Parallel Detection of Regulatory Elements with gMP

Bertil Schmidt, Lin Feng,
School of Computer Engineering
Nanyang Technological University
Singapore 639798,
{asbschmidt,asflin}@ntu.edu.sg

Amey Laud, Yusdi Santoso
Helixense Pte Ltd
73 Science Park Drive, 02-05 Science Park
Singapore 118254
{alaud,ysantoso}@helixense.com

Abstract

The detection of regulatory elements from a large set of regulatory regions is a challenging problem in computational genomics. However, computational methods to extract this biological meaningful information suffer from high computational requirements. In this paper we present a parallel algorithm to detect regulatory elements using correlation with gene expression data and its implementation with gMP. gMP is a new purely Java-based interface that adds MPI-like message-passing and collective communication to the genomics Research Network Architecture (gRNA). The parallel implementation leads to significant runtime savings on our distributed gRNA system.

1. Introduction

A fundamental question in biology is how the expression level of several thousands of genes is regulated. A general assumption is that the regulation is controlled by regulatory elements (so-called motifs) in the regulatory region of a gene (the upstream region). Therefore, several computational methods have been introduced to detect putative motifs from the upstream regions and micorarray gene expression data [4,5,21]. Because of the large genomic scale data sets involved, high computing power is required for their calculation. Since even more genomic data will be available in the future, parallel and distributed processing is needed to get high quality results in reasonable time [14].

In this paper we present how to detect regulatory elements using correlation with gene expression in parallel using the gRNA Message Passing interface (gMP). gMP is a Java-based communication library that adds MPI-like functionality to the genomics Research Network Architecture (gRNA). The gRNA is a highly programmable and extensible platform specifically designed to facilitate the implementation of genome-centric tools. It provides a development environment consisting of application programming interfaces (APIs) in which new applications can be quickly written and tested using object-oriented methodologies. The deployment framework of the system enables these applications to use biological databases and computing resources systematically and transparently.

The rest of this paper is organized as follows. Section 2 provides a description of the gRNA architecture. The new message passing interface gMP is introduced in Section 3. Section 4 illustrates the development of the parallel application to detect regulatory elements. Its performance is then evaluated on a distributed gRNA system. Our approach is compared to previous work in Section 5. Section 6 concludes the paper with an outlook to further research topics.

2. The gRNA framework

The gRNA [15,18] comprises a development platform in which to prototype and build new applications, and a deployment environment in which to provide access to distributed computing and data resources. The development platform consists of application programming interfaces (APIs) designed to provide abstractions for and the foundational basis for new functionality in bioinformatics applications. Figure 1 shows an architectural overview of the development platform. The APIs are inter-related, yet decoupled. This means that the APIs can be used, extended and improved independently.

Three of the APIs are focused on organizing, modeling and querying biological data. The Data Hounds API is used for transporting, wrapping, storing and indexing external databases. The Data Services API provides object-relational abstractions built from multiple primary sources of data; and the XomatiQ API provides a query language to systematically query and correlate

multiple warehouses of life-sciences data (see [1] for more details).

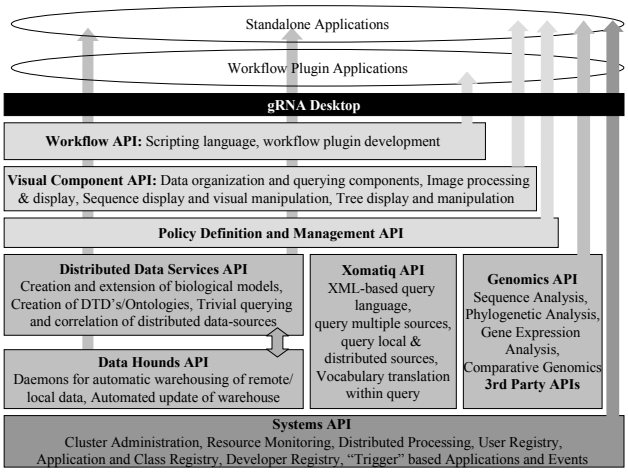

Figure 1. The gRNA development platform.

gRNA consists of two domain dependent APIs. The Genomics API provides specific abstractions and functionality from the biological world; and the Visual Component API provides multi-modal displays for displaying biological data and visual building blocks for typical experiments. The independent Workflow API is a generic, configurable engine to pipeline a set of tasks and data into a unified process.

Within the gRNA, we refer to the following as resources: computers, computing clusters, data warehouses, implemented Data Hounds, implemented Data Services, toolkits and Document Type Definitions (DTDs). Accordingly, the gRNA maintains a centralized directory of all resources, their status and properties. A Policy API allows for the imposition of ownership and conditions for the use of various resources. This means that owners of resources are allowed to assign such conditions as "view only", "read only", "write" and "execute" to various resources, as applicable. Owners are also allowed to embed terms and conditions to define the basis of agreement between a third party user wanting to use resource under the given party's ownership. Lastly, the Systems API provides access to underlying systemic tasks such as distributed programming and access to centralized directories of available resources.

We believe that the simplicity of the deployment environment makes for an important consideration in the successful adoption of new programmable systems. The gRNA environment operates on a clustered computing environment. We refer to the system as the gRNA Grid (see Figure 2).

The typical application written for the gRNA must therefore execute as a multiple tier application, with parts of it executing on the client computer, and parts executing on several server-side, distributed computers. Applications operate from the client's computer by communicating with the cluster through a single computer that hosts an Enterprise Java Bean (EJB) server as application server. This coordinating server then identifies one or more "processing nodes" which are computers running a small footprint EJB server, to perform the task of executing the server-side functionality of the application.

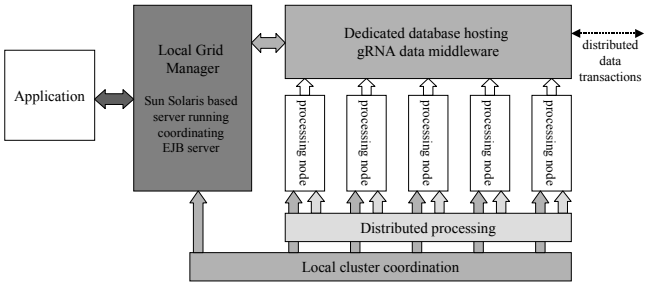

Figure 2: The gRNA grid - clustered deployment environment

The use of a single coordinating server is beneficial in several ways. The cluster can decide, at the point of initiation of the application, decide the number of computers that need to be assigned for that application, depending on availability and necessity. Moreover, the coordinating server also maintains directories of available resources discussed in the previous section, using an LDAP server. As processing nodes are essentially treated as computing resources, the LDAP server can be configured to interface with heterogeneous hardware as well as specialized resources such as vector computers.

The gRNA has been primarily written using Java, with interoperability support for Perl and Python scripting languages (fairly popular within the bioinformatics community). Where performance is a consideration, functionality can be written in C or C++ and interfaced with the overall system using the Java Native Interface (JNI).

3. The gRNA message passing interface

Distributed programming in gRNA is provided through the *gRNA Message Passing* (gMP) API. gMP is a Java-based message-passing tool and comes as part of the gRNA platform. To provide with a highly efficient tool, the gMP API is built directly on top of sockets. Furthermore, the gMP API abstracts the physical configuration of the servers from the program point of view. It will automatically manage virtual processors to run on the available machines.

Since it has been implemented as a set of pure Java APIs, the gMP API can seamlessly use (or be used from) any gRNA and Java library. It is also designed to be scalable by allowing new machines to be added and

removed from the setup transparently. The configurations are done dynamically and automatically without the need of manual intervention.

The gMP process management follows the Java thread management paradigm closely. Processes are grouped. A group is created before any execution is started, and all corresponding processes will join the group. The process that initiates the execution is called the *master* process. This master process plays an important role in execution. It takes care of gathering results, distributing parameters, and so on. The other processes are called *slave* processes. Each process is given a rank. The group ID and rank together provide the process with a unique identifier. Following subsections will describe different aspects of the gMP API in more detail.

3.1 Group management

The gMP API uses the concept of a group for grouping related processes into an easily managed entity. The group is formed before the gMP execution takes place. To ensure sound communication semantics, the grouping is done before any execution is performed. Once the group is formed, it is immutable (cannot be modified). The immutability of the group is necessary to ensure the consistency of group members and ranks among different processes.

```
void startGroup()
void startGroup( int numOfProcs )
void startGroup( GroupConfig grpCfg )
void deleteGroup()
```

The creation of a group is started with a call to the *startGroup* method. There are three flavors of *startGroup* – one with no group size, one with the group size specified, and another one with the exact group configuration specified. When the version with no parameter is invoked, *N* processes are automatically allocated, where *N* is the number of available machines at that time.

The second variant of the *startGroup* method asks for the number of processes to be used. Depending on the number of available machines, extra processes might be run on one or more machines. The allocation of virtual processes is done transparently and tuned to make the process distribution balanced.

The third version of *startGroup* allows more control over the process distribution mechanism. When used together with *gMPGatewayAPI* (that manages running machines), this method allows exact control over which set of processes is executed where.

The method *deleteGroup* has to be executed at the end of a computation. This is to ensure that resources are returned to the server and processes are cleaned. As we have mentioned before, each process in the group is given a unique rank. The number of processes created in the group is given by the *getGroupSize()* command. The ranks of the processes are from 0 to *GroupSize*–1. The rank of the current process can be obtained from the *getRank()* method. To check if the current process is the master process (process 0), we can use the *isMaster()* function.

3.2 Communication primitives

These primitives can be categorized as either synchronous or asynchronous communication.

```
void sendSync ( int destination,
     java.io.Serializable data )
void sendAsync ( int destination,
     java.io.Serializable data )
java.io.Serializable receive
                ( int source )
int available ( int source )
java.io.Serializable rendezvous
     ( int destination,
       java.io.Serializable data )
```

The *sendSync* function performs the synchronous send operation. This method call is blocking. *Receive* can be used to receive a synchronous or an asynchronous message. This method is blocking and will only return if any *sendSync* or *sendAsync* messages are received. The *sendAsync* command performs asynchronous send. The method will return without waiting for the other side to respond. If the destination process never calls *receive*, this message will be kept in the message queue until the next *receive* command is executed.

The method *available* can be used to have a peek on the message queue for incoming messages. The number of available messages is given by the return value. This method is non-blocking and will return directly after counting the number of messages.

Unlike the other primitives, *rendezvous* is a symmetric method. This means that the same method has to be invoked from two processes. These processes will exchange data. This method is also blocking and will not return until a matching *rendezvous* is called.

3.3 Collective communication

Unlike end-to-end communication between two machines, collective communications are communication operations that deal with groups of processes. There are few operations that fall under this category. The semantics of these operations follow from those in MPI [12], but have been adjusted to work naturally with Java. Basically, MPI messages consist of arrays of given data types. Although, important for many scientific codes, arrays cannot serve as general-purpose data structure in

Java's object model. Instead, collective operations should deal with serializable objects in the most general case.

Broadcast is one of the most commonly used collective communication methods. When we broadcast a piece of data, this data is made available to all processes that participate in the group. The master process will supply the data and send it asynchronously to all other processes. The master is not blocked when broadcasting the message, but the slaves are blocked until the broadcast message is received.

The *barrier* operation is commonly used for synchronization among distributed processes. All processes that call this method will be blocked until every other process has reached the barrier statement.

```
java.io.Serializable broadcast(
     java.io.Serializable data )
void barrier()
java.io.Serializable scatter
     ( hlx.gmp.Partitionable data )
hlx.gmp.Partitionable gather
     ( hlx.gmp.Partitionable rootObj,
    java.io.Serializable data )
hlx.gmp.Partitionable allGather
     ( hlx.gmp.Partitionable rootObj,
       java.io.Serializable data )
java.io.Serializable reduce
     ( java.io.Serializable data,
       hlx.gmp.Reducer reducer )
java.io.Serializable allReduce
     ( java.io.Serializable data,
       hlx.gmp.Reducer reducer )
```

When a *scatter* operation is performed, each process in the group will obtain a portion of a bigger set of data being scattered by the master process. This mechanism relies on a data structure defined by the *hlx.gmp.Partitionable* interface. The *Partitionable* interface defines how an object can be partitioned dynamically:

```
public interface Partitionable
     extends Serializable {
     public void setElementAt
           ( int index,
             int groupSize,
           Serializable object );
     public Serializable elementAt
           ( int index,
             int groupSize ); }
```

The *setElementAt* method is used to set the i^{th} element of data of size N, where N is the virtual size given by the *groupSize* parameter. Of course, the actual data itself might not be of size N, but virtually it is divided into N parts. By allowing an object to be divisible consistently we can construct a very flexible data structure that allows division into an almost arbitrary group size. There is also an *elementAt* method that allows one to retrieve the i^{th} element from data of size N. Any data structure that wants to use these operations has to implement the *Partitionable* interface.

While *scatter* disperses data, the *gather* operation collects data from different processes and presents them to the master process. Each part of the data given by the data parameter is passed to the master process. The master process then populates its *Partitionable* object using the given data portion. This method will return the complete *Partitionable* object only to the master. Other processes will receive a *null* value. The method *allGather* is a variant of the *gather* operation. It is equivalent to *gather* followed by the *broadcast* operation.

The method *reduce* defines a group of operations that perform global reductions such as summation or maximum on a set of data, e.g. finding the maximum of N numbers distributed over N processes. To provide the ability to flexibly define the reduction operation, the *hlx.gmp.Reducer* interface can be implemented. The binary reduction operator should be commutative and associative.

```
public interface Reducer {
     public Serializable reduce(
           Serializable obj1,
           Serializable obj2); }
```

Once the reducer class is created, the method *reduce* can be performed on that set of data. The result is returned to the master process. The slave processes get a *null* value as the output. *allReduce* is similar to *reduce*, but subsequently broadcasts the result to all members

4. Parallel detection of regulatory elements

The search for regulatory DNA elements (motifs) upstream of genes is of high importance in biology. These motifs act as sites for the binding of trans-activating factors and therefore play an important role in the activation and inhibition of those genes.

A common approach to motif detection is to group genes into clusters based on similarity in expression under different experimental condition [21]. The upstream regions of these clustered genes are then analyzed to find statistically significant common motifs. However, this method is imprecise, as there are several genes, which respond to the same stimuli but do not share the same motif. There are also genes, which have the same motif but do not respond to the same stimuli.

Recently, Bussemaker et al [4] have published of REDUCE ("Regulatory element detection using correlation with detection"), a new way to identify statistically significant motifs within a given set of gene expression data.

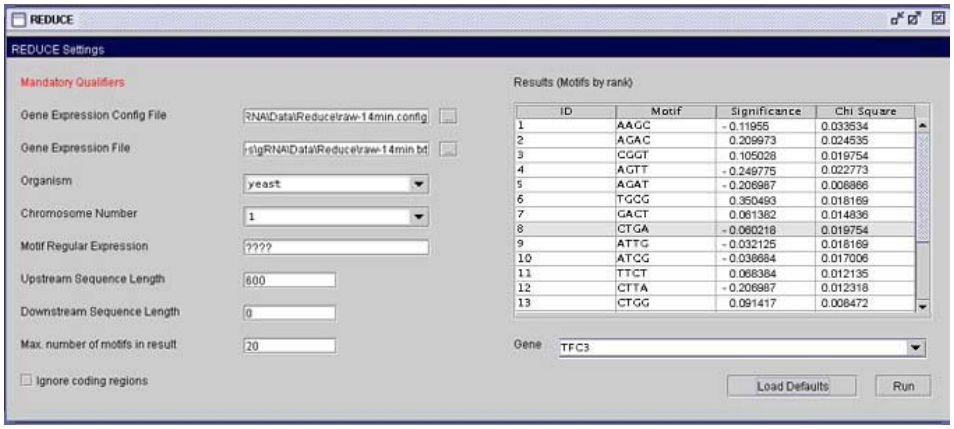

Figure 3: Graphical user interface of the REDUCE application.

Table 1: Motif frequency counter for six ORFs of length 16 base-pairs. The occurrences of each possible motif up to length 2 are counted.

ORFs	A	C	G	T	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
TACATCAATCCACCGC	5	7	1	3	1	2	0	2	3	2	1	0	0	1	0	0	1	2	0	0
ATCTCGAACTTCTTGG	3	4	3	6	1	1	0	1	0	0	1	3	1	0	1	0	0	3	1	2
AATGTGAAAAATTTTT	7	0	2	7	5	0	0	2	0	0	0	0	1	0	0	1	0	0	2	4
AGTATATTATCATGTA	6	1	2	7	0	0	1	4	1	0	0	0	0	0	0	2	4	1	1	1
TAAAGTACAGTCTACG	6	3	3	4	2	2	2	0	1	0	1	1	0	0	0	2	3	1	0	0
AACAAATTTCGAGTCA	7	3	2	4	3	1	1	1	2	0	1	0	1	0	0	1	0	2	0	2

It is based on a model in which upstream motifs contribute additively to the expression level of each gene. Because all genes are fit and are not biased by the clustering, the authors claimed this method to be advantageous over the conventional method. This algorithm is very interesting from several points of view and is definitely worth implementing and testing on both public available and local gene expression data. gRNA provides an ideal environment for the rapid maturation of an algorithm into an application. Figure 3 shows the GUI of our gRNA implementation of REDUCE.

A bottleneck of the application is the high runtime, since the complexity depends on the product of number of possible motifs and that of genes. In the following, we will describe an efficient parallelization of the REDUCE algorithm and its implementation using gMP.

4.1 REDUCE algorithm

The REDUCE algorithm consists of two parts: a *motif frequency counter* and a *significant motif finder*.

The frequency counter simply calculates the occurrences of each possible DNA motif up to a given length (typically between 7 and 11 nucleotides) in each given upstream region (a so-called Open Reading Frame (ORF)). Only those motifs that occur within an upstream region of the translation start-site of each ORF are counted. A typical length of this region is 600 base-pairs, because most of the known transcription binding sites fall into that range [22]. Table 1 illustrates a small example.

In order to find a significant motif, all motifs are ranked as follows. Firstly, the occurrence vector $\boldsymbol{n}_\mu$ of every motif μ is normalized. Secondly, the dot product of each vector $\boldsymbol{n}_\mu$ and the normalized vector of the logarithms of gene expression ratios $\boldsymbol{a}$ is calculated: $\boldsymbol{a} \bullet \boldsymbol{n}_\mu$. All motifs are then ranked according to the square of this dot product $(\boldsymbol{a} \bullet \boldsymbol{n}_\mu)^2$ and the largest one μ_{max} is selected. We proceed by calculating the residual vector $\boldsymbol{a'} = \boldsymbol{a} - (\boldsymbol{a} \bullet \boldsymbol{n}_{\mu max})\, \boldsymbol{n}_{\mu max}$ and normalizing it.

Afterwards, all motifs except μ_{max} are ranked again by using $(\boldsymbol{a'} \bullet \boldsymbol{n}_\mu)^2$. This procedure is iterated until the most significant n motifs are found (a typical value is n=20). Table 2 shows again an example. We proceed by calculating the residual vector $\boldsymbol{a'} = \boldsymbol{a} - (\boldsymbol{a} \bullet \boldsymbol{n}_{\mu max})\, \boldsymbol{n}_{\mu max}$ and normalizing it. Afterwards, all motifs except μ_{max} are ranked again by using $(\boldsymbol{a'} \bullet \boldsymbol{n}_\mu)^2$. This procedure is iterated until the most significant n motifs are found (a typical value is n=20). Table 2 shows again an example.

Table 2: Finding significant motifs: all occurrence vectors from Table 1 are normalized. Subsequently the dot product of each vector with the vector of gene expression ratios *a* is calculated. This example uses *a* = (0.5816, 0.2522, 0.2886, –0.5947, –0.1595, –0.3683). All motifs are ranked by the square of the dot product. In this example *GT* is then selected as the most significant motif. The computation would proceed by finding the second significant motif by using the normalized residual gene expression vector *a'* = (0.3037, –0.2382, 0.4749, –0.3252, 0.3907, –0.6059).

Motifs	*NORMALIZED OCCURRENCE VECTORS*						$(\boldsymbol{a} \bullet \boldsymbol{n}_\mu)$	$(\boldsymbol{a} \bullet \boldsymbol{n}_\mu)^2$
A	(–0.1980,	–0.7921,	0.3961,	0.0990,	0.0990,	0.3961)	–0.4212	0.1774
C	(0.7303,	0.1826,	–0.5477,	–0.3651,	0.0,	0.0)	0.5299	0.2808
G	(–0.6931,	0.4951,	–0.0990,	–0.0990,	0.4951,	–0.0990)	–0.2905	0.0844
T	(–0.5626,	0.2164,	0.4761,	0.4671,	–0.3029,	–0.3029)	–0.2584	0.0668
AA	(–0.25,	–0.25,	0.75,	–0.5,	0.0,	0.25)	0.2133	0.0455
AC	(0.5,	0.0,	–0.5,	–0.5,	0.5,	0.0)	0.3641	0.1326
AG	(–0.3651,	–0.3651,	–0.3651,	0.1826,	0.7303,	0.1826)	–0.7022	0.4931
AT	(0.1091,	–0.2182,	0.1091,	0.7637,	–0.5455,	–0.2181)	–0.2469	0.0610
CA	(0.7013,	–0.4463,	–0.4463,	–0.0638,	–0.0638,	0.3188)	0.0972	0.0095
CC	(0.9129,	–0.1826,	–0.1826,	–0.1826,	–0.1826,	–0.1826)	0.6371	0.4059
CG	(0.2887,	0.2887,	–0.5774,	–0.5774,	0.2887,	0.2887)	0.2651	0.0703
CT	(–0.2462,	0.8616,	–0.2462,	–0.2462,	0.1231,	–0.2462)	0.2205	0.0486
GA	(–0.4082,	0.4082,	0.4082,	–0.4082,	–0.4082,	0.4082)	0.1410	0.0199
GC	(0.9129,	–0.1826,	–0.1826,	–0.1826,	–0.1826,	–0.1826)	0.6371	0.4060
GG	(–0.1826,	0.9129,	–0.1826,	–0.1826,	–0.1826,	–0.1826)	0.2763	0.0764
GT	(–0.5,	–0.5,	0.0,	0.5,	0.5,	0.0)	–0.7941	**0.6306**
TA	(–0.085,	–0.3405,	–0.3405,	0.681,	0.4256,	–0.3405)	–0.5812	0.3378
TC	(0.2132,	0.6396,	–0.6396,	–0.2132,	–0.2132,	0.2132)	0.1831	0.0335
TG	(–0.3651,	0.1826,	0.7301,	0.1826,	–0.3652,	–0.3651)	0.1286	0.0165
TT	(–0.4423,	0.1474,	0.7372,	–0.1474,	–0.4423,	0.1474)	0.0967	0.0093

4.2 Parallelization with gMP

The parallel REDUCE algorithm consists of three parts: (i) a parallel motif frequency counter, (ii) a matrix transposition and (iii) a parallel significant motif finder. These parts are now explained in more detail.

(i) The computation of motif frequencies in each ORF is performed independently (computation of each row in Table 1). Thus, we split the set of given ORFs into equal sized pieces and distribute them evenly across the number of available processing nodes. Each node then calculates the respective occurrences in parallel (see Figure 4 (i)).

(ii) After Step (i) the occurrence vectors are scattered across the nodes. For efficient parallelization in Step (iii) it is advantageous to store each occurrence vector within a single node. This partitioning can be achieved by transposing the motif frequency matrix. The transposition is implemented by gMP's asynchronous send operation as shown in Figure 4(ii).

(iii) The significant motif finder normalizes the locally stored occurrence vectors within each node first. Afterwards, all squares of dot product are computed concurrently and each node determines its local maximum. Finally, a global maximum computation is required. This global reduction operation is performed with gMP's *Allreduce(obj,reducer)* operation.

The *reducer* interface allows for flexible implementations of our own reduce operation. In our example the reducer is a simple maximum operation that will take two motif objects consisting of their DNA sequence, frequency vector and significance value and return the one with the higher significance value.

Afterwards, the most significant motif is available to all nodes in the group. Subsequently, the next iteration step can be initiated by computing the residual gene expression vector ***a'*** in each node.

4.3 Performance evaluation

We have evaluated the performance of our parallel implementation with gMP on an gRNA installation on a Compaq Alpha system. It consists of a cluster of eight AlphaServer SC/ES45 connected by a high-speed Alpha SC 16-Port switch and ELAN PCI adapter cards. Each server contains four Alpha EV68 processors.

Table 3: Timing (in seconds) and speedup of our parallel program for 7090 gene expressions of yeast, ORFs of length 600 and motifs up to length 7 on different number of nodes. The bisection throughput of the matrix transposition operation (in MBytes/s) is also given.

#nodes	1	2	4	8	16
Overall Runtime (speedup)	355	182 (1.9)	104 (3.4)	61 (5.8)	39 (9.1)
Frequeny Counter	74	36	18	8	4
Transpose (throughput)	N/A	10 (15)	6 (38)	4 (66)	3 (91)
Motif finder	281	136	80	49	32

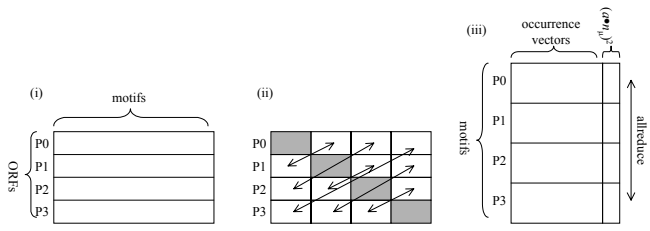

Figure 4: The three phases of our parallel algorithm using 4 processes P0,…,P3: (i) parallel frequency counter (ii) transposition of frequency matrix (iii) parallel significant motif finder.

Table 3 shows the runtime for a different number of nodes using Compaq Java FastVM. Our evaluation uses a data set of 7090 gene expressions of yeast (dataset form [21]), yeast ORFs of length 600 base-pairs each and motifs up to length 7 (i.e. there are 21844 motifs altogether). The 20 most significant motifs are computed.

The measurements show that the overall runtime scales pretty well with the number of processing nodes. The frequency counter scales perfectly since there is no communication involved. The motif finder also scales with the number of nodes, but cannot achieve perfect speedup due to the collective communication involved. Some overhead compared to the sequential program is introduced by the matrix transposition. However, it is only a fraction of the overall computing time and its throughput also scales with the number of nodes.

5. Related work

There exist a number of systems that provide modular and configurable mechanisms for many issues in computational biology. A number of notable systems target a key bottleneck within bioinformatics problem-solving, by providing systematic approaches to the issue of biological data integration and management. DiscoveryLink [17] follows the approach of providing configurable wrappers as consistent interfaces to the wide range of remote data sources. The Kleisli [7] system provides is a systematic approach to managing and integrating external databases, and uses a functional query language to perform correlation across nested databases.

There are a number of toolkits designed to encapsulate functionality from specialized areas within computational biology. Notable examples include BioJava [2] and BioPerl [3], primarily designed for sequence analysis; and the Phylogenetic Analysis Library (PAL) [9]. The Ensembl initiative at the European Molecular Biology Labs (EMBL) [10] and the related Distributed Annotation System (DAS) [8] are systems that provide extensible approaches to the issue of annotating genomic data.

The gRNA distinguishes itself by factoring in the whole range of foundational requirements that go into typical applications in computational biology. It emphasizes on providing decoupled, yet inter-related subsystems that are essentially designed with the ease of third-party programmability as a key criterion. The gRNA also emphasizes on the ability to easily deploy new applications, and provides that as part of an integrated development and deployment environment.

There have also been previous approaches of using Java as a platform for high performance computing. These approaches can be grouped into two categories (1) bindings into native message passing APIs, such as MPI or PVM, where some native libraries are called by Java programs through JNI wrappers (e.g. JavaMPI [11], DOGMA [16], MPJ [6]), and (2) pure Java implementations (e.g. JMPI [19], CCJ [20]).

Although, the native approach provides efficient communication, it does not allow for seamless integration with larger scale Java applications. MPI for instance does not execute only a particular program segment; instead it will try to duplicate the execution of the caller program on all nodes.

JMPI, CCJ and gMP are pure Java message passing interfaces and allow for a certain degree clean integration into Java's object-oriented framework. However, JMPI and CCJ are implemented on top of Java's RMI (Remote Method Invocation), which is inherently slower compared to the raw socket communication used in gMP. CCJ tries to overcome this problem by using RMI on top of Manta (optimized RMI implementation for Myrinet networks).

Unfortunately, this solution is not portable. The CCJ and JMPI frameworks also suffer from the fact that they are not designed with respect to integration. Thus, they lack several features that can greatly assist application development and deployment, such as dynamic class reloading, simplified server management and automated application deployment, which are provided by gMP through the various gRNA services.

6. Conclusions and future work

In this paper we have demonstrated how the gRNA provides an efficient and systematic mechanism for the development of genome-centric applications. In particular, design and usage of gMP has been discussed, a purely Java-based API that seamlessly integrates MPI-like message passing and collective operations into gRNA. We have presented the development of a parallel application for detecting regulatory elements using correlation with gene expression data with gMP that leads to almost linear speedups.

The exponential growth of genomic databases demands more parallel and distributed processing in life science research in the future. Because genomics and bioinformatics are evolving fields, novel algorithms are discovered on a daily basis, e.g. the extensions to REDUCE recently proposed in [13], which require an even higher computing power. Thus, scientists need a high performance computing platform that allows easy implementation, integration and extension of those algorithms. Therefore, we advocate the use of a number of specialized programming interfaces for the management, integration and analysis of biological data.

Our future work on gMP will include identifying communication patterns that are frequently used on popular data structures in computational biology such as sequences, trees and matrices. The results of this study will influence our design of a new API, which will extend the functionality of gMP to these patterns and thus allows easy parallelization of many compute-intensive genomics applications.

References

[1] Bhowmick, S., Cruz P., Laud, A.: *XomatiQ: Living with Genomes, Proteomes, Relations and a Little Bit of XML*, to appear in Proc. IEEE ICDE 2003, Bangalore, India, 2003.

[2] BioJava Project: www.biojava.org.

[3] BioPerl Project: www.bioPerl.org.

[4] Bussemaker, H.J., Li, H., Siggia, E.D.: *Regulatory element detection using correlation with expression*, Nature Genetics 27, pp. 167-171, 2001.

[5] Bussemaker, H.J., Li, H., Siggia, E.D.: *Regulatory Element Detection using a Probabilistic Segmentation Model*, Proc. Int. Conf. Intell. Syst. Mol. Biol. 8, pp. 67-74, 2000.

[6] Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: *MPJ: MPI-like Message Passing for Java*, Concurrency: Practice and Experience, 12 (11), pp. 1019-1038, 2000.

[7] Chung, S.Y., Wong, L.: *Kleisli: A New Tool for Data Integration in Biology*, Trends Biotechnology 17 (9), pp. 351-355, 1999.

[8] Dowell, R.D., Jokerst, R.M., Day A.: *The Distributed Annotation System*, BMC Bioinformatics 2:7, 2001.

[9] Drummond, A., Strimmer, K.: *PAL: An Object-Oriented Programming Library for Molecular Evolution and Phylogenetic*, Bioinformatics 17, pp. 662-663, 2001.

[10] www.ebi.ac.uk/embl/.

[11] Getov, V.: *MPI and Java-MPI: Contrasts and Comparison of Low Level Communication Performance*, in Supercomputing'99, Portland, OR, 1999.

[12] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: *A High-performance, Portable Implementation of the MPI Message Passing Interace Standard*, Parallel Computing, 22 (6), pp. 789-828, 1996.

[13] Grossman, D.M., Ruzzo, W.L.: *Extensions to REDUCE: Regulatory Element Detection using Correlation with Expression*, CSE Qualifying Project, University of Wahington (http://www.cs.washington.edu/homes/grossman/projects/qualsproject/), 2002.

[14] Hampson, S. Baldi, P., Kibler, D., Sandmeyer, S.: *Analysis of Yeast's ORFs Upstream Regions by Parallel Processing, Microarrays, and Computational Methods*, Proc. ISMB'00, AAAI Press, pp. 190-201, 2000.

[15] Helixense Pte Ltd, www.helixense.com.

[16] Judd, G., Clement, M., Snell, Q.: *DOGMA: Distributed Object Group Metacomputing Architecture*, Concurrency: Practice and Experience, 10, pp. 977-983, 1998.

[17] www3.ibm.com/solutions/lifesciences/discovery.html.

[18] Laud, A., Bhowmick, S., Cruz, P., Singh D.T., Rajesh, G.: *The gRNA: A Highly Programmable Infrastructure for Prototyping, Developing and Deploying Genonics-Centric Applications*, Proc. VLDB 2002, Hong Kong, 2002.

[19] Morin, S., Koren, I., Krishna, C.M., *JMPI: Implementing the Message Passing Standard in Java*, Proc. Int. Parallel and Distributed Processing Symposium (IPDPS 2002), Ft. Lauderdale, FL, 2002.

[20] Nelisse, A., Maassen, J., Kielmann, T., Bal, H.E.: *CCJ: Object-based Message Passing and Collective Communication in Java*, Joint ACM JavaGrande-ISCOP 2001 Conf., pp. 11-20, Stanford, 2001.

[21] Roth F.R., Hughes, J.D., Estep, P.W., Church, G.M.: *Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation*, Nature Biotechnology 16, pp. 939-945, 1998.

[22] Spellman, P.T. et al.: *Comprehensive identification of cell cycle-regulated genes of the yeast Saccharomyces cerevisiae by microarray hybridization*, *Mol. Biol. Cell*. **9**, pp. 3273-3297, 1998.