Benchmarking of BioPerl, Perl, BioJava, Java, BioPython, and Python for Primitive Bioinformatics Tasks and Choosing a Suitable Language

Taewan Ryu
Dept of Computer Science, California State University, Fullerton, CA 92834, USA

ABSTRACT

Recently many different programming languages have emerged for the development of bioinformatics applications. In addition to the traditional languages, languages from open source projects such as BioPerl, BioPython, and BioJava have become popular because they provide special tools for biological data processing and are easy to use. However, it is not well-studied which of these programming languages will be most suitable for a given bioinformatics task and which factors should be considered in choosing a language for a project.

Like many other application projects, bioinformatics projects also require various types of tasks. Accordingly, it will be a challenge to characterize all the aspects of a project in order to choose a language. However, most projects require some common and primitive tasks such as file I/O, text processing, and basic computation for counting, translation, statistics, etc. This paper presents the benchmarking results of six popular languages, Perl, BioPerl, Python, BioPython, Java, and BioJava, for several common and simple bioinformatics tasks. The experimental results of each language are compared through quantitative evaluation metrics such as execution time, memory usage, and size of the source code. Other qualitative factors, including writeability, readability, portability, scalability, and maintainability, that affect the success of a project are also discussed. The results of this research can be useful for developers in choosing an appropriate language for the development of bioinformatics applications.

Keywords: *A programming language comparison, BioPerl, BioJava, BioPython.*

1. INTRODUCTION

Bioinformatics is the application of mathematical and computational techniques to the area of molecular biology to solve problems arising from the management and analysis of biological data, eventually to understand biological processes [2, 9, 31]. Common tasks in bioinformatics include the creation, search, retrieval, and analysis of biological data, mapping, manipulating, and analyzing DNA and protein sequences, alignment of those sequences to compare them, 3-D modeling of protein structures, etc. [1], [9], [14], [21], [26]. To perform these tasks, many different programming languages can be used. Out of all the different programming languages, Perl [29], Python [11], and Java [27] have received the most attention from developers of bioinformatics applications mainly because those languages are platform independent and support flexible and powerful features such as string manipulation, text processing, file handling, etc.

Perl is a high-level, general-purpose, and interpreter-based programming language that was originally developed by Larry Wall in 1987, as a Unix scripting language to make report processing easier [24]. Since then, it has undergone many revisions and has become widely popular among programmers for system administration, web application, and text and file processing for many other applications. Particularly, for bioinformatics application development, as developers write similar code to implement common bioinformatics tasks [20], they have formed a community for bioinformatics application developers using Perl and started an open source project called BioPerl [6, 12]. This community allows for the sharing of reusable code that reduces the amount of development time and effort. BioPerl is basically a collection of Perl modules for many of the typical tasks of bioinformatics programming. It is one of the active open source software projects supported by the Open Bioinformatics Foundation [20]. With a basic understanding of Perl, including how to use Perl references, modules, objects and methods, a developer can take advantage of BioPerl to implement sophisticated tasks by using only a few lines of code, significantly saving time and effort in developing applications compared to using the standard Perl.

Java is a general-purpose, object-oriented and compiler-based programming language that derives much of its syntax from C and C++. Similar to BioPerl, BioJava is also an open source project that provides Java-based library for processing biological data and other various mundane bioinformatics tasks [4], [5].

Python is a general-purpose high-level programming language with a design focusing on code readability [11]. Python provides a large and comprehensive standard library supporting multiple paradigms that are primarily object-oriented, imperative, and functional. The language has an open, community-based development model managed by the non-profit Python Software Foundation. BioPython is another open source project based on the Python programming language and is also supported by the Open Bioinformatics Foundation [7]. For our convenience in this paper, the open source projects, BioPerl, BioJava, and BioPython will be referred to as Bio*languages and their base languages, Perl, Java, and Python as native languages.

For a given project, choosing a right language is important since the selected programming language can, in part, affect the success of the project. With a wide range of open source projects and traditional programming languages available, application developers may have difficulty in choosing the right programming language for a project, leading them to ask: what

* *Corresponding author. E-mail : tryu@fullerton.edu*
Manuscript received Feb. 12, 2009 ; accepted Jun. 3, 2009

factors do we need to consider when choosing a language? Some important factors to be considered may include the available project period, available computer resources, ease of collaboration with other team members especially for a large project, efficiency and maintainability of programs, and so on. Furthermore, when it comes to implementation of bioinformatics tasks, one can be easily tempted to use Bio*languages, mainly for quick implementation without realizing whether or not the implemented codes will comply with the intended project goals.

However, while a comparison of languages C, C++, C#, Java, Perl, and Python has been done [13], [23], the pros and cons of each language, especially a comparison between the Bio*languages and their native languages based on these factors in implementing various bioinformatics tasks were not well studied in the past.

Therefore, the main objective of this paper is to provide developers with the valuable information needed in selecting an appropriate language for their projects by evaluating each of the six popular languages including the Bio*languages discussed above. The quantitative evaluation metrics, execution time, memory usage, and size of the source code for each language are measured in several common and basic bioinformatics tasks including: (1) performing disk input/output (I/O) with GenBank and FASTA files, (2) finding a sequence in a GenBank file with a LOCUS name [3], (3) computing the reverse complement of a DNA sequence, (4) counting the residues in a sequence, and (5) translating a DNA sequence to proteins [2], [19]. The paper presents benchmarking results of the six languages for each of these tasks. For some bioinformatics applications, many other complex tasks than these may need to be implemented. However, this research was not intended to cover all aspects of bioinformatics applications but rather to focus on the language-related issues in implementing simple tasks. Therefore, we intentionally avoided sophisticated tasks, especially those involving other systems or special hardware such as database management systems [10], visualization, and specialized computations [30], in order to simplify the experiment without being biased by external systems or hardware. Although these tasks, among many other bioinformatics tasks, are rather simple and elementary, they are also fundamental and sufficient in measuring the strength and weakness of a language used. In addition, other qualitative factors such as writeability, readability, portability, scalability, and maintainability that also affect the project success will be discussed.

The rest of the paper is organized as follows. Section 2 gives the detailed benchmarking results and discussion. Section 3 concludes the paper with a summary and an overall analysis of the research.

2. BENCHMARKING RESULTS

It is well-known that there can be many different ways to implement an algorithm. In order to evaluate each language fairly for each task, a common implementation approach regarding data structure, algorithm, and coding style is first defined and then implemented in each of the six languages. In other words, even though a task can be more optimally implemented through a different technique, the better implementation is ignored if the standard implementation approach can be used in the language. As for the experimental environment, each program was executed on a 500 MHz Pentium 4 with 512 MB of RAM and Windows XP operating system. The languages and versions used in this benchmarking are Perl 5.8.0, BioPerl 1.2.2, Java 1.4.2_01, BioJava 1.3pre1, Python 2.2.3, and BioPython 1.21.

2.1. Performing Disk I/O with GenBank and FASTA files

The main operations in this task are to extract the sequence data and the annotations from a GenBank file and write them into a file in FASTA format. This reading and writing are mainly disk input/output (I/O) operations with some text parsing. The following source codes present the major statements of implementation in each language. The complete version of the source codes and the related information is available on the web: http://tryu.ecs.fullerton.edu/biolanguagebenchmarking.zip.

(1) Bioperl

```
@seq_object_array =
read_all_sequences($infilename,'genbank');
write_sequence(">$outfilename",  'fasta',
@seq_object_array);
```

(2) Biojava

```
SequenceIterator iter =
 (SequenceIterator)SeqIOTools.fileToBiojava(4,
br);
SeqIOTools.writeFasta(new
FileOutputStream("out_biojavaNC_002950.fa"),
iter);
```

(3) Biopython

```
feature_parser = GenBank.FeatureParser()
gb_iteratorFeature = GenBank.Iterator(gb_handle,
feature_parser)
        ......
fasta = Fasta.Record()
while 1:
   cur_feature = gb_iteratorFeature.next()
   if cur_feature is None:
        break
   titleStr = cur_feature.name + " " +
cur_feature.description
   fasta.title = titleStr
   fasta.sequence = cur_feature.seq.data
   wfile.write( str(fasta))
   wfile.write( '\n')
   ......
```

(4) Perl

```
foreach my $line (@GenBankFile) {
   if( $line =~ /^LOCUS/ ) {   # Beginning of gi
number
      ......
   } elsif( $in_origin ) {
      $origin = substr($line, 10, 66);
      $origin =~ s/\s//g;
      $origin = uc($origin);
      print OUT "$origin\n";
   }
}
```

(5) Java

```
......
BufferedReader reader = new BufferedReader(r);
BufferedWriter writer = new BufferedWriter(w);
......
while ( null != line ) {
   if( line.startsWith("LOCUS") ) {
        ......
   } else if( 1 == in_origin ) {
      if(line.length() >= 10){
      StringTokenizer st = new
StringTokenizer(line.substring(10));
      int count = st.countTokens();
```

```
      while( count > 0){

      writer.write(st.nextToken().toUpperCase());
         count--;
      }
      writer.write("\n");
   }
}
```

(6) Python

```
while ( line != ""):
   ......
   elif line.startswith('LOCUS'):# Beginning of
gi number
     locus = line[10:20]
     locus2= locus.strip()     # Removes trailing
& leading hitespaces
     print >>  outfile, '>' + locus2,
   ......
   elif in_origin == 1:
     origin = line[10:76]
     origin = origin.strip()
     origin = origin.replace(' ', '')
     print >> outfile, origin.upper()
```

Execution Time

Figure 1. shows the results of experiment on execution time. The six test data sets used in this study include (A) *AF165912.gbk* (10k) with 1 sequence, (B) *HomoSapiens5_rs.gbk* (120k) with 5 sequences, (C) *HomoSapiens71_rs.gbk* (519k) with 71 sequences, (D) *AC125735.gbk* (726k) with 1 sequence, (E) *NC_002950.gbk* (4,925k) with 1 sequence, and (F) *HomoSapiens500_rs.gbk* (7,022k) with 500 sequences, in the order of their size. Note that the unit of the data size represented in k means KB. To avoid any effect from other running processes, programs were executed right after the system was booted. All programs ran more than five times to minimize the variance among different runs, and their execution times were averaged for four runs excluding the first execution time. The first execution time is intentionally excluded as it might be adversely affected by additional loading time. According to the experimental results, it is apparent that the implementations of Perl, Java, and Python have performance advantages over their corresponding open source projects regardless of the data set sizes. For data set C, Perl ran about 5 ~ 19 times faster than BioPerl, Java ran about 10 times faster than BioJava, and Python ran about 30 ~ 40 times faster than BioPython. BioPython had the worst performance of all the test data sets. The possible cause of a poor performance seen in open source projects may be due to the extra overhead for loading unnecessary modules included in the language into the system. One interesting result is that Perl, BioPerl, and Python slightly outperformed Java and BioJava for small data sets A, B, C, and D while Java and BioJava outperformed Perl and BioPerl for larger data sets E and F. The result is somewhat contradictory to the idea that script languages, such as Perl and Python, are usually slower than compiler-based languages such as Java. To further identify why Java performs poorly compared to Perl when handling small data sets, additional experiments to test the performance of only I/O operations for each language were conducted.

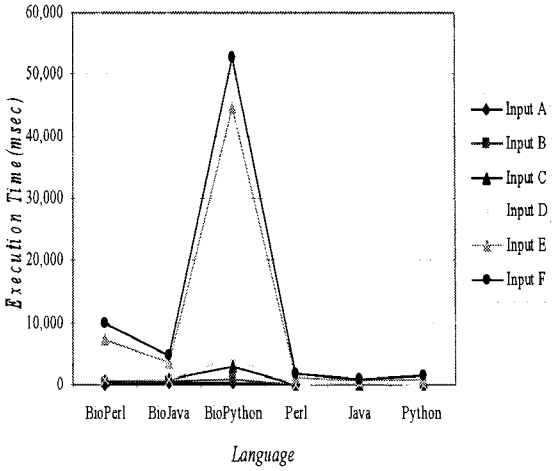


Fig. 1. Comparison of average execution times

Consistent with the results of the previous experiment, as shown in Table 1, Perl outperformed Java for small data sets while Java outperformed Perl for large data sets.

Table 1. Comparison of execution times for I/O operations

Exec. Time (msec) / Language	Data Set	A	B
	Size(K)	10	7,022
Perl		2.50	1,449.50
Java		15.00	463.00
Python		1.54	402.36

On the other hand, the results show that Python significantly outperformed both Perl and Java for both data sets A and B. However, Java slightly outperformed Python for larger data sets, E and F. In Java, the BufferedReader class was wrapped around the FileReader class in order to convert the underlying character stream to buffered I/O, which can be more efficient than the one without buffering. Python uses the buffered I/O operations provided by the C library. We think that the management of buffering in Python may be more efficient than the ones in other languages.

Memory Usage

Since the memory usage varies significantly over runtime, we measured the peak memory usage [32]. Figure 2 shows the memory usage of six languages for six different data sets. It is apparent that BioPerl and BioPython used higher consumption of memory than the others. Among the six languages, Python consumed the least amount of memory for I/O operations. BioPython consumed the most memory, followed by BioPerl. In general, the Bio* languages consumed more memory than their native languages. Again, the main reason for this can be that Bio* languages require loading many unnecessary modules to perform the task. Intuitively, one can expect that memory usage would be increased as input data size grows. However, the experimental results with the relatively large data set F using BioJava and BioPython proved to be counterintuitive. Our observation is that processing a single larger sequence seems to require more memory than multiple smaller sequences. Particularly, the reduction in memory usage in BioPython was significant. It is also interesting to see that the memory usages of

Python and Java were not affected by the size of inputs; they had constant memory usages of about 6K and 3K bytes for all data sets.

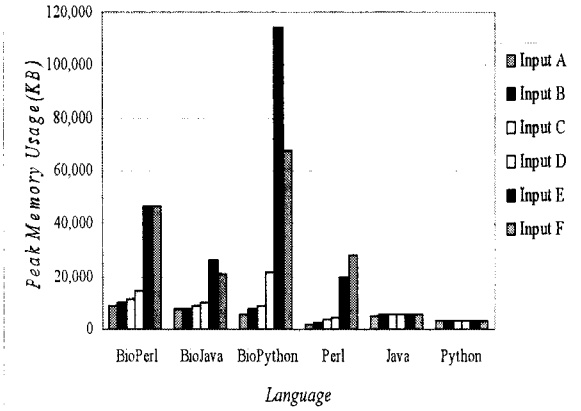


Fig. 2. Comparison of memory usages

Size of Code

The size of a program is an important factor in choosing a language because it indicates the complexity of code and the development time. Fig. 3 illustrates the total number of lines of source code in the six languages to perform this task. As expected, the Bio* languages required less lines of code than their native languages. Of the non-Bio* languages, Perl required the least lines of code required by the defined coding style.

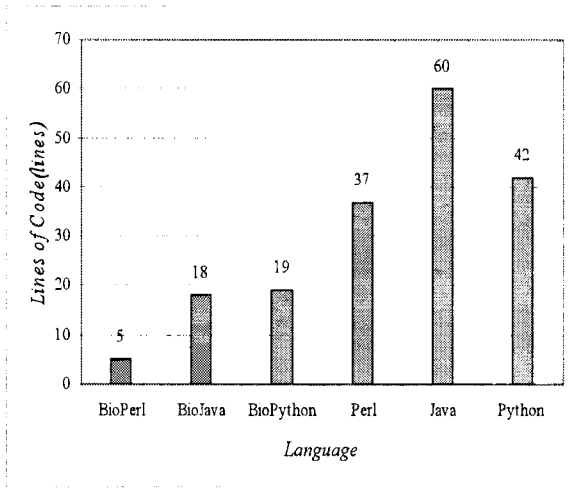


Fig. 3. Comparison of code size for each language

Java, a compiler-based language, required the largest number of lines of code. Compared to BioPerl, Java required almost twelve times as many lines of code. Other experiments showed similar results to figure 3.

2.2. Finding a Sequence in a GenBank File with a LOCUS Name

This task is to search for a particular sequence with the LOCUS name by processing one line at a time, which requires both basic string processing and I/O operation. The major statements for this task in each of the languages are given below:

(1) BioPerl

```
@seq_object_array =
read_all_sequences($infilename,'genbank');
foreach my $seq_object (@seq_object_array) {
   if ($seq_object->id() eq $findID) {
        #print "FOUND\n";
        last;
   }
}
```

(2) BioJava

```
SequenceIterator seqs =
SeqIOTools.readGenbank(br);
String findID = "AZ574568";
while (seqs.hasNext()) {
   Annotation anno =
seqs.nextSequence().getAnnotation();
   if (findID.equals(((String)
anno.getProperty("LOCUS")))) {
        break; //found
   }
}
```

(3) BioPython

```
gb_file = "HomoSapiens1000_rs.gbk"
gb_handle = open(gb_file, 'r')
feature_parser = GenBank.FeatureParser()
gb_iteratorFeature = GenBank.Iterator(gb_handle,
feature_parser)
findID = "AZ574568"
while 1:
   cur_Feature = gb_iteratorFeature.next()
   if cur_Feature is None:
        break
   if(findID == cur_Feature.name):
       break #print "Found"
```

(4) Perl

```
open ( IN , "HomoSapiens1000_rs.gbk" ) or die
( "Cannot open file!");
my @GenBankFile = ();
my $findID = "AJ574568";
@GenBankFile = <IN>;
foreach my $line (@GenBankFile) {
   if ($line =~ /^LOCUS/) {
        $name = substr($line, 12, 34);
        $name =~ s/\s//g;              #Remove all
the spaces between characters
        if ($findID eq $name) {
            print "FOUND";
            last;
        }
   }
}
```

(5) Java

```
FileReader r = new FileReader( new
File("HomoSapiens1000_rs.gbk") );
BufferedReader reader = new BufferedReader(r);
String line = reader.readLine();
String findID ="AZ574568";
while ( null != line ) {
    if ( line.startsWith("LOCUS")) {
        if
(findID.equals(line.substring(12,34).trim())) {
            break; //FOUND
        }
    }
    line = reader.readLine();
}
```

(6) Python

```
try:
   infile  = open( "HomoSapiens1000_rs.gbk",
"r");
```

```
   except IOError:
       print >> sys.stderr, "Input file could
not be opened"
       sys.exit( 1 ) ;
   line = infile.readline()    # Get the first
record
   findID = "AZ574568"

while ( line != ""):
   if line.startswith('LOCUS'): # Beginning of
ACCESSION
       ID = line[12:34]
       ID = ID.strip()
       if ID == findID :
          break        #print "FOUND"
       line = infile.readline()
```

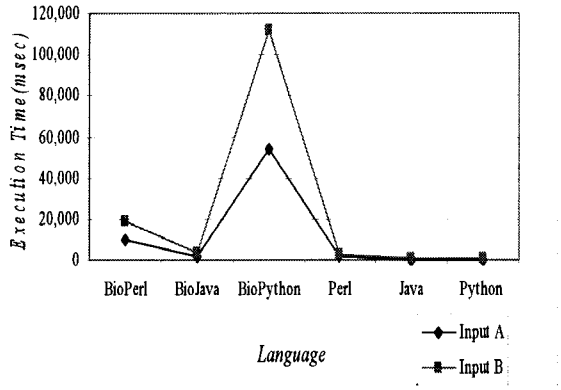


Fig. 4. Comparison of average execution times. The data sets used in this experiment are (A) HomoSapiens500_rs.gbk (7,022k) with 500 sequences and (B) HomoSapiens1000_rs.gbk (14,043k) with 1,000 sequences.

Execution Time

Figure 4 compares the results on the execution times of the six languages for two different data sets. The resulting graph for this task is similar to the ones for the previous task. Consistent with the previous result, BioPython again ran the slowest. For data set B, the best language (Java) performed about 245 times better than the worst language (BioPython). In general, the native languages ran faster than the Bio* languages.

Memory Usage

As shown in figure 5, BioPython consumes the largest memory for both data sets. Given that the size of data set B is approximately two times larger than the data set A, it is of interest that the memory consumption for this task seems to be insensitive to the size of input data set, except when using BioPerl and Perl, as shown in Fig. 5.

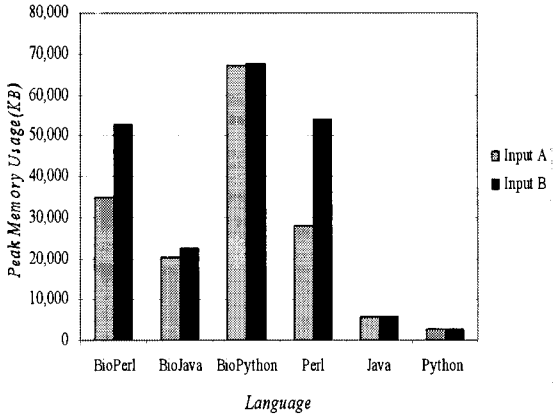


Fig. 5. Comparison of memory usages in each language

2.3. Computing the Reverse Complement of a DNA Sequence

This task requires simple string manipulation (e.g., reverse the string and compute the complement of it) without involving I/O operations.

Execution time

In this experiment, as shown in figure 6, BioPython and Python showed the worst performance among the six languages whereas BioJava significantly outperformed all other languages. It is of interest that BioJava and Perl both outperformed Java. The results indicate that the string manipulation in both BioJava and Perl is very efficient.

Memory Usage

As shown in figure 7, BioPython and Python used almost up to 50% more memory than the other languages for data set A, and BioPerl and Python noticeably used more memory than the other languages for data set B. As was generally expected, the memory usages of Bio* languages was higher than the native languages.

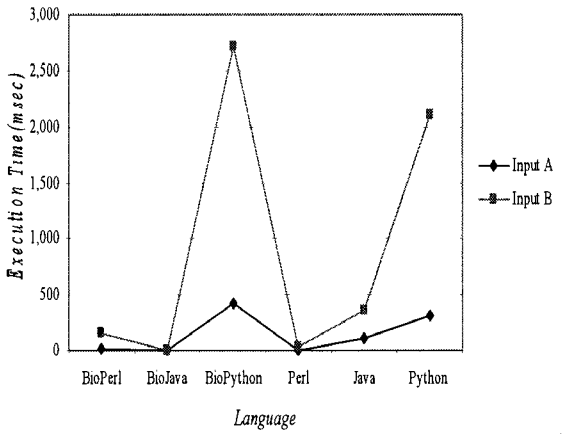


Fig. 6. Comparison of average execution times. The data sets used in this experiment are (A) AC125735.fa (387k) with 1 sequence and (B) NC_002950.fa (2,422k) with 1 sequence.

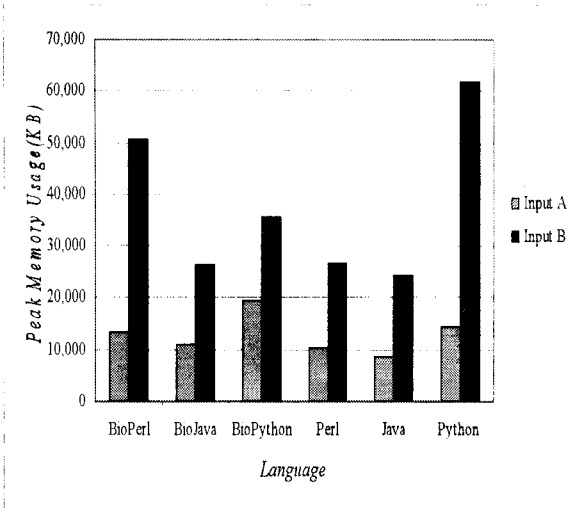


Fig. 7. Comparison of memory usages

2.4. Counting the Residues in a Sequence

In this study, we evaluated the performance in counting a specific string in a sequence excluding the I/O operations.

Execution Time

As with the previous task, shown in figure 8, Python and BioPython again exhibited the poorest performance out of the six languages. Based on these results, we can speculate that the string manipulation in Python may be inefficient. Java including BioJava achieved the best performance.

Memory Usage

Figure 9 shows the comparative results of memory consumption for the six languages. According to the results, as the size of input data is increased, the memory usage in BioPython, Perl, and Python is significantly increased. The results confirm that the memory management in the compiler-based languages is more efficient than the management in the interpreter-based languages, which also implies the good scalability of Java and BioJava.

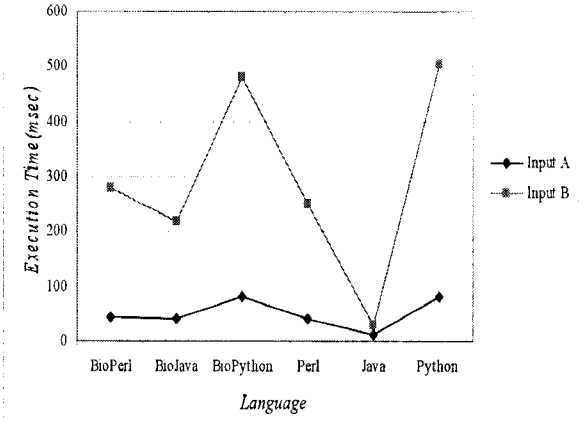


Fig. 8. Comparison of average execution times. The data sets used are (A) AC125735.fa (397k) with 1 sequence and (B) NC_002950.fa (2,422k) with 1 sequence.

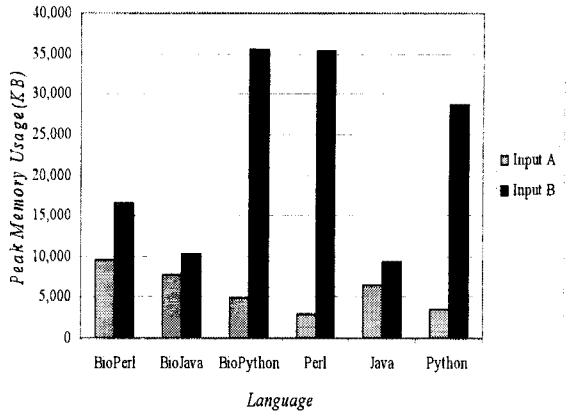


Fig. 9. Comparison of memory usages

2.5. Translating a DNA Sequence to Protein

This task involves both string manipulation and I/O operations. To implement an efficient genetic code look-up table, the hash data structure was used in the native languages. In order to measure the overhead of function calls, two functions were implemented—one function to read a DNA sequence from a FASTA file and return the sequence and another function to return the protein sequence from the genetic code table. The major statements for this task in each of the languages are given below:

(1) BioPerl

```
#$t = Benchmark::Timer->new();
#$t->start('time_tag');
$seqobj = $seq_object->translate();
#$t->stop('time_tag');
```

(2) BioJava

```
long start = System.currentTimeMillis();
SymbolList symL = DNATools.createDNA(seqAll);
long start = System.currentTimeMillis();
//transcribe to RNA
symL = RNATools.transcribe(symL);
//translate to protein
symL = RNATools.translate(symL);
long stop = System.currentTimeMillis();
```

(3) BioPython

```
my_seq = Seq(seqAll, IUPAC.unambiguous_dna)
start = time.clock()   # Timer ON
transcriber = Transcribe.unambiguous_transcriber
# Get the transcriber
standard_translator =
Translate.unambiguous_dna_by_id[1] # Get the
proper translator
result_seq =
standard_translator.translate(my_seq) #
Translate a sequence
end = time.clock() # Timer OFF
```

(4) Perl

```
#$t = Benchmark::Timer->new();
#$t->start('time_tag');
$seqAll =~ tr/T/U/;    ## change T into U
my $protein = '';
# Translate each three-base codon into an amino
acid, and append to a protein
my $len = length($seqAll) - 2;
for(my $i=0; $i < $len ; $i += 3) {
    $protein .=
$transTable{substr($seqAll,$i,3)};
```

```
}
#$t->stop('time_tag'); ## STEP3:TIMER
```

(5) Java

```
long start = System.currentTimeMillis();
seqAll = seqAll.replace('T','U'); //
Transcription
long len = seqAll.length()-2;
StringBuffer protein = new StringBuffer("");

for (int i=0; i < len ; i += 3) {

protein.append( table.get(seqAll.substring(i,i+3
)) );
}
long stop = System.currentTimeMillis();
```

(6) Python

```
#start = time.clock()
seqAll = seqAll.replace('T', 'U')     #
Transcription
i = 0
allProtein = ""
comLen = len(seqAll)-2
while(1):
  #allProtein+=(proteinDict[seqAll[i: i+3]])
  allProtein = ''.join(proteinDict[seqAll[i:
i+3]])
  i += 3
  if (i >= comLen):
  break
end = time.clock()
```

Execution Time

The experimental results are shown in figure 10. As we expected, compiler-based language Java and BioJava were faster than the interpreter-based languages. For large data set like B, BioJava is about approximately ten times faster than BioPerl.

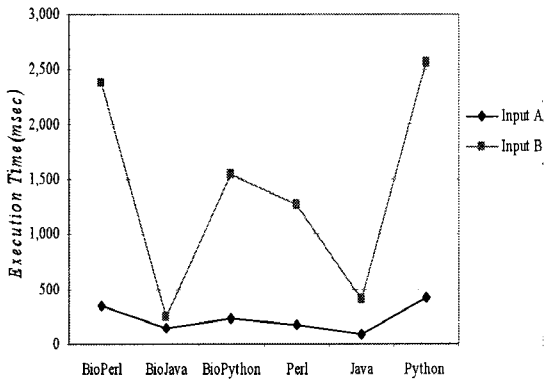


Fig. 10. Comparison of average execution times. The data sets used include (A) AC125735_2.fa (397k) with 1 sequence and (B) NC_002950_2.fa (2,422k) with 1 sequence.

Table 2 illustrates the overhead of function calls in three languages.

Table 2. Comparison of function call overhead. The number of function calls is defined by the number of codons in a DNA sequence.

Exec. Time (msec) / Language	Data Set	A	B
	# of Function Calls	128,192	781,158
Perl		100.14	595.86
Java		5.00	7.50
Python		121.12	737.49

As expected, the overhead of function calls in Java is significantly smaller compared to the interpreter-based languages Perl and Python regardless of the size of input data set. The result may not be surprising because a compiler-based language like Java is generally equipped with a powerful code optimizer. On the other hand, it is known that code optimization is limited for interpreter-based languages. This result can explain why Java and BioJava showed the best performance for this task.

Memory Usage

According to the result shown in figure 11, Python and BioPython require more memory than other languages. Moreover, the experimental results show that the memory consumption in a program depends on the size of the input data sets.

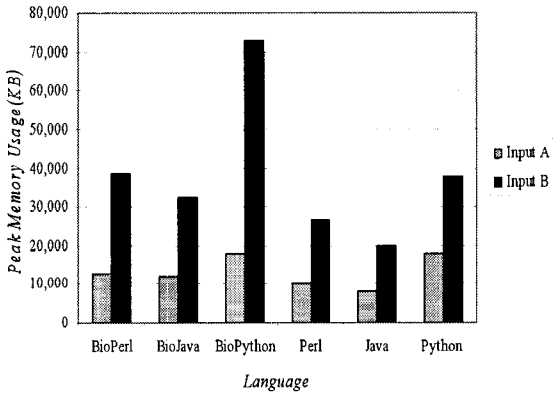


Fig. 11. Comparison of memory usages

This is mainly due to the specific implementation in this study (e.g., a variable is used to hold the whole sequence for sequence translation).

2.6. Overall Evaluation

In this section, the overall evaluation of the six languages based on evaluation metrics is discussed. Figure 12 shows the overall execution time for all five tasks combined. According to the results, it is apparent that the native languages run faster than Bio*languages and the compiler-based languages in general run faster than the interpreter-based languages.

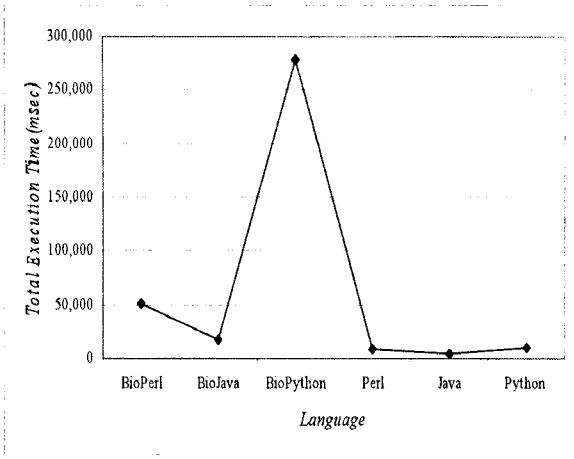


Fig. 12. The overall execution time for all tasks

In addition, BioPython seems to be the slowest language among these languages.

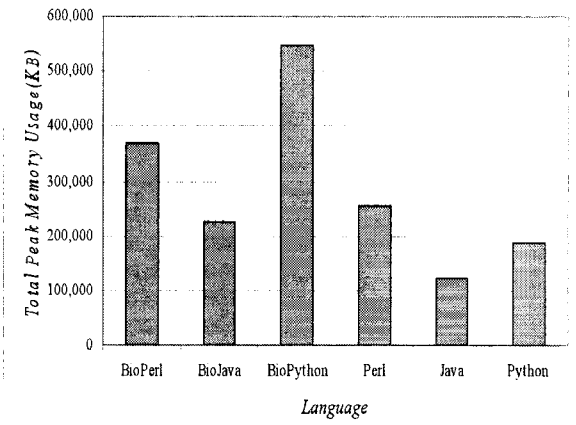


Fig. 13. The overall memory usage for all tasks

Figure 13 shows that the Bio*languages in general require more memory than the corresponding native languages. The compiler-based languages require less memory than the interpreter-based languages. In general, BioPython requires more memory than any other language we have used in this experiment. On the other hand, Java and Python use the least memory for most tasks. To compare Perl and Python, Python in general outperforms Perl in terms of execution time and memory usage, which is consistent with the results of algorithms implemented by Fourment and Gillings that require string manipulation, I/O, and memory management [13].

Figure 14 shows a summary of code size for each language. According to the results, Bio*languages require much less code than their corresponding native languages. BioPerl requires the shortest lines of source code among the six languages while Java, a strongly-typed language, requires the longest lines of code [25].

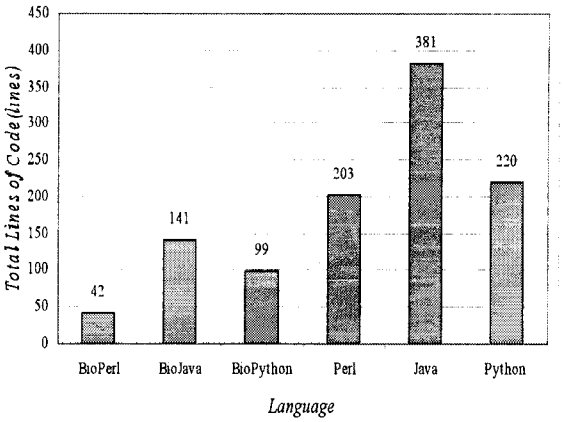


Fig. 14. The overall code size for all tasks

3. SUMMARY AND CONCLUSIONS

It is well known that no particular language is best suited for all types of applications. Each language has unique features that can be better applied to a particular type of application than any other. Therefore, it is important to consider many different factors in a given development environment when choosing a programming language for the project.

According to the experimental results for the tasks we selected to measure the quantitative perspective of a language, compiler-based languages expectedly perform better and are more scalable than interpreter-based languages in terms of execution time and memory management because of powerful code optimization. However, they require more lines of source code and higher programming skills, which means more development time, while the interpreter-based languages are more flexible and have a shorter development time than the compiler-based languages.

In comparison to native languages, Bio*languages obviously require less lines of code but, in general, perform poorly in terms of both execution time and memory management. This result may be somewhat surprising, especially to those who consider the languages as being the same because Bio*languages are based on the native languages with additional modules or libraries.

In an evaluation of individual languages, Java shows the best overall performance in most tasks in terms of both execution time and memory management. BioJava effectively takes advantage of its native language, Java, unlike BioPython. Interestingly, Perl and BioPerl outperformed Java and BioJava when processing small data sets. In addition, Perl is very powerful for string manipulation, even compared to both Java and Python. According to the experimental results, the string manipulation operation in Python seems to be inefficient compared to Perl and Java; the inefficiency is also shown in the experiment by Fourment and Gillings based on the BLAST parsing program [13]. On the other hand, Python seems to be better in memory management than Perl. Compared to other languages used in this experiment, BioPython performed very poorly for most tasks. BioPython was much slower and consumed more memory in all tasks than any other language used in this experiment. This result is surprising to us and indicates that BioPython may need a significant improvement in language implementation.

Guidelines and other factors to consider in choosing a suitable language

Other qualitative factors such as writeability, readability, portability, and scalability need to be considered when evaluating a programming language. Writeability is related to a programmer's learning curve and productivity, which is important for a project with a tight budget and deadline. Readability is related to an ease of collaboration and maintainability, which is important for a large project with many members involved. Portability is important when a system needs to run on multiple platforms. Scalability can be defined as how well a program continues to function with a growing number of users or input data size. It is an important measurement of a system's expandability.

In general Bio*languages are considered to have good writeability and readability due to code simplicity and the availability of many built-in modules, requiring shorter lines of codes and a lower learning curve. These advantages allow easier maintenance and faster development. In regards to the readability of Python, the syntax of Python is interesting; in Python, indentation showing where a block begins and ends is important, thus ensuring good readability. All six languages are considered to be portable.

Java is an efficient and reliable an object-oriented language but requires a higher learning curve; it takes time to learn the language and needs skilled programmers for implementation. As a result, BioJava may instead be a better choice for a large project that involves many members and high maintainability because of its strongly supported object-oriented programming equipped with many built-in libraries. BioJava takes advantage of Java's quick execution time and effective memory management as well as the flexibility and easy programming of Bio*languages. According to the experimental results considering memory use and execution time, BioJava, like Java, is scalable. BioJava is also reliable since it is a strongly-typed language. Python can also be well suited for a large project involving many programmers since it supports object-orientation.

Thus, if either program efficiency or good memory management (because of large input data sets) is a key consideration, a compiler-based language, such as Java, BioJava, or C/C++ if necessary [13], [23], would be the best choice. For beginners, a Bio*language would be a good choice since they provide built-in, easy-to-use, and usually well-tested modules. For manipulation of a small data set and quick implementation of a task, Perl or BioPerl can be a good choice. However, Perl may not be appropriate for a large project as it can become very complex and difficult to manage and maintain. On the other hand, developers may need to be cautious in using BioPython and wait for an improved version of the language.

Future research will be done to compare languages in more sophisticated bioinformatics tasks involving databases, Web, image processing, visualization, and other complex algorithms.

REFERENCES

[1] P. Baldi and S. Brunak, *Bioinformatics: The Machine Learning Approach*, 2nd edition, MIT Press, 2001.

[2] A.D. Baxevanis and B.F.F. Ouellette, eds., *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, 3rd edition, Wiley, 2005.

[3] D.A. Benson, M. Boguski, D. J. Lipman, J. Ostell, B.F. Ouellette, B.A. Rapp, and D.L. Wheeler, "GenBank," *Nucleic Acids Research*, 27, 12-17, 1999.

[4] R.C.G. Holland, T. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Drager, A. Yates, M. Heuer, M.J. Schreiber, "BioJava: an Open-Source Framework for Bioinformatics," *Bioinformatics, Vol. 24(18)*, 2008, pp. 2096-2097.

[5] BioJava Web information, http://www.biojava.org, 2009.

[6] BioPerl Web information, http://www.bioperl.org, 2009.

[7] BioPython Web information, http://www.biopython.org, 2009.

[8] M. Catanho, D. Mascarenhas, W. Degrave, A. Miranda, "BioParser: a tool for processing of sequence similarity analysis reports," *Applied Bioinformatics*, 5 (1): 49–53, 2006.

[9] N. Cristianini and M.W. Hahn, *Introduction to Computational Genomics*, Cambridge University Press, 2006.

[10] O. Croce, M. Lamarre, R. Christen, "Querying the public databases for sequences using complex keywords contained in the feature lines," *BMC Bioinformatics* 7:45, 2006.

[11] H. Deitel, P. Deitel, J. Liperi, and B. Wiedermann, B., *Python: How to Program, Prentice Hall*, 2002.

[12] J. Dugan, Open Source Initiatives in Bioinformatics, *A report submitted to health science initiative application working group Internet2*, 2001.

[13] M. Fourment and M.R. Gillings, "A comparison of common programming languages used in bioinformatics," BMC Bioinformatics, *Vol. 9:82*, 2008.

[14] W. Keedwell, Intelligent Bioinformatics: The Application of Artificial Intelligence Techniques to Bioinformatics Problems, *Wiley*, 2005.

[15] R. Khaja, J. MacDonald, J. Zhang, S. Scherer, "Methods for identifying and mapping recent segmental and gene duplications in eukaryotic genomes," *Methods Molecular Biology* 338: 9–20, 2006.

[16] B. Landsteiner, M. Olson, R. Rutherford, "Current Comparative Table (CCT) automates customized searches of dynamic biological databases," *Nucleic Acids Research 33*, 2005.

[17] B. Lenhard, W. Wasserman, "TFBS: Computational framework for transcription factor binding site analysis," *Bioinformatics 18 (8)*: 1135–6, 2002.

[18] A.M. Lesk, Introduction to Bioinformatics, *Oxford University Press*, 2008.

[19] D.W. Mount, Bioinformatics: Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, *Cold Spring Harbor, New York*, 2001.

[20] Open Bioinformatics Foundation, http://www.open-bio.org, 2009.

[21] L. Pachter and B. Sturmfels, *Algebraic Statistics for Computational Biology*, Cambridge University Press, 2005.

[22] P.A. Pevzner, Computational Molecular Biology: An Algorithmic Approach, *The MIT Press*, 2001.

[23] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl," *IEEE Computer Vol. 33*, 23-29, 2000.

[24] R. Schwartz, T. Phoenix, and B. Foy, *Learning Perl, 5th Edition, O'Reilly*, 2008.

[25] R.W. Sebesta, Concepts of programming languages, *Addison Wesley*, 206-208, 2006.

[26] S. Shah, G. McVicker, A. Mackworth, S. Rogic, B. Ouellette, B., "GeneComber: combining outputs of gene prediction programs for improved results," *Bioinformatics 19 (10)*: 1296–7, 2003.

[27] J. Shirazi, Java Performance Tuning, *O'Reilly*, 2003.

[28] J. Stajich, D. Block, K. Boulez, S. Brenner, S. Chervitz, C. Dagdigian, G. Fuellen, J. Gilbert, I. Korf, H. Lapp, H.

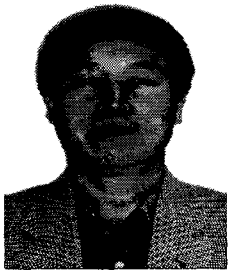
Lehväslaiho, C. Matsalla, C. Mungall, B. Osborne, M. Pocock, P. Schattner, M. Senger, L. Stein, E. Stupka, M. Wilkinson, E. Birney, "The Bioperl toolkit: Perl modules for the life sciences" *Genome Res 12 (10)*: 1611–8, 2002.

[29] J.D. Tisdall, Beginning Perl for Bioinformatics, *O'Reilly*, 2001.

[30] N. Trivedi, K.T. Pedretti, T.A. Braun, T.E. Scheetz, and T.L. Casavant, "Alternative parallelization strategies in EST clustering," *Lecture Notes in Computer Science, Vol. 2763*, 384 – 394, 2003.

[31] M.S. Waterman, Introduction to Computational Biology: Sequences, Maps and Genomes, *CRC Press*, 1995.

[32] J. Zobel, S. Heinz, and H.E. Williams, "In-memory hash tables for accumulating text vocabularies," *Information Processing Letters, Vol. 80:6*, 271 – 277, 2001.



Taewan Ryu

He received the PhD degree in Computer Science from University of Houston, Houston, Texas. Currently, he is an associate professor of the Department of Computer Science in California State University, Fullerton, California. His research interest includes data mining, Bioinformatics, computational finance, and software engineering.