IUPUI

Indiana University Purdue University Indianapolis

Department of Computer & Information Science

Iterators - An Overview

Authors:
Abhinandan Jayanth
Akshay Virkud
Arvind Nair
Hitendra Rathod
Pulkit Sood

Under the guidance of:
Dr. Rajeev Raje

December 14, 2015

Contents

List of Figures

Abstract

Iterators are an effective means of traversing a collection of objects while hiding the internal representation of the process. In this report we aim to give an overview of this programming construct in various programming paradigms and also look at available alternatives in the case of absence of iterator implementations. We also describe the various types of iterators and their applications in good software practices. The simulations section encapsulates our experiments with iterators in a few languages like C++, Python, CLISP, Java and also the offbeat language Julia.

1 Introduction

“An iterator is a module that allows clients to access each element of a composite data type while hiding the representation of the structure” [1]

“An Iterator is used for defining how the objects in a collection are obtained” [14]

CLU was the first programming language to introduce the concept of an iterator. [15]

In computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists. Various types of iterators are often provided via a container’s interface. An iterator is behaviorally similar to a database cursor.

Iterators are a powerful technique in object-oriented programming and one of the fundamental design patterns. [2]

2 Types Of Iterators

Iterators can be classified on the basis of access and on their operations.

2.1 Basis of Access

2.1.1 Read-Only Access

Some languages like C++ have the concept of constant iterator in which the data members of the class are not modified [21]. These Iterators have the syntax as follows:

```
vector<std::string>::const_iterator iter
```

They can be used for functionalities like printing out values of a vector which does not change the values of the vector.

2.1.2 Read-Write Access

These are the Iterators which are normally used if there is a need to modify the data members of the class of objects which are stored. They can be used for functionalities like executing the command objects stored in a vector.

2.2 Basis of Operation

Iterators can be classified on the basis of operation as described below: [23]

2.2.1 Input Iterator

It is a Read Only Iterator which moves in forward direction. It is simple and can be used to check for equality, dereferencing and incrementing to the next element in the sequence.

2.2.2 Output Iterator

An output iterator has the opposite function of an input iterator. Output iterators can be used to assign values in a sequence, but cannot be used to access values.

2.2.3 Bidirectional Iterator

Bidirectional iterators are a combination of input and output iterator features. They support the decrement operator, operator - -(), making traversal in forward as well as backward direction through the elements of a container.

2.2.4 Random Access Iterator

They are used in algorithms that have complex operations. They allow values to be accessed by subscript, to obtain the number of elements in a specific range or changed by arithmetic operations. Random access iterators are used in algorithms for computing generic operations like sorting and binary search.

2.2.5 Forward Iterator

A forward iterator is a combination of input and output iterator features. It allows accessing and modifying values.

2.2.6 Reverse Iterator

A reverse iterator yields values in exactly the reverse order of values given by the standard iterators. For a vector or a list, a reverse iterator generates the last element first, and the first element last. For a set it generates the largest element first, and the smallest element last. The table below shows a summary of iterators on the basis of their operations.

Iterator Category	Ability	Providers
Input Iterator	Reads Forward	Istream
Output Iterator	Writes Forward	ostream_iterator(), inserter(),front_inserter(),back_inserter()
Forward Iterator	Reads & writes forward	Hashset
Bidirectional Iterator	Reads & writes forward and backwards	list, set, multiset, map, multimap
Random Access Iterator	Reads & writes with random access	vector, array
Reverse Iterator	Reads & writes backward	vector

2.3 Special Types of Iterators

This section looks at a few special type of iterators like the interruptible iterator, parallel iterator and the zip iterator.

2.3.1 Interruptible Iterator

Interruptible Iterators are enhanced coroutines in which the loop body can interrupt the iterators to perform updates. These kind of iterators provide a convenient & efficient manner of handling additional requests like element removal. Similar to exceptions, interrupts are a nonlocal control mechanism but on the other hand unlike exceptions, a handled interrupt results in resumption, and interrupts propagate logically downwards in the call stack than moving upwards. Interruptible Iterators are a mechanism that extend coroutine iterators to handle update operators through interrupts. The core logic behind an interrupt is that when there is an operation to be performed, the code raises an interrupt which interrupts the iterator. The iterator handles the interrupt, carries out the necessary operations and returns the control back to the place where the interrupt was raised. Iterator interrupts and exceptions are similar as they signal with non-local handlers but also have two essential differences. The first difference is exceptions have termination semantics while interrupts have resumption semantics, and the second difference is the direction of propagation. Regular exceptions propagate up the call stack while interrupts propagate down the call stack. [5]

2.3.2 Parallel Iterator

An important aspect of the parallel iterator is that it focuses on objected oriented programming. The following Java application code that traverses a collection of files and processes each one [6]:

```
Collection<File> elements = ...
Iterator<File> it = elements.iterator();
while ( it.hasNext() ) {
  File file = it.next();
  processFile(file);
}
```

This code only contains one thread - therefore only one processor core will be employed to execute the loop while the other cores remain idle. To use multiple threads to execute this in parallel, new threads need to be created. The following problems arise in such a scenario:
What is the scheduling policy?
How will such a scheme be implemented?
How will such a scheme perform?
In most case, a scheduling policy is implemented manually. Unfortunately, the simplest schemes to implement tend to be the least efficient, while the most efficient schemes tend to be too complicated and error prone.

This is where the Parallel Iterator comes in. It is a thread-safe Iterator that is shared amongst the threads, without worrying about implementing the underlying scheduling policy (including the tedious parallelization concerns). Below is the same example from above but using the Parallel Iterator [6]:

```
Collection<File> elements = ...
ParIterator<File> it =
    ParIteratorFactory.createParIterator(elements,
    threadCount);
while ( it.hasNext() ) {
  File file = it.next();
  processFile(file);
}
```

The difference is that the Parallel Iterator is thread-safe, unlike the standard sequential Iterator. This allows all the threads to share it, while the distribution of the elements is handled internally. Other than that, notice how the Parallel Iterator has the same interface as the standard sequential Iterator. In fact, the Parallel Iterator interface even extends the sequential Iterator's interface.

Since multiple threads are sharing the same Parallel Iterator, it is important that none of the threads continue executing past the loop until all the other threads have also finished. This is because code following the loop might rely on results of the completed loop. Therefore, the call to hasNext() will actually block a thread until all the other threads finish executing their respective iterations. After this block, false is returned to all threads and they therefore break out of the loop together.

2.3.3 Zip Iterator

The zip iterator provides the ability to iterate over several controlled sequences simultaneously. A zip iterator is constructed from a tuple of iterators. Moving the zip iterator moves all the iterators in parallel. [19]

In Python we can print out a tuple as follows:

```
Y1 = [1,2,3,4]
Y2 = [5,6,7,8]
for x1,x2 in zip(Y1,Y2):
  print x1,x2
```

This would give the pairs (1,5), (2,6), (3,7) and (4,8). However, the two size of the two containers must be the same for zip iteration.

Advantages of Zip Iterator

- The first one concerns runtime efficiency: If one has several controlled sequences of the same length that must be processed (e.g., with the for_each algorithm) then it is more efficient to perform just one parallel-iteration rather than several individual iterations. [20]

- The second important application of the zip iterator is as a building block to make combining iterators. A combining iterator is an iterator that parallel-iterates over several controlled sequences and, upon dereferencing, returns the result of applying a functor to the values of the sequences at the respective positions. [20]

3 Use of Iterators in Good Software Practices

3.1 Open Closed Principle

The Iterators can be made as a base class [22] and we can extend and write many new Iterators like Iteration over odd indices or Iteration over even indices by extending the base class. Thus, different Iterators can be made as per one's choice by just extending (open for extension) and the base class is closed for modification. [17]

3.2 Modularity

The Iterators can be made as different classes and hence this leads to modularized code. This helps in easily identifying errors in the particular module or unit and fix them. It also helps in better testing of code and harder for bugs to hide.

3.3 Separation of Concerns

Iterators' logic can be made separate from the main source code and new Iterators can be added, removed and modified as required. [18]

3.4 Abstraction

Iterators provide a level of abstraction and their code can be changed and tested separately.

3.5 Anticipation of Changes

Changes can be anticipated as the software grows and evolves and accordingly pre patched in places which will not break the code's functionality.

3.6 Generality

Iterators can be used interchangeably with different data structures which are similar in representation using Template Method pattern.

3.7 Incremental development

Software can be developed in an incremental manner and Iterators can be changed as per the changes in the data structure representations.

3.8 Consistency

The base class here is consistent and the other iterators inherit from this base/super class and the interface will remain the same. This can be done using Strategy pattern.

3.9 Iterator Pattern

This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Iterator pattern can help in traversal of a particular data structure in different ways without bloating the interfaces. Many new traversal methods can be represented as types of Iterators and created using the Abstract Factory method. The common logic can be abstracted or factored out using the Template Method pattern. [2]

Many modern imperative languages like Java, C#, C++, Python etc. have a built in iterator class.

4 Iterator Alternatives in Programming Paradigms

4.1 Functional Languages

Functional programming is a form of declarative programming. Declarative programming uses expressions to construct programs. And, so in the functional style of programming we consider mathematical functions to represent computation. The main focus is on the 'What?'rather than the 'How?'. As we have learnt in

class, here we would use recursion rather than Iterators to perform list processing. Such functional programming languages which use recursion and don't provide looping constructs are said to be Turing Complete just like most imperative programming languages. CLISP is a Turing Complete language.
However, Imperative programming involves changing the state using commands. Here we focus on the 'How? 'rather that the 'What? '. Most of the popular programming languages these days are from this programming paradigm. Imperative programs give us better control over the flow of control. This gives a programmer a lot of flexibility in order to maximize the efficiency of algorithms and the logic. C++, Julia, Java and Python used for simulations in this report are imperative programming languages. As we have seen in imperative programming paradigm, we make use of Iterators.

We wanted to compare the performance of the two approaches. For this purpose we chose Python from Imperative paradigm (refer section 8.3.1) and CLISP from functional paradigm (refer section 8.3.2). Both are interpreted languages. Though they are listed as Interpreted languages, they are not strictly interpreted. Both these languages are shipped with compilers of their own which compile the code into an intermediate representation (much like the 'Bytecode'representation done in Java). After this intermediate representation is generated, it is interpreted in a regular manner. The main thing to observe is that both these languages use the same process of execution. For the simulations in this report, the programs were executed on *Pegasus* and the results obtained were tabulated. Hence, the comparisons should be fair.
Advantages: The main advantage of using recursion to iterate over lists in CLISP is that, it results in relatively easier code constructs. Such code constructs invariably result in 'elegant'looking programs, which are also easier to understand for anyone new to this language.
Disadvantage: The main disadvantage of this programming language is that, recursion is generally slower and requires more memory (to maintain the call stack). And although it is Turing Complete, it makes it very difficult and cumbersome to be able to write complex algorithms for random memory accesses in aggregated data structures. [8]

4.2 Logic Programming

Logic programming, as the name suggests, is a programming paradigm in which programs are written based on facts and rules. The main way these programs execute is by using an inference engine to validate rules based on the facts in the database.

Prolog is a logic programming language based on 'Predicate Calculus'. Prolog uses 'Tail Call Optimization'to improve its performance when predicates involving tail calls are present. This method is based on discarding the stack frame if there is a deterministic tail call going through, thereby, executing that recursive call with a constant stack space.To iterate over an aggregated data structure (such as an array) in Prolog, we enter in the facts and rules governing each iteration. The facts represent the knowledge, and the rules give a way to derive conclusions based on the facts. Prolog has a built-in backward chaining inference engine that uses these facts and rules to iterate over the list. Such recursion is the only iterative method available in Prolog. However, tail recursion can often be implemented as iteration. The following definition of the factorial function is an Iterative definition because it is tail recursive. It corresponds to an implementation using a while-loop in an imperative programming language.

```
fac(0,1).
fac(N,F) :- N > 0, fac(N,1,F).

fac(1,F,F).
fac(N,PP,F) :- N > 1, NPP is N*PP, M is N-1,
    fac(M,NPP,F).
```

Advantages: The inference engine of Prolog assumes that the database that it inferences from is the only known universe. And hence, everything available inside this universe is taken to be true. As a consequence, anything that is not mentioned in this universe, even if deducible with human intuition, is taken to be false. This is known as the 'Closed World Assumption'. This makes it easy for programmers to enter in only the facts and rules that are necessary for the proper functioning of the program, which utilizes less memory eventually. Hence, Prolog can be used in memory constrained systems.
Disadvantages: The main disadvantage of this programming language is that the inference engine needs to go back and forth to validate the rules based on facts in the database. This backtracking makes processing of the program relatively slower when compared to some popular languages. Another disadvantage is that the 'Closed World Assumption ', although being an advantage, also becomes a disadvantage when the program is expected to deduce certain facts based on the ones already present in the database.

4.3 Data Flow Programming

Lustre is a synchronous data flow language. It is mainly used in reactive systems since they require critical applications.

Iterators in Lustre are mainly used for loop-code generation. Iterators make the generation of loop-code easier because they keep the dependences among array elements simple. Lustre iterators are justified by the following advantages [9]:

1. **Size:** Applying a computation N times requires more than 1 copy of that computation. But, using iterators and hence loop-code we can reduce the number of copies to 1.

2. **Execution Time:** Executing N assignments is almost always slower than executing a loop-code with 1 assignment N times.

3. **Memory:** Using loop-code it is possible to remove unwanted intermediate variables from complex computations.

4. **Readability:** The iterator operators proposed in Lustre are easy to manipulate.

The basic iterator operators used in Lustre are map, red, fill and map-red. To understand them in more detail we can consider that n is an integer with a known value, T and T'are arrays of size n. [9] τ is the type and τ^n is an array of size n and elements of type τ

4.3.1 Operators on Iterators in Lustre

Map: If g = λ t.t'. We can have an abstract syntax of the map operator as T'= map(g,T). Semantically, it is given by:

$$\{T[i] = g(T[i])\} where\ i \in range(T)$$

If the signature of a node N is:

$$\tau 1\ X\ \tau 2\ X\ ...\ X\ \tau l \rightarrow \tau 1'\ X\ \tau 2'\ X\ ...\ X\ \tau k'$$

then map «N,n» gives a node with signature:

$$\boxed{\tau_1 n\ \ x\ \tau_2 n\ \ x\ .\ .\ .\ x\ \tau_l\text{^}n\ \rightarrow \tau_1'\text{^}n\ \ x\ \tau_2'\text{^}n\ \ x\ .\ .\ .\ x\ \tau_k'\text{^}n}$$

Red: If g = λ (t,accu).accu', then r = red(init, T, g) is the reduction of the array T using g. Here init is the initialization expression of the reduction. Semantically

$$r0 = init; ri + 1 = g(ri, T[i])i \in range(T); r = rsize(T)$$

If the signature of node N is:

$$\tau\ X\ \tau 1\ X\ \tau 2\ X\ ...\ X\ \tau l \rightarrow \tau'$$

then red«N,n» is a node of signature:

$$\boxed{\tau\text{^}n\ \ x\ \tau 1\text{^}n\ \ x\ \tau 2\text{^}n\ \ x\ .\ .\ .\ x\ \tau l\text{^}n\ \rightarrow \tau'}$$

Fill: If g =λ the fill is given by T = fill (init,g) where init is the initialization of the filling process. Semantically:

$$\{r0 = init; ri + 1, T[i] = g(ri)i \in range(T)\}$$

If N has the signature:

$$\tau \rightarrow \tau\ X\ \tau 1'\ X\ \tau 2'\ X\ ...\ X\ \tau k'$$

the fill«N,n» is a node with signature:

$$\boxed{\tau \rightarrow \tau 1'\text{^}n\ x\ \tau 2'\text{^}n\ x\ .\ .\ .\ x\ \tau k'\text{^}n}$$

map_red: If g = λ(accu,t).(accu ',t'), then map_-red is given by

$$(T, r) = map\_red(init, T, g)$$

Semantically:

$$\{r0 = init; ri + 1, T'[i] = g(ri, T[i])i \in range(T); \\ r = rsize(T)\}$$

If N has a signature:

$$\boxed{\tau\ \ x\ \tau 1\ x\ \tau 2\ x\ .\ .\ .\ x\ \tau l\ \rightarrow \tau\ \ x\ \tau 1'\ x\ \tau 2'\ x\ .\ .\ .\ x\ \tau k'}$$

then map_red«N,n» has a signature:

$$\boxed{\tau\ \ x\ \tau 1\text{^}n\ x\ \tau 2\text{^}n\ x\ .\ .\ .\ x\ \tau l\text{^}n\ \rightarrow \tau\ \ x\ \tau 1'\text{^}n\ x\ \tau 2'\text{^}n\ x\ .\ .\ .\ x\ \tau k'\text{^}n}$$

5 Simulations

This section consists of the various simulations carried out.

5.1 Performance of Iterator vs For-Loop for a Linked list

In a linked list data structure [4], the elements are not stored sequentially in memory as compared to an array. So, instead of using index operator to retrieve elements like in an array we use the get_element function to retrieve an element at a particular index (refer section 8.1.1). So, if we need element at index 5 we call get function which will scan all elements from index 0 to 4 using the next operation and retrieve the particular element. In an array, it would be just for example data[4]. So, a linked list has O(n) operation and an array has O(1) operation. We have done a simple operation to print out all the elements of a linked list. To go over the size of the linked list would be O(n) operation as the for loop runs over n elements and inside the For loop the get function has O(n) complexity. Hence the total complexity to print out all the elements of a linked list would be $O(n^2)$. However, if we use an Iterator we would be able to store the previous element accessed in the Iterator class as a data member and use that to access the next element (refer section 8.1.2). Then the data member can be updated to point to the value returned. This would result in an O(1) operation and the Iterator would just use two functions i.e., advance which will return the element and the is_done to check if the iteration is over. Thus, the is_done is placed inside a while loop and its complexity would be O(n) as it has to go over all the elements.

So, the combined complexity of the while loop and Iterator advance function would be O(n) + O(1) i.e., O(n) complexity similar to accessing elements in an array. We printed out elements upto 200 thousand and compared the performance of For-Loop vs Iterator in a linked list and our results confirm that For-loop has $O(n^2)$ complexity and Iterator has O(n) complexity.

Advantages: : Iterators are very useful in scenarios where all the elements are to be accessed in sequence and they are not stored in contiguous memory locations.

Disadvantages: : If only one or two specific elements are to be accessed then it would be better to use get function.

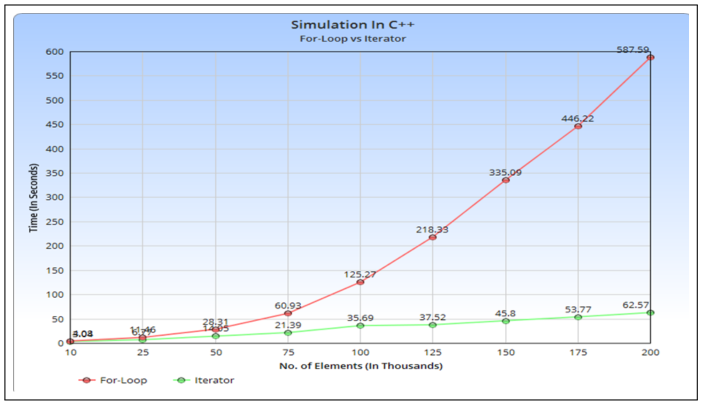


Figure 1: Simulation in C++

5.2 Simulation in Julia

Julia is a high-level, high-performance dynamic programming language for technical computing, founded in 2012. [12] For implementing sequential iteration in Julia we need three methods (refer section 8.2.2):

1. **start(iter)** Returns the initial iteration state
2. **next(iter, state)** Returns the current item and the next state
3. **done(iter, state)** Tests if there are any items remaining

Any object that implements the above three methods is iterable in Julia.

5.2.1 Comparison Of For Loops and Iterators Using Programs Written In Julia

The traditional For Loop and Iterators differ slightly in performance. Both have almost same speed, however the iterator needs more space than For Loop. For Loop only has one counter to keep track of the current index of the collection (refer section 8.2.1). However iterator iterates over each object of the collection and while doing so it keeps track of the object state and as a result needs more space (refer section 8.2.2). Due to the extra space requirement Iterator is slightly slower than For Loop. However, it is possible to use the local availability of object state to our advantage. If we change the loops to have more than one statement that access the value, then in such scenario the Iterator should work faster because the value is locally available. In order to test the above facts and study the performance of Iterators vs the traditional for loop, we carried out the following experiment. We wrote a simple program to calculate the sum of all elements of an array using For Loop (refer section 8.2.1) and Iterator approach (refer section 8.2.2) and observed their execution time for three arrays of integers of size 10 million, 50 million and 75 million.

For testing the second scenario i.e accessing the array more than once, we modified the sum program such that sum was getting calculated using the formula: $new\_sum = previous\_sum + arrayElement + arrayElement + arrayElement + arrayElement + arrayElement$

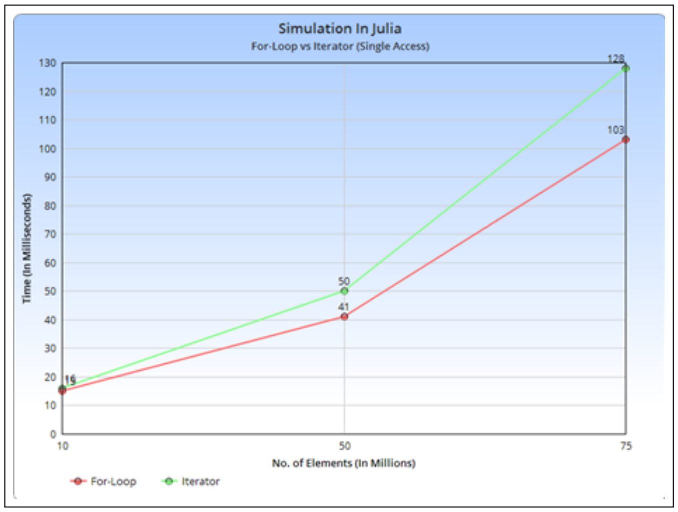


Figure 2: Simulation in Julia - For Loop vs Iterator-Single Access

We observed the following results:

1. When we are trying to access the array once (Figure 2), we can see that the For Loop performs better than the Iterator as the number of records in the array increase.

2. On the other hand, when we are accessing the array multiple times (Figure 3) Iterator always perform better than the For Loop and the time gap between For Loop and Iterators increases drastically as the array size becomes larger. Thus iterators perform better for large array size.

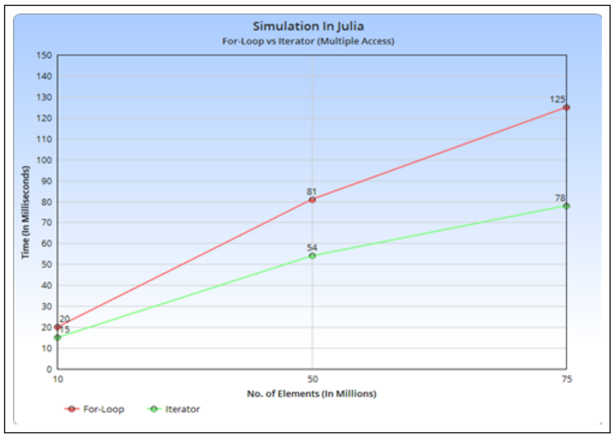


Figure 3: Simulation in Julia - For Loop vs Iterator-Multiple Access

The For Loop is slightly faster than the Iterator if the array size is large and when the array is getting accessed only once per iteration. However if our program accesses the array more than once in each iteration then For Loop is slower and hence it is advisable to use Iterators.

5.3 Performance Evaluation of Iterators in Imperative Paradigm vs Recursion in Functional Paradigm

We created an array-list of elements and performed a linear search on the list in both paradigms. The element to be found did not exist in the list and hence the comparison iterated over the entire array. It would take O(n) time. So, we create an array of 1000, 2000 and 3000 elements respectively, each time initializing every location to 1. In CLISP, we made use of the makelist() function to create the list of the specified number of elements and initialized it to integer value 1. We then searched for the integer value 2. We timed only for the search effort and not for the effort of creating the lists. (refer section 8.3.2)
In Python, we created an iterator and initialized it with the list to be traversed (refer section 8.3.1). Then we compare the current element with the search item. If they are equal we raise a flag and exit the loop, otherwise we call iter.next() on the current item, which fetches the next item and continue with the loop. The Iteration stops if the element is found, and if the list becomes empty it will raise a StopIteration Exception which will stop the Iteration.

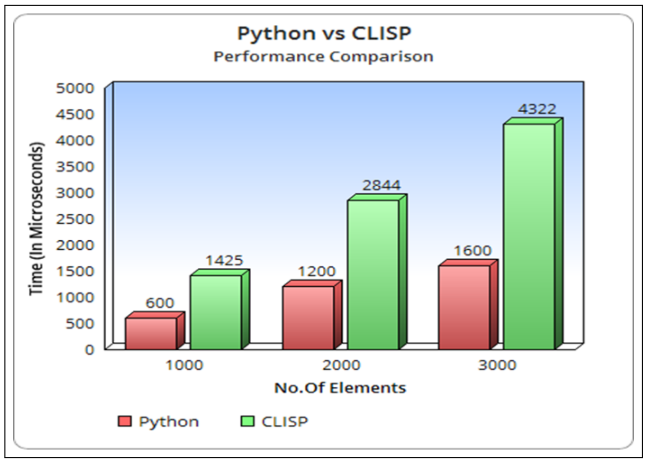
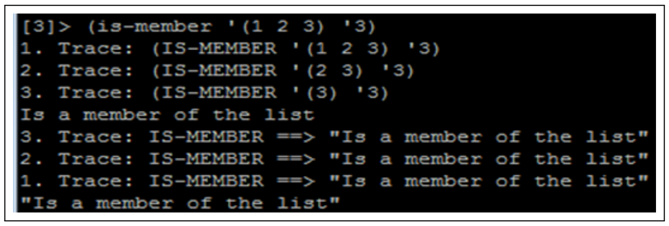


Figure 4: Python vs CLISP - Performance Comparison



```
[3]> (is-member '(1 2 3) '3)
1. Trace: (IS-MEMBER '(1 2 3) '3)
2. Trace: (IS-MEMBER '(2 3) '3)
3. Trace: (IS-MEMBER '(3) '3)
Is a member of the list
3. Trace: IS-MEMBER ==> "Is a member of the list"
2. Trace: IS-MEMBER ==> "Is a member of the list"
1. Trace: IS-MEMBER ==> "Is a member of the list"
"Is a member of the list"
```

Figure 5: Trace for CLISP program

As shown in the CLISP trace above, the car element is used to check if the element is found otherwise we call the function recursively on the cdr, which is essentially the rest of the elements in the list apart from the car element. Then, after all the elements in the list are exhausted the function returns out of the successive function calls and displays that it is not a member of the list. We timed the effort by both approaches and our results showed that Iteration is faster than recursion. This must be due to the fact that the system stack is involved during function calls and the state of operation has to be stored in order to proceed onto another call target. When the element is not found and the function is exiting from recursive function calls the stack has to unwind for the control to get back to the caller, whereas in Iteration it simply runs over all the elements comparing each one and returning if found. The only state that an iterator has to keep track of is the index at which it was iterating before successive calls to iterate.

5.4 Simulation in Java using threads

We tried to evaluate the usefulness of Iterators in a distributed scenario. For that, we made use of threads in Java (refer section 8.4.3). We used an external Jar library called as Par Iterator [11]. We created Job Objects which have a mathematically intensive calculation of raising each element up to 10 million to the power of 1.5 (refer section 8.4.1). Without threads, the Main class will iterate over the Job Objects stored in a container (refer section 8.4.2). Using this library, we can pass the reference of the container and the number of threads and the Worker Threads will internally split the container and perform Iteration over the parts in parallel. As per the experiments we carried out, if we have an array of 8 Job Objects then it would take almost 1/2 the time using 2 threads and 1/4 the time using 4 threads as compared to no threads.

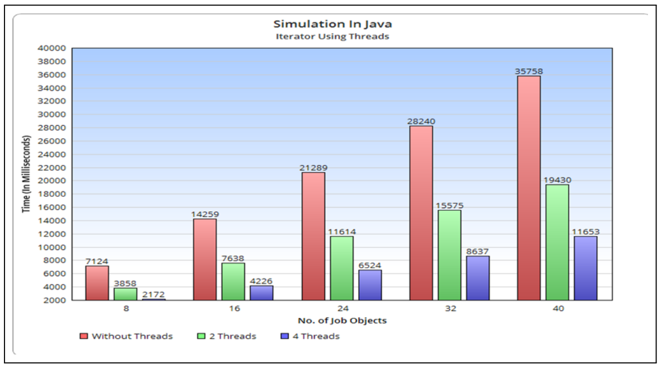


Figure 6: Simulation in Java

- **Pros:** This approach offers a considerable performance boost up. Also, the design of Iterator pattern makes it relatively easy to keep a unified interface rather than having to use for loops each time.

- **Cons:** These Job Objects are considered to be independent of each other. If the Job Objects are dependent then the performance would be slower as the Objects would have to wait for data from other Job Objects.

6 Benefits of Iterators

1. Decoupling of data and algorithm leading to reuse. This means that the logic and the implementation can be separated. For instance if we have an even & an odd iterator and a change needs to be made in the even iterator, only that particular code needs change rather than the entire section.

2. Iterators allow access to large lists in an efficient manner as seen in the simulation carried out on Linked lists.

3. Iterators store execution state which can be used to continue execution from a desired point as illustrated in the simulations done on the linked lists.

4. A single iterator interface can be used for multiple containers. This can be achieved using the iterator pattern as the common traversal logic is placed in a separate iterator class.

7 Drawbacks of Iterators

1. When there are dependencies amongst data structures, use of Iterators can reduce efficiency. This is due to the fact that these data structures may be waiting for some other response in order to execute their task.

2. The structure that is being iterated can't be updated because of the way the iterator stores its position. In the scenario that an string is being iterated over, erasing elements from the same string at the same time is not possible as this reduces the string length.

3. The iterator syntax may look ugly & complicated and hence we can use an enhanced For-Loop (refer section 7.1). The enhanced for loop internally makes use of an iterator. However, in the scenario where one needs to use an even and odd iterator, the enhanced for loop can not be applied.

7.1 Enhanced For-Loop

The for_each is a programming construct used in programming languages. It is a syntactic sugar for using Iterators. It hides the index value and all unnecessary details. After for, generally a variable is written which takes value after colon. This construct is used to simplify the code. The Iterator operations like hasNext, isDone etc. are not called. [7] Eg: Consider the following method, which takes a collection of timer tasks and cancels them:

```
void cancelAll(Collection<TimerTask> c) {
for (Iterator<TimerTask> i = c.iterator();
    i.hasNext(); )
i.next().cancel();
}
```

The iterator is just clutter. Furthermore, it is an opportunity for error. The iterator variable occurs three times in each loop. The for-each construct gets rid of the clutter and the opportunity for error. Here is how the example looks with the for-each construct:

```
void cancelAll(Collection<TimerTask> c) {
for (TimerTask t : c)
t.cancel();
}
```

The for-each construct combines beautifully with generics. It preserves all of the type safety, while removing the remaining clutter.

8 Code Snippets

8.1 C++

8.1.1 For-Loop

Get function printing Linked List using For-Loop:

```
for (int i = 0; i < list.size(); i++)
{
  std::cout<<list.get_element(i)<<std::endl;
}
```

8.1.2 Iterator

Iterator printing Linked List:

```
Linked_List_Iterator<int> li(list);
while (!li.is_done())
{
  std::cout<<li.advance()<<std::endl;
}
```

Element in the iteration of the passed array:

```
template <typename T>
inline
T Linked_List_Iterator<T>::advance (void)
{
  T data = curr_->data_;
  curr_ = curr_->next_;
  return data;
}
```

Check for status of iteration:

```
template <typename T>
inline
bool Linked_List_Iterator<T>::is_done (void)
{
  return curr_ == 0;
}
```

8.2 Julia

8.2.1 For-Loop

Method to Compute Sum of Array Elements Using For Loop:

```
function forLoopSingleAccessMethod(myArray)
  sum=0
     for i in 1:length(myArray)
       sum = sum + myArray[i]
     end
  sum
end
```

8.2.2 Iterator

Method to Compute Sum of Array Elements Using Iterators:

```
function iteratorSingleAccessMethod(myArray)
  sum = 0
  state = start(myArray)
  while !done(myArray, state)
    (iter,state) = next(myArray, state)
      sum = sum + iter
  end
  sum
end
```

8.3 Python & CLISP

8.3.1 Python

Find element using Iterator:

```
iterator = iter(myList)
flag = True
found = False
try:
  while flag:
    item = next(iterator)
    if(search_item == item):
      flag = False
      found = True

except StopIteration:
  pass
finally:
  del iterator
```

8.3.2 CLISP

Find element using recursion (also displays run time):

```
(defun linear-search(A1 X)
  (time (is-member A1 X)
  )
)
(defun is-member (A1 X)
  (cond
    ((null A1) (princ "Not a member of the list"))
    ((= (car A1) X) (princ "Is a member of the
        list"))
    (t (is-member (cdr A1) X)))
)
```

8.4 Java

8.4.1 Job-Class

Execute method for Job Class.

```
void execute ()
{
 double[] results = new double[10000000];
 for (int i = 0; i<10000000; i++)
 {
    results[i] = Math.pow(i, 1.5);
 }
}
```

8.4.2 Iterator without threads

```
int i=0; //counter of job object being processed.
while (Iter.hasNext())
{
 System.out.println(i);
 Iter.next().execute();
 i++;
}
```

8.4.3 Iterator with threads

Create and start a pool of worker threads:

```
Thread[] threadPool = new
    WorkerThread[threadCount];
final long startTime = System.currentTimeMillis();
for (int i = 0; i < threadCount; i++)
{
 threadPool[i] = new WorkerThread(i, pi);
 threadPool[i].start();
}
```

9 Critique

We initially planned to compare across 3 programming paradigms, namely, Imperative, Functional and Logical. However, we ran into problems while trying to capture the accurate time with the desired decimal precision. The time was being shown as 0.0 seconds but we wanted the time in microseconds comparable to those taken for Python and CLISP. Below is the code snippet of the program we tried to implement, which works perfectly.

```
prol(L1,L3) :- do_list(L1, L2),list_sum(L2,L3).
do_list(N, L) :- do_list1(N, [], L).
do_list1(0, L, L) :- !.
do_list1(N, R, L) :- N > 0, N1 is N-1,
    do_list1(N1, [1|R], L).
list_sum([], 0).
list_sum([Head | Tail], TotalSum) :-
list_sum(Tail, Sum1),
TotalSum is Head + Sum1.
```

While working on the simulations across different programming constructs and various paradigms we missed the subtle differences that different programming languages in the same paradigm have. For example, in C++ we generally use the is_done method to stop iteration when the list is completely traversed, whereas, in Python we raise the StopIteration exception. We could not incorporate these kind of syntactic and semantic variations of iterators in our report.

While doing the simulations for CLISP, we found that CLISP throws a stack overflow exception if the number of elements in the input list exceeded 3383. This proved to be a big challenge for us because we were unable to get good time readings for the Python program for such low numbers of input size of the lists. Hence we had to repetitively run the Python program numerous times for the input sizes of 1000, 2000 and 3000. We then had to average out the time values for such repetitions and consider them for 1000, 2000 and 3000 list sizes. This is how we were able to tackle this challenge.

CLISP and Prolog are not exclusively meant for complex algorithmic computations. CLISP is mostly used for List Processing and creation of macros (meta-programming) [16] Prolog is mainly used in Artificial Intelligence systems. In our simulations we tried to emulate the features which are mainly executed using imperative programming languages. In essence, we were stretching the capabilities of CLISP and Prolog. Therefore, our naive representations of Iterators in these languages were not the best representations of what these programming languages are actually capable of.

Also, we could not accurately emulate iterators in CLISP and Prolog. This is mainly due to the fact that we have been learning programming in imperative programming languages. We are considerably new to functional and logical programming paradigms. We have applied the knowledge we have gained in our course and some research done ourselves. Time constraints also played a major part in restricting our research abilities.

References

[1] David Alex Lamb *Specification Of Iterators* Dept. of Comput. & Inf. Sci., Queen's Univ., Kingston, Ont., Canada Software Engineering, IEEE Transactions - Volume:16 , Issue: 12 , Dec 1990

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides *Design patterns* Addison-Wesley Publishing Company, 1995

[3] Sterling and Shapiro *The Art of Prolog* MIT Press, Cambridge, Mass. 1986.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, 3rd Edition* MIT Press, 2009

[5] Jed Liu Aaron Kimball Andrew C. Myers *Interruptible Iterators* Department of Computer Science Cornell University liujed,ak333,andru@cs.cornell.edu

[6] http://homepages.engineering.auckland.ac.nz/ parallel/ParallelIT/PI_QuickStart.html

[7] https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html

[8] https://www.gnu.org/software/emacs/manual/html_node/eintr/Recursion.html#Recursion

[9] Lionel Morel *Efficient compilation of array iterators for Lustre* SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002) , Volume 65, Issue 5, July 2002, Pages 19 - 26

[10] http://nvie.com/posts/iterators-vs-generators/

[11] http://homepages.engineering.auckland.ac.nz/ parallel/ParallelIT/PI_API/pi/ParIterator.html

[12] http://julialang.org/

[13] http://www.chartgo.com/

[14] Babara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert *Abstraction Mechanisms in CLU* Massachusetts Institute of Technology

[15] Barbara Liskov *A History of CLU* Laboratory for Computer Science , Massachusetts Institute of Technology, Cambridge, MA 02139 , April l992

[16] http://www.nist.gov/lispix/doc/lispix/lisp-new.htm

[17] Robert C Martin *Design Principles and Design Patterns*, 2000 [online] Available: http://www. objectmentor.com

[18] Edsger W. Dijkstra *Selected Writings on Computing: A Personal Perspective* Springer-Verlag, 1982. ISBN 0 387 90652 5 Pages: 60 - 66

[19] https://docs.python.org/3.3/library/functions.html#zip

[20] http://www.boost.org/doc/libs/1_41_0/libs/iterator/doc/zip_iterator.html

[21] https://msdn.microsoft.com/en-us/library/352sf8za.aspx

[22] http://www.cplusplus.com/reference/iterator/iterator/

[23] http://stdcxx.apache.org/doc/stdlibug/2-2.html