Universidade do Minho
Instituto de Educação

Bucket Sort

From the implementation to its parallelization

Benjamim Coelho PG47164
Henrique Neto PG47238

Presentation Steps

Implementation of the sequential version:

- First Implementation based on Linked Lists
- Second Implementation based on Arrays
- Optimization of the sorting algorithm

Implementation of the parallel version:

- Parallelization of the Linked List Implementation
- Parallelization of the Array Implementation

Performance Tests and Analysis

- Analysis of the gain with multiple secondary algorithms (*Insertion Sort* vs *Merge Sort*)
- Analysis of each implementation's sequential and parallel version
- Analysis of the scalability of the best implementation across different machines

Basic Bucket Sorting Process

1. Calculate the maximum, minimum.
   - The values are then used to calculate the range each bucket
2. Scatter all the elements across the buckets
   - Each element is indexed to its bucket ($index = (value\ - minimun)$ % range)
3. Call a secondary Sorting function on each bucket
   - In our case, either Merge Sort or Insertion Sort
4. Regroup each ordered array into the original array.
   - Only performed on the first implementation

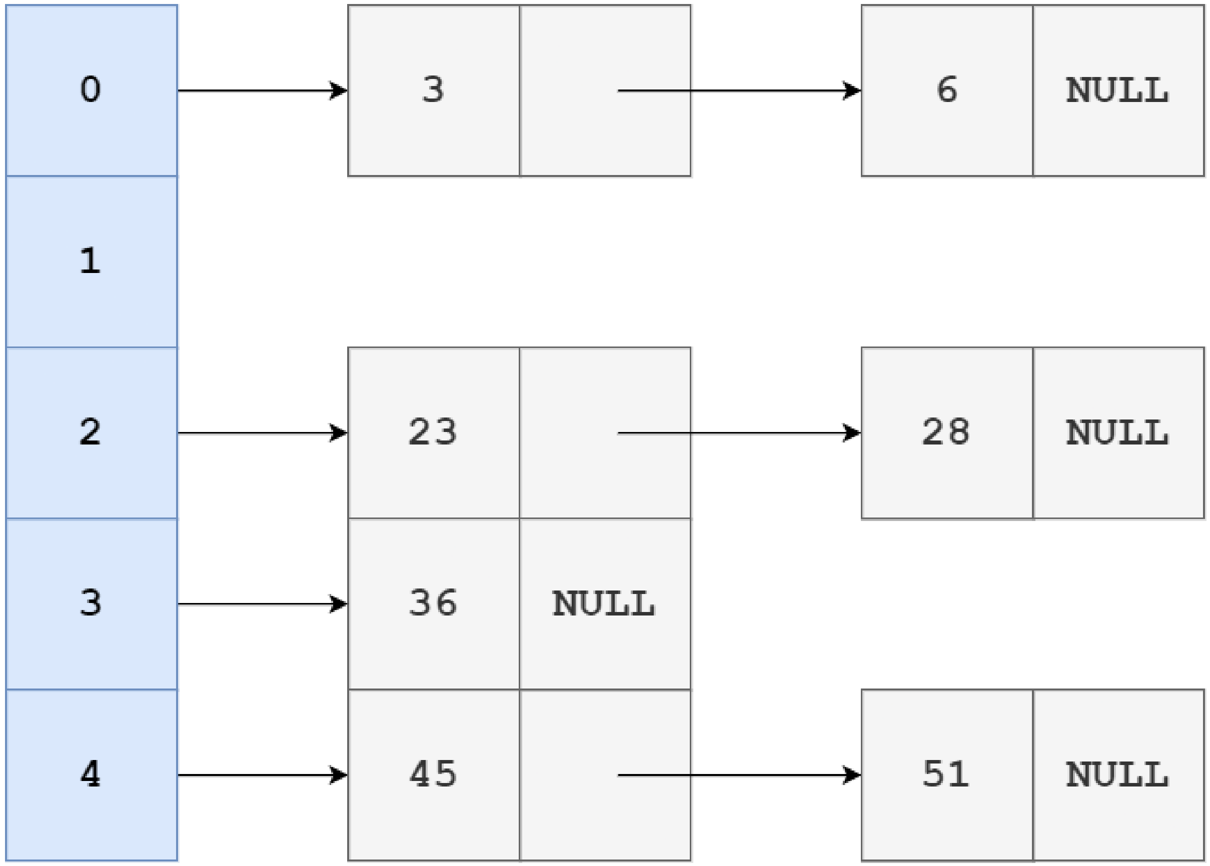
Sequential version based on Linked Lists

Vantages

- Simple and Dynamic Structure
- Easy to Use and Manage
- Easy to visualize

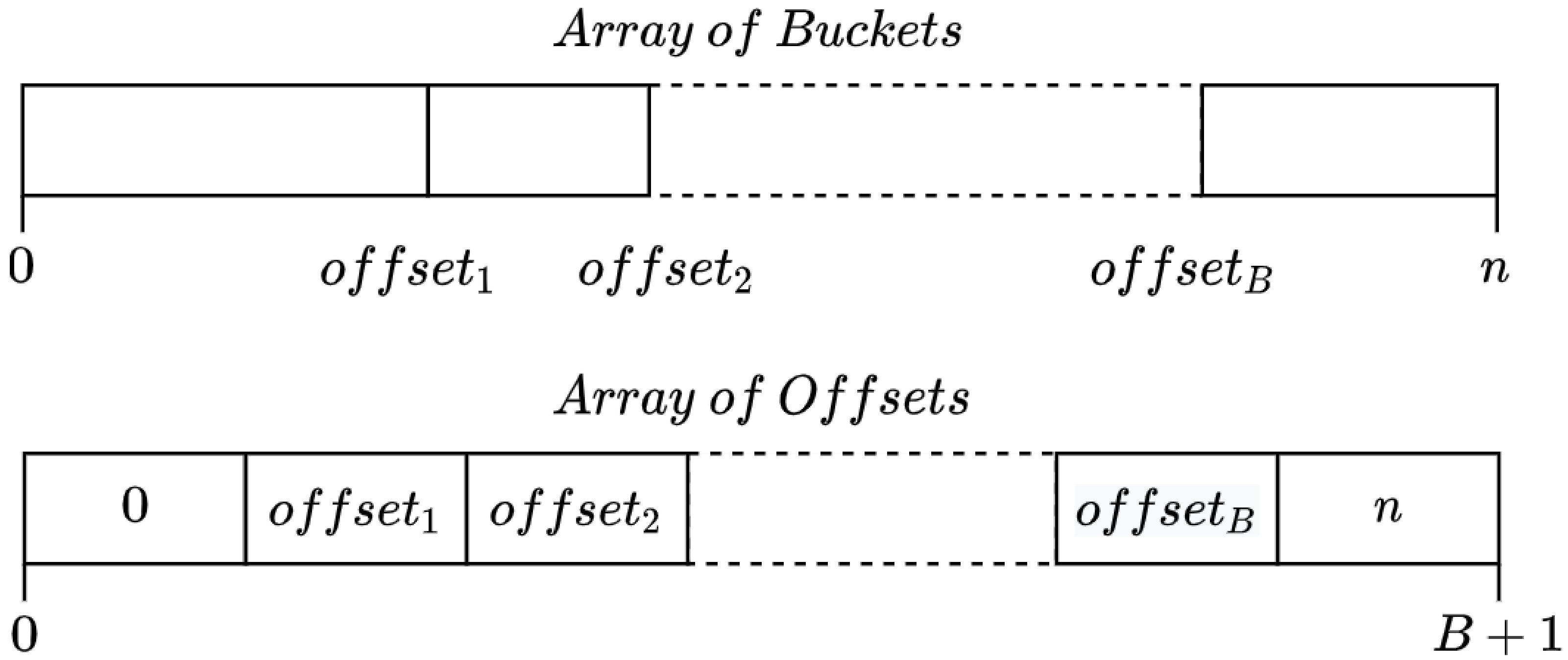
Disadvantages

- Extremely long allocation times when in large scale
- Noncontiguous in memory, no cache locality.



Sequential version based on Arrays

- Uses two arrays to sort the original array (*bucket_array and offsets_array)*
- The *bucket_array contains all the buckets*
- The *offsets_array* contains all the interval offsets.



Sequential version based on Arrays

Vantages over Linked Lists

- Efficient memory usage (no need to store a pointer with each value)
- Contiguous elements and bucket, allow for the use of cache spacial locality
- No need to regroup the buckets after each sorting
- Constant number of allocations per sorting

Disadvantages over Linked Lists

- Increased complexity on the scattering step
- Not optimal for distributed memory parallelism

Optimization of the sorting algorithm

- There wasn't any significant gains that could be easily achieved from loop unrolling.
- On the initial version of the first implementation, we used Insertion Sort, so the overhaul algorithm scaled terribly $\theta(n^2)$. We switched to Merge Sort $\theta(n \log(n))$.
- The initial version of the second implementation first used the original array to store the buckets requiring a non contiguously access to the original array in order to not lose data. We choose to use more memory to store the *bucket_array* on a new array, which allowed for contiguous reads on the input array decreasing the amount of cache misses.

Parallelization of both implementations

We considered three ways to parallelize the algorithm, however only one proved to be significant and was implemented.

1. We proposed parallelizing the scattering step, however, due to the bucket's dependencies, most heavy operations required to be done sequentially.
2. We proposed simply scattering the sorting of each individual bucket across multiple threads. This came with no drawbacks and was extremely significant to the overall performance.
3. We proposed to increase the granularity of the previous proposal and scatter each bucket’s sorting algorithm through multiple threads. This resulted in an excessive number of threads which halted the performance.

Parallelization of both implementations

To guarantee the even distribution of work across all threads we used a guided schedule in order to counteract the inconsistencies caused by the different size of each bucket.

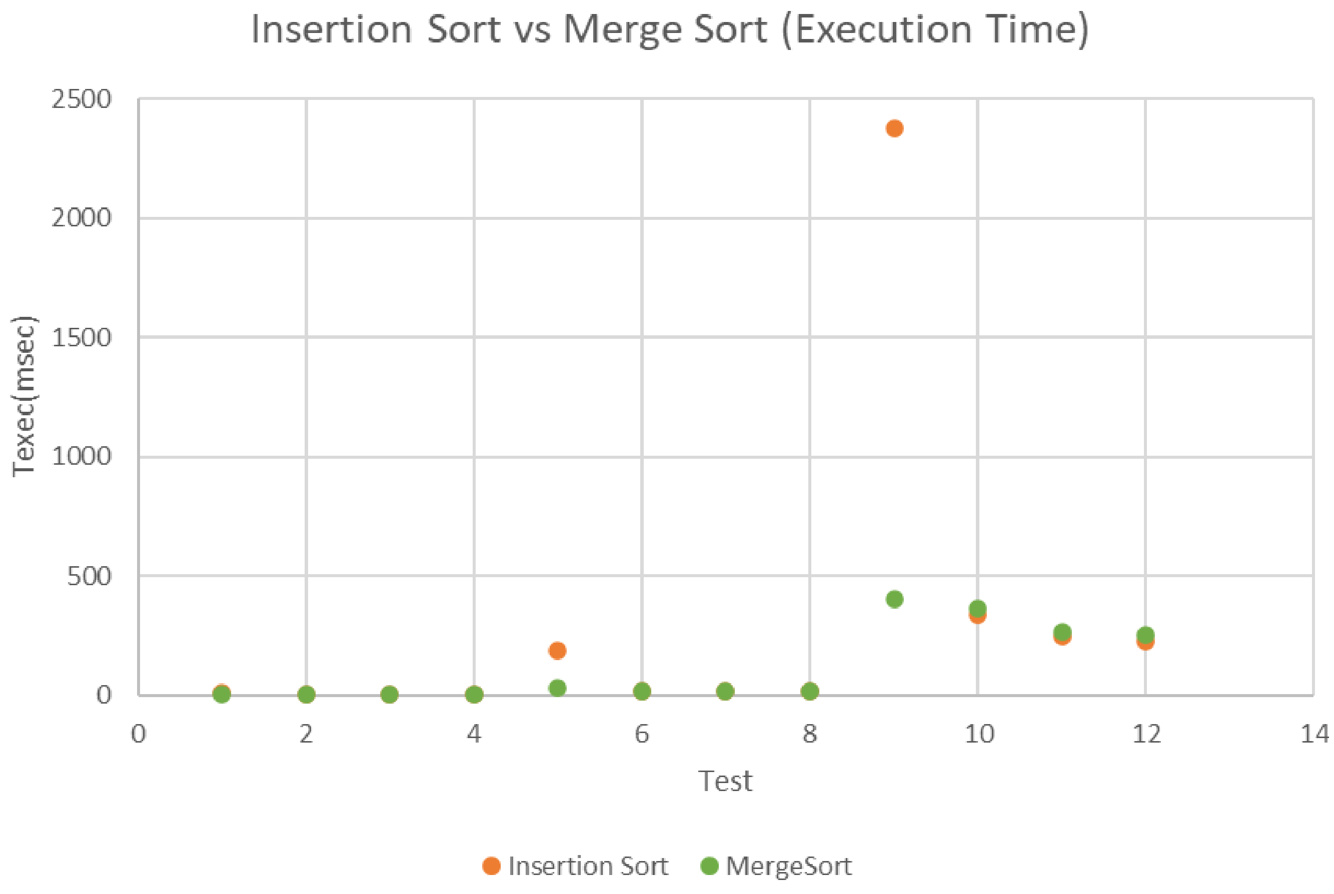
The result consisted of adding a pragma omp for schedule(guided) to each implementation.

Tests and Analysis

- We used the PAPI API to measure the execution time, #CC, #I and L1/L2 data cache misses.
- We also used the *perftool* to measure the overhead of each function in each implementation.
- For each situation, there was an off-record warmup run followed by five runs. Each run ordered a unique and randomly generated array, which was built before any measurement took place.

Tests and Analysis - Insertion vs Merge sorting

Test	N	Buckets	Insertion	Merge
1	10000	10	9.43	2.19
2	10000	100	1.413	1.662
3	10000	1000	1.276	2.189
4	10000	10000	1.572	2.229
5	100000	100	185.169	29.823
6	100000	1000	18.244	19.344
7	100000	10000	17.548	20.148
8	100000	100000	16.944	17.578
9	1000000	1000	2378.943	401.544
10	1000000	10000	338.146	363.13
11	1000000	100000	247.592	262.09
12	1000000	1000000	228.403	252.511

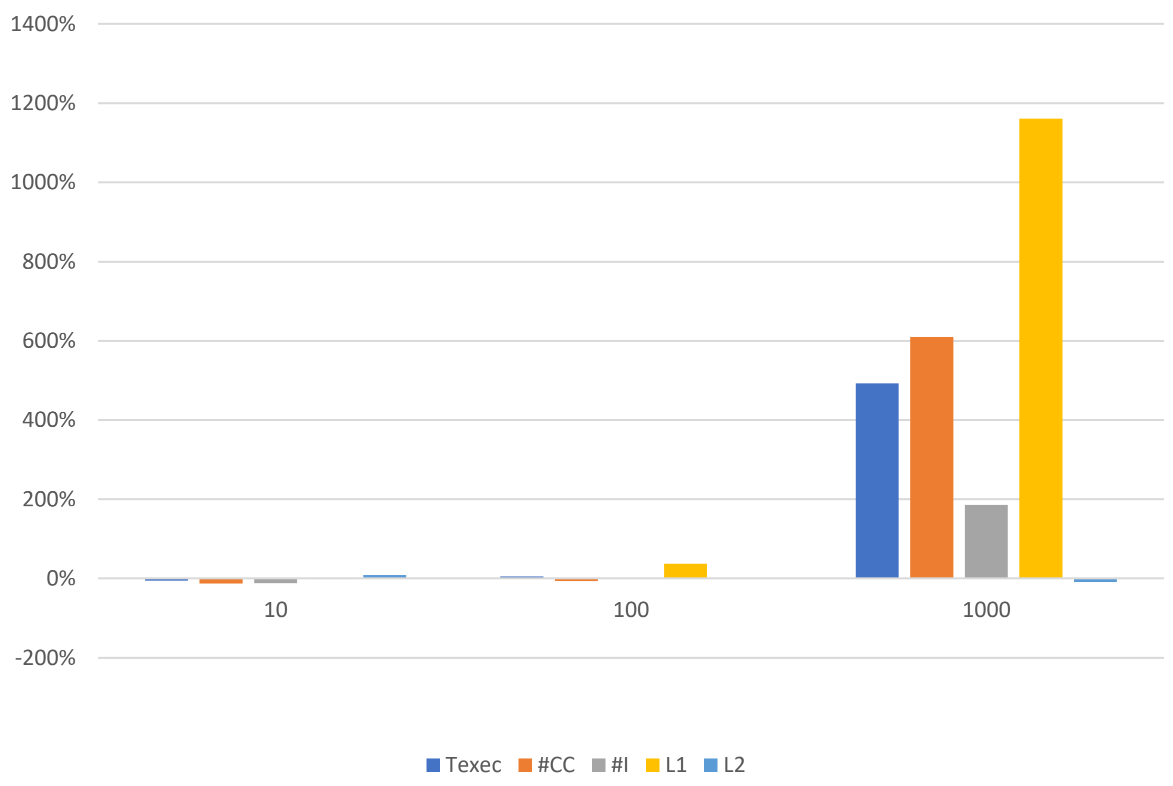


Tests and Analysis - Insertion vs Merge sorting

Since our arrays are completely random, we can consider:

$$Average\ Bucket\ Length \approx \frac{Length}{\#Buckets}$$

AVG	T	#CC	#I	L1	L2
10	-6%	-13%	-12%	2%	2%
100	5%	-7%	1%	38%	-1%
1000	492%	609%	186%	1161%	-9%



Overhead analysis – Perf-report tool

Overhead - Insertion Sort

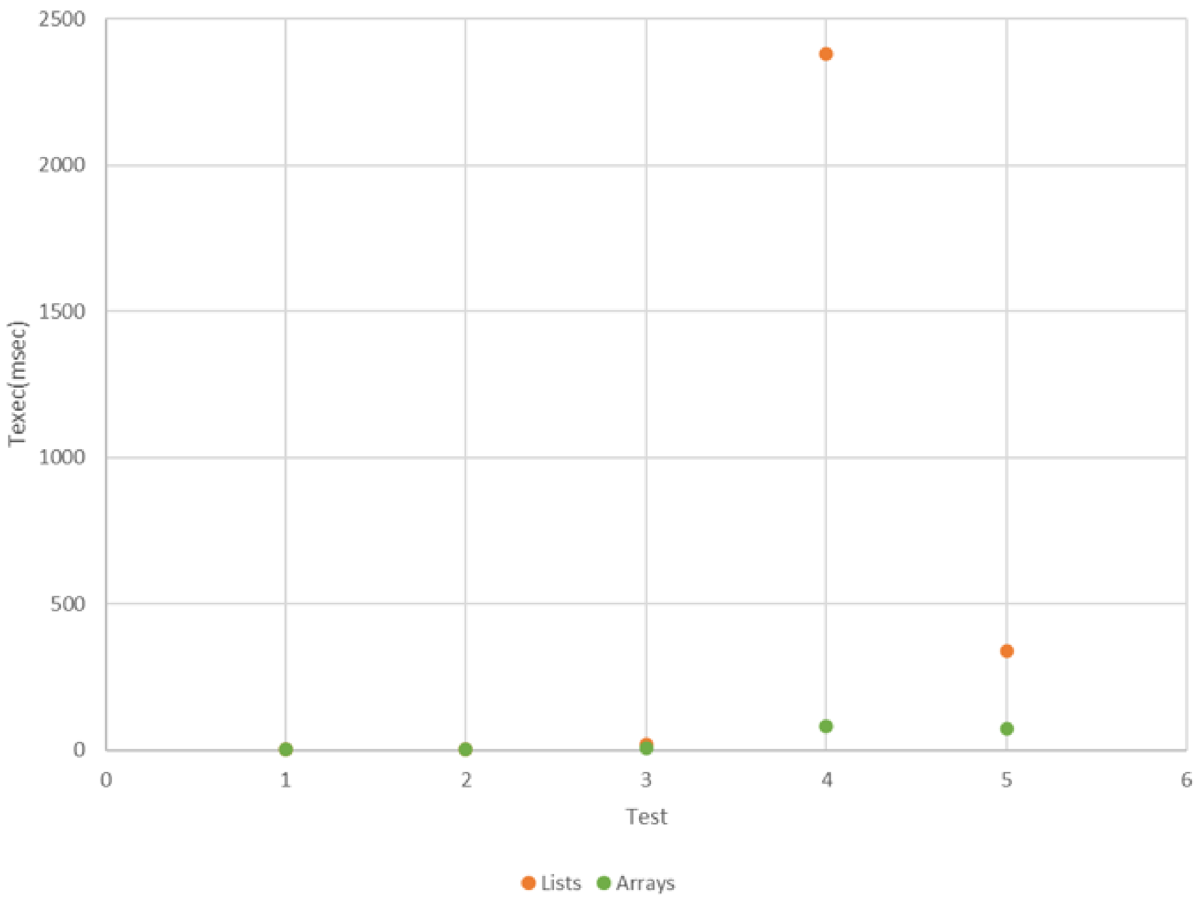
Overhead	Samples	Command	Shared Object	Symbol
29.71%	407	main.out	main.out	[.] InsertionSort_List
25.49%	350	main.out	main.out	[.] BucketSortSequen-tial_Lists_aux
23.32%	332	main.out	libc-2.17.so	[.] malloc_consolidate

Overhead - Merge Sort

Overhead	Samples	Command	Shared Object	Symbol
24.78%	352	main.out	main.out	[.] BuckerSortSequen-tial_Lists_aux
22.25%	318	main.out	libc-2.17.so	[.] malloc_consolidate
19.08%	273	main.out	main.out	[.] MergeSort_List
12.87%	184	main.out	main.out	[.] SortedMerge_List

Tests and Analysis – Lists vs Arrays Execution Time

Id Test	N	Buckets	T Lists	T Arrays
1	10000	10	2190	973
2	10000	100	1662	778
3	100000	1000	18244	6671
4	1000000	1000	2378943	81844
5	1000000	10000	338146	73058



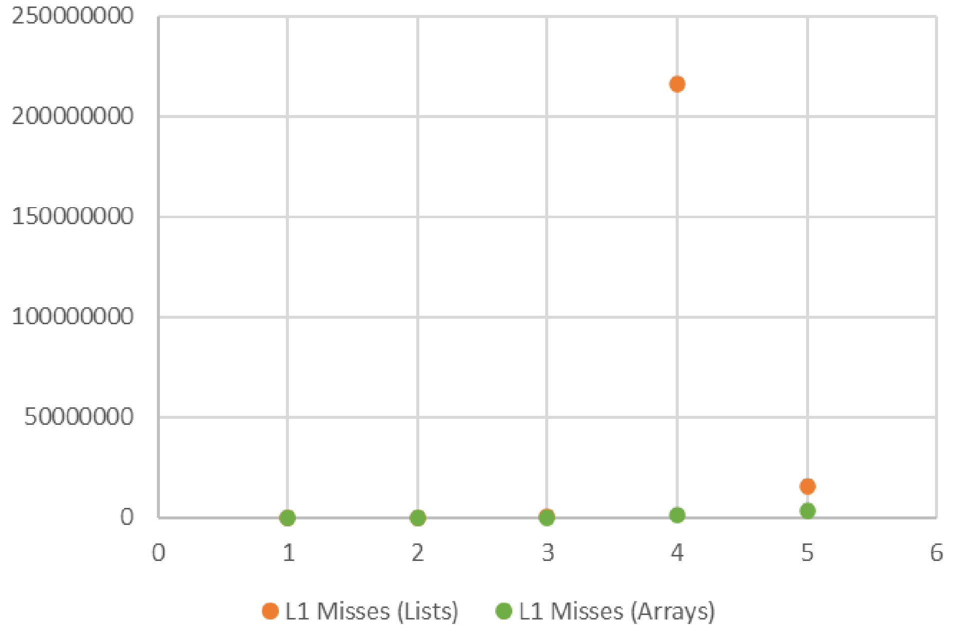
Overhead analysis – Perf-report tool

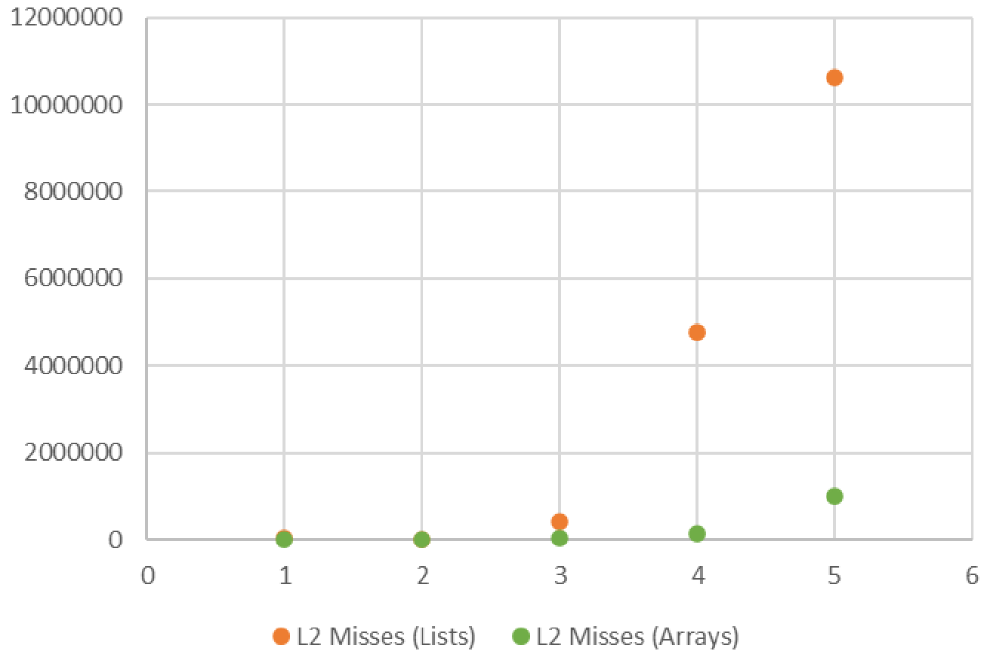
Overhead - Arrays' Implementation – Merge Sort

Overhead	Samples	Command	Shared Object	Symbol
70.17%	218	main.out	main.out	[.] MergeSort_Array
9.44%	30	main.out	main.out	[.] sort_into_bucket_array
6.01%	29	main.out	[unknown]	[k] 0xffffffffb418c4ef
4.95%	16	main.out	lib-2.17.so	[.] __random_r

Tests and Analysis – Lists vs Arrays Misses Cache L1 and L2

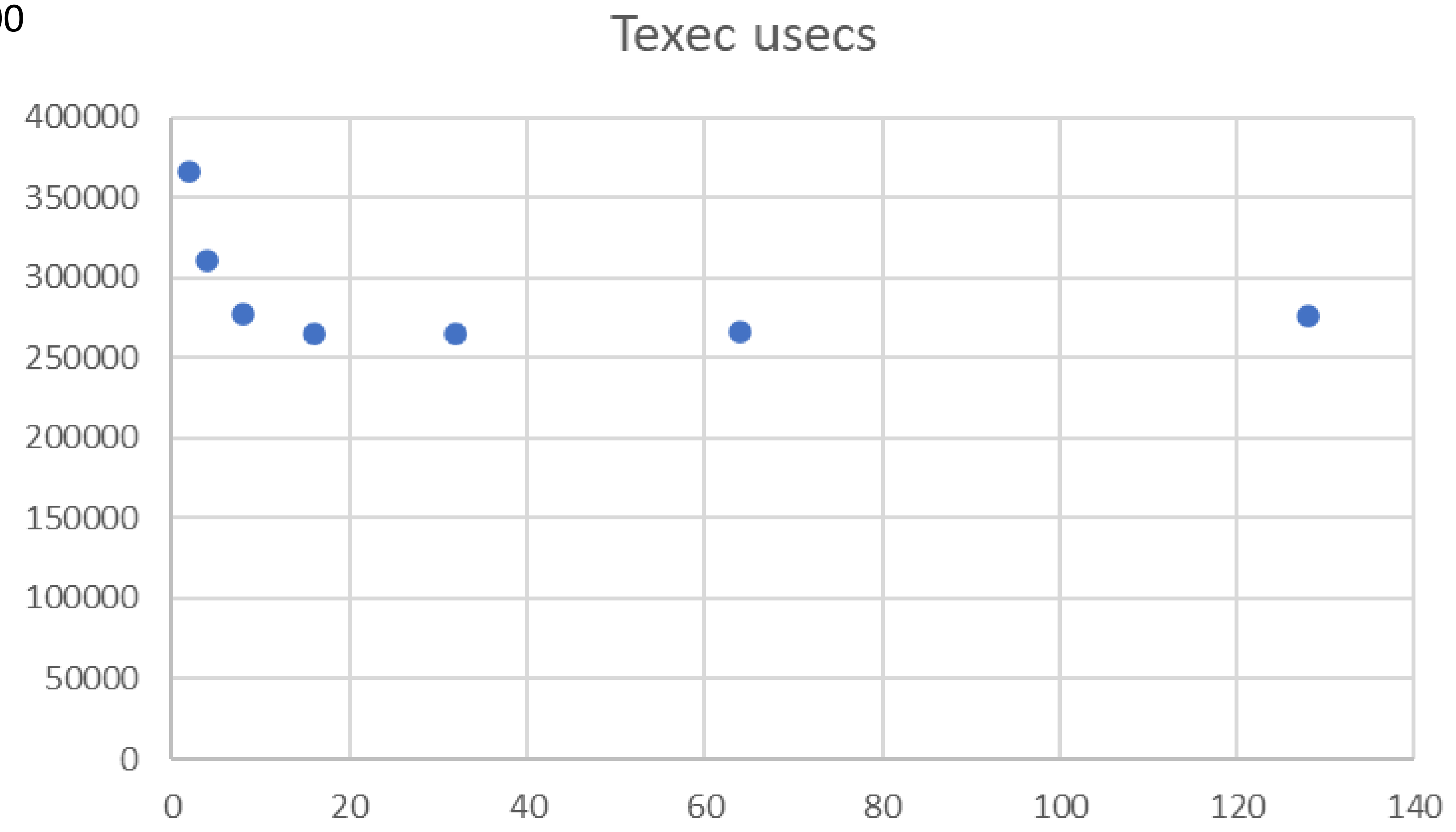
Test Id	N	Buckets	L1 Misses (Lists)	L2 Misses (Lists)	L1 Misses (Arrays)	L2 Misses (Arrays)
1	10000	10	109975	25845	6460	381
2	10000	100	45475	14907	6395	435
3	100000	1000	518601	397429	124892	15667
4	1000000	1000	215979985	4764159	1285953	131949
5	1000000	10000	15374112	10619778	3429509	999184



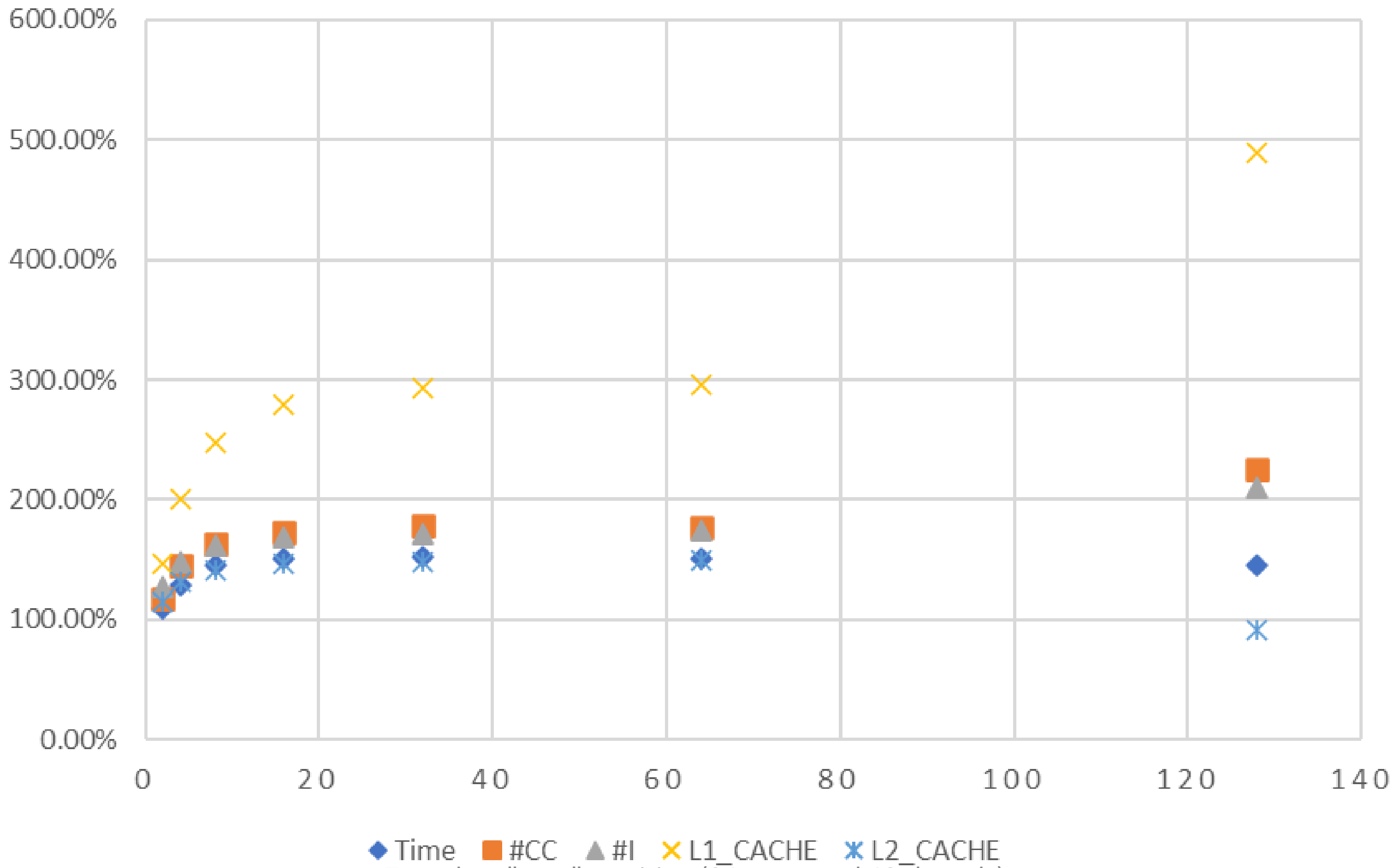


Tests and Analysis – Lists – Parallel Version

- N = 1000000
- B = 1000



Lists – Gains
From Sequential to Parallel



Tested on "cpar" partition (24 cores and 48 threads)

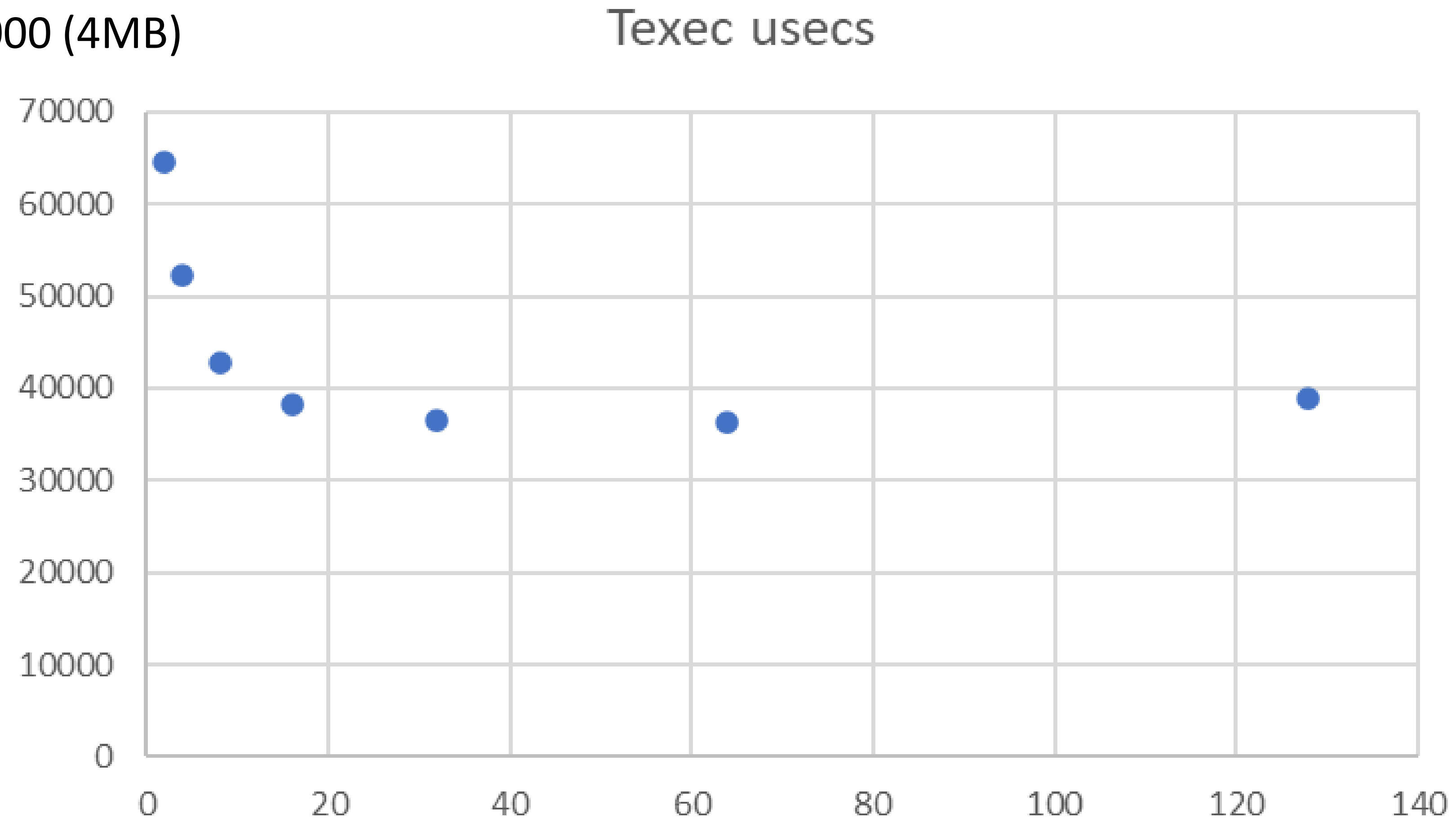
Overhead analysis – Perf-report tool

Overhead - Parallelized Lists' Implementation – Merge Sort

Overhead	Samples	Command	Shared Object	Symbol
26.58%	702	main.out	main.out	[.] MergeSort
24.05%	623	main.out	main.out	[.] SortedMerge
14.25%	348	main.out	main.out	[k] BucketSortParallel_aux

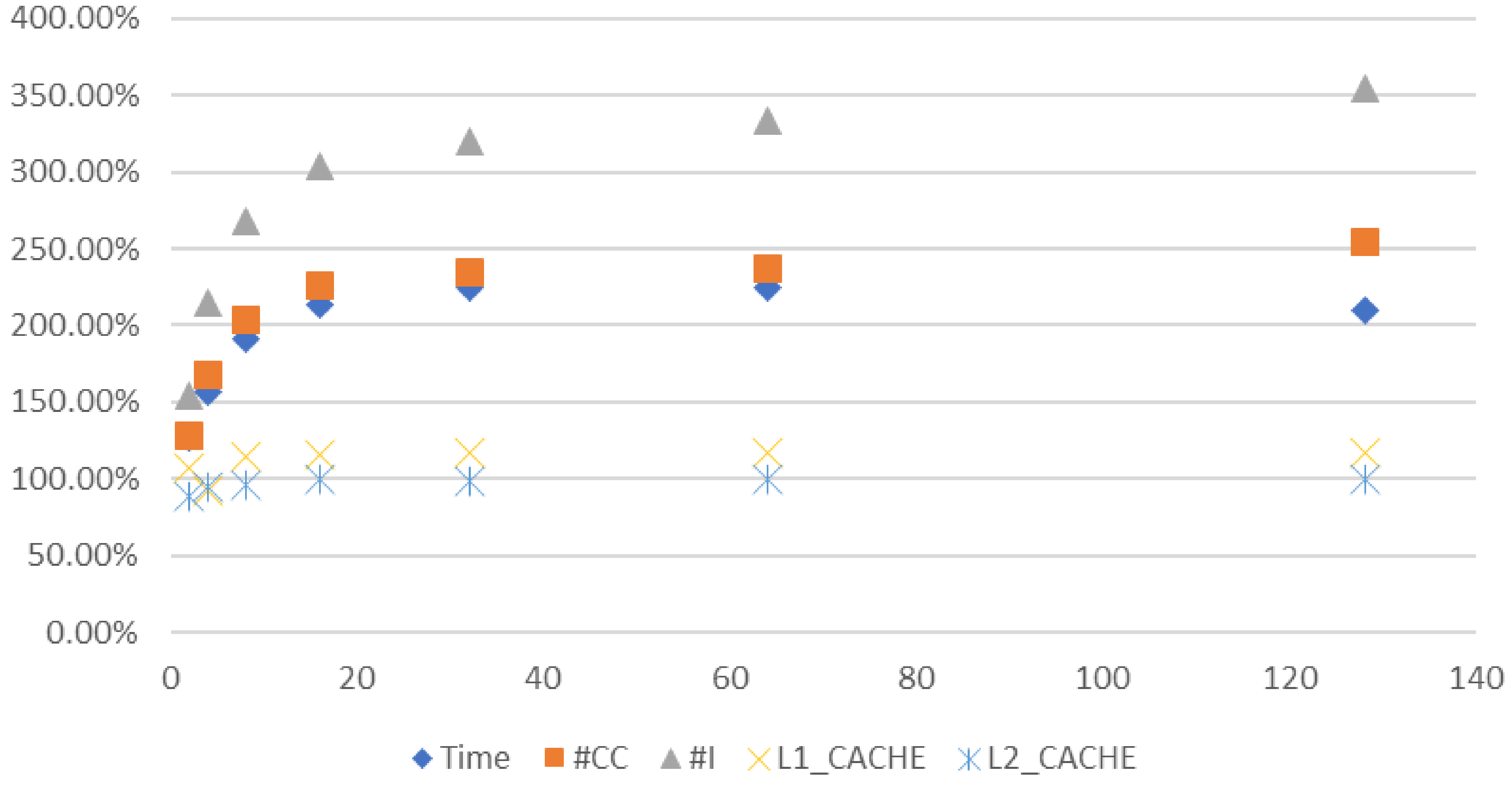
Tests and Analysis – Arrays Parallel Version

- N = 1000000 (4MB)
- B = 1000



Tested on "cpar" partition (24 cores and 48 threads)

Arrays – Gains
From Sequential to Parallel



Tested on "cpar" partition (24 cores and 48 threads)

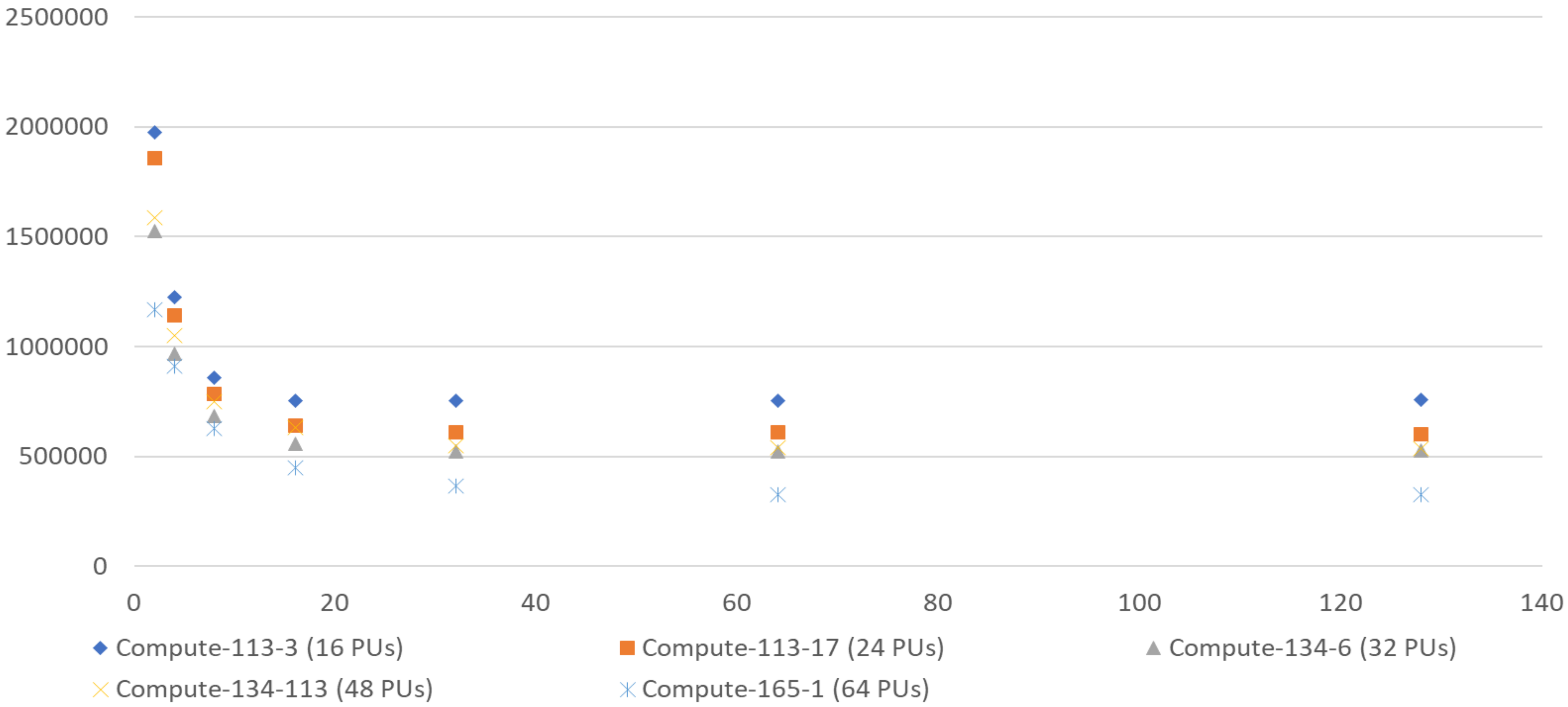
Overhead analysis – Perf-report tool

Overhead - Parallelized Arrays' Implementation – Merge Sort

Overhead	Samples	Command	Shared Object	Symbol
62.38%	367	main.out	main.out	[.] MergeSort_Array
8.60%	189	main.out	[unkown]	[k] 0xffffffffb4196098
7.04%	33	main.out	main.out	[k] sort_into_bucket_array

Tests and Analysis – Different Hardware

- N = 25000000 (100 MB)
- B = 1000



Tested on different machines of "day" partition; Execution times are in microseconds