Progressive Multiple Biosequence Alignments on the ALiCE Grid

Yong Meng Teo\textsuperscript{1,2}
Yew-Kwong NG\textsuperscript{1}
Xianbing Wang\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, NUS, Singapore 117543
\textsuperscript{2}Singapore-MIT Alliance, Singapore 117576
\{teoym, wangxb\}@comp.nus.edu.sg

Outline

- ALiCE & Motivation
- PMSA Problem
- Distributed PMSA on ALiCE
- Performance Evaluation
- Conclusion
ALiCE

- ALiCE is a portable middleware designed for developing and deploying general-purpose grid applications and application programming models.
- ALiCE is designed to meet a number of design goals, such as flexibility, scalability and platform independent.

Motivation

- Develop an adaptable programming toolkit to facilitate the implementation of medium- to large-scale bioinformatics applications on grid environments
- Demonstrate the capability of ALiCE in deploying data-intensive and compute-intensive problems for parallel execution on grids
ALiCE Architecture

ALiCE Applications and Toolkits
- DES Key Search
- Ray Tracer
- Satellite Image Processing
- Biosequence Comparison

ALiCE Extensions
- Programming Template
- Runtime Support
- Data Services

ALiCE Core
- Compute Grid Services
- Data Grid Services
- Monitoring and Accounting
- Object Network Transport Architecture
- Security Infrastructure

Java Technologies
- JVM, Jini™, JavaSpaces™, GigaSpaces™, JNI, RMI

ALiCE RunTime System

ALiCE Core
- Task Farm Manager (Java/C – Sparc Solaris)
- Task Farm Manager (Java/C – Intel Solaris)
- Task Farm Manager (Java/C – Intel Linux)
- Task Farm Manager (Java/C – Intel Windows)

Internet / LAN
- Resource Broker
- Data Server
### ALiCE Programming Template

- **ALiCE** adopts the *TaskGenerator-ResultCollector* programming model.
- **TaskGenerator** runs on a task farm manager machine and allows tasks to be generated for scheduling by the resource broker.
- **Task** runs on a producer machine, and it specifies the parallel execution routine at the producer.
- **Result** models a result object that is returned from the execution of a task.
- **ResultCollector** runs on a consumer machine, and handles user data input for an application and the visualization of results thereafter.

---

**TaskGenerator Template**

```java
import alice.consumer.*;
import alice.data.*;
public class TASKGEN_CLASSNAME extends TaskGenerator {
    public TASKGEN_CLASSNAME() {
        // Place your initialization code here
    }

    public void init() {
        // Place your initialization code here
    }

    public void main(String args[]) {
        // This is where the tasks are generated, usually in a loop
        TASK_CLASSNAME t = new TASK_CLASSNAME();
        process(t);
        // To open a data file, read and write from/to it
        DataFile f = Data.openFile("file_name", this);
        READ_BUFF = f.read(POSITION, LENGTH);
        f.write(WRITE_BUFF, POSITION, LENGTH);
        // To send/receive an object
        OBJECT_CLASSNAME obj = new OBJECT_CLASSNAME();
        sendObject(obj, "snd_str_id");
        OBJECT_CLASSNAME rcvObj = (OBJECT_CLASSNAME)requestObject("rcv_str_id");
        // To receive a string message from the result collector:
        String msg = getStringMessage();
    }
}
```

**Task Template**

```java
import alice.consumer.*;
import java.io.*;
public class TASK_CLASSNAME extends Task {
    // Place variables here
    public TASK_CLASSNAME() {
        // Place your initialization code here
    }

    public Object execute() {
        // This is where you do your computations. The results can be any kind of
        // object
        // You can generate and send a new task to be produced
        O_TASK_CLASSNAME t = new O_TASK_CLASSNAME();
        process(t);
        // To open a data file, read and write from/to it
        DataFile f = Data.openFile("file_name", this);
        READ_BUFF = f.read(POSITION, LENGTH);
        f.write(WRITE_BUFF, POSITION, LENGTH);
        // To send/receive an object
        OBJECT_CLASSNAME obj = new OBJECT_CLASSNAME();
        sendObject(obj, "snd_str_id");
        OBJECT_CLASSNAME rcvObj = (OBJECT_CLASSNAME)requestObject("rcv_str_id");
        // To receive a string message from the result collector:
        String msg = getStringMessage();
    }
}
```

**Result Template**

```java
import java.io.*;
public class MyResult implements Serializable {
    public DATA_TYPE var;
    public MyResult() {
        var = NULL;
    }
}
```

**ResultCollector Template**

```java
import alice.result.*;
public class RESCOL_CLASSNAME extends ResultCollector {
    // Place variables here
    public RESCOL_CLASSNAME() {
        // Place your initialization code here
    }

    public void collect() {
        // Place here the result collection and processing code to obtain
        // number of results ready call
        int resReady = getResultsNoReady();
        // To get a new result call
        RES_CLASSNAME res = (RES_CLASSNAME)collectResult();
    }
}
```
Progressive Multiple Sequence Alignment

- The simultaneous alignment of multiple sequences is now an essential methodology in molecular biology.
- To find diagnostic patterns to characterize protein families
- To detect or demonstrate homology between new sequences and existing families of sequences
- To help predict the secondary and tertiary structures of newly sequenced genes
- To suggest special primers for Polymerase Chain Reaction, etc.

The Three Successive Stages

1. Sequence Databases
2. Pairwise Comparison
3. Guide Tree
Stage 1: Pairwise Comparison

- **Objective** – To derive the *similarities* between each pair of sequences in the database
- **Similarity Score Function of biosequences** \((s, t)\)

\[
sim(s[1..i], t[1..j]) = \max \begin{cases} 
\sim(s[1..i], t[1..j−1]) - 2 \\
\sim(s[1..i−1], t[1..j−1]) + p(i, j) \\
\sim(s[1..i−1], t[1..j]) - 2 
\end{cases}
\]

---

Stage 1: Pairwise Comparison-II

- **Similarity Scores Matrix example (after comparison)**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hbb_Human</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hbb_Horse</td>
<td>0.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hba_Human</td>
<td>0.41</td>
<td>0.40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hba_Horse</td>
<td>0.41</td>
<td>0.41</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myg_Phyca</td>
<td>0.83</td>
<td>0.33</td>
<td>0.87</td>
<td>0.27</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

---

11

12
Stage 2: Guide Tree Construction

- **Objective** – To produce the sequence in which progressive alignment will take place later, so that the more closely related sequences and profiles are aligned first.

Stage 2: Guide Tree Construction-II

- Build tree based on best (lowest) similarity score values in matrix at each stage.
Stage 3: Progressive Alignment

Objective – To cluster the sequence profiles according to the guide tree branches to produce the final alignment of all sequences.

PMSA Algorithm

```
Progressive_Multiple_Sequence_Alignment (D)
1 n = |D|
2 M = (n x n) distance matrix
3 for all s ∈ D do
4   for all t ∈ D – {s} do
5     M[index(s), index(t)] ← sim(s, t)
6     T = guide tree
7     G = number of sequence groups remaining
8     G ← D
9   while |G| > 1 do
10    v ← minimum value in M
11    c ← sequence group formed by s and t, where M[index(s), index(t)] = v
12    Recompute values in M that involves s and t
13    G ← G – {s, t}
14    G ← G ∪ {c}
15    Add c to T, and add edges from c to s and t respectively in T
16   d ← depth of T
17   while d > 0 do
18      for all s ∈ T, t ∈ T, depth(s) = d, depth(t) = d, parent(s) = parent(t) do
19        align_s, align_t ← Align s and t
20    parent(t) ← merge(align_s, align_t) // creates alignment profile
21    d ← d - 1
22
23 return T – {c | c is not a leaf of T}
```
Parallelization of MSA

Given a database with \( n \) sequences,
- Stage 1: Each task compares a sequence to \( d \) other sequences in the database.
  \[ \text{Total no. of tasks} \approx \frac{n(n-1)}{2d} \]
- Stage 2: No parallelization
- Stage 3: Each task performs the alignment of a pair of sequence profiles. All alignments at the same depth in the guide tree can proceed in parallel, since they are mutually independent tasks. Total no. of tasks = No. of profile alignments

Implementation in ALiCE

- TaskGenerator
- ResultCollector
- MSAVisualizer
- Visualization Module
- SmithWaterman Algorithms Module
- FileParser
- Statistics Module
- DataServer
- DataClient
- SequencesProcessor
- Statistics
- LinearSpace
- Phylogeny
- Parallelization Module
- MSAEngine
- ParallelCompTask
- ProfilesAlignTask
- ProfilesAlignResult
PMSA Implementation

- **Algorithms** encompasses the bioinformatics algorithms used in PMSA.
- **Parallelization** handles the distribution and execution of the tasks in the pairwise comparison and profiles alignment stages.
- **Sequence Retrieval** retrieves the necessary sequences for computation.
- **Statistics** compile meta-information.
- **Visualization** collects and presents the execution results to the user via a GUI.

Performance Evaluation

- **Experimental Environment**
  - **Grid I** (*Homogeneous* cluster grid)
    - 64 Intel Xeon 1.4GHz dual processors nodes, each with 1GB RAM, connected via a *Myrinet* switch
  - **Grid II** (*Heterogeneous* cluster grid)
    - 26 Intel Xeon 1.4GHz dual processors nodes, each with 1GB RAM, connected via a *Myrinet* switch
    - 16 Pentium II 400MHz, each with 256MB RAM
    - 8 Pentium III 866MHz, each with 256MB RAM
    - *Fast Ethernet* connection between the 2 grid segments
Performance Evaluation-II

- Study how performance scales with computational power
- Sequence database centralized on NFS
- Database size = 8,000 sequences
  (~ 10MB)

Performance Evaluation-III

- Sequential execution time = 237.6 hours

<table>
<thead>
<tr>
<th>#Producers</th>
<th>Execution Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Grid I</td>
</tr>
<tr>
<td></td>
<td>PC</td>
</tr>
<tr>
<td>2</td>
<td>148.1</td>
</tr>
<tr>
<td>4</td>
<td>98.6</td>
</tr>
<tr>
<td>8</td>
<td>52.2</td>
</tr>
<tr>
<td>16</td>
<td>32.4</td>
</tr>
<tr>
<td>32</td>
<td>17.8</td>
</tr>
<tr>
<td>48</td>
<td>10.2</td>
</tr>
</tbody>
</table>
Performance Evaluation-IV

Grid I

Grid II

Number of Producers

Exe Time Fraction

Profiles Alignment
Guide Tree Construction
Profiles Alignment
Guide Tree Construction

Performance Evaluation-V

Execution Time (hours)

Number of Producers
Conclusion and Future Works

- Parallelized PMSA problem on the ALiCE grid.
- Study the performance scalability of grid-based PMSA.
- The next approach is to develop a grid programming model for life sciences applications.
- Facilitates the deployment of complicated bioinformatics problems on the grid without re-implementing commonly used component algorithms that can be very tedious to parallelize.

Q & A