Jobs enable the decoupling of tasks and resource assignment. This leads to effective task and data level parallelization when possible and helps to operate on heterogeneous resources with ease. Cloud computing provides resources on demand with no wait time and are apt for the heterogeneous mix of algorithms. It offers new business models which also provide cost effective options.

Finally, *e*Thread pipeline is a cost effective and viable option for genome-scale annotation and in Next Generation Sequencing analytics which can be easily extended at ease to any other kind of distributed infrastructure.

Appendix A

Appendix

This section details the necessary tools required, their versions and the installation procedure to replicate the *e*Thread pipeline.

Table A.1: Threading tools incorporated in *e*Thread and their software version used

Threading Tool	**Version**
CSBLAST	2.1.0
pfTools	2.3.4
HMMER	v3.1b1
pGenThreader	8.9
HHpred	2.0
Threader/PSIPRED	3.5/v3.3
COMPASS	3.1
SPARKS2	20050315
SP3	20050315
BLAST	2.2.25
SAM-T2K	3.5

Table 7 presents the versions of each single-threading algorithm used for installation of *e*Thread pipeline. The BigJob version used in our experiment is 0.64.5.

A.1 Installation Details

In order to work with the python program explained in chapter 3 the necessary installation procedure is explained in this section. Working with pilot-job based *e*Thread program is a two part installation. One part requires the installation of modules for the protein threading tools. Other is the pilot-job (BigJob) layer which helps in decoupling the application from the underlying infrastructure.

A.1.1 Protein Threading Modules

It is important to choose a suitable OS to have a hassle free installation and execution. In most of the scenarios, LINUX based OS variants are chosen for threading tools. During installation few threading tools like SP3, SPARKS and SAM-T2K seemed to have installation issues with Red-Hat version. Hence, CentOS was chosen for our experimentation which provided a hassle free process. AMI templates were built for each individual threading tool as each of the threading tools required considerable amount of memory.

The following PERL modules were installed as prerequisite for the required software set:

- Al::NaiveBayes1
- Algorithm::NeedlemanWunsch
- Bit::Vector
- Compress::Zlib
- File::Slurp
- Math::MatrixReal
- Math::Trig
- Statistics::Descrptive
- Uniq
- YAML

After the installation of the basic software tools, the *e*Thread package was installed using tar and make package [51]. Upon successful installation, the following files were present in the ethread-1.0/bin/:

- econtact

- econtact_model
- econv-compass
- econv-csiblast
- econv-fix1
- econv-hhpred
- econv-hmmer
- econv-pftools
- econv-pgenthreader
- econv-samt2k
- econv-threader
- eextract
- emodel
- erank-modeller
- erank-tasser
- ethread
- ethread_model

After building the models, threading library packages were downloaded and unpacked in a location where the individual threading tools had to be installed. These steps remain common to all the threading tools. Once it was completed, the protein threading/fold recognition tools were installed on the desired instances. Next, the Amazon EC2 AMIs templates were created from the instance and used for creating customized instances when required. This completes the initial installation process of *e*Thread.

A.1.2 Installation of BigJob

BigJob was installed at the local environment from where the *e*Thread jobs were submitted to the remote resource. A virtual environment was created and activated for using BigJob. BigJob was installed using pip. SSH password-less login was set up to enable submitting the jobs to the remote resource. The commands used are as follows:

$ virtualenv $HOME/.bigjob/ethread

$ source $HOME/.bigjob/ethread/bin/activate

$ pip install bigjob

The BigJob version used during the experimentation was 0.64.5. The implementation of *e*Thread program in our experiment is specific to Amazon EC2 cloud environment. If it requires to be installed in any other standalone system, please follow the procedure stated in [51]. This is the final step towards the installation of required packages and software modules.

A.2 Accessing the Existing *e*Thread Application

The application program and the required input and python program files can be accessed from the bitbucket repository at [52]. This experimentation is based on Amazon EC2 infrastructure. The input sequences used, JSON input parameter configuration file and the python program required to execute the BigJob based *e*Thread application used for the experimentation are available in the repository. After the basic installation process is complete, the JSON configuration file requires necessary modification based on the infrastructure used. The *pilot_compute_description* arguments and parameters would change accordingly. For arguments specific to a particular Grid/Cluster refer to the BigJob documentation in [53]. If the system is implemented on Amazon EC2 environment, the security credentials, the AMI IDs and the location of input files alone change as per need in the JSON configuration file. The python program can be executed using a simple readme available in the repository.

References

[1] http://www.rcsb.org/pdb/home/home.do, july 2014.

[2] Alexandra M. Schnoes, Shoshana D. Brown, Igor Dodevski, and Patricia C. Babbitt. Annotation error in public databases: Misannotation of molecular function in enzyme superfamilies. *PLoS Comput Biol*, 5(12):e1000605, 12 2009.

[3] Yang Zhang. Protein structure prediction: when is it useful? *Current Opinion in Structural Biology*, 19(2):145 – 155, 2009. Theory and simulation / Macromolecular assemblages.

[4] Michal Brylinski and Daswanth Lingam. ethread: A highly optimized machine learning-based approach to meta-threading and the modeling of protein tertiary structures. *PLoS ONE*, 7(11):e50200, 11 2012.

[5] John Moult, Krzysztof Fidelis, Andriy Kryshtafovych, and Anna Tramontano. Critical assessment of methods of protein structure prediction (casp)round ix. *Proteins: Structure, Function, and Bioinformatics*, 79(S10):1–5, 2011.

[6] M Brylinski and WP Feinstein. Setting up a meta-threading pipeline for high-throughput structural bioinformatics: ethread software distribution, walkthrough and resource profiling. *J Comput Sci Syst Biol*, 6:001–010, 2012 2012.

[7] Sitao Wu and Yang Zhang. Lomets: A local meta-threading-server for protein structure prediction. *Nucleic Acids Research*, 35(10):3375–3382, 2007.

[8] J. Lundstrom, L. Rychlewski, J. Bujnicki, and A. Elofsson. Pcons: a neural-network-based consensus predictor that improves fold recognition. *Protein Sci.*, 10(11):2354–2362, Nov 2001.

[9] Shantenu Jha, Daniel S. Katz, Andre Luckow, Andre Merzky, and Katerina Stamou. Understanding Scientific Applications for Cloud Environments. In Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski, editors, *Cloud Computing: Principles and Paradigms*, chapter 13, page 664. March 2011.

[10] Thilina Gunarathne, Tak-Lon Wu, Jong Youl Choi, Seung-Hee Bae, and Judy Qiu. Cloud computing paradigms for pleasingly parallel biomedical applications. *Concurrency and Computation: Practice and Experience*, 23(17):2338–2354, 2011.

[11] Ronald Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1, 2010.

[12] Pradeep Mantha, Nayong Kim, Joohyun Kim, Andre Luckow, and Shantenu Jha. Understanding MapReduce-based Next-Generation Sequencing Alignment on Distributed Cyberinfrastructure. In *3rd International Workshop on Emerging Methods in Computational Life Sciences, Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12*. ACM, June 2012.

[13] Sharath Maddineni, Joohyun Kim, Yaakoub el Khamra, and Shantenu Jha. Advancing Next-Generation Sequencing Data Analytics with Scalable Distributed Infrastructure. 2012. Concurrency and Computation: Practise and Experience (in press).

[14] Andre Luckow, Lukas Lacinski, and Shantenu Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 135–144, 2010.

[15] Joseph C Jacob, Daniel S Katz, G Bruce Berriman, John C Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, et al. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

[16] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing, Second European Across Grids Conference, AxGrids 2004, Nicosia, Cyprus, January 28-30, 2004, Revised Papers*, pages 11–20, 2004.

[17] Robert H. Swendsen and Jian-Sheng Wang. Replica monte carlo simulation of spin-glasses. *Phys. Rev. Lett.*, 57:2607–2609, Nov 1986.

[18] Yuji Sugita and Yuko Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, 314(1):141–151, 1999.

[19] Abhinav Thota, Andre Luckow, and Shantenu Jha. Efficient large-scale Replica-Exchange Simulations on Production Infrastructure. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1949):3318–3335, 2011.

[20] Emilio Gallicchio, Ronald M Levy, and Manish Parashar. Asynchronous replica exchange for molecular simulations. *Journal of computational chemistry*, 29(5):788–794, 2008.

[21] Z. Li and Manish Parashar. Grid-based asynchronous replica exchange. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, pages 193–200, Austin, TX, USA, September 2007. IEEE Computer Society Press, IEEE Computer Society Press.

[22] Li Zhang, Manish Parashar, Emilio Gallicchio, and Ronald M Levy. Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications. In

Parallel Processing, 2006. ICPP 2006. International Conference on, pages 127–134. IEEE, 2006.

[23] Andre Luckow, Shantenu Jha, Joohyun Kim, Andre Merzky, and Bettina Schnor. Adaptive Distributed Replica–Exchange Simulations. In *Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure Proceedings of the UK e-Science All Hands Meeting*, volume 367, 2009.

[24] Michael C Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[25] Michal Brylinski. Unleashing the power of meta-threading for evolution/structure-based function inference of proteins. *Frontiers in genetics*, 4, 2013.

[26] Hongyi Zhou and Yaoqi Zhou. Sparks 2 and sp3 servers in casp6. *PROTEINS: Structure, Function, and Bioinformatics*, 61(S7):152–156, 2005.

[27] Andreas Biegert and Johannes Soding. Sequence context-specific profiles for homology searching. *Proceedings of the National Academy of Sciences*, 106(10):3770–3775, 2009.

[28] Johannes Soding. Protein homology detection by hmm–hmm comparison. *Bioinformatics*, 21(7):951–960, 2005.

[29] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[30] Philipp Bucher, Kevin Karplus, Nicolas Moeri, and Kay Hofmann. A flexible motif search technique based on generalized profiles. *Computers and Chemistry*, 20(1):3 – 23, 1996.

[31] Anna Lobley, Michael I. Sadowski, and David T. Jones. pgenthreader and pdomthreader: new methods for improved protein fold recognition and superfamily discrimination. *Bioinformatics*, 25(14):1761–1767, 2009.

[32] Ruslan Sadreyev and Nick Grishin. Compass: a tool for comparison of multiple protein alignments with assessment of statistical significance. *Journal of molecular biology*, 326(1):317–336, 2003.

[33] Richard Hughey and Anders Krogh. Hidden markov models for sequence analysis: extension and analysis of the basic method. *Computer applications in the biosciences : CABIOS*, 12(2):95–107, 1996.

[34] David T Jones, WR Taylort, and Janet M Thornton. A new approach to protein fold recognition. 1992.

[35] David T Jones. Protein secondary structure prediction based on position-specific scoring matrices. *Journal of molecular biology*, 292(2):195–202, 1999.

[36] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[37] Jeffrey Skolnick, Hongyi Zhou, and Michal Brylinski. Further evidence for the likely completeness of the library of solved single domain protein structures. *The Journal of Physical Chemistry B*, 116(23):6654–6664, 2012.

[38] Andre Luckow, Mark Santcroos, Ole Weidner, Andre Merzky, Pradeep Mantha, and Shantenu Jha. P*: A Model of Pilot-Abstractions. In *8th IEEE International Conference on e-Science 2012*, 2012.

[39] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[40] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[41] J.T. Moscicki. Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data. In *Nuclear Science Symposium Conference Record, 2003 IEEE*, volume 3, pages 1617–1620 Vol.3, Oct 2003.

[42] A Tsaregorodtsev, M Bargiotti, N Brook, A Casajus Ramo, G Castellani, Ph Charpentier, C Cioffi, J Closier, R Graciani Diaz, G Kuznetsov, et al. Dirac: a community grid solution. In *Journal of Physics: Conference Series*, volume 119, page 062048. IOP Publishing, 2008.

[43] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: A fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 43:1–43:12, New York, NY, USA, 2007. ACM.

[44] Po-Hsiang Chiu and Maxim Potekhin. Pilot factory–a condor-based system for scalable pilot job generation in the panda wms framework. In *Journal of Physics: Conference Series*, volume 219, page 062041. IOP Publishing, 2010.

[45] Pieter van Beek. Topos - a token pool server for pilot jobs, july 2014.

[46] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 283–289. IEEE, 2000.

[47] Edward Walker, Jeffrey P Gardner, Vladimir Litvin, and Evan L Turner. Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pages 95–103. IEEE, 2006.

[48] Andre Luckow, Mark Santcroos, Ashley Zebrowski, and Shantenu Jha. Pilot-data: an abstraction for distributed data. *arXiv preprint arXiv:1301.6228*, 2013.

[49] Bigjob library reference, http://saga-project.github.io/bigjob/sphinxdoc/library/index.html, 2014.

[50] Aws manual on t1.micro instances, http://docs.aws.amazon.com/awsec2/latest/userguide/concepts_m 2014.

[51] ethread installation requirements, http://brylinski.cct.lsu.edu/content/installation-and-requirements, 2014.

[52] ethread repository, https://bitbucket.org/anjaniragothaman/e-thread-repo, 2014.

[53] Bigjob manual, http://saga-project.github.io/bigjob/sphinxdoc/library/index.html, 2014.