Optimization Technique and Implementation Project Report

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Professor Brandon Bass

Student: Suresh Ghimire

UC ID: 005038288

Date: August 9, 2025

Introduction

High-Performance Computing (HPC) systems depend on optimized algorithms and data structures to handle large-scale problems effectively. While parallelism is a key factor for performance, considering efficiency of the memory hierarchy is nearly as important (Hennessy & Patterson, 2019). The literature recently has shown through many different empirical studies that reformulations targeting data locality and data-structure level of efficiency can lead to significant performance improvements (Yoo et al., 2013).

In this project, we will evaluate one of these optimization approaches, which replaces pointer-based data structures with contiguous arrays and vectorized operations and pre-allocated buffers as a Python implementation. The goal is to replicate the performance improvement type and effect found in HPC applications and quantify these benefits with a controlled experiment.

Background

An empirical study of HPC performance bugs found that data structure optimization and re-ordering memory access are two of the ways to improve performance (Yoo et al., 2013). Transaction time (including memory management) can be improved specifically by replacing linked lists and fragmented memory with contiguous data structures (e.g., using an array rather than allocating memory for each entry or varying the memory structure for the HPC task). Better performing software has a lot of uses for core advances especially for cache utilization (ensuring contiguous data is retained nearby) by improving cache locality (temporal and data locality improves cache exploitation), thereby reducing TLB page misses (Jia et al., 2018); data LOC is optimized or vectorized and using SIMD (single-instruction, multiple-data) where data structures are memory layouts that can be SIMD friendly and high-level vectorized operations (Intel, 2022); avoid heap overflow inside a loop reduces dynamic memory allotment for fragmentation and having to repeat the overhead of allocation for subsequent iterations or memories (Lam et al., 2015).

Optimization Technique

The optimization technique described in the Empirical Study on High Performance Computing Performance Bugs (Yoo et al., 2013) is data structure optimization specifically to improve data locality. Data locality relates to how data is accessed in memory over the course of a computation. In other words, we want the CPU to access data as it is requested, and if it has good locality, the CPU will most likely be requesting that data from the fast-access cache, rather than from the slower main memory.

In high performance computing (HPC), data locality is important because modern processors can execute instructions much faster than the time it takes to access data in memory. This means poor data locality will lead to a lot of cache misses, forcing the CPU to stall, waiting for that data to come out of memory, and thus destroying your application's performance. The finding of the empirical study stated many performance bugs can be fixed if programmers transferred their scattered, pointer-based data structures to contiguous data structures, which allow the CPU to fetch the data more compellingly using the pre-fetching hardware, as well as allowing the prefetching data to be reused more effectively.

Why data-locality technique?

There are several reasons why the data-locality is effective on high performance calculation. Followings are some reasons:

- **Hight impact in HPC system**: The studies to fix performance defects for optimizations that change the data locality has a significant composition of all performance fixes. The approaches studied have been tested in the high-performance computing realm using real workloads, so there's evidence that it is one of the best ways you can improve execution speed of scientific and engineering workloads.
- **Applicable in python**: Even with it's criticism that Python is slower than compiled languages, you can use libraries like NumPy to access contiguous memory layouts and low-level optimizations as you would expect from C or Fortran from Python code.
- **Relevance data structure performance**: The implementation is concerned with data structure performance. This technique is directly related to data storage and memory access.

Implementation in python

To illustrate the effect of improved data locality, we replaced the original representation of points from a Python list of tuples with a NumPy ndarray, which stores elements as a contiguous block of memory. This means that all of the coordinate values will be stored closer together in memory, which in theory should largely benefit cache hits and reduce latency of memory access.

Though we have the new layout in memory as the main optimization, there are two corresponding practices that were used to further the gains:

- **Vectorization**: Instead of calculating the pairwise distance with nested Python loops, the implementation allows NumPy to broadcast the operation and effectively executes it as a single bulk operation. Not only does this allow the optimized C and BLAS routines take care of the maneuvers involved (which can take advantage of CPU vector instructions SIMD internally but it also allows us to benefit from using all the available cores on the CPU at once.
- **Pre-allocation**: The output distance matrix is pre-allocated with a call to np.empty() so that repeated calls to "new" memory do not occur during the computation of the total distance. This will ensure that fragmentation of the heap will be reduced, and all allocation overhead will be taken out of the inner loops.

Ultimately, there are not independent optimization techniques from these strategies for the purpose of this project, they were merely implementation strategies that enabled the optimization of the data structure implementational technique.

Strength and Weakness

Strength: Followings are the strengths of data-locality on high performance computing

- **Performance Gain:** Data-locality lowers the execution time by having low cache miss and interpret overhead
- **Memory Efficiency:** Since the execution reduced the pre-element overhead on python objects because of contiguous storage. It cause low memory use.
- **Scalability:** As the dataset size grows it can be scale horizontally, making it beneficial for HPC

- **Broad Applicable:** Popular languages like python, C, C++ can be used to apply this technique.

Weakness: Followings are the weaknesses of the high-performance computing

- **Learning Curve:** As it involves computing performance, it requires high learning curve on libraries like NumPy
- **Limited Flexibility:** This technique works well with the fixed dataset. It may not be efficient on dynamic datasets.
- **Memory trade-off:** Pre-allocation of large contiguous dataset may cause capturing unused space.

Code Implementation

Following code snipped shows the baseline implementation for the high-performance computation. The data will be generated on csv file

```python
HighPerformanceComputingBenchmark.py > ...
import numpy as np
import math, random, timeit
import tracemalloc
import matplotlib.pyplot as plt
import pandas as pd

# Baseline: Python lists + loops
def baseline_pairwise(n=1000):
    points = [(random.random(), random.random()) for _ in range(n)]
    distances = []
    for i in range(len(points)):
        row = []
        for j in range(len(points)):
            dx = points[i][0] - points[j][0]
            dy = points[i][1] - points[j][1]
            row.append(math.sqrt(dx*dx + dy*dy))
        distances.append(row)
    return distances

# Optimized: NumPy vectorization + preallocation
def optimized_pairwise(n=1000):
    points_np = np.random.rand(n, 2).astype(np.float32)
    distances_np = np.empty((n, n), dtype=np.float32)
    diffs = points_np[:, np.newaxis, :] - points_np[np.newaxis, :, :]
    distances_np[:] = np.sqrt(np.sum(diffs**2, axis=-1))
    return distances_np
```

```python
# Helper: measure time and memory
def measure(func, *args, **kwargs):
    tracemalloc.start()
    start_time = timeit.default_timer()
    _ = func(*args, **kwargs)
    elapsed = timeit.default_timer() - start_time
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return elapsed, peak / (1024*1024)  # seconds, MB

# Run experiments
sizes = [200, 400, 600, 800, 1000]
baseline_times, optimized_times = [], []
baseline_mem, optimized_mem = [], []

for n in sizes:
    print(f"Running baseline for n={n}...")
    t, m = measure(baseline_pairwise, n)
    baseline_times.append(t)
    baseline_mem.append(m)

    print(f"Running optimized for n={n}...")
    t, m = measure(optimized_pairwise, n)
    optimized_times.append(t)
    optimized_mem.append(m)

# Save results table
df = pd.DataFrame({
    "n": sizes,
    "Baseline Time (s)": baseline_times,
    "Optimized Time (s)": optimized_times,
    "Baseline Mem (MB)": baseline_mem,
    "Optimized Mem (MB)": optimized_mem,
    "Speedup (x)": [b/o for b, o in zip(baseline_times, optimized_times)]
})

df.to_csv("performance_results.csv", index=False)
```

```python
df.to_csv("performance_results.csv", index=False)
print("\nSaved results table as performance_results.csv")

# Plot results
plt.figure(figsize=(8,5))
plt.plot(sizes, baseline_times, 'r-o', label="Baseline (lists)")
plt.plot(sizes, optimized_times, 'g-o', label="Optimized (NumPy)")
plt.xlabel("Number of Points (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Baseline vs Optimized Performance")
plt.legend()
plt.grid(True)
plt.savefig("performance_comparison.png")
print("Saved performance plot as performance_comparison.png")

# Display summary
print("\n\tBenchmark Summary")
print(df)
```

Results and analysis

Following screenshot shows the results of the above script

```
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Final_Project> python -u "c:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Final_Project\HighPerformanceComputingBenchmark.py"
Running baseline for n=200...
Running optimized for n=200...
Running baseline for n=400...
Running optimized for n=400...
Running baseline for n=600...
Running optimized for n=600...
Running baseline for n=800...
Running optimized for n=800...
Running baseline for n=1000...
Running optimized for n=1000...

Saved results table as performance_results.csv
Saved performance plot as performance_comparison.png

       Benchmark Summary
      n  Baseline Time (s)  Optimized Time (s)  Baseline Mem (MB)  Optimized Mem (MB)  Speedup (x)
0   200           0.150320            0.004106           1.253510            0.950352    36.612441
1   400           0.641152            0.008210           4.933083            3.697742    78.091166
2   600           1.581484            0.021948          11.390175            8.276905    72.056301
3   800           2.236443            0.026916          19.994423           14.687122    83.088195
4  1000           3.062934            0.031531          31.400978           22.928394    97.139157
```

The benchmark was run on datasets of 200, 400, 600, 800, and 1,000 data points. For each dataset size, the baseline implementation (Python lists with nested loops) and optimized implementation (NumPy arrays with vectorization and pre-allocation) were also run. Execution time and peak memory usage were captured as well as the calculated speedup factor.

Following table shows the output of the above computational script

N(points)	Baseline Time(s)	Optimized Time(s)	Baseline Memory (MB)	Optimized Memory (MB)	Speedup
200	0.150320	0.004106	1.253510	0.950352	36.61
400	0.641152	0.008210	4.933083	3.697742	78.09
600	1.581484	0.021948	11.390175	8.276095	72.06
800	2.236443	0.026916	19.994423	14.687122	83.09

1000	3.062934	0.031531	31.400978	22.928394	97.14

Observation

Reduction in Time to Complete Task:

- The optimized version exhibits speedup ranging from 36× (for the smallest dataset) to over 97× (for the largest dataset).
- The largest gains are associated with larger datasets because of reductions in Python interpreter overhead as well as better cache efficiency for vectorized operations.

Memory Savings:

- Memory usage is approximately 20–30 % lower for all dataset sizes.
- This is due to contiguous storage in NumPy arrays, which eliminates the overhead of Python objects for each element.

Scalability:

- The performance improvement gap increases with size of input dataset, thereby demonstrating that the proportional benefits of vectorization and data-locality are bigger for larger computations.
- The optimized approach is scalable with respect to computation time but has a much smaller constant factor.

These results are consistent with Yoo et al. (2013), who cite optimization of data structures, loop transformations, and modifications to memory access patterns, as major contributors to the performance gains attained in HPC applications.

Conclusion

This project illustrated that, when deploying data-locality optimization, vectorization, and pre-allocation in tandem, it had significant performance improvements of up to 97× faster execution time and 20–30% less memory than a baseline Python list-based implementation. These results are in line with the research as reported in the Empirical Study of High Performance Computing Performance Bugs (Yoo et al., 2013), which indicated that data structure optimizations and memory access optimizations tend to be two of the best they found for HPC. The scalability observed demonstrates that while small datasets may not benefit as much, more efficient memory layout and bulk computation will not only lead to improved performance at a large scale, but actually yield performance that could be equal to a hardware upgrade (e.g., reduced the memory requirements that normally requires a few hundred GB of RAM), thus representing a really beneficial strategy to employ for research prototypes and production scale scientific computing modules.

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Hennessy, J. L., & Patterson, D. A. (2019). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

Intel. (2022). *Intel® 64 and IA-32 architectures optimization reference manual*. Intel Corporation.

Jia, Z., Zaharia, M., & Aiken, A. (2018). Beyond data and model parallelism for deep neural networks. *Proceedings of the 2nd Conference on Systems and Machine Learning*.

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Yoo, R. M., Romano, A., & Amarasinghe, S. (2013). An empirical study of high performance computing performance bugs. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 15–26.