Final Report: Performance and Memory Usage Comparison Between Haskell (Functional List) and Python

# 1. Introduction

Objective: To compare the performance (time), memory usage, and potential benefits of parallelization for list operations in Haskell (functional list) and Python (imperative programming).

Operations Tested: Create, traverse, prepend, append, map, filter, reverse.

Test Environment: Both implementations were run with a functional list of 1,000,000 elements in Haskell and a regular Python list of 1,000,000 elements to assess scalability with large data sets.

# 2. Methodology

The following operations were implemented and measured:  
• Create: Initialize a list with 1,000,000 elements.  
• Traverse: Traverse the list and apply a simple operation (e.g., return each element).  
• Prepend: Prepend an element to the list.  
• Append: Append an element to the list.  
• Map: Apply a function (\*2) to each element of the list.  
• Filter: Filter out elements based on a predicate (e.g., even numbers).  
• Reverse: Reverse the list.

# 3. Performance Comparison (Time)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Haskell (Functional List) (Time in seconds) | Python (Time in seconds) | Difference |
| Create | 2.796875 | 0.370762 | Haskell (Functional List) is slower |
| Traverse | 1.515625 | 0.985402 | Haskell (Functional List) is slower |
| Prepend | 0.578125 | 0.155354 | Haskell (Functional List) is slower |
| Append | 0.937500 | 0.159659 | Haskell (Functional List) is slower |
| Map | 1.343750 | 1.370818 | Python is slightly slower |
| Filter | 1.109375 | 1.141082 | Python is slightly slower |
| Reverse | 1.484375 | 0.151199 | Haskell (Functional List) is slower |

# 4. Memory Usage Comparison

Memory Usage can be difficult to measure directly in Haskell because of its lazy evaluation. It doesn't allocate memory for elements until they are needed, which can reduce memory usage in some cases but may also cause high memory usage spikes if large parts of the list are evaluated all at once.  
Python, being an eager language, allocates memory upfront for all list elements, so memory usage is predictable but may be higher for large lists.

# 5. Parallelization Potential

## Haskell (Functional List):

Haskell has strong built-in support for parallelism and concurrency with its `Control.Parallel` library, allowing operations to be split across multiple processors. The functional nature of Haskell makes it well-suited for parallel execution, as data can be processed independently.  
Parallelism is expected to show a significant benefit for operations like `map` and `filter` on large datasets.

## Python:

Python has parallelism options available through libraries such as `multiprocessing`, `concurrent.futures`, and `joblib`.  
Python’s parallelism is easier to implement with simpler concurrency models like threads and processes, but the Global Interpreter Lock (GIL) limits true parallelism for CPU-bound tasks.

## Parallelization Comparison:

Haskell (Functional List) would likely show better performance when parallelizing operations like `map` and `filter` due to its functional nature and support for parallelism. Python could benefit from parallelism but is limited by the GIL, especially in CPU-bound tasks.

# 6. Conclusion

Performance: Python generally performs better for list operations due to its eager evaluation and optimized memory management. Haskell (Functional List), with its lazy evaluation, shows slower performance for many operations but may offer better performance for operations that benefit from lazy evaluation.  
Memory Usage: Haskell (Functional List) can potentially use less memory when working with large datasets due to lazy evaluation, while Python allocates memory upfront.  
Parallelization: Haskell (Functional List) is better suited for parallelization due to its immutability and efficient parallel computation libraries. Python’s parallelism is hindered by the GIL for CPU-bound operations.