Comparisons

Persistent Array
Persistent LinkedList
Persistent Stack
Persistent Queue
Persistent Search Tree

We have done BenchMarking using: GoogleBenchMark Tool
https://github.com/google/benchmark

PROJECT MEMBERS: ANANNYO DEY, SOUMYAJIT RUDRA SARMA , DEBASMIT ROY, KANKO GHOSH AND KUSHAL DAS

Persistent Array

Persistent Array

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Array

Strategies	Update(index, version, newVal)	RetrieveData(index, version)
Copy_On_Write	O(length_of_array_at_version) ~ O (N)	O(1)
Fat_Node	**O(1)**	**Between O(1) and O(V)[Worst Case] V = number of versions O(1) (Average)**
Log_log_time method (Deitz, 1989)	**O(log(log(min(n, v))), n = size of array v = number of versions**	**O(log(log(min(n, v))), n = size of array v = number of versions**

Persistent Array

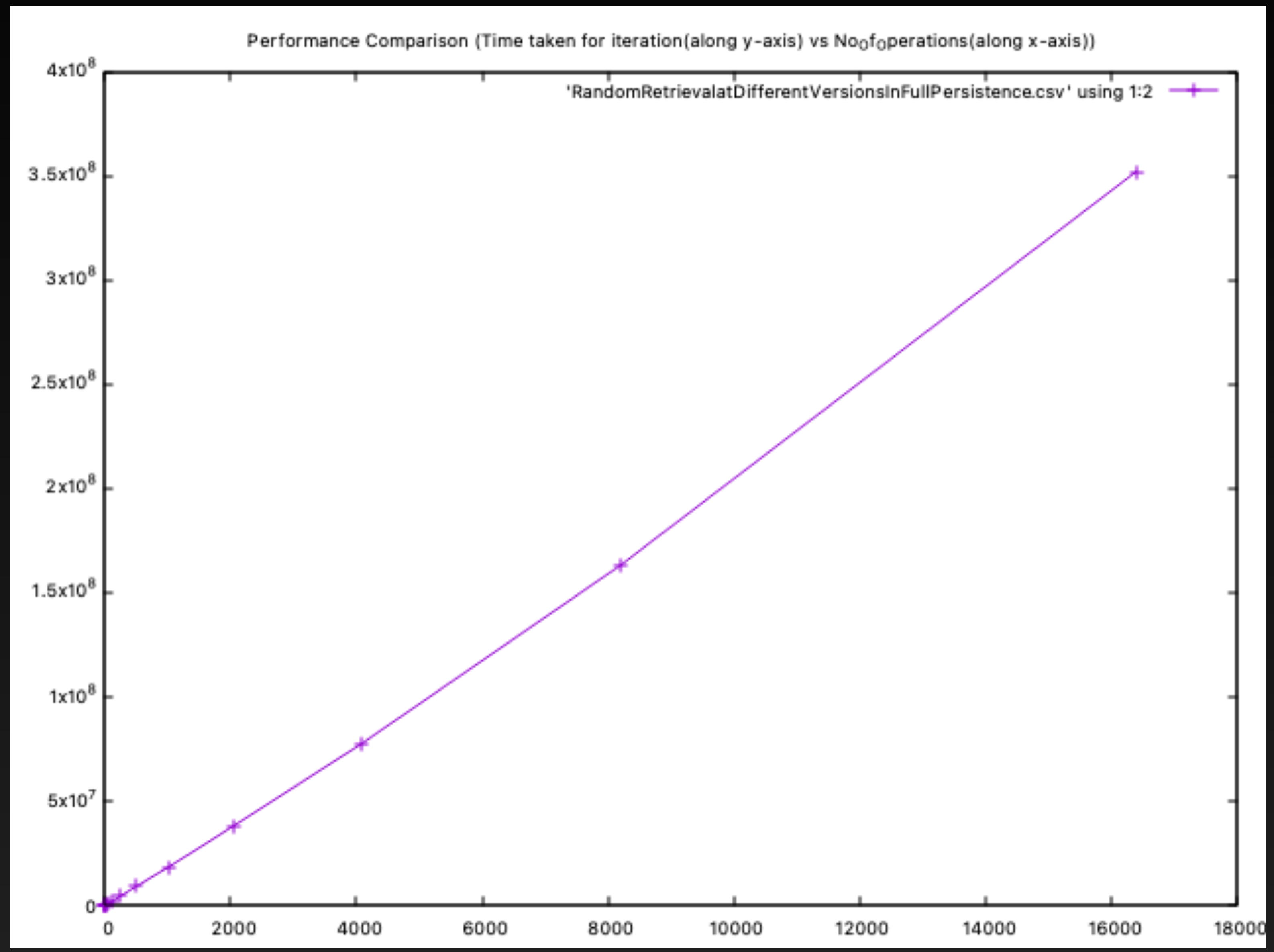
Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Array

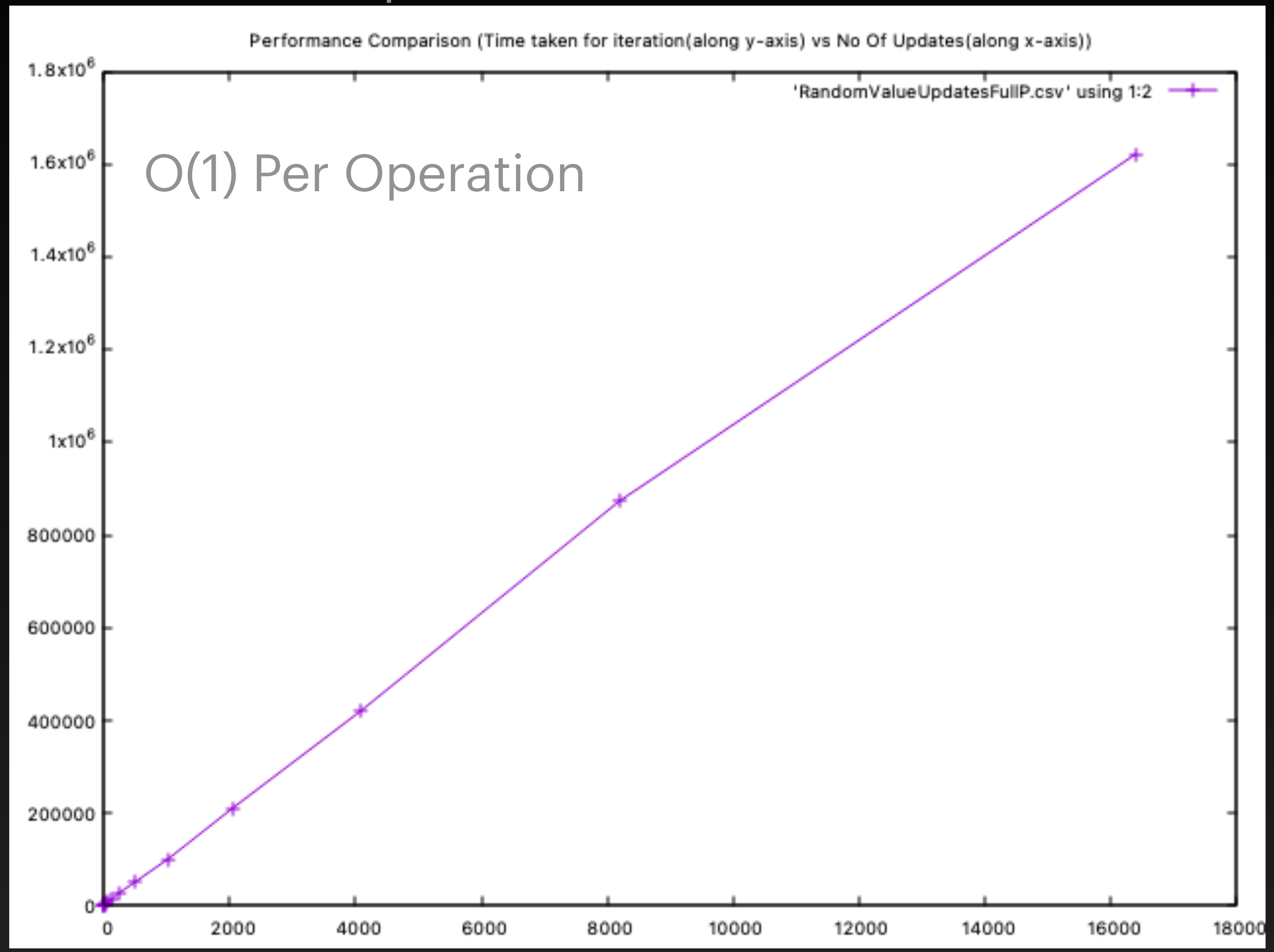
Strategies	Category1	Category2
Copy_On_Write	~ O (NV) To Hold All The Copies of Array At Different Versions	O(V) To Hold The Mapping From Version To Array
Fat_Node	**O(N + V)** **N = size of initial array,** **V = number of version**	**O(V)** **To Hold The Mapping From Version To immediate ancestor version**
Log_log_time method (Deitz, 1989)	**O(N + V)** **N = size of initial array,** **V = number of version**	**O(V)** **to arrange the versions in form of list order maintenance tree**

Benchmarking Of FatNode Model

RandomRetrievalatDifferentVersionsinFullPersistence

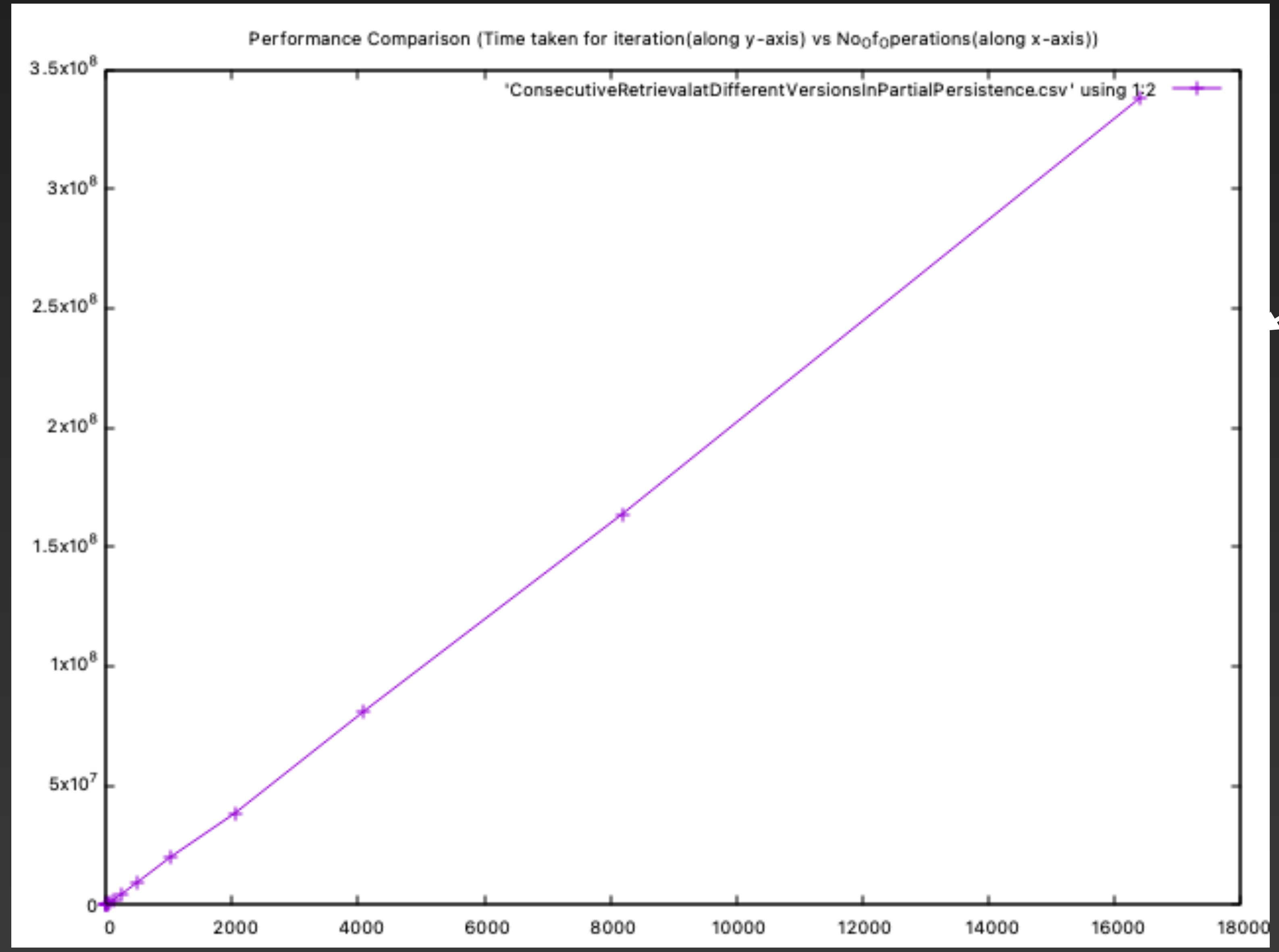


RandomValueUpdatesFullPersistence

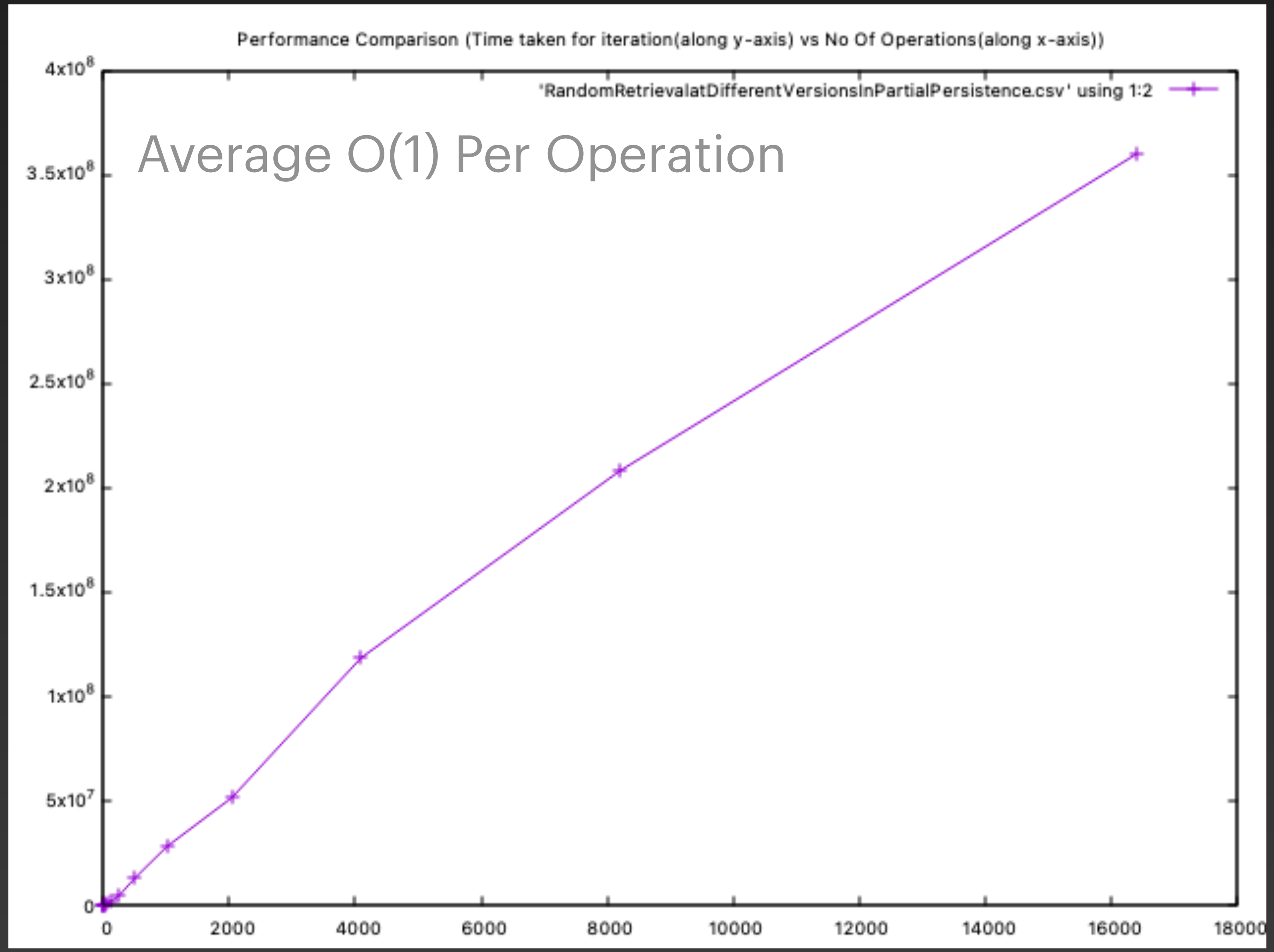


Average O(1) Per Operation

ConsecutiveRetrievalatDifferentVersionsinPartialPersistence



RandomRetrievalatDifferentVersionsinPartialPersistence



Partial Persistent Linked List

Partial Persistent Linked List

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions,
m = no. of modifications in a particular position

Strategies	InsertAfter(position) #	DeleteAfter(position) #	UpdateData (position) #	RetrieveData (postition,version)	traverseWholeLL atVer(version)
Fat Node	O(1)	O(1)	O(1)	O(log_x m) [If the versions are stored in a Balanced x_array Tree/ Trie] O(1) [Amortised]	O(N * log_x m)
Path Copying	O(1) for at extreme position O(N) In Average	O(1) for at extreme position O(N) for at Rear	O(1) for at extreme position O(N) for at Rear	O(N)	O(N)
Pointer Machine	O(1) [Amortised]	O(1) [Amortised]	O(1) [Amortised]	O(N)	O(N)
Ephemeral Linked List	O(1)	O(1)	O(1)	O(N) [Only Current Version Supported]	O(N) [Only Current Version Supported]

time to reach the node at that position is not considered

Partial Persistent Linked List

Auxiliary Space

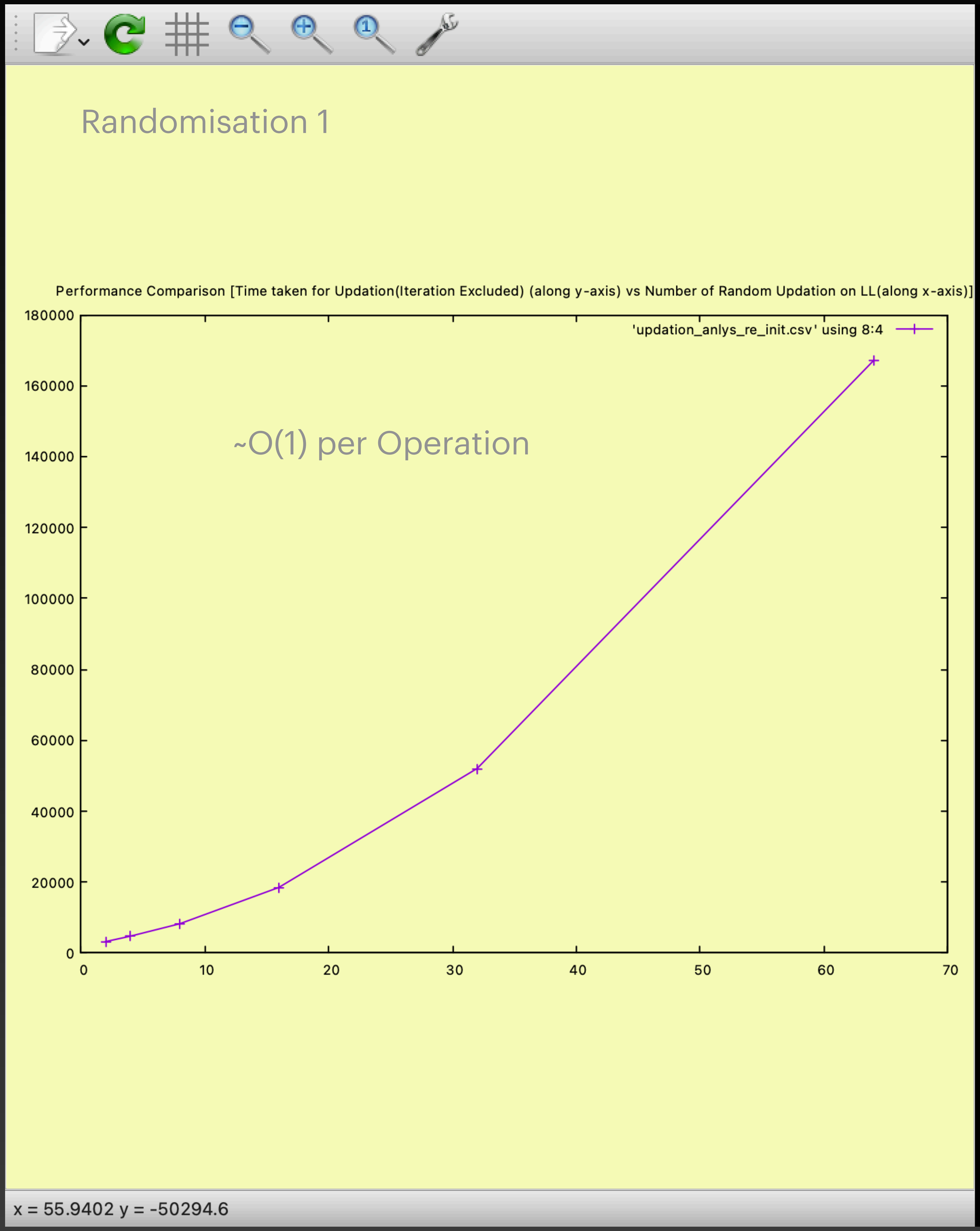
Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

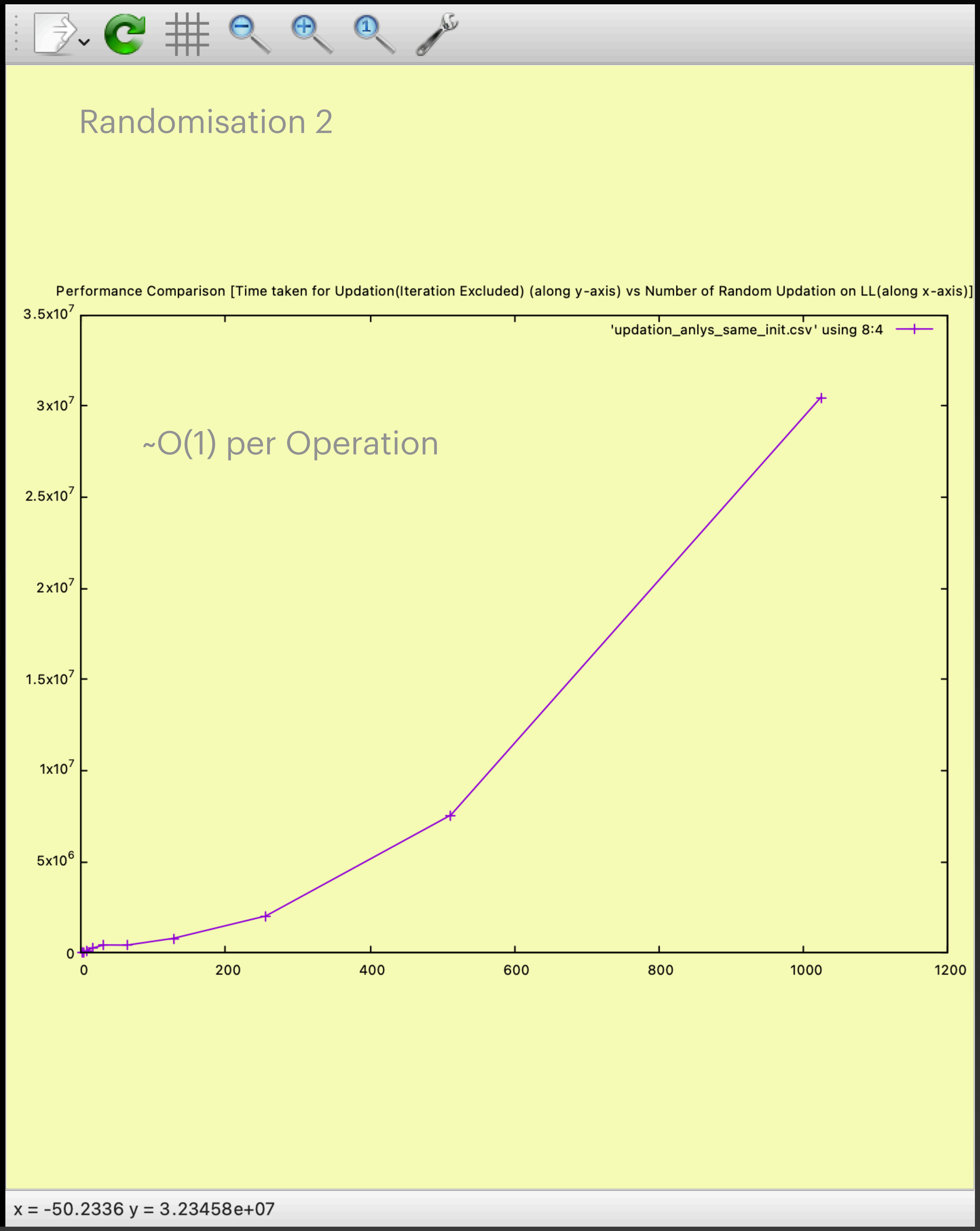
Strategies	Category 1	Category 2
Fat Node	Node Size: # ~ 20byte	$O(N + V)$ to Hold The LinkedList
Path Copying	Node Size: # ~ 12 byte	Best Case: $O(N) + O(V)$ Worst Case: $O(N^2) + O(V)$ to Hold The Tree and Staring Pointers
Pointer Machine	Node Size: # ~ 140 byte	$O(V)$ Amortised to Hold The LinkedList
Ephemeral Linked List	Node Size: # ~ 12 byte	$O(N)$ to Hold The LinkedList

to store 4 byte Integer | 8 Bye pointers

Benchmarking Of Pointer Machine Model

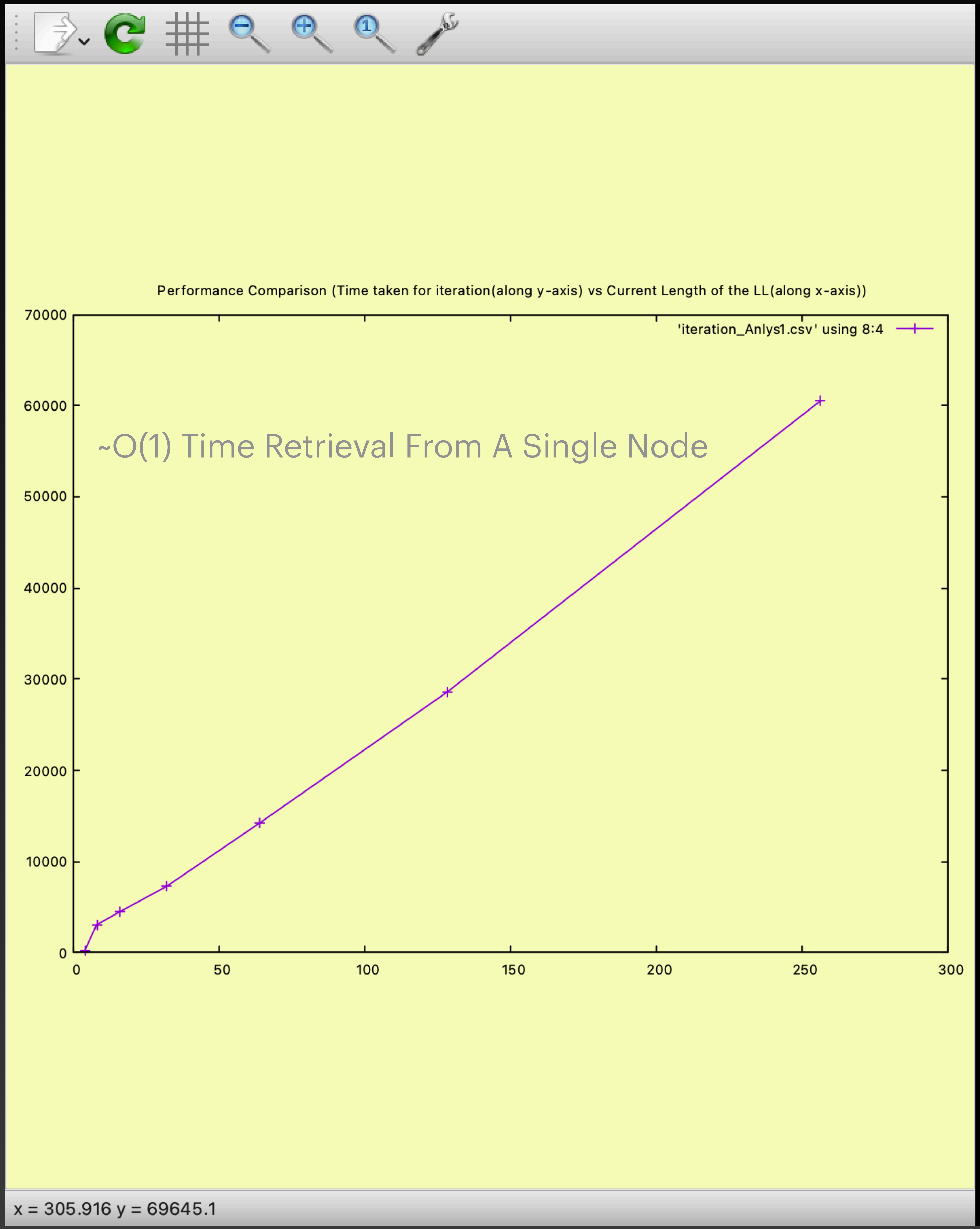
Updation (Value/ Insertion/ Deletion) Randomised

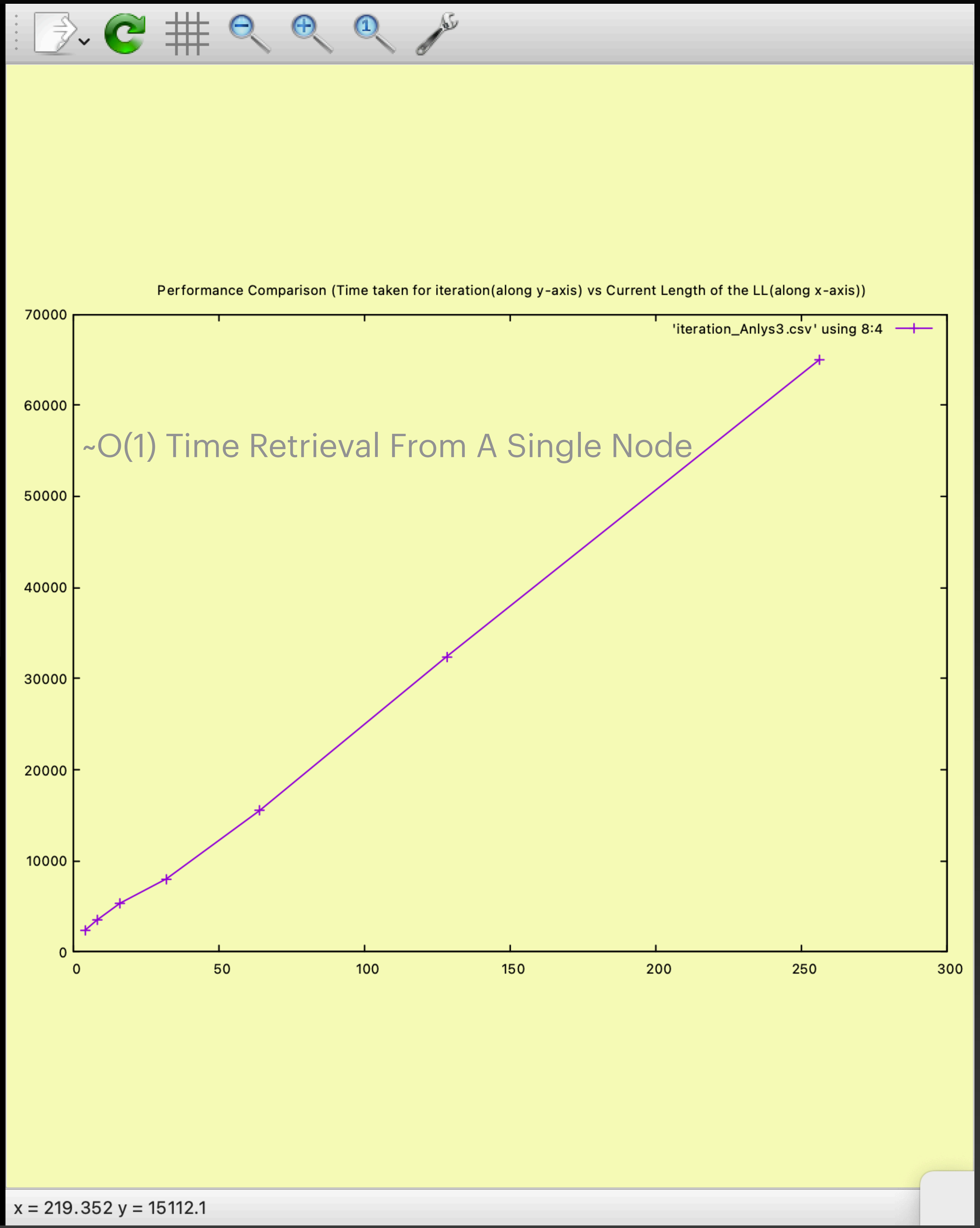




Benchmarking Of Pointer Machine Model

Iteration Through
Whole LL
At
Randomised Versions





Full Persistent Linked List

Full Persistent Linked List

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

Strategies	InsertAfter(position,ver) #	DeleteAfter(position,ver) #	UpdateData (position,ver) #	RetrieveData (postition,version)	traverseWholeLL atVer(version)
Fat Node Pointer Machine [With List Maintenance]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(N) + O(1) [Amortized] [additional amortised O(1) due for look up at version tree]	O(N) * O(1) [additional amortised O(1) due for look up / insertions at version tree]
Path Copying	O(1) for at front O(N) for at Rear	O(1) for at front O(N) for at Rear	O(1) for at front O(N) for at Rear	O(N)	O(N)
Pointer Machine [With List Maintenance]	O(1) [Amortised]	O(1) [Amortised]	O(1) [Amortised]	O(N)	O(N)
Ephemeral Linked List	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(N) [Only Current Version Supported]	O(N) [Only Current Version Supported]

Full Persistent Linked List

Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

Strategies	Category 1	Category 2	Category 3
Fat Node Pointer Machine [With List Maintenance]	Node Size: # ~ 20byte	O(N + V) to Hold The LinkedList	O(V) to hold the Version Tree Here A ScapeGoat Tree
Path Copying	Node Size: # ~ 12 byte	Best Case: O(N) + O(V) Worst Case: O(N^2) + O(V) to Hold The Tree and Staring Pointers	-
Pointer Machine [With List Maintenance]	Node Size: # ~ 200 byte	O(V) [Amortised] to Hold The LinkedList	O(V) to hold the Version Tree Here A ScapeGoat Tree
Ephemeral Linked List	Node Size: # ~ 12 byte	O(N) to Hold The LinkedList	-

to store 4 byte Integer | 8 Bye pointers

Persistent Stack

Persistent Stack

Model Developed By Our Team

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Stack Considering All The Versions

Strategies	push(data, version)	pop(version)	getTop(ver)
Using PPL with PM Model	O(1)	O(1)	O(1)
DAG Model	O(1)	O(1)	O(1)
Ephemeral std::stack (C++)	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]

Persistent Stack

Model Developed By Our Team

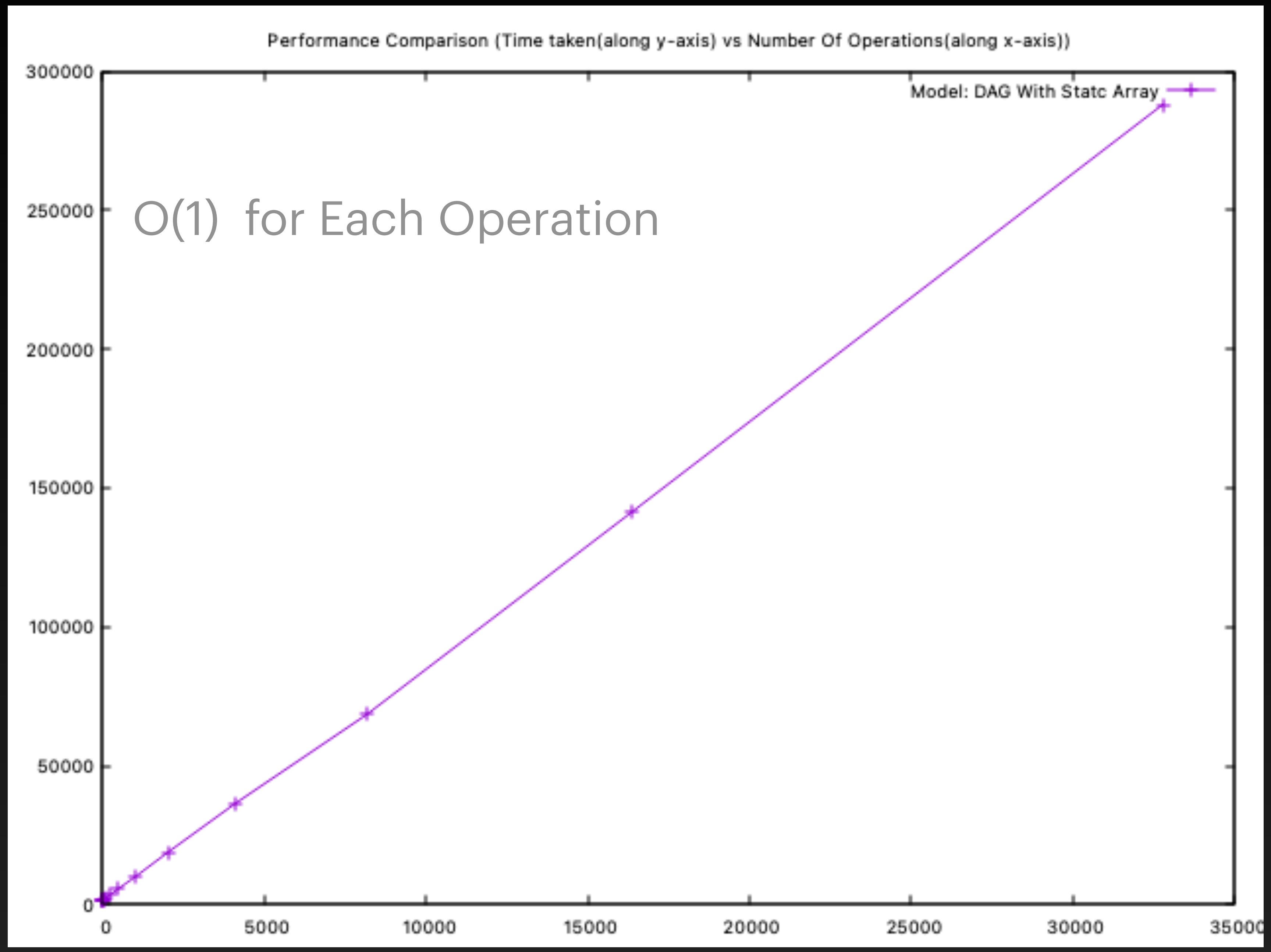
Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Stack Considering All The Versions

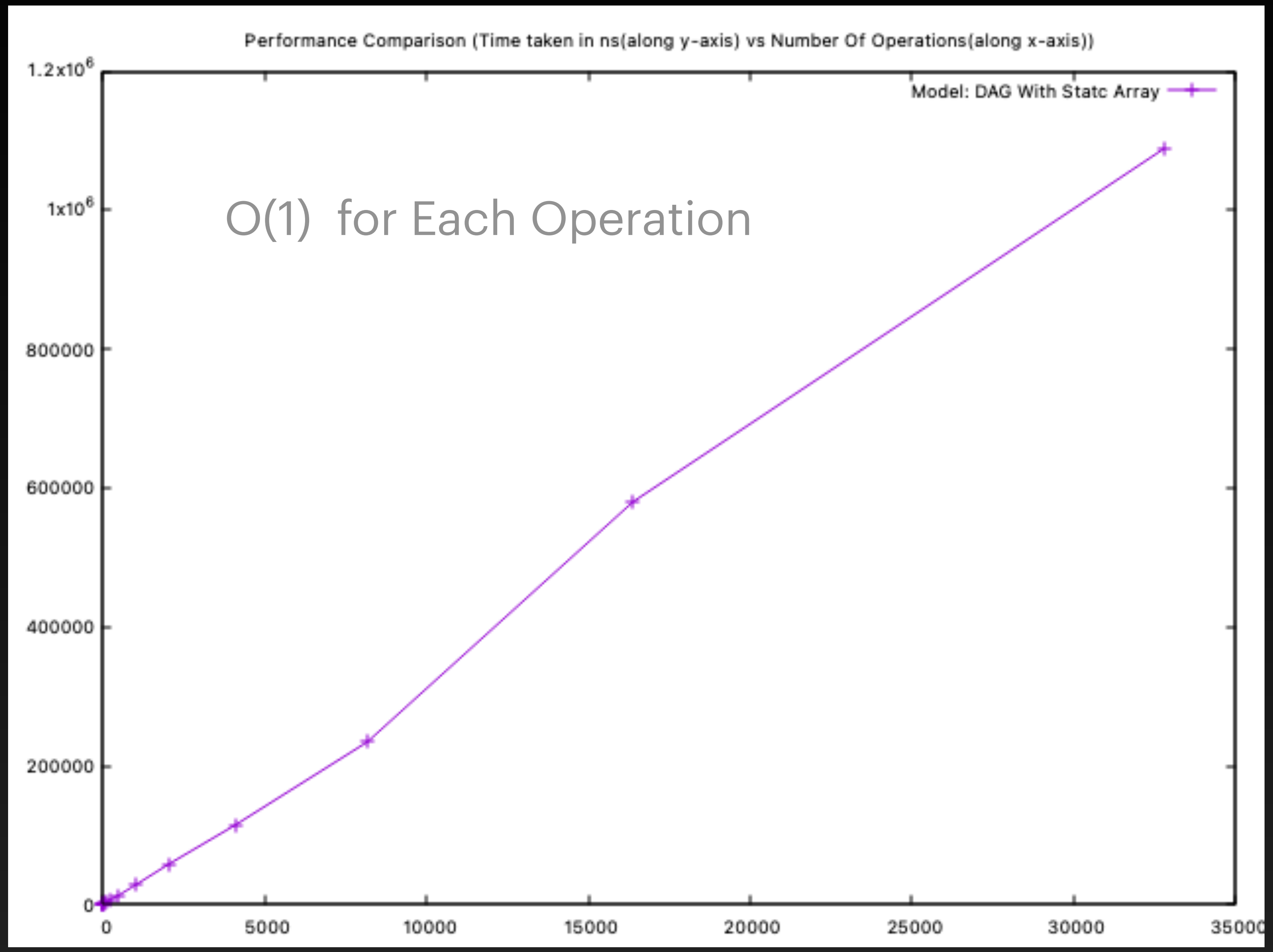
Strategies	Category 1	Category 2	Category 3
Using PPL with PM Model	O(V) [Amortized] To Hold The Linked List		
DAG Model	O(V) To Hold The MAP/DAG		
Ephemeral std::stack (C++)	O(N)	-	-

Benchmarking Of DAG Model

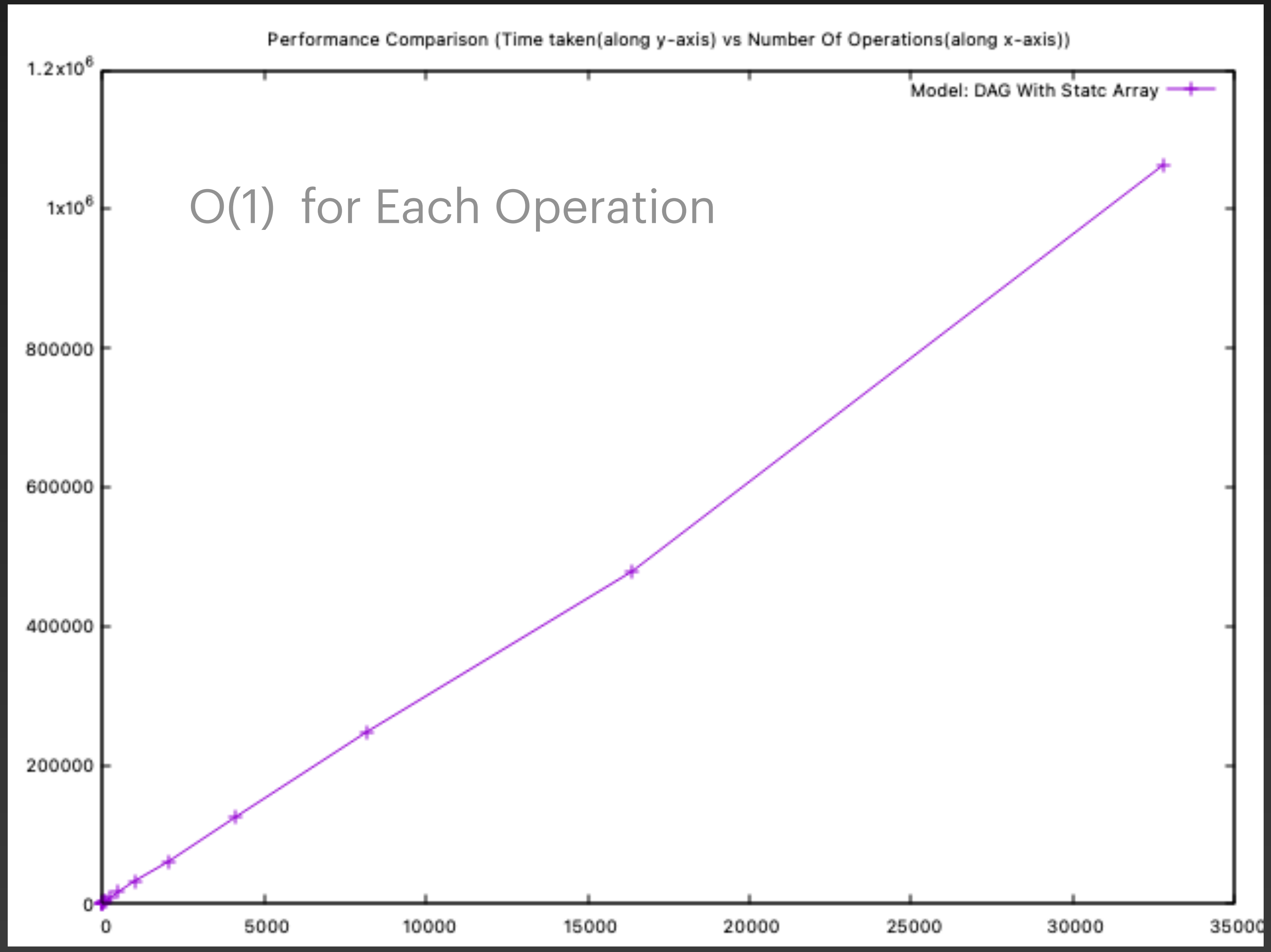
Simulation Of Full Persistence



OnlyPush In Randomised Versions

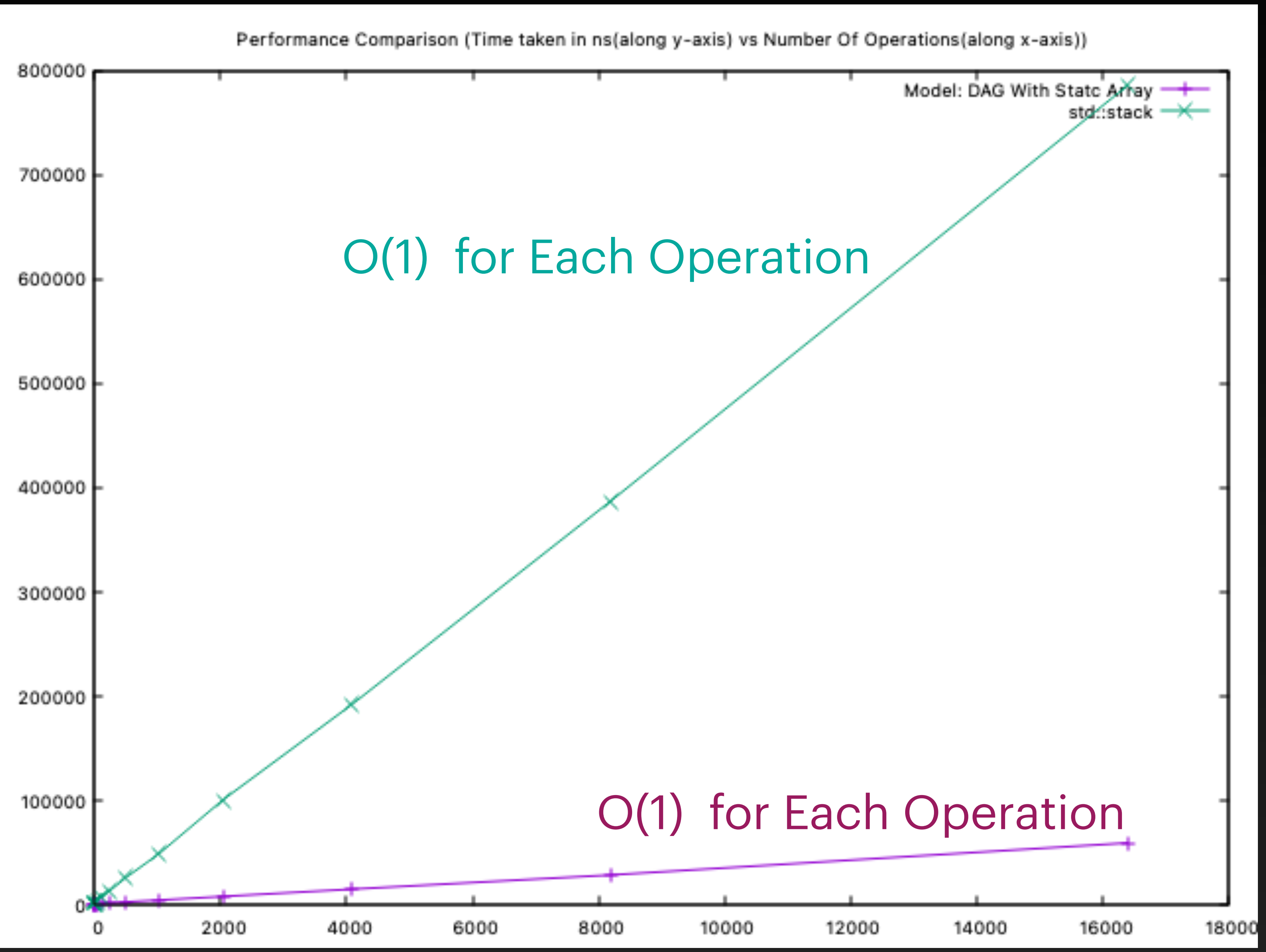


Randomised Push/Pop In Randomised Versions



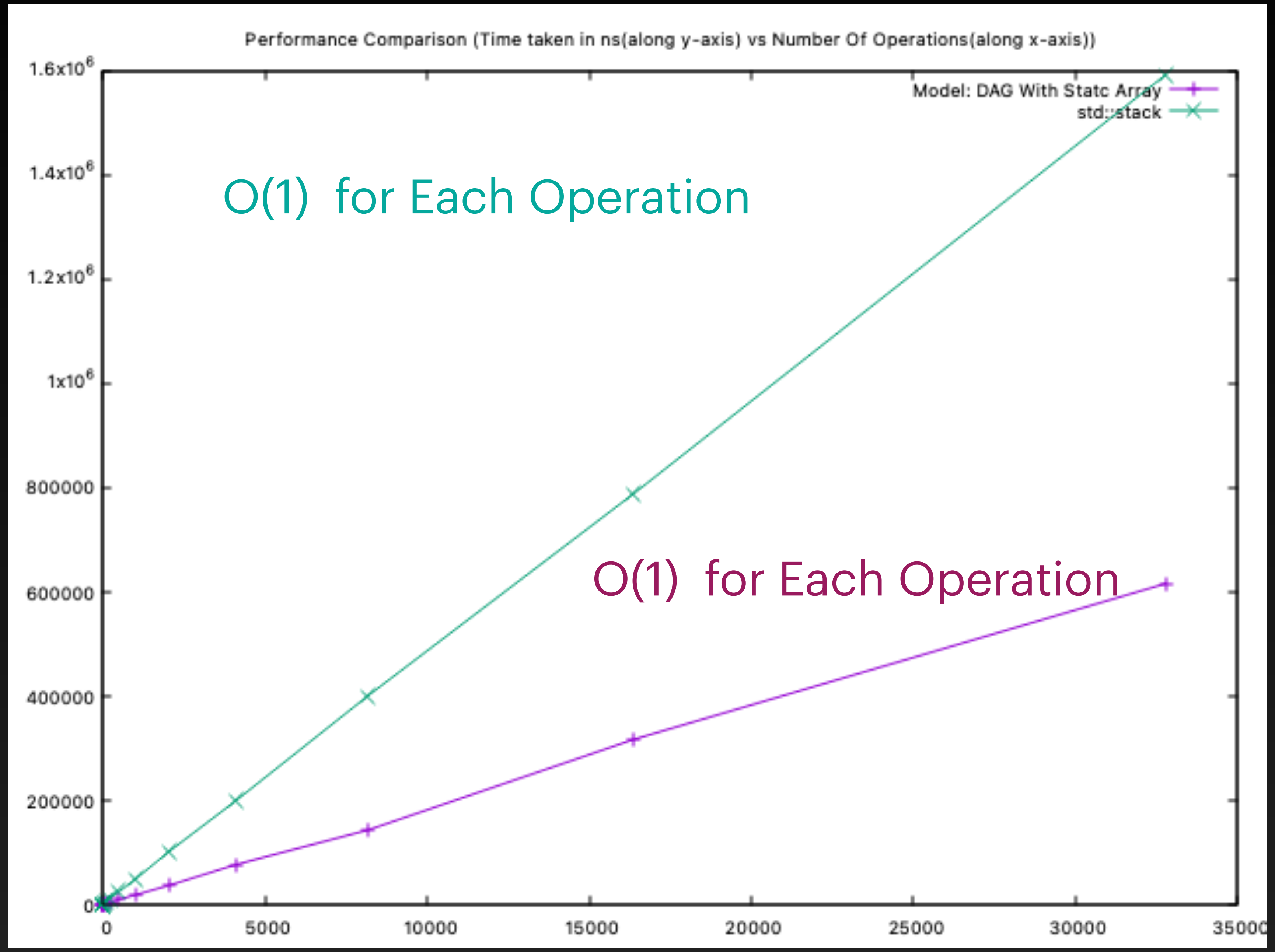
OnlyPop In Randomised Versions
(Pushing Operation is Not Time Profiled)

Benchmarking: DAG Model Vs std::stack

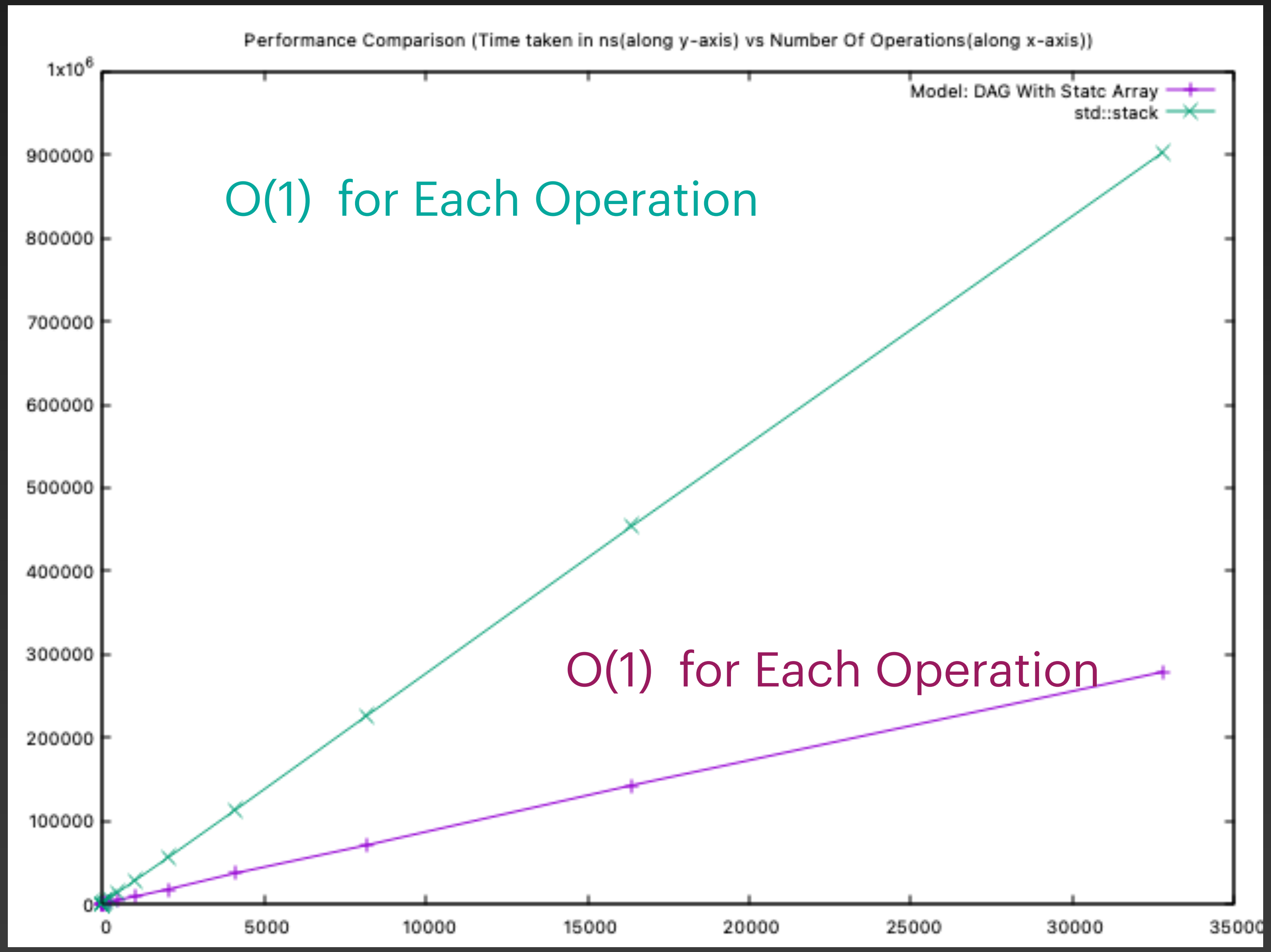


OnlyPush In latest Versions

Simulation Of Partial Persistence



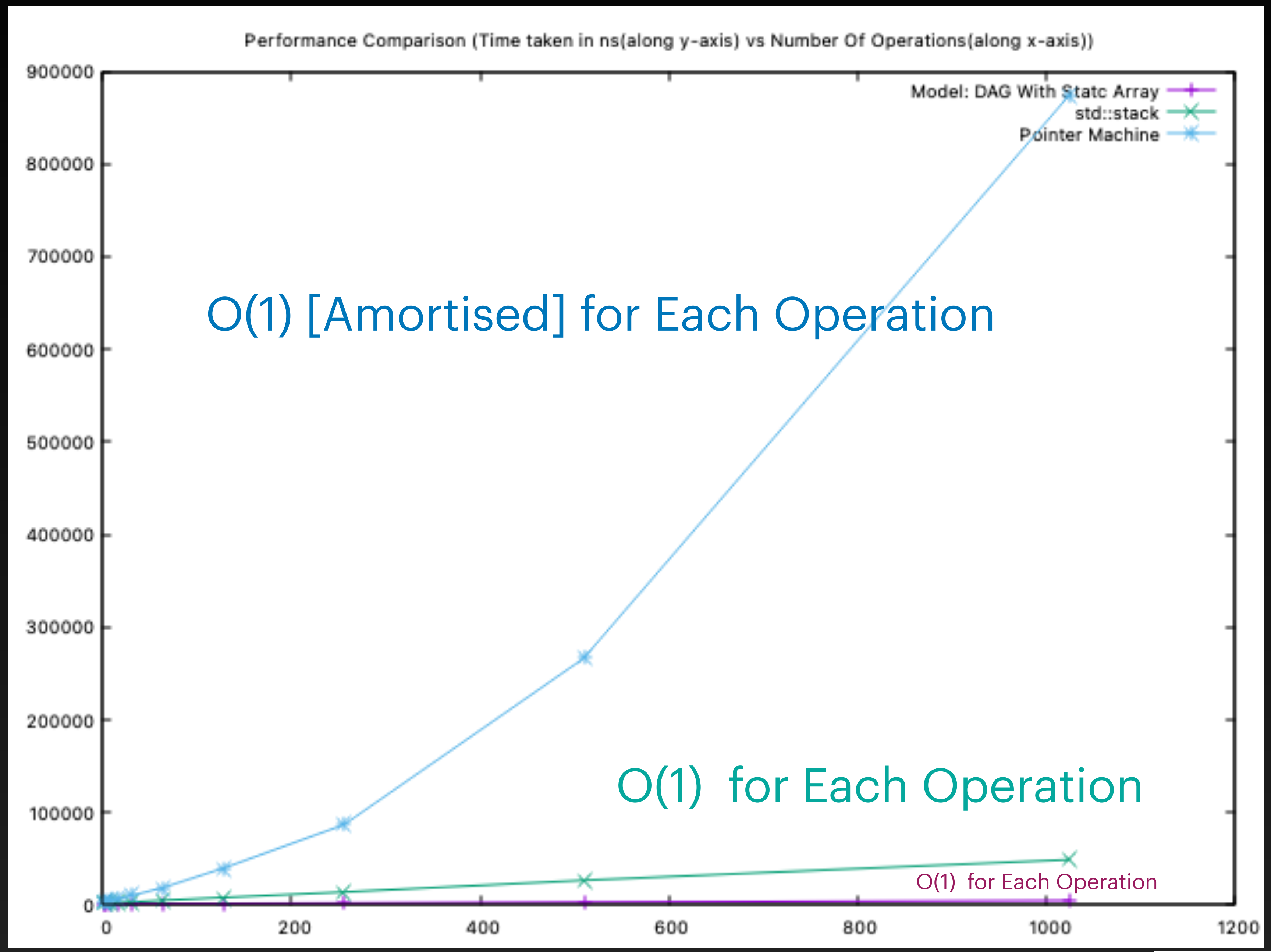
Randomised Push/Pop In Latest Versions



OnlyPop In Latest Versions
(Pushing Operation is Not Time Profiled)

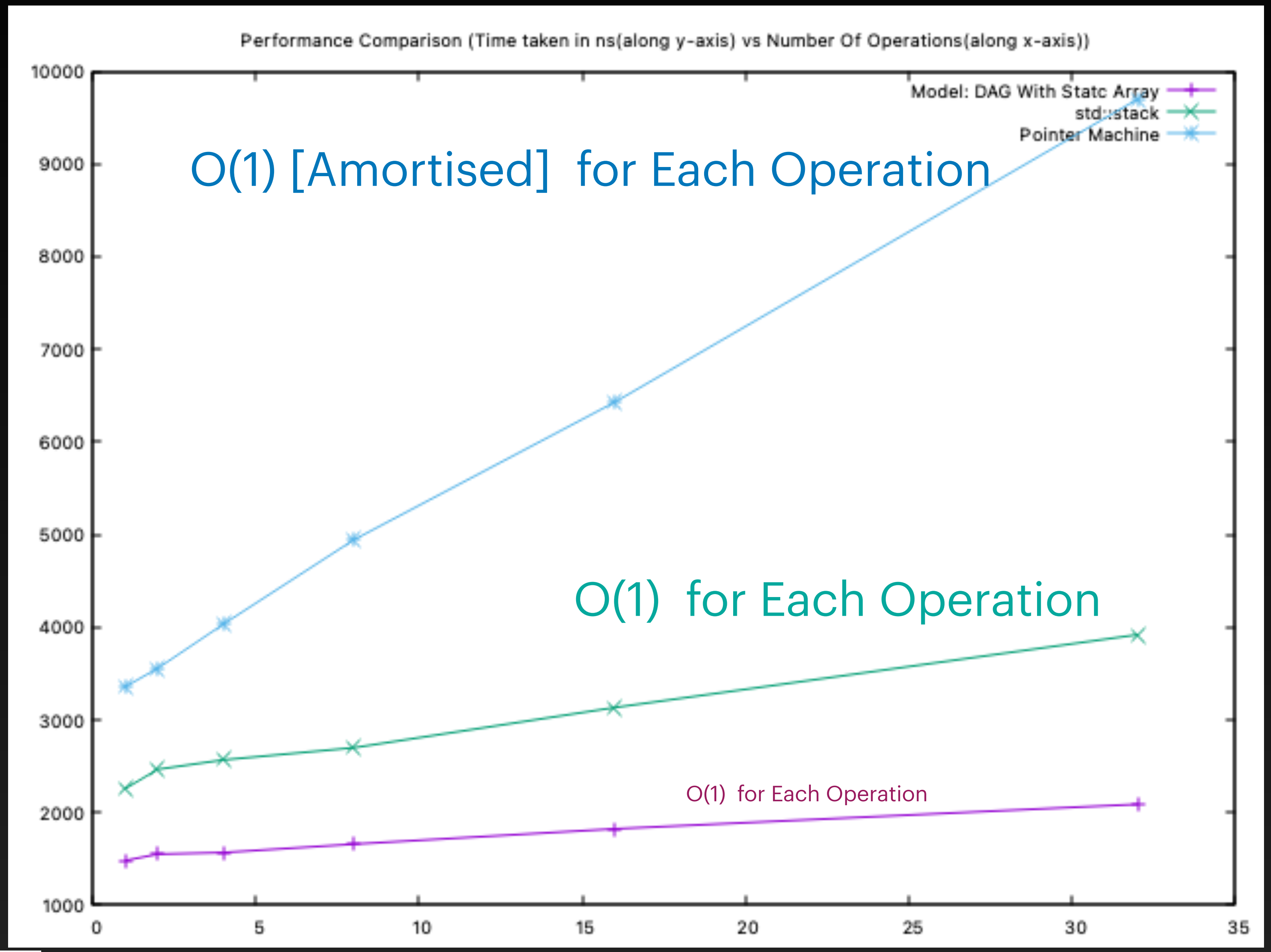
NOTE: We Used Static Array To Handle In DAG Model, So, It Is Slightly Faster Than Std::Stack

Benchmarking: DAG Model Vs std::stack Vs Stack_Using_PM_PPL

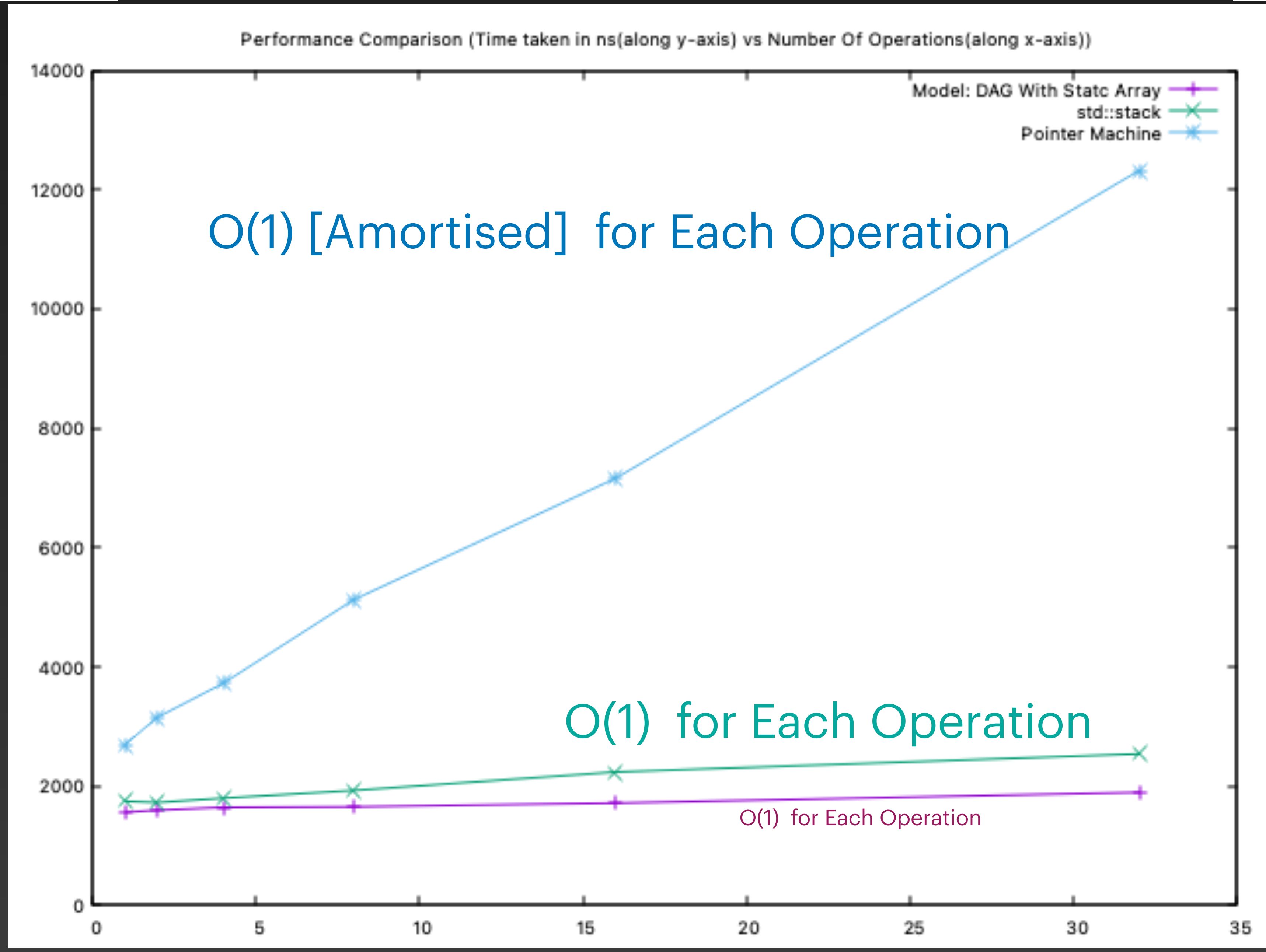


OnlyPush In Latest Versions

Simulation Of Partial Persistence



Randomised Push/Pop In Latest Versions



OnlyPop In Latest Versions
(Pushing Operation is Not Time Profiled)

NOTE: We Used Static Array To Handle In DAG Model, So, It Is Slightly Faster Than Std::Stack

Persistent Queue

Persistent Queue

Model Developed By Our Team

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Queue Considering All The Versions

Strategies	enqueue(data, version)	dequeue(version)	getFront(ver)	getRear(rear)
Using PPL with PM Model	O(1)	O(1) [Amortized]	O(1)	O(1)
Reduced Sparse Matrix Model	O(log v) [Here v is current version] O((log V!)/V) ~ O(1)	O(log v) [In Average Case] O(v * log v) [In Worst Case]	O(1) [In Average Case] O(log v) [In Worst Case]	O(1)
Threaded_FPQ #	~O(1)	~O(1)	~O(1)	~O(1)
Ephemeral std::queue (C++)	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version

Persistent Queue

Model Developed By Our Team

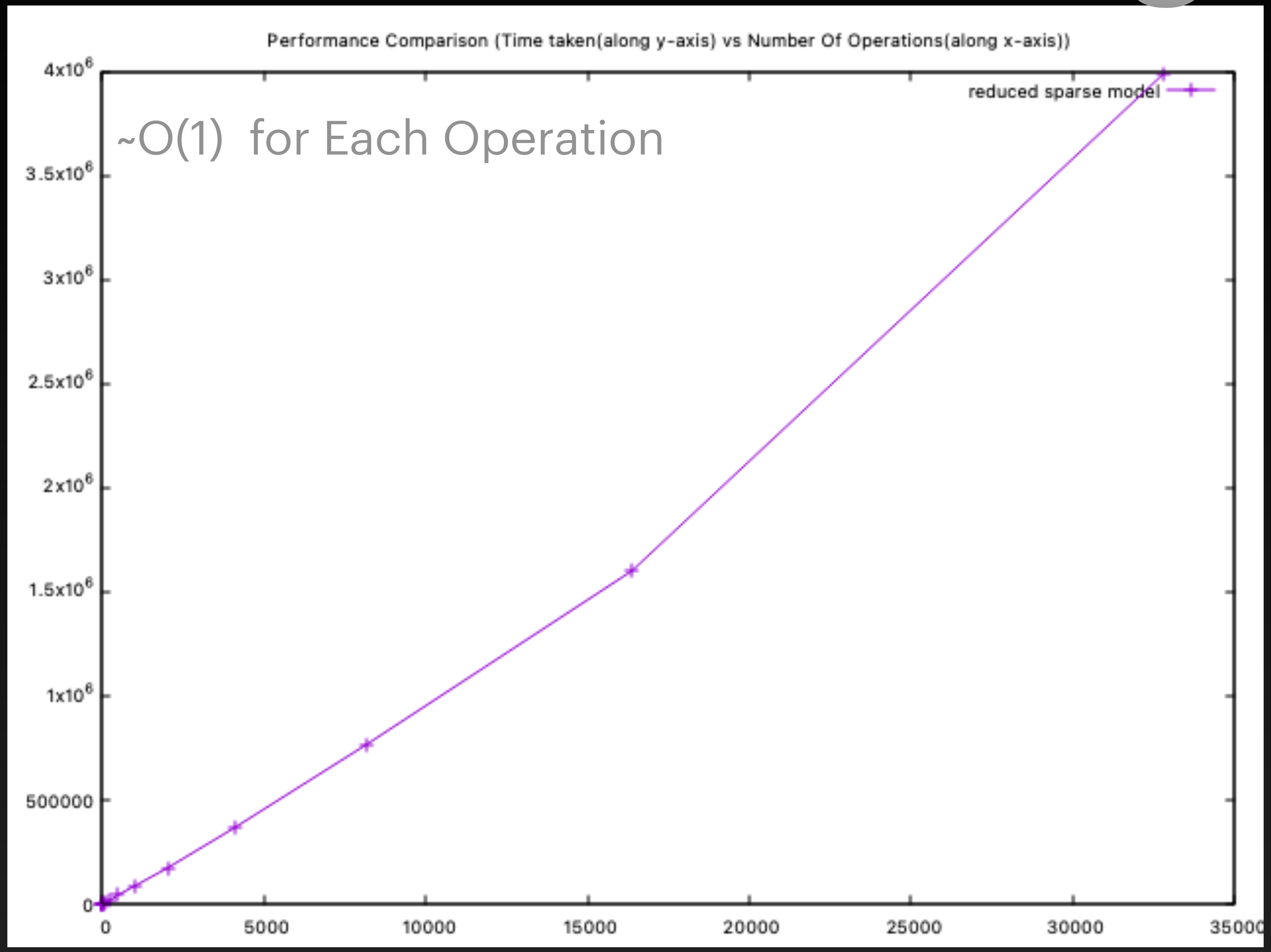
Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Queue Considering All The Versions

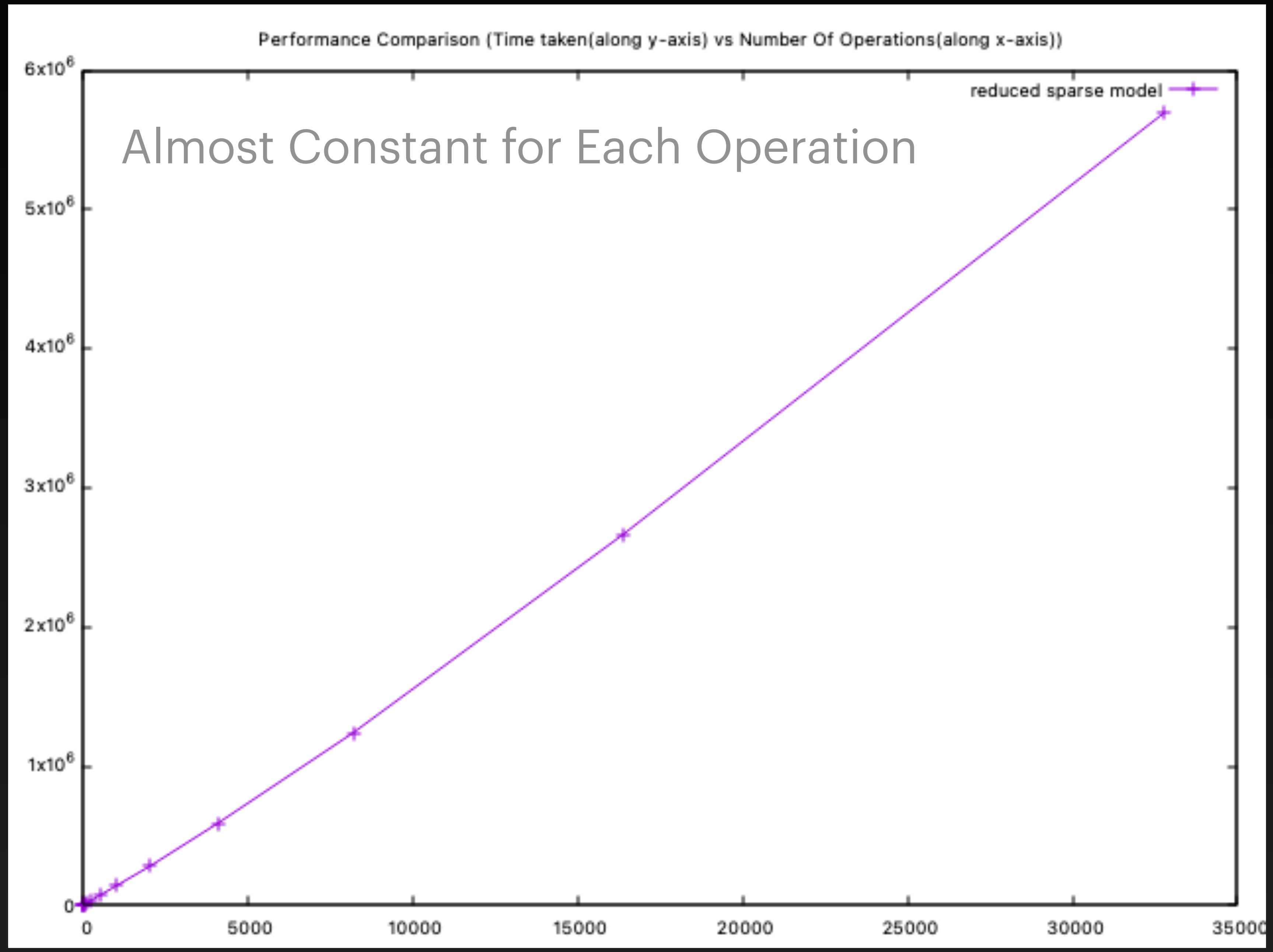
Strategies	Category 1	Category 2		Category 3
Using PPL with PM Model	O(V) [Amortized] To Hold The Linked List	O(V) to Hold The MAP for version->rear		
Reduced Sparse Matrix Model	• O (V + log (V!)) to Hold The UP_TABLE	• O (V) to Hold the MAP	• O (V) to Hold the TYPE_OF_VER	
Threaded FPQ	*O(no_of_enqueues = n) to Hold The Nodes*	*O(V) to Hold The Versions*	*O(1) [Best Case] O(log V) [Balanced Average Case , O(N*V) [Worst Case] To Hold The Thread Directions*	Disadvantage Very Low I/O Rate due to Heavy Access To The Heap Memory
Ephemeral std::queue (C++)	O(N)			-

Benchmarking Of Reduced Sparse Matrix Model

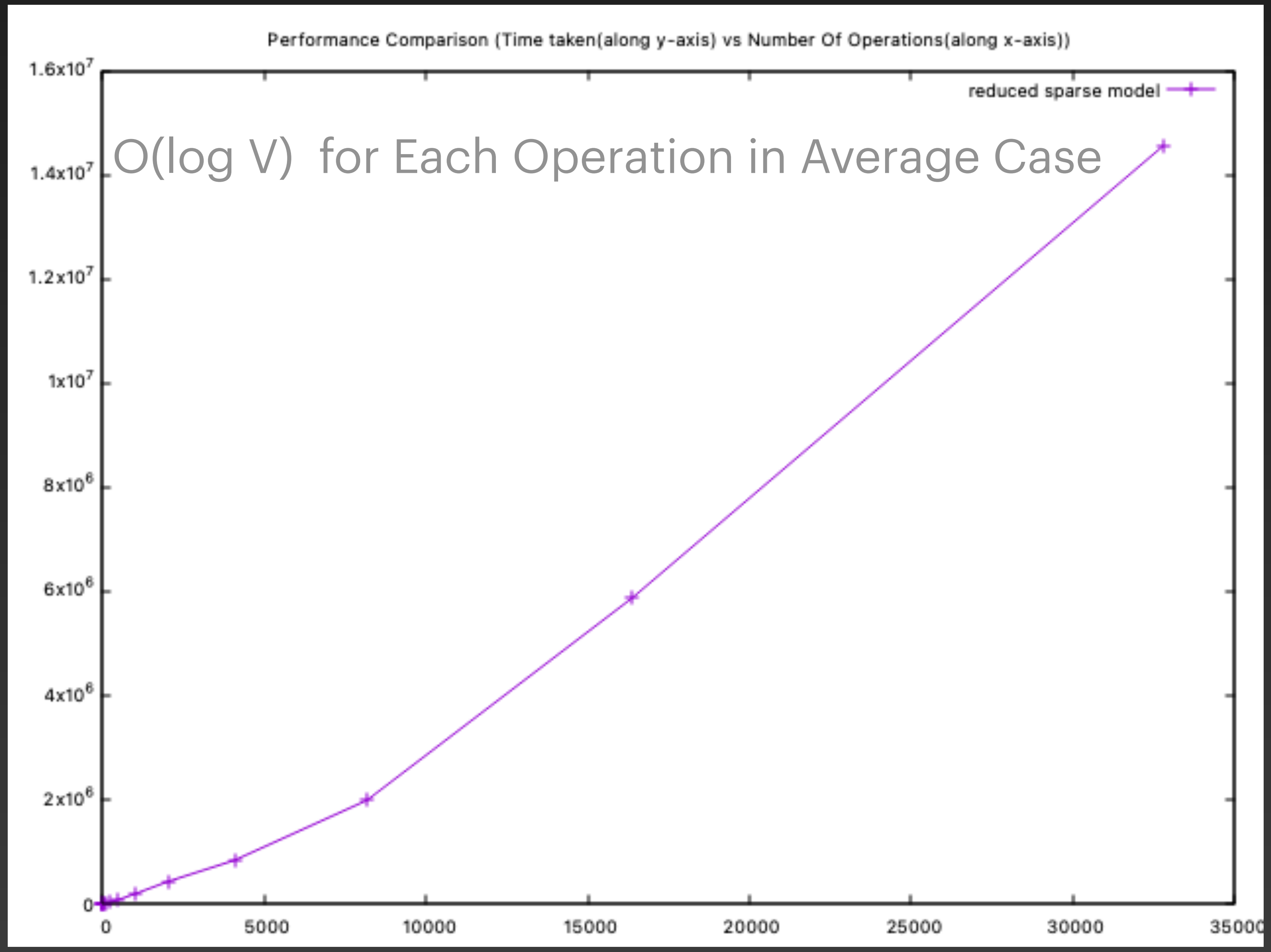
Simulation Of Full Persistence



Only Enqueue In Randomised Versions



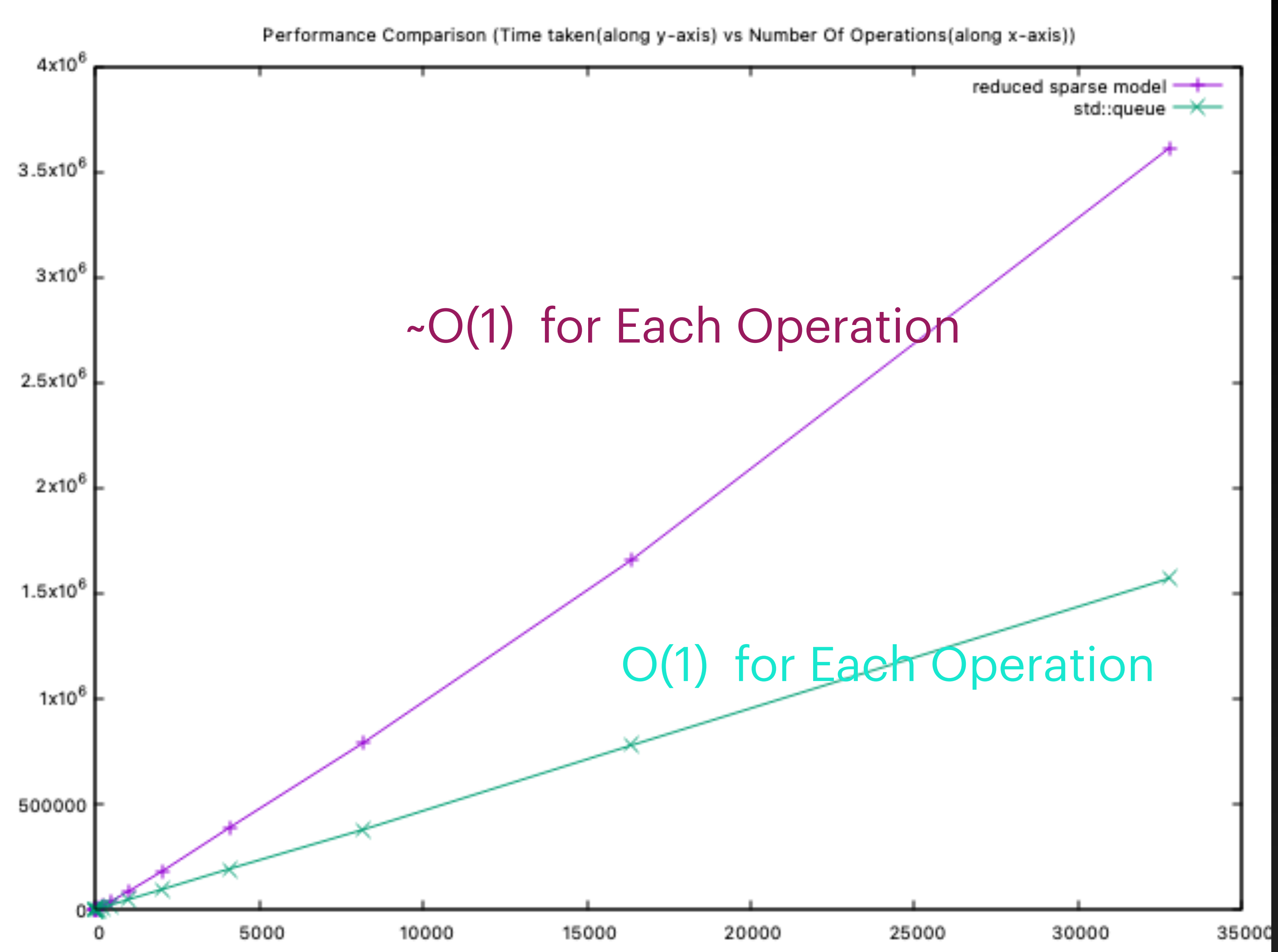
Randomised Enque/Deque In Randomised Versions



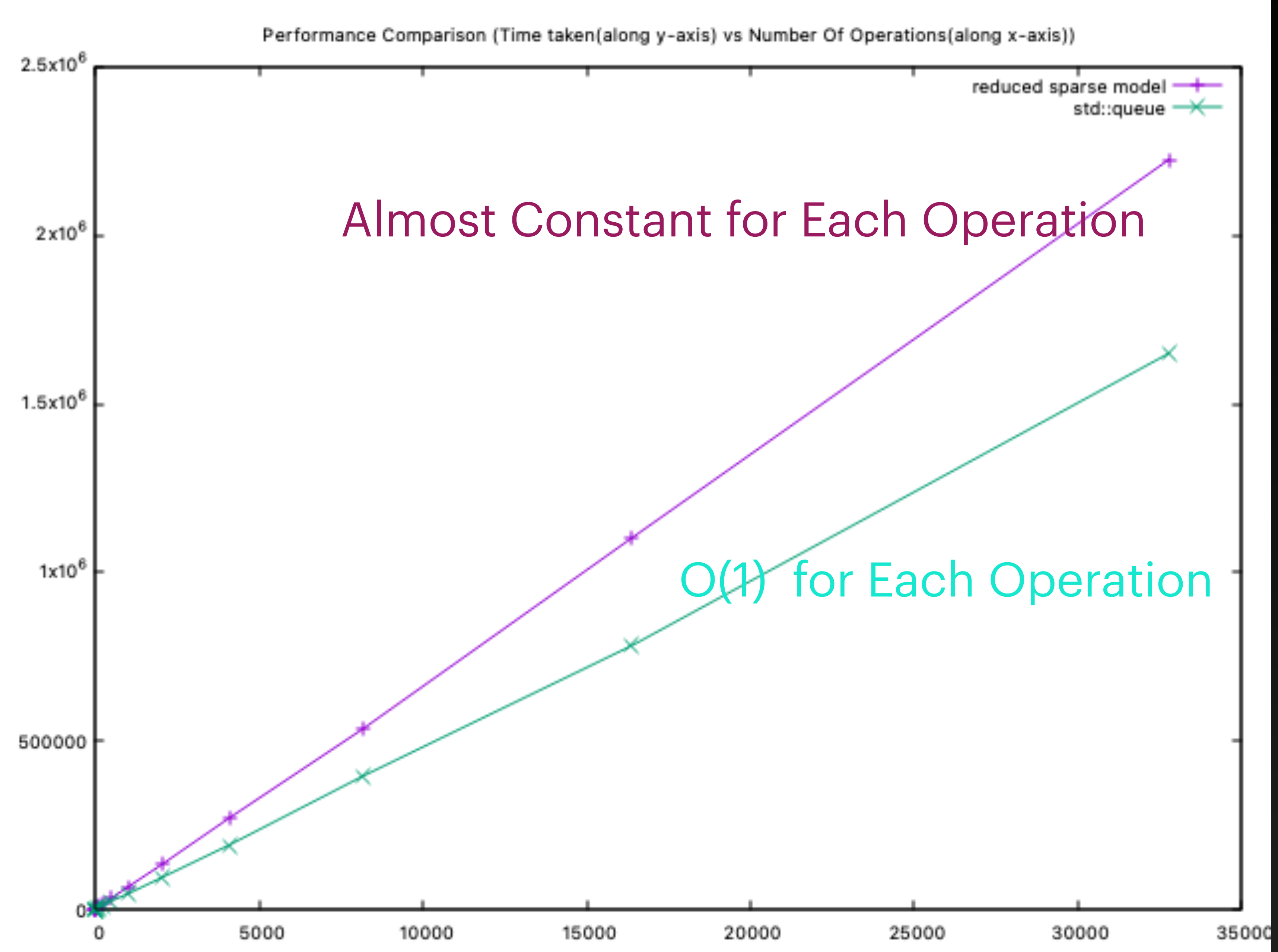
Only Deque In Randomised Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model vs std::queue

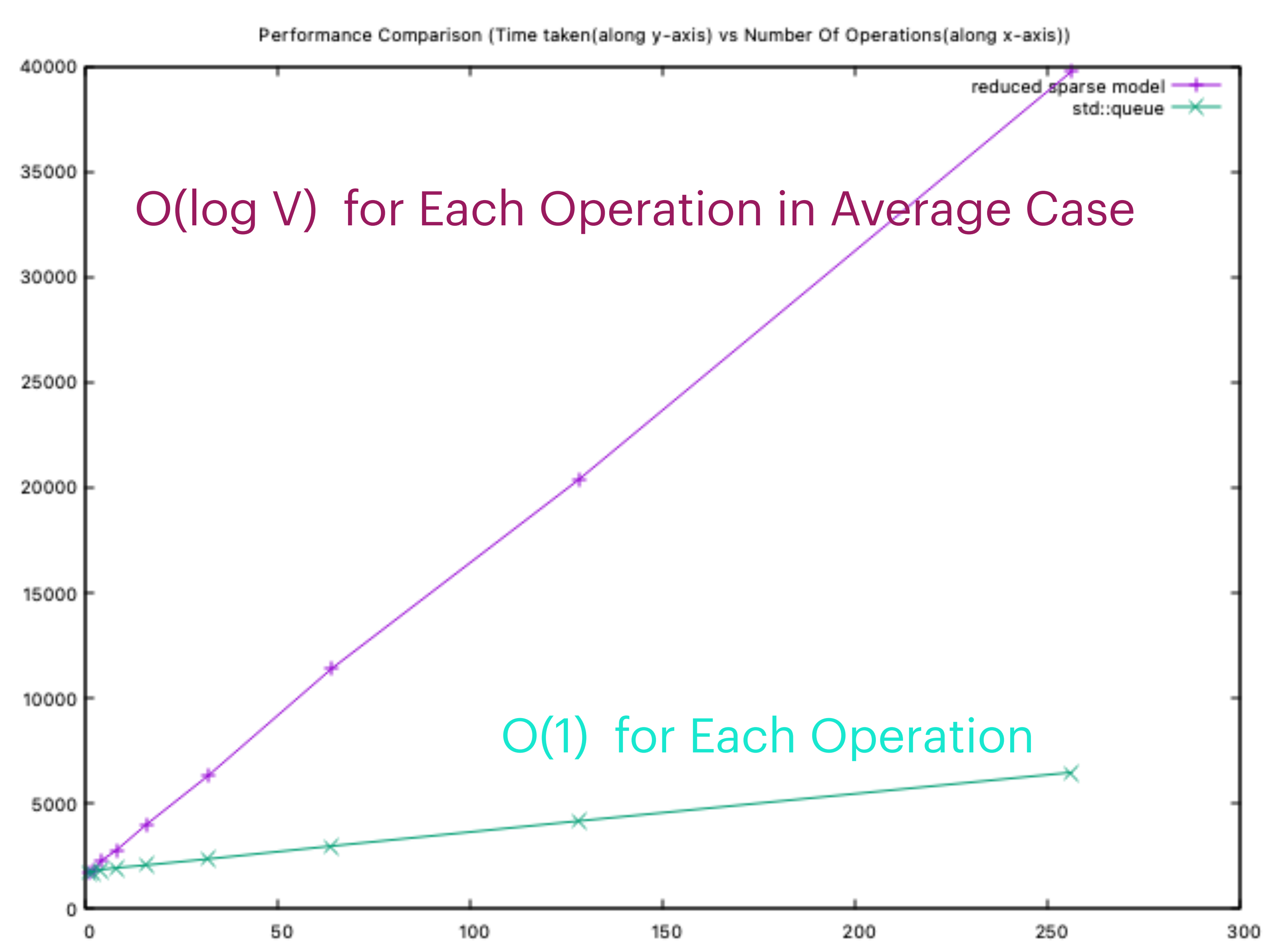
Simulation Of Partial Persistence



Only Enqueue In Latest Versions



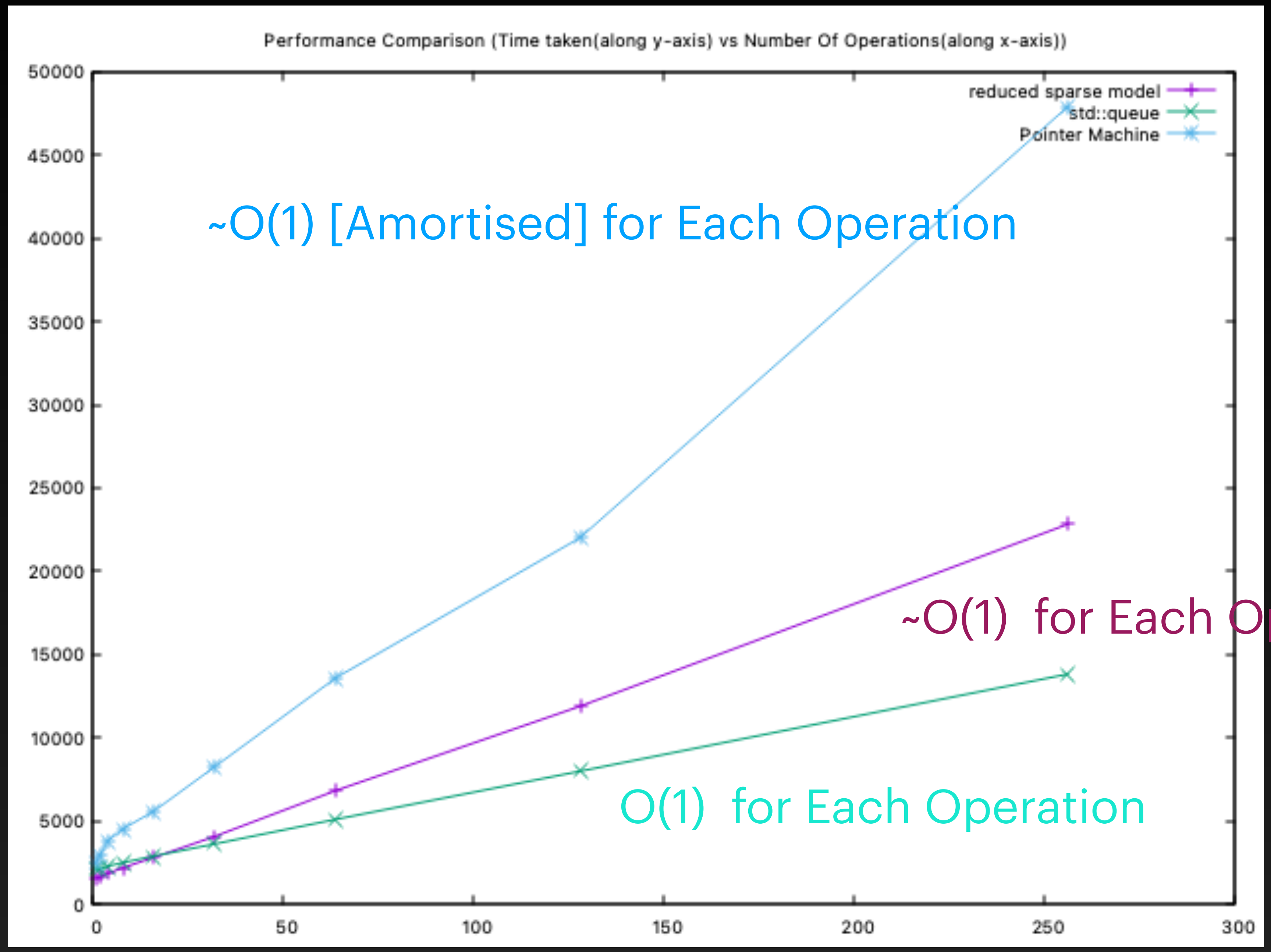
Randomised Enque/Deque In Latest Versions



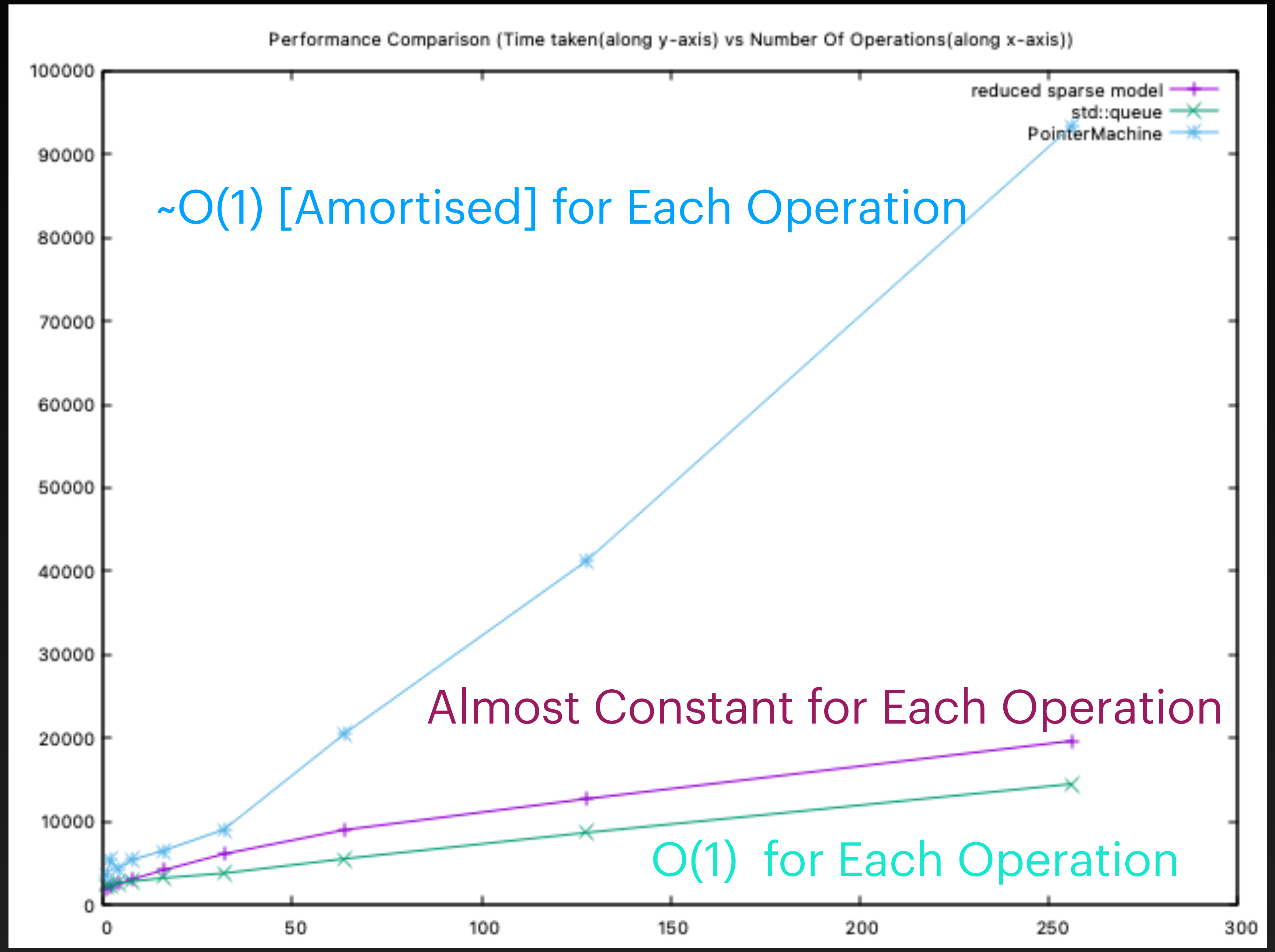
Only Deque In Latest Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model vs std::queue Vs Queue_Using_PM_PPL

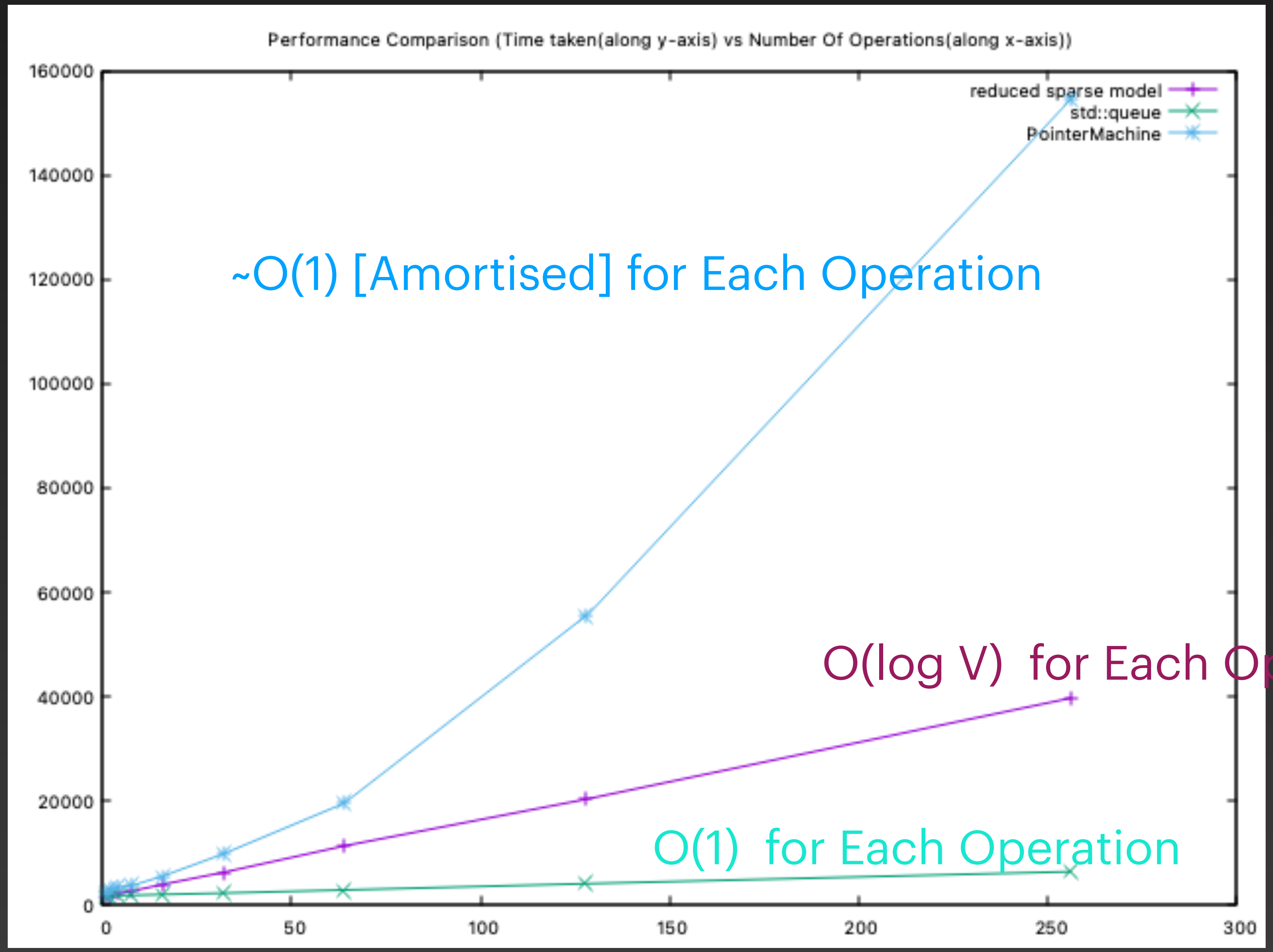
Simulation Of Partial Persistence



Only Enqueue In Latest Versions



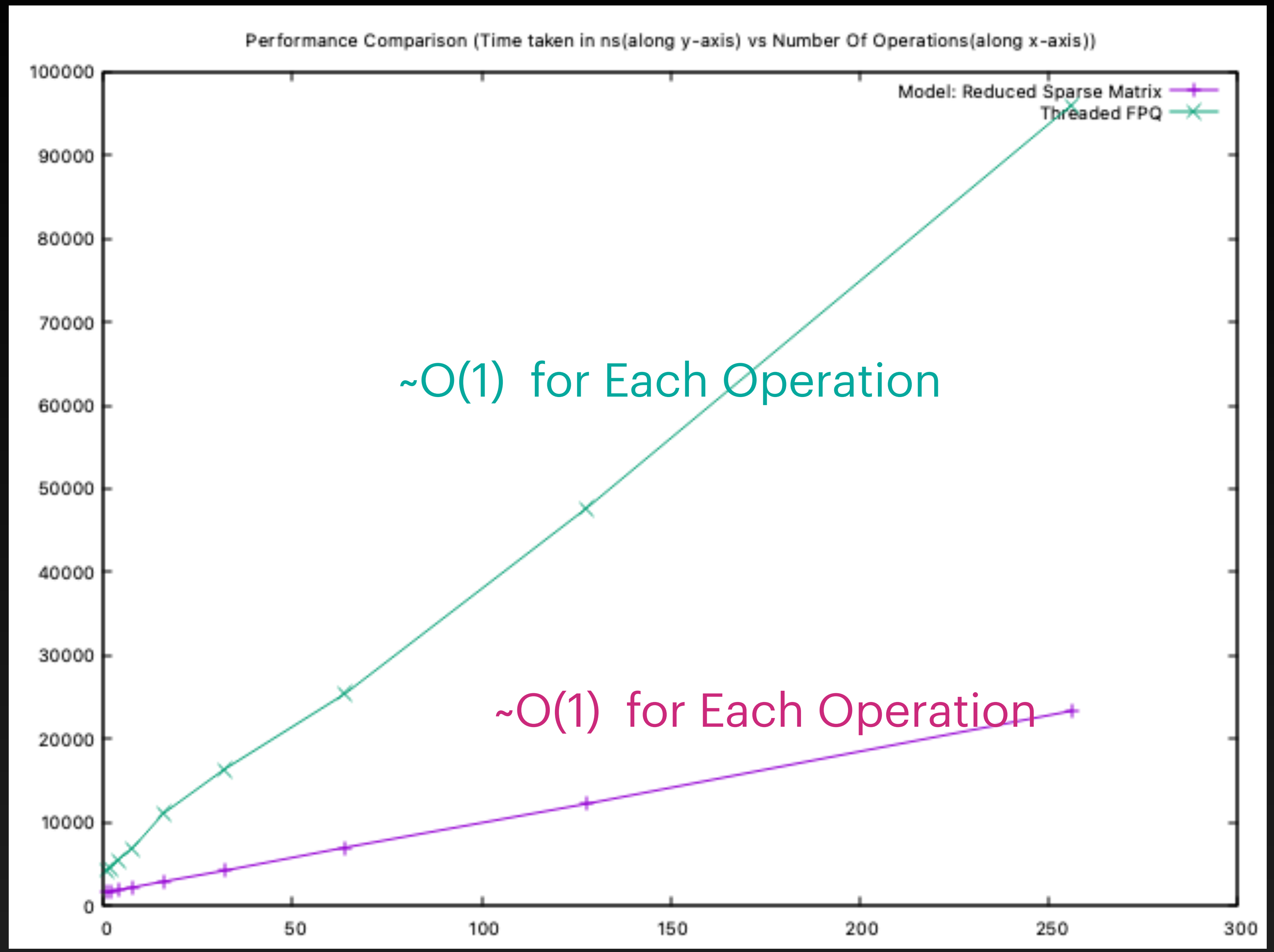
Randomised Enque/Deque In Latest Versions



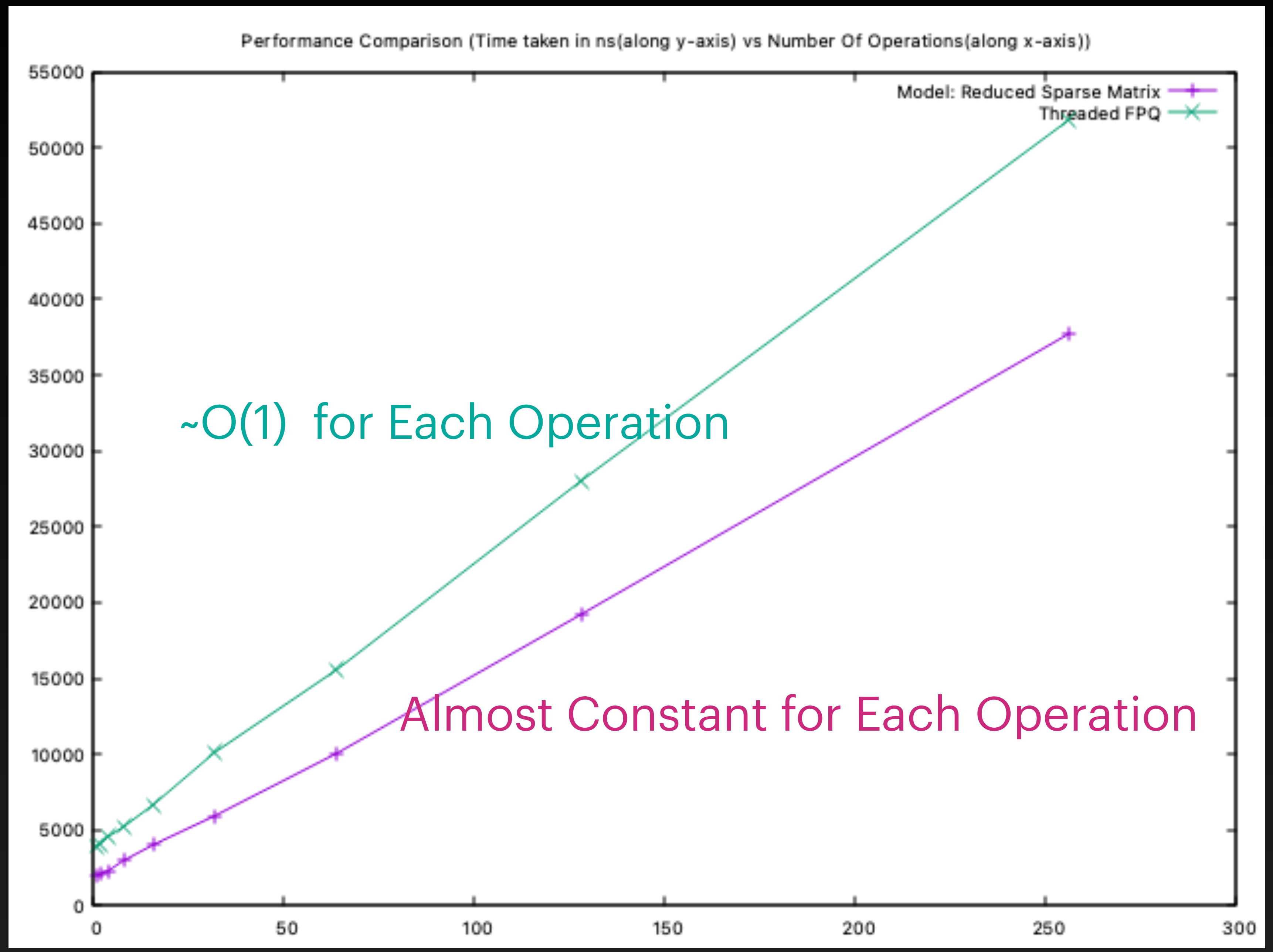
Only Deque In Latest Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model Vs Threaded FPQ

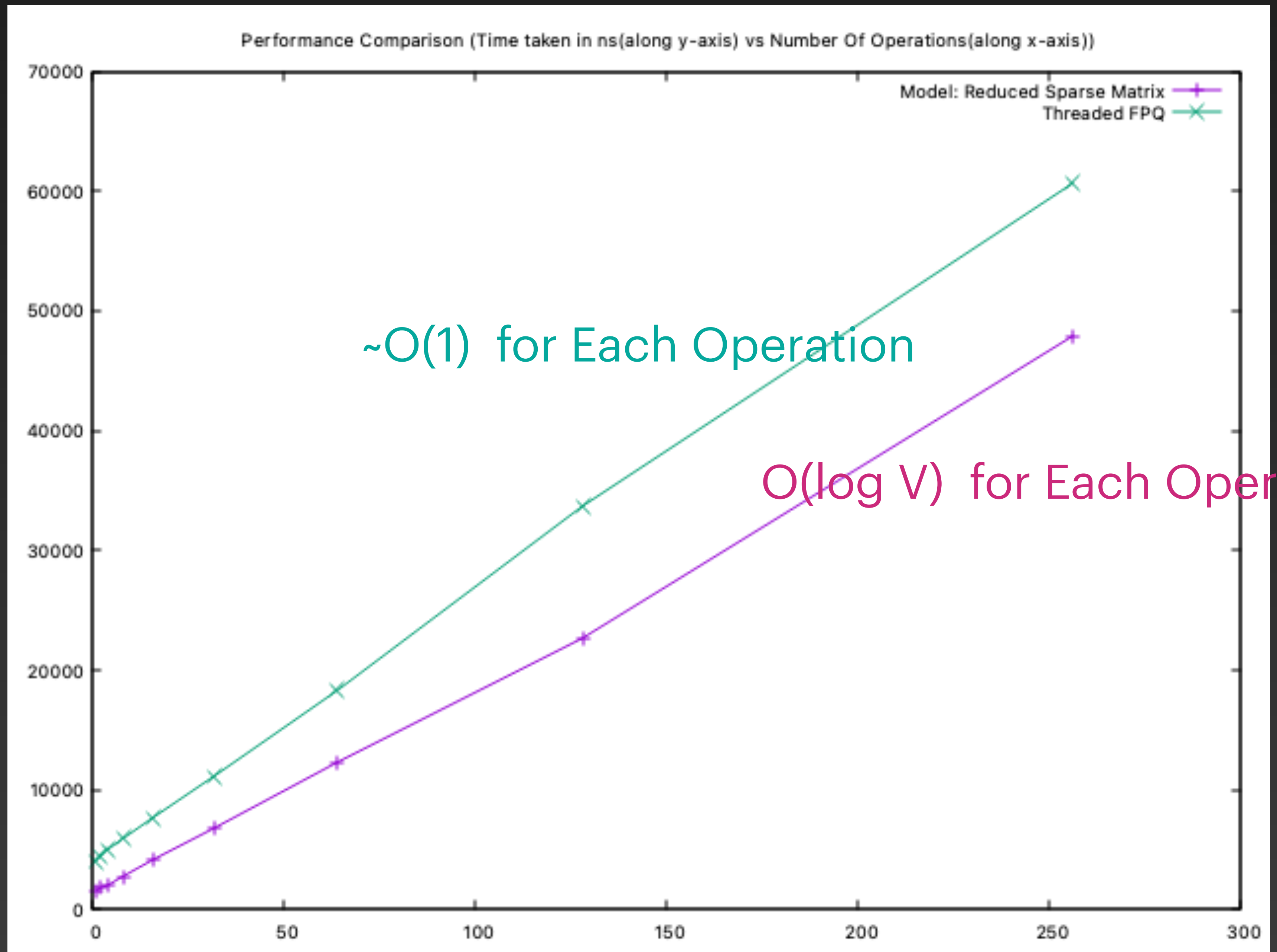
Simulation Of Full Persistence



Only Enqueue In Randomised Versions



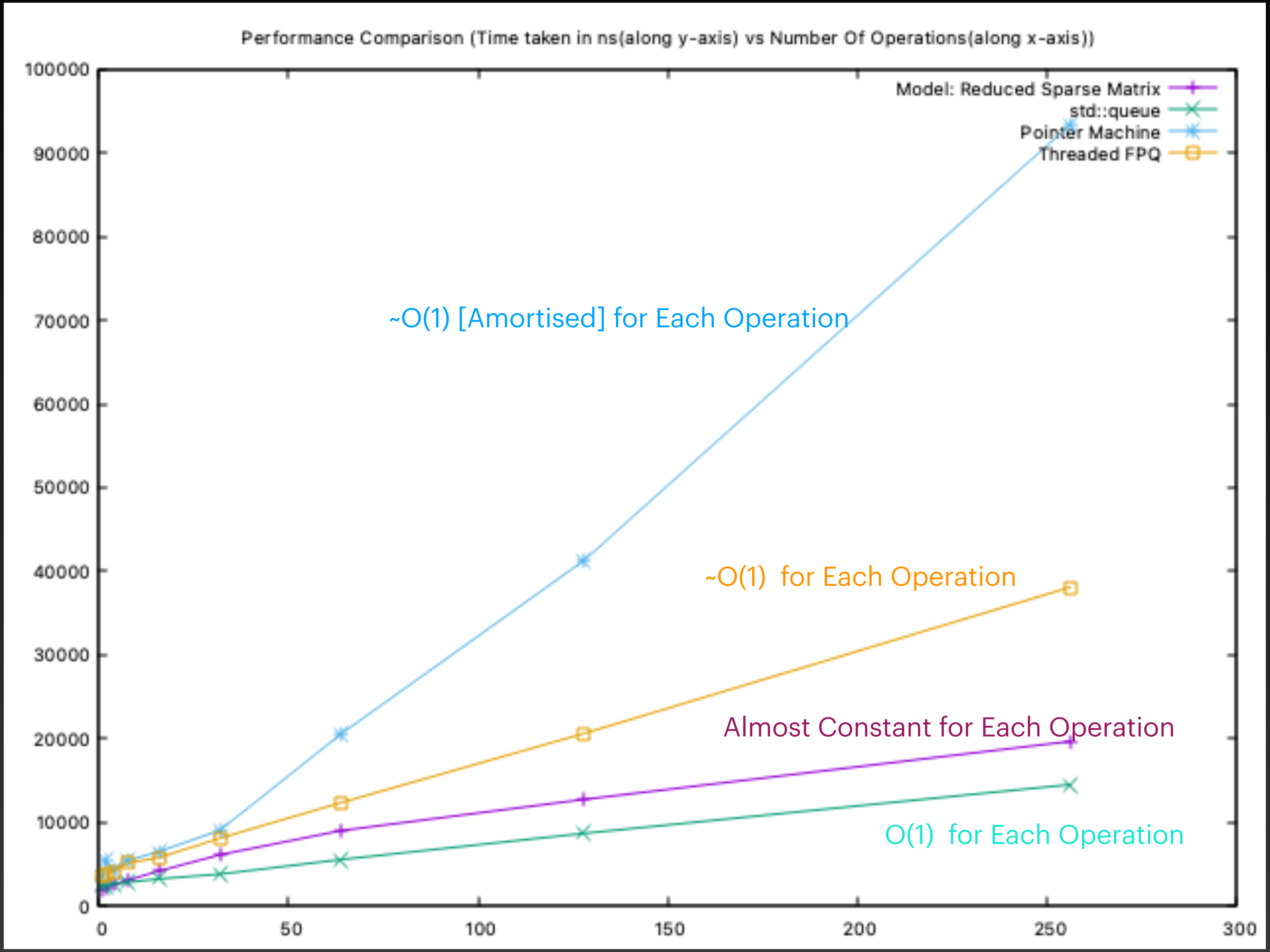
Randomised Enque/Deque In Randomised Versions



O(log V) for Each Operation in Average Case

Only Deque In Randomised Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model vs std::queue Vs Queue_Using_PM_PPL vs Threaded FPQ

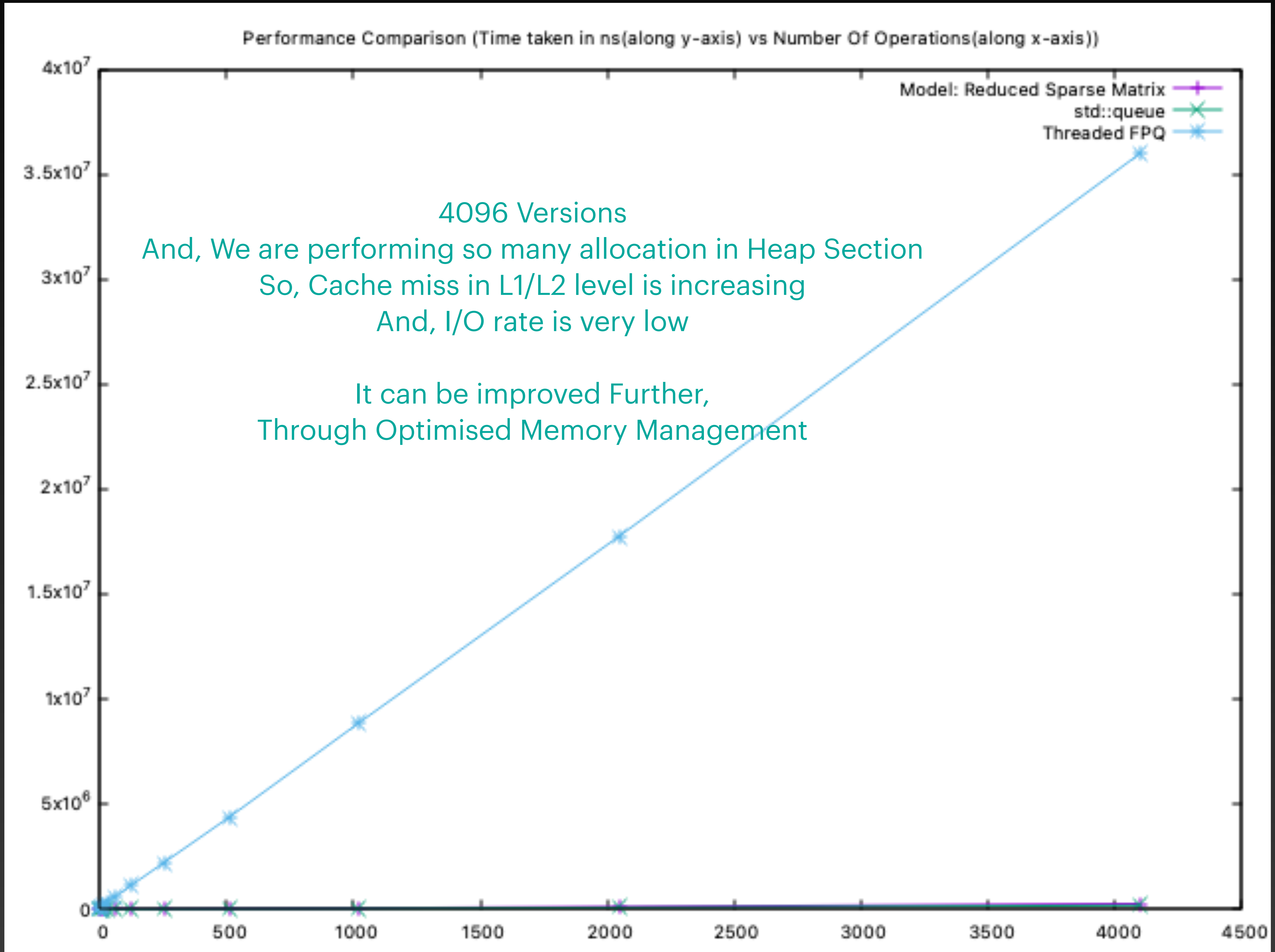


Simulation Of Partial Persistence

Randomised Enque/Dequeue In Consecutive Versions

Issue With Threaded FPQ Model ...

Low I/O Rate Due to poor memory management in C



Issue With Threaded FPQ Model … / BenchMarking Data

~/Desktop/programming/XcodePrj/CM_SIR_PROJECT/Annannyo_q2 — -zsh | ~/Desktop/persistent/MyCode/BenchMarkingWithGoogleBenchmark/src/main/gpp_run/QUEUE_Anannyo — -zsh

```
  L1 Data 48 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 6144 KiB (x1)
Load Average: 1.40, 1.98, 2.12
------------------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------
BM_QUEUE_Wraper/1          19629 ns        19169 ns        46360 items_per_second=52.167k/s 1
BM_QUEUE_Wraper/2          32168 ns        31566 ns        20668 items_per_second=63.3589k/s 2
BM_QUEUE_Wraper/4          53112 ns        52226 ns        12222 items_per_second=76.59k/s 4
BM_QUEUE_Wraper/8          93951 ns        92851 ns         7158 items_per_second=86.1592k/s 8
BM_QUEUE_Wraper/16        171014 ns       169404 ns         4016 items_per_second=94.4486k/s 16
BM_QUEUE_Wraper/32        312674 ns       310172 ns         2236 items_per_second=103.169k/s 32
BM_QUEUE_Wraper/64        602828 ns       599170 ns         1174 items_per_second=106.815k/s 64
BM_QUEUE_Wraper/128      1180954 ns      1171348 ns          587 items_per_second=109.276k/s 128
BM_QUEUE_Wraper/256      2344332 ns      2331366 ns          295 items_per_second=109.807k/s 256
BM_QUEUE_Wraper/512      4598501 ns      4579066 ns          152 items_per_second=111.813k/s 512
BM_QUEUE_Wraper/1024     9090952 ns      9056254 ns           71 items_per_second=113.071k/s 1024
BM_QUEUE_Wraper/2048    19536364 ns     19327947 ns           38 items_per_second=105.961k/s 2048
BM_QUEUE_Wraper/4096    40346193 ns     39580063 ns           16 items_per_second=103.486k/s 4096
BM_QUEUE_Wraper_BigO     9733.55 N       9571.93 N     1
BM_QUEUE_Wraper_RMS            5 %             4 %     1
(base) debasmitroy@DEBASMITs-MacBook-Air QUEUE_Anannyo % g++ FP_QUEUE_MAIN_ENQ.cpp -std=c++11 -isystem benchmark/include \-Lbenchmark/build/src -lbenchmark -lpthread -o mybenchma
rk
ld: warning: directory not found for option '-Lbenchmark/build/src'
(base) debasmitroy@DEBASMITs-MacBook-Air QUEUE_Anannyo % ./mybenchmark --benchmark_out="rand_enq_FP_256.csv" --benchmark_out_format=csv
2022-01-23T12:29:27+05:30
Running ./mybenchmark
Run on (8 X 1100 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 6144 KiB (x1)
Load Average: 1.51, 1.93, 2.10
------------------------------------------------------------------------------
Benchmark                           Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------
BM_QUEUE_Wraper/1          4349 ns        4286 ns       198018 items_per_second=233.318k/s 1
BM_QUEUE_Wraper/2          4823 ns        4690 ns       140919 items_per_second=426.455k/s 2
BM_QUEUE_Wraper/4          5671 ns        5563 ns       132090 items_per_second=719.071k/s 4
BM_QUEUE_Wraper/8          7155 ns        7044 ns        80764 items_per_second=1.1357M/s 8
BM_QUEUE_Wraper/16        11422 ns       11129 ns        73691 items_per_second=1.43769M/s 16
BM_QUEUE_Wraper/32        16605 ns       16381 ns        42350 items_per_second=1.95348M/s 32
BM_QUEUE_Wraper/64        25580 ns       25460 ns        26858 items_per_second=2.51378M/s 64
BM_QUEUE_Wraper/128       48136 ns       47762 ns        14310 items_per_second=2.67998M/s 128
BM_QUEUE_Wraper/256       97095 ns       95942 ns         7683 items_per_second=2.66827M/s 256
BM_QUEUE_Wraper_BigO     382.95 N       378.79 N     1
BM_QUEUE_Wraper_RMS          15 %           14 %     1
```

Very Poor I/O Rate @ 4096 Versions

Very decent I/O Rate @ 256 Versions

Persistent Search Tree

Persistent Search Tree

Time Complexity

Here, V = Total No. Of Version, N = Average Number Of Elements in The Tree Considering All The Versions

Strategies	Updation/Insertion/ Deletion of Data	Retrieval Of Data
Path Copying With Normal BST	O(N) in Worst Case O(1) in Best Case	O(V) + O(N) in Worst Case O(V) + O(1) in Best Case
Path Copying With AVL/RB Tree	O(log2N) in Worst Case O(1) in Best Case	O(V) + O(log2N) in Worst Case O(V) + O(1) in Best Case
Path Copying With Hash Array Mapped Trie	O(log32_(2^64))~O(12)	O(V) + O(log32_(2^64))~O(12)
With Pointer Machine	O(1) Amortised	O(1) Amortised

Persistent Search Tree

Auxiliary Space

Here, V = Total No. Of Version, N = Average Number Of Elements in The Tree Considering All The Versions

Strategies	Category 1	Category 2
Path Copying With Normal Threaded BST	Node Size: # ~ 28 byte	O(N + V) to Hold The Tree and Staring Pointers
Path Copying With AVL/RB Tree	Node Size: # ~ 32 byte	O(N + V) to Hold The Tree and Staring Pointers
Path Copying With Bitmaped Hash Array Mapped Trie	Node Size: # ~ (4+4+64*8) byte [Worst Case] ~ (4+4) byte [Best Case]	O(N + V) to Hold The Tree and Staring Pointers
With Pointer Machine	Node Size: # ~ 250 byte	O(N) [Amortised] to Hold The LinkedList

to store 4 byte Integer | 8 Bye pointers

BenchMarking Of Search Tree

std::set<T>

Red Black Tree
Ephemeral

std::unordered_set<T>

Custom Hashing Function
Ephemeral

Hamt<T>

Path Copying
With Bitmaped Hash Array Mapped Trie
Persistent

BenchMarking Tool:

Nonius

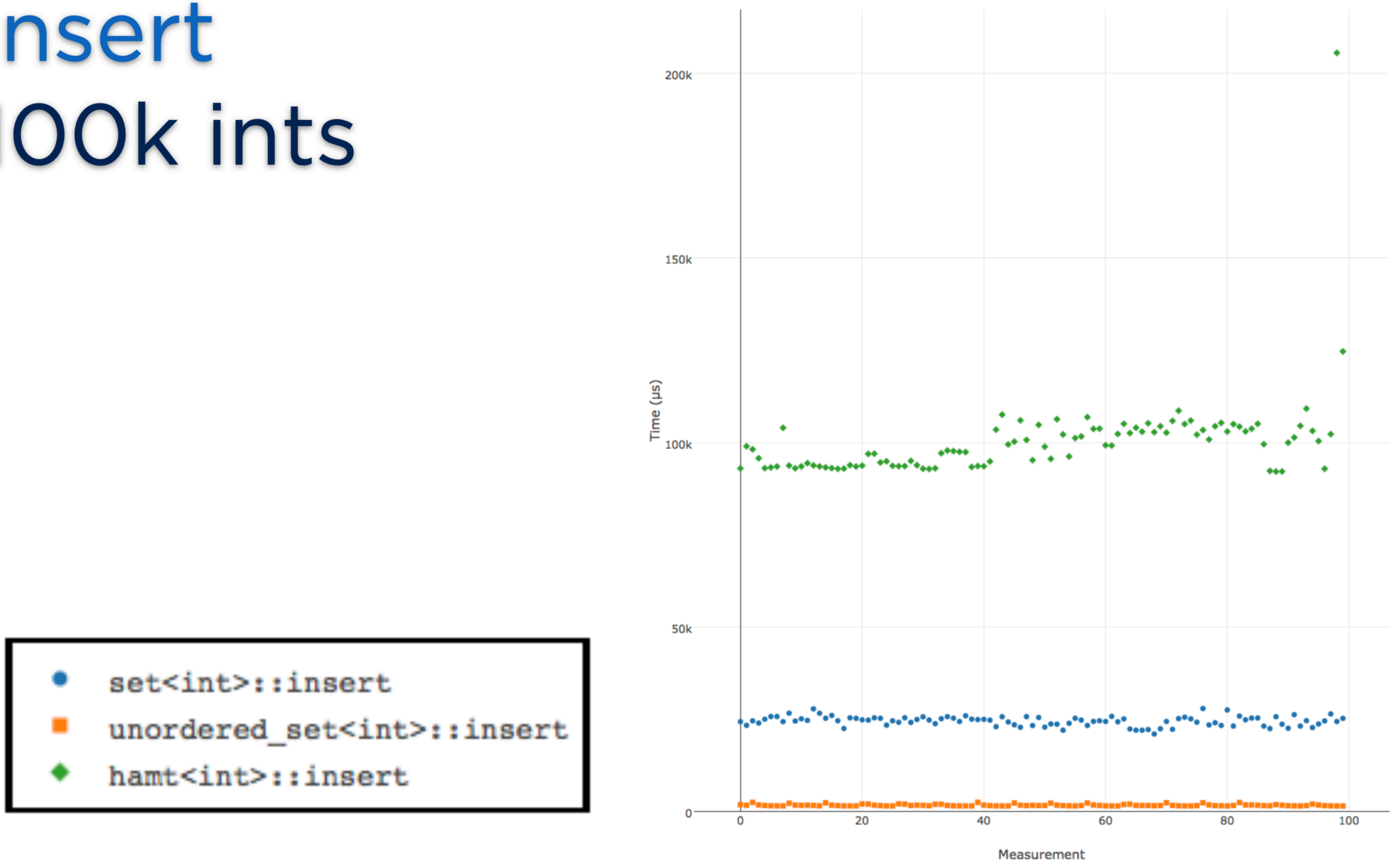
A C++ micro-benchmarking framework

Benchmarking Data Taken From A Talk By
Phil Nash,
Developer Advocate

JET BRAINS

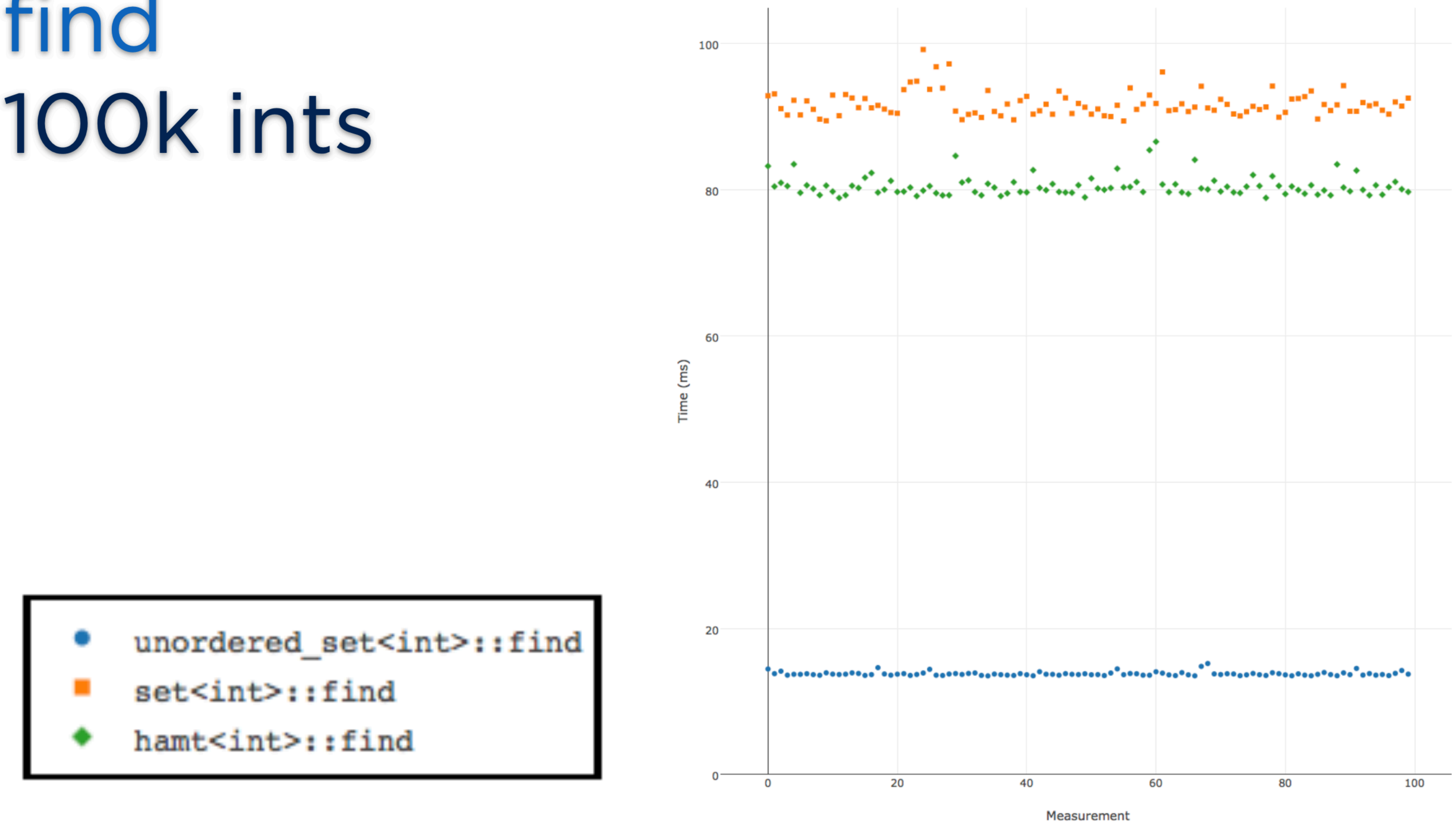
BenchMarking Of Search Tree

insert
100k ints



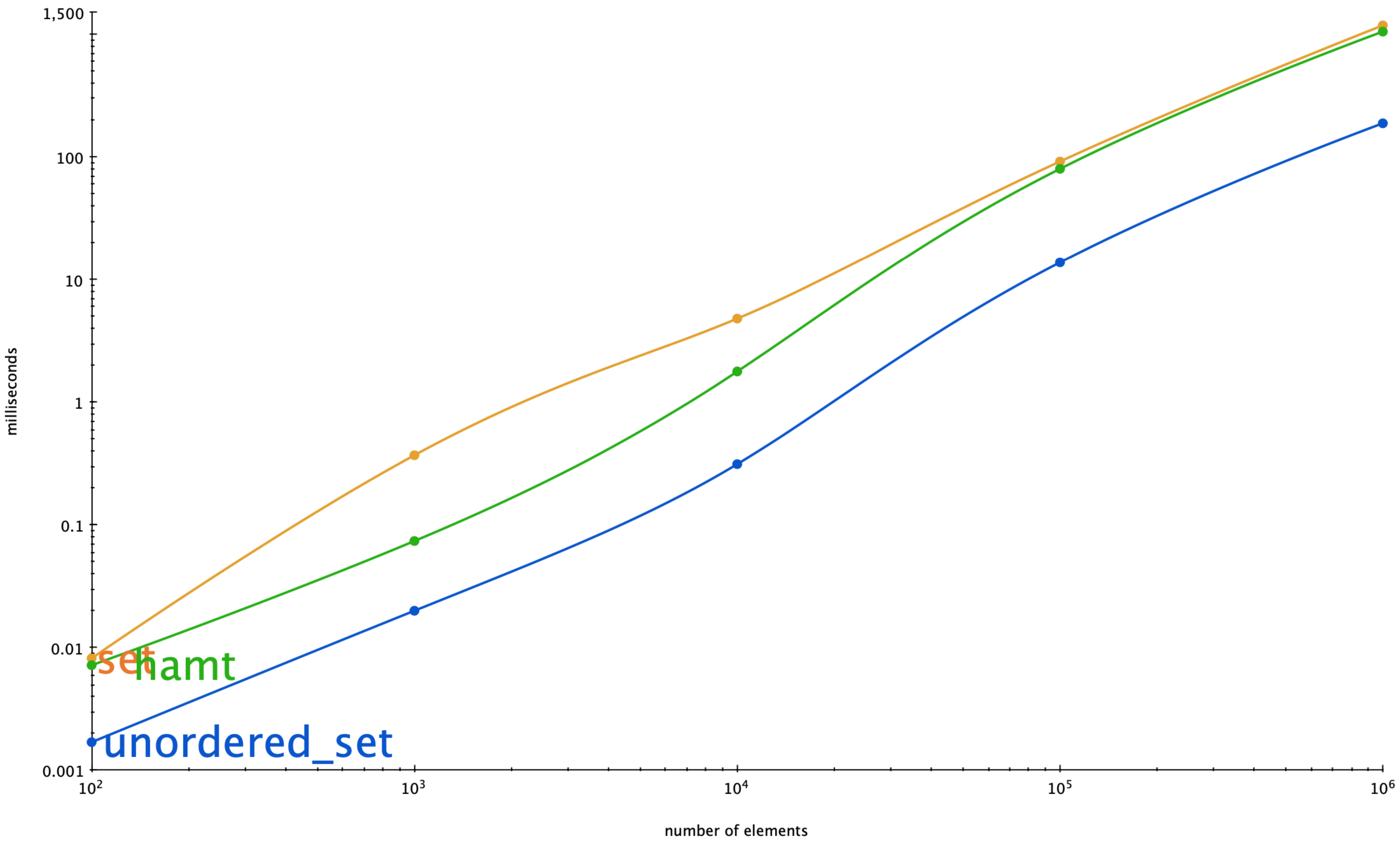
BenchMarking Of Search Tree

find
100k ints



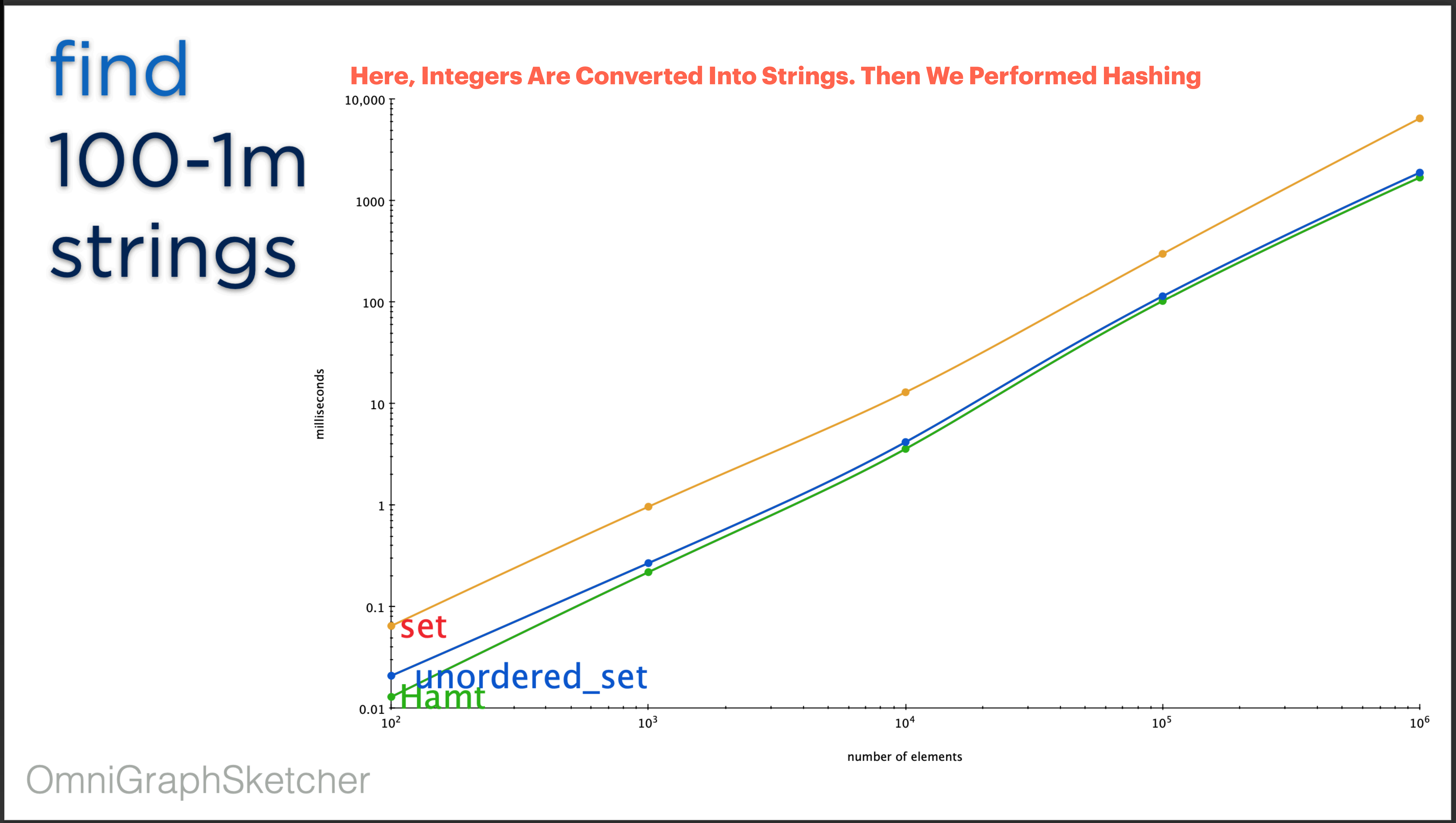
BenchMarking Of Search Tree

find
100-1m
ints



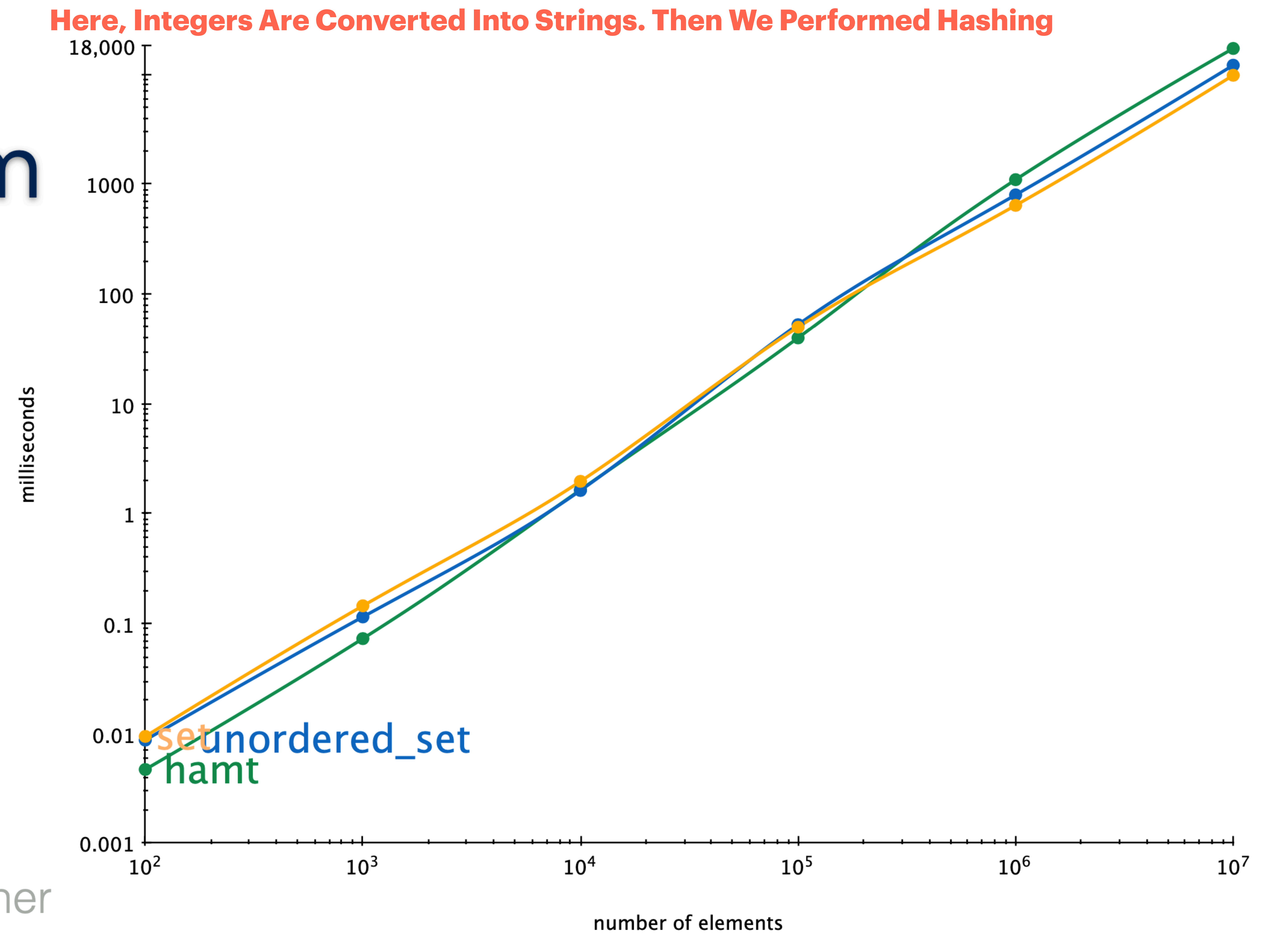
OmniGraphSketcher

BenchMarking Of Search Tree



BenchMarking Of Search Tree

find
100-10m
hashes



OmniGraphSketcher