VRIJE
UNIVERSITEIT
BRUSSEL

PARALLELISM PROJECT

Parallelizing the Wikipedia Search Engine

Rayane Kouidane - 0587073

15 May 2024

Sciences and Bio-engineering Sciences

Contents

1 Implementation

The implementation of this project can be divided into two phases. The first phase involves parallelizing the processing of articles and the second phase involves counting the number of occurences of a keyword in the text of an article in parallel.

1.1 Phase 1

The class that implements the first phase is the *SearchTask* class which extends the *RecursiveTask*. This class follows the map pattern which operates on each element of a collection independently to create a new collection of the same size. The goal of this class is to parallelize the processing of articles by dividing the dataset into smaller subsets until the size of these subsets falls below the sequential threshold T. If the size of the subset falls below this threshold, the class will perform a sequential search. This will evaluate the relevance of the given subset, and sort the result. The results from the left and right tasks are merged using a merge-sort algorithm. This ensures that the the final output remains sorted.

1.2 Phase 2

For the second phase the *CountTask*, which extends the *RecursiveTask*, was implemented. This class follows the reduction pattern by producing a single answer from a collection via the associative operator $+$. The purpose of this class is to count the occurrences of a keyword within a given text in parallel. The text is divided into smaller parts until reaching the sequential threshold T. Once the size is below this threshold the class counts the occurrences sequentially. The sum of all the results is taken as final output.

2 Evaluation

2.1 Experiments and experimental set-up

The purpose of the experiments was to determine how different levels of parallelism and thresholds affect the implementation's overhead and speed-up under various computational loads. I employed the Java Microbenchmark Harness (JMH) framework, configured the benchmark to save the results in a csv file, and enabled garbage collection between runs to avoid slowdowns. The slowdowns are introduced by the garbage collector when suspending the program execution to ensure the safety of the object graph. I adjusted the following parameters to evaluate their impact:

- p (Number of threads): To see how adding cores affects performance, this parameter was adjusted from 1 to the maximum number of available cores. On my desktop, these values were 1, 4, 8, 16, and 20; on the Firefly server, they were 1, 8, 16, 32, 64, and 128.
- T (Sequential threshold phase 1): To observe the impact of the threshold on the execution time, this parameter was adjusted from 1 to the number of articles in the dataset. It increases exponentially in powers of 10. If there is a significant gap then I added an intermediate value to guarantee coverage across the range. For the benchmark this threshold was called *searchCutoff*.
- $T2$ (Sequential threshold phase 2): This parameter follows the same structure as T, but it is adjusted from 1 to the length of the longest article. For the benchmark this threshold was called *countCutoff*.

I ran different warmup and measurement iterations depending on the dataset size:

- *Small*: 5 warmup iterations, 10 measurement iterations.
- *Medium*: 5 warmup iterations, 10 measurement iterations.
- *Large*: 5 warmup iterations, 10 measurement iterations.
- *Firefly*: 2 warmup iterations, 3 measurement iterations.

All the properties (overhead and speed-ups) are determined for T = $+\infty$ as mentioned in the assignment in section 4.2, except when observing the impact of different thresholds.
Desktop Specifications:

- **Hardware**:
  - **CPU**: 13th Gen Intel(R) Core(TM) i5-13600KF (14 cores @ 3.50GHz base, 5.1GHz boost, 20 threads)
  - **RAM**: 32GB DDR4-3600
  - **SSD**: 2x1TB nvme
- **Software**:
  - **OS**: Windows 11 Familiy Edition
  - **Java**: correto version 19.0.2 (Java 19)
  - **Heap size**: 2048Mb

Firefly Specifications:

- **Hardware**:
  - **CPU**: AMD Ryzen Threadripper 3990X Processor (64 cores @ 2.9Ghz base, 4.3Ghz boost, 128 threads)
  - **RAM**: 128 GB DDR4-3200
  - **SSD**: 2x2TB nvme
- **Software**:
  - **OS**: Ubuntu 22.04.4 LTS
  - **Java**: openjdk version 19.0.2 (Java 19)
  - **Heap size**: 100Gb

2.2 Overhead and speed-up

What is the overhead of your implementations? Is this overhead acceptable? What causes this overhead?
The overhead for the different dataset sizes can be found in figure 1 along with the 99% confidence interval. The overhead $\frac{T_1}{T_{seq}}$ is the amount of additional time required for the parallel execution compared to the sequential execution. For instance, the overhead of the firefly dataset is $1,639012027 \pm 0,034768653$. This means that on average the parallel execution with one core takes 1.64 times longer than the sequential execution. This overhead is introduced by the Fork/Join framework during task creation, scheduling and joining. The more tasks we split, the more overhead it causes. Overhead also comes from the synchronisation and communication of the tasks, such as when merging the results of different tasks. In the context of parallelizing a search engine, the main objective is to reduce the execution time, despite the presence of overhead. Given the significant speed-up achieved with the addition of more cores, as shown in figure 3 and 4, this overhead is considered acceptable.

What are the speed-ups of your implementations? Are these speed-ups higher or lower than expected? Why?
Figure 3 and 4 show the application and computational speed-ups, respectively along with the 99% confidence interval. The application speed-up $\frac{T_{seq}}{T_p}$ is defined as the ratio of the sequential execution time to the parallel execution time, whereas the computational speed-up $\frac{T_1}{T_p}$ compares the parallel execution time with one core to that on p cores. For example, the application speed-up is $4,938966896 \pm 0,081363547$ with 20 cores on the large dataset. This shows that the parallel program is nearly five times faster than its sequential counterpart. The computational speed-up for the identical case is $10,55049957 \pm 0,173806808$. This shows that compared to executing the tasks on a single core, the execution with 20 cores is more than ten times faster. The graphs' trends align with Gustafson's law. By increasing the size of the problem and the number of cores, larger workloads can be managed efficiently.

What speed-up is larger: the computational or application speed-up? Is this what you expect? Why?
It is evident from figure 2 that the computational speed-up is larger than the application speed-up, which aligns with the expectations. The application speed-up shows how efficiently the parallel implementation performs compared to the sequential, taking into account the overhead. The computational speed-up shows the improvement achieved when parallelizing the process over multiple cores compared to one core,

without taking the overhead into account. Computational speed-up focuses only on the raw speed-ups from the parallel processing. This is why the computational speed-up is larger than the application speed-up.

How well do your implementations scale? Why do they (not) scale well?
Figures 3 and 4 show that both the application and computational speed-ups increase with the number of cores, which indicate a good scaling. Especially with the medium and large dataset. However, for the small and firefly dataset, the speed-up reaches a plateau and even decreases slightly at 4 and 32 cores respectively. For the small dataset this can be explained given it only contains 9 articles. It becomes inefficient to process each article on a core since certain cores won't be utilized. The decrease on the firefly dataset might be explained by Amdahl's law, which states that a program's maximum speed is limited by the time needed by the sequential fraction of the task.

Based on your experiments, which of both implementations is the best? Why?
Parallel vs Sequential
Based on the experiments, the parallel implementation is the best approach. In the context of a Wikipedia search engine, users expect to find relevant articles quickly in a dataset containing millions of articles. It is impractical to let users wait two minutes for results when this time can be greatly reduced by parallelizing the search. The improvement justifies the use, although the overhead involved.

Phase 1 vs Phase 2
As discussed in the next section the second phase has a minimal impact on the speed-ups. Parallelizing the second phase is ineffective because it doesn't bring any significant improvement.

2.3 Optimal threshold

How does the threshold influence the overhead and speed-up of your implementations? What happens when you make the threshold larger or smaller? Explain.
Figure 5 shows how the thresholds of phase 1 and 2 affect the overhead. Instead of reading the values this graph is meant to show the overhead trends as the threshold varies. My expectation was that as the threshold increased, the overhead would decrease. When increasing the threshold less tasks are created, which reduces the

overhead associated with managing these tasks. However, the figure shows another trend. For the large and firefly dataset the overhead decreases slightly as expected. But for the smaller datasets the values of the overhead varies. This could be the result of load imbalance due to less articles to process.

Figures 6a, 7a, 8a, 9a, 10a, 11a, 12a and 13a show how the thresholds of the first phase affect the application and computational speed-up.The closer the threshold gets to the number of articles (increase), the smaller the application and computational speed-up become. When the threshold is increased, the number of tasks decreases and the workload increases. There are not enough tasks to distribute among the cores when large tasks are reached.

The impact of the threshold of the second phase is shown in figures 6b, 7b, 8b, 9b, 10b, 11b, 12b and 13b. The impact is not significant, because counting the occurrences of a keyword in a text has a low computational complexity.

Suppose you would need to determine a threshold naively (i.e., without benchmarking). How would you do this? What would you consider?
If I would need to determine a threshold naively I would first consider the size of the problem. In this context it is the number of articles. Then I would also consider the size of articles to ensure that I don't introduce load imbalance. Because then some cores would end up with more work than others. The threshold will be based on these two factors.

Based on your measurements, what is the best threshold? Why?
Since the firefly dataset is the one that most closely mirrors the real-world setting in which the search engine will be deployed, we will find its best threshold. I developed a T-score that combines the speed-ups and overhead to be able to determine the best threshold. In order to do that, I first scaled the overhead (O), the application speed-up (AS) and computational speed-up (CS) to be between 0 and 1. At threshold T, the T-score is finally computed as $AS_T + CS_T - O_T$. The T-scores for various thresholds are shown in figure 14. We may conclude that the best threshold for phase 1 is $T = 100000$ and for phase 2 it is $T2 = 100000$. This indicates that a task is processed sequentially when its size reaches 100000. A high threshold results in fewer tasks to manage and less overhead. It is the ideal balance between overhead and speed-up.

2.4 Implementation characteristics

Let n be the number of articles in the dataset, r the effort done to evaluate the relevance of an article and m the number of words in the text of an article.

What is the work of the computation?
We keep splitting the total number of articles in half until reaching the threshold T. In worst case scenario, the threshold is extremely small, leading to n splits, which is in $O(n)$. The relevance of each article must be evaluated which is $O(n \times r)$. The work for the first phase is then $Work_{search} = O(n) + O(n \times r) = O(n \times r)$
The splitting of the second phase follows the same logic as above and is then $O(m)$. Each word is compared to the keyword, character by character. Since the length of a word is a constant factor, it can be omitted. Comparing each word is then in $O(m)$. The work for the second phase is then $Work_{count} = O(m) + O(m) = O(m)$.
The total work is then $Work = O(n \times m)$. For the merge-sort, the work is: $Work_{merge} = O(nlog(n))$.

What is the span of the computation?
The span is the time needed by the "most expensive" path (i.e., the longest sequential path) in the tree of computations. For the first phase, each recursive call splits the tree in half. The maximum depth of the tree is then $log_2(n)$. The span is $Span_{search} = O(log(n))$
Every recursive call also splits the tree in half, for the second phase. The maximum depth of the tree is then $log_2(m)$ and thus the span is $Span_{count} = O(log(m))$.
The total span is $Span = O(log(n) + log(m))$. The span of the merge-sort is still $O(nlog(n))$ because it is a sequential implementation.

How many tasks are created by your implementation?
The height h of the binary tree, that represent the task splitting, is $log_2(n)$ with n articles. A complete binary tree has $2^{h+1} - 1$ nodes, which becomes $2^{log_2(n)+1} - 1$ and simplifies to $2n - 1$. Which is the number of tasks created.
Similarly, in the second phase, the height of the tree is $log_2(m)$ with m words. The total number of tasks for the second phase is then $2m - 1$.
The tasks of the second phase will be created only in the leaves of the tree created in the first phase. The number of leaves in the binary tree of the first phase is 2^h, where $h = log_2(n)$, so there are n leaves. The total number of task is the sum of the number of tasks of the first phase and the tasks created at each leaf of the tree of the first phase, which is: $2n - 1 + (n \times 2m - 1) = 2mn + n - 1$

3 Appendix

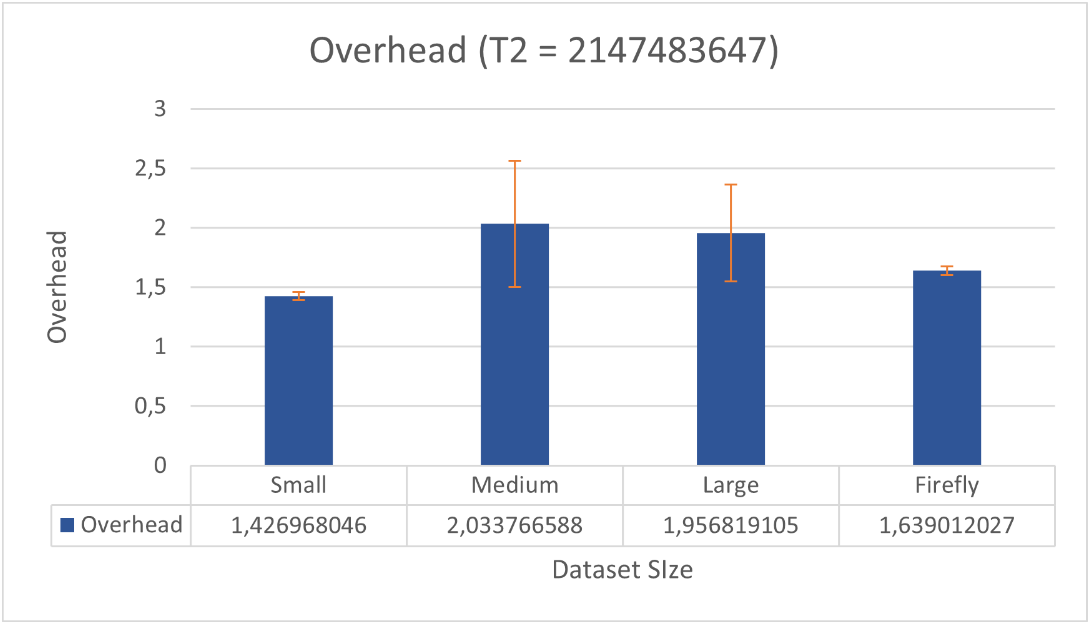


Figure 1: Overhead for each dataset size

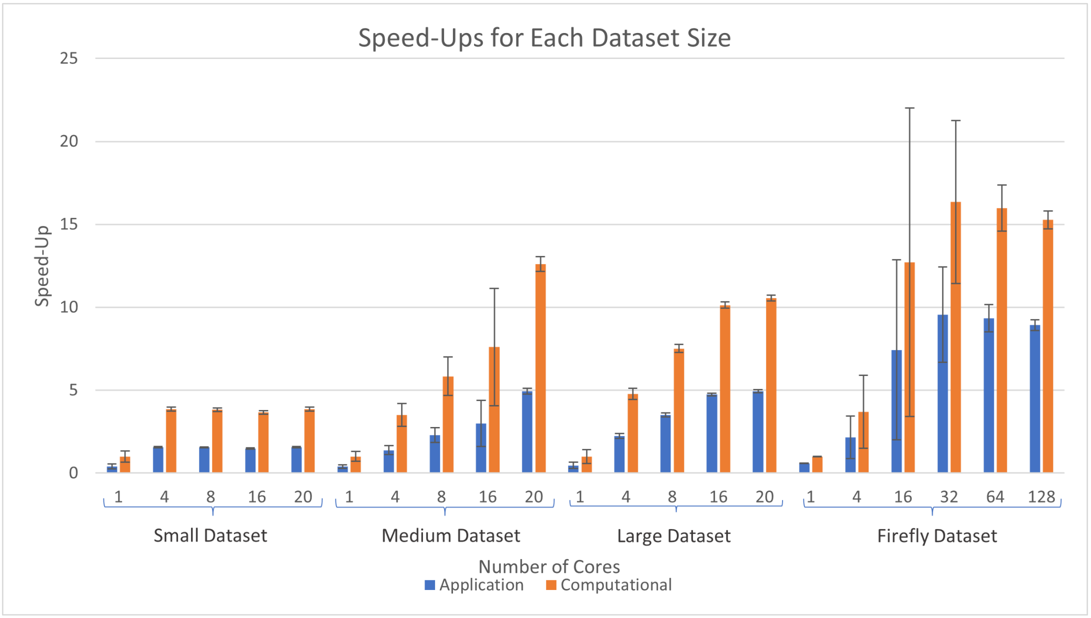
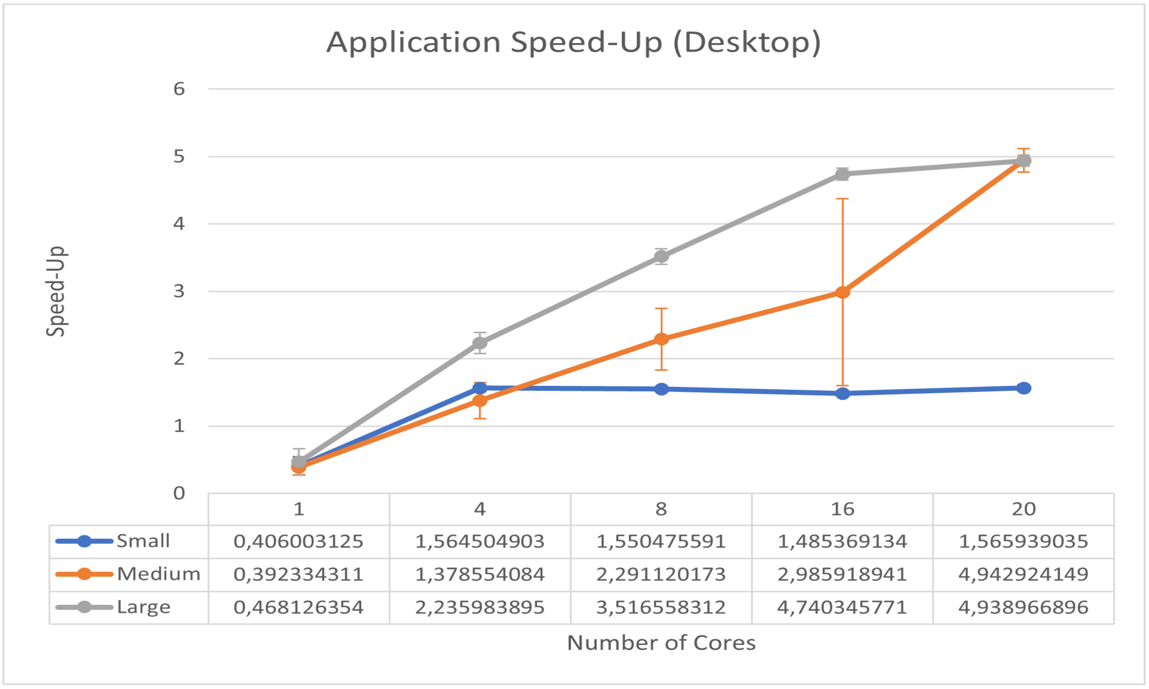
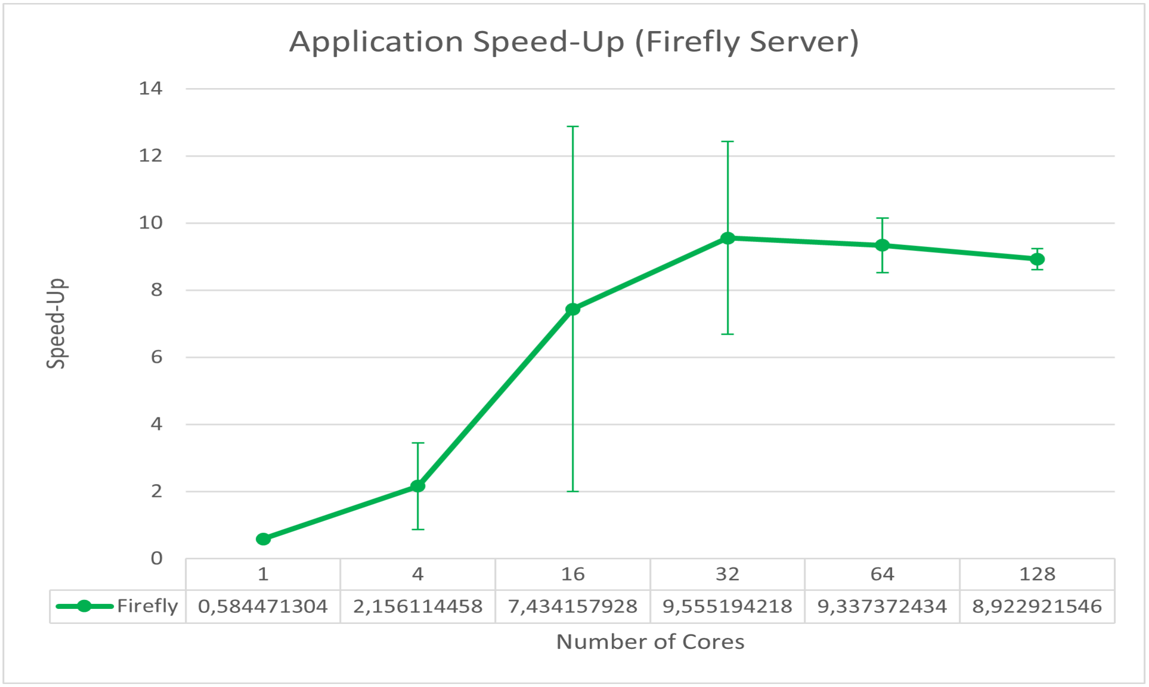


Figure 2: Speed-ups for each dataset size and different number of cores



	1	4	8	16	20
Small	0,406003125	1,564504903	1,550475591	1,485369134	1,565939035
Medium	0,392334311	1,378554084	2,291120173	2,985918941	4,942924149
Large	0,468126354	2,235983895	3,516558312	4,740345771	4,938966896

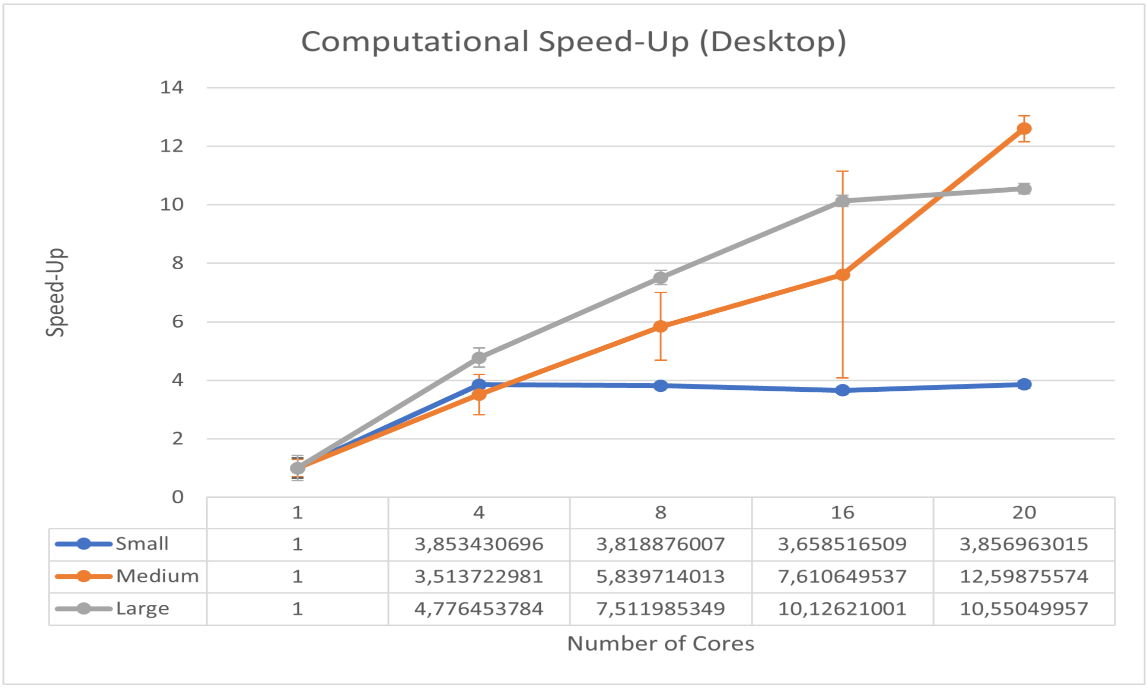
(a) Application speed-ups for the small, medium and large dataset



	1	4	16	32	64	128
Firefly	0,584471304	2,156114458	7,434157928	9,555194218	9,337372434	8,922921546

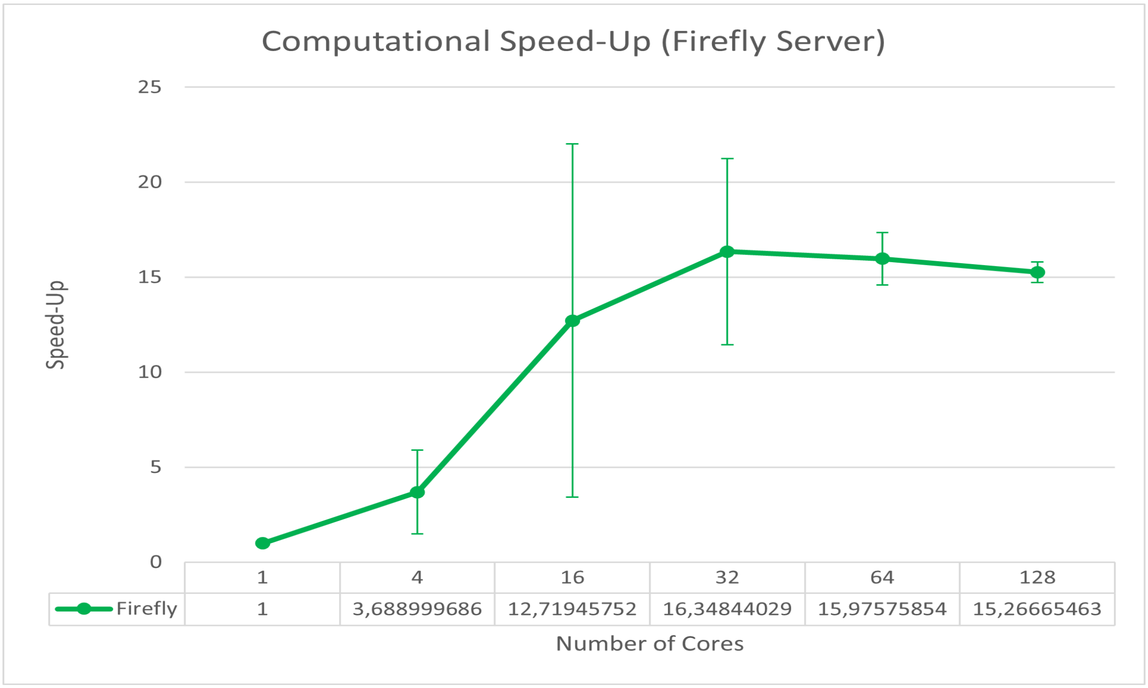
(b) Application speed-ups for the firefly dataset

Figure 3: Application speed-ups for different dataset sizes and various number of cores



	1	4	8	16	20
Small	1	3,853430696	3,818876007	3,658516509	3,856963015
Medium	1	3,513722981	5,839714013	7,610649537	12,59875574
Large	1	4,776453784	7,511985349	10,12621001	10,55049957

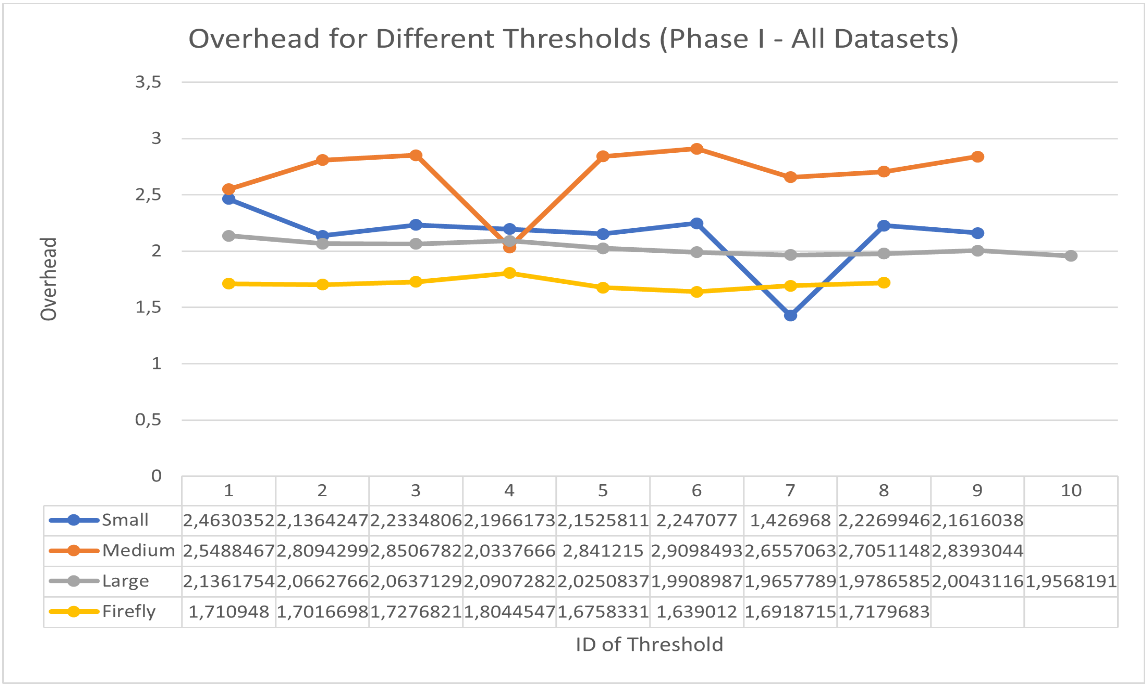
(a) Computational speed-ups for the small, medium and large dataset



	1	4	16	32	64	128
Firefly	1	3,688999686	12,71945752	16,34844029	15,97575854	15,26665463

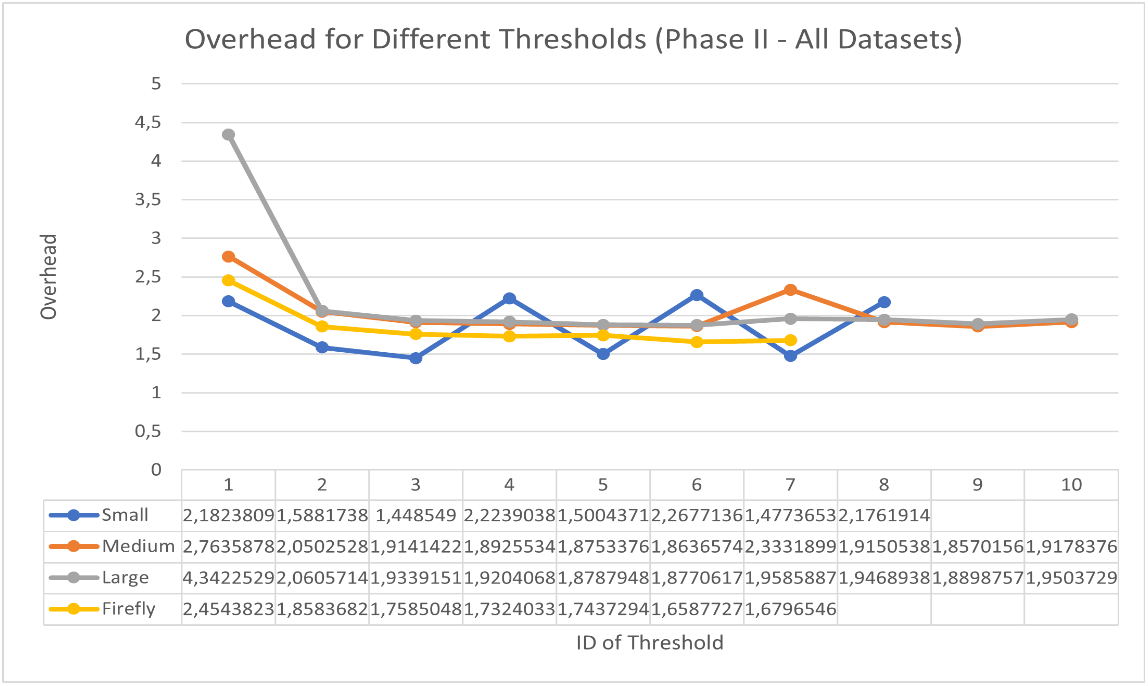
(b) Computational speed-ups for the firefly dataset

Figure 4: Computational speed-ups for different dataset sizes and various number of cores



	1	2	3	4	5	6	7	8	9	10
Small	2,4630352	2,1364247	2,2334806	2,1966173	2,1525811	2,247077	1,426968	2,2269946	2,1616038	
Medium	2,5488467	2,8094299	2,8506782	2,0337666	2,841215	2,9098493	2,6557063	2,7051148	2,8393044	
Large	2,1361754	2,0662766	2,0637129	2,0907282	2,0250837	1,9908987	1,9657789	1,9786585	2,0043116	1,9568191
Firefly	1,710948	1,7016698	1,7276821	1,8044547	1,6758331	1,639012	1,6918715	1,7179683		

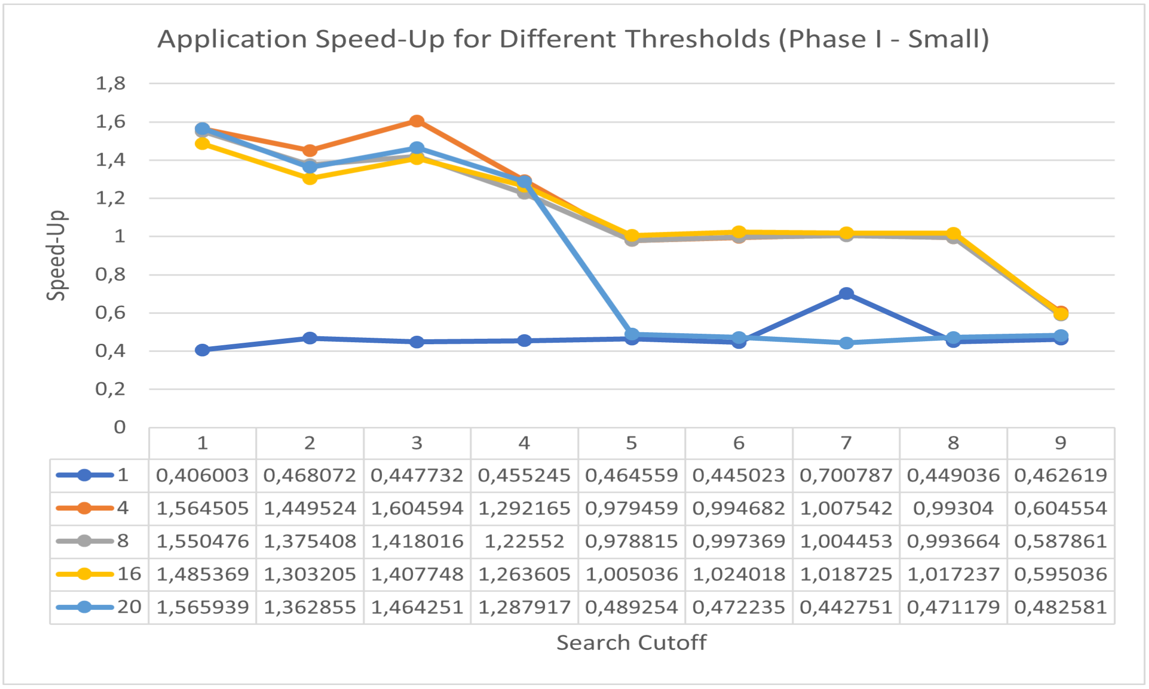
(a) Overheads for for the small, medium and large dataset with T2 $= +\infty$



	1	2	3	4	5	6	7	8	9	10
Small	2,1823809	1,5881738	1,448549	2,2239038	1,5004371	2,2677136	1,4773653	2,1761914		
Medium	2,7635878	2,0502528	1,9141422	1,8925534	1,8753376	1,8636574	2,3331899	1,9150538	1,8570156	1,9178376
Large	4,3422529	2,0605714	1,9339151	1,9204068	1,8787948	1,8770617	1,9585887	1,9468938	1,8898757	1,9503729
Firefly	2,4543823	1,8583682	1,7585048	1,7324033	1,7437294	1,6587727	1,6796546			

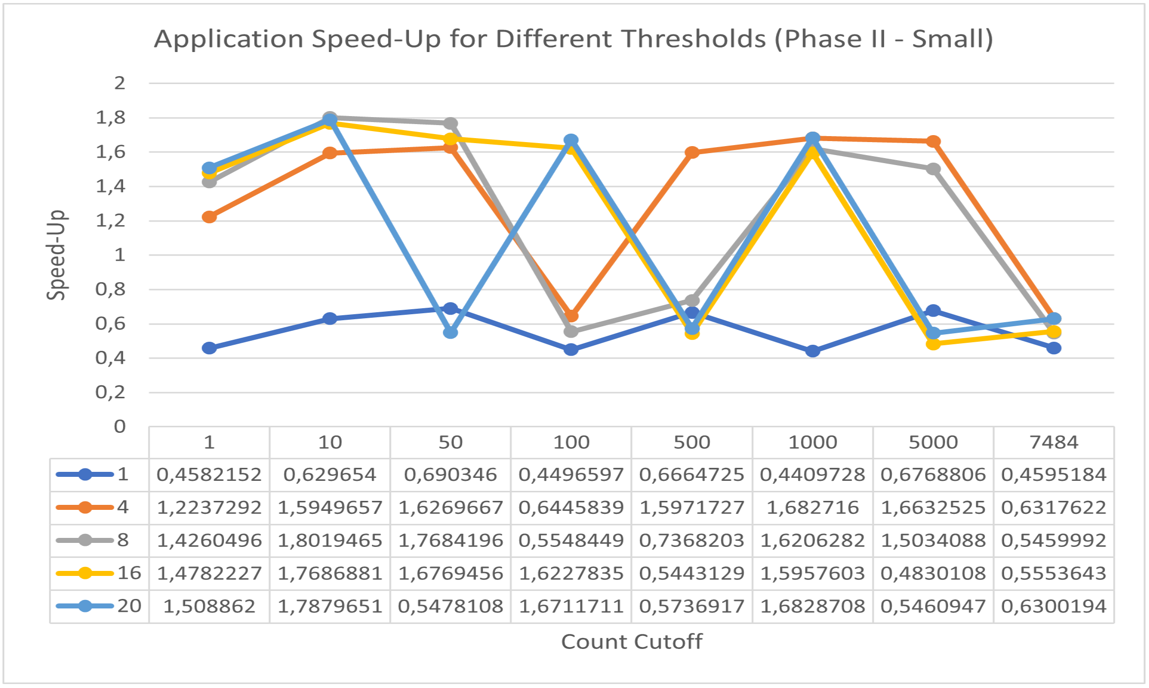
(b) Overheads for for the small, medium and large dataset with T $= 1$

Figure 5: Overheads for all datasets and different thresholds



	1	2	3	4	5	6	7	8	9
1	0,406003	0,468072	0,447732	0,455245	0,464559	0,445023	0,700787	0,449036	0,462619
4	1,564505	1,449524	1,604594	1,292165	0,979459	0,994682	1,007542	0,99304	0,604554
8	1,550476	1,375408	1,418016	1,22552	0,978815	0,997369	1,004453	0,993664	0,587861
16	1,485369	1,303205	1,407748	1,263605	1,005036	1,024018	1,018725	1,017237	0,595036
20	1,565939	1,362855	1,464251	1,287917	0,489254	0,472235	0,442751	0,471179	0,482581

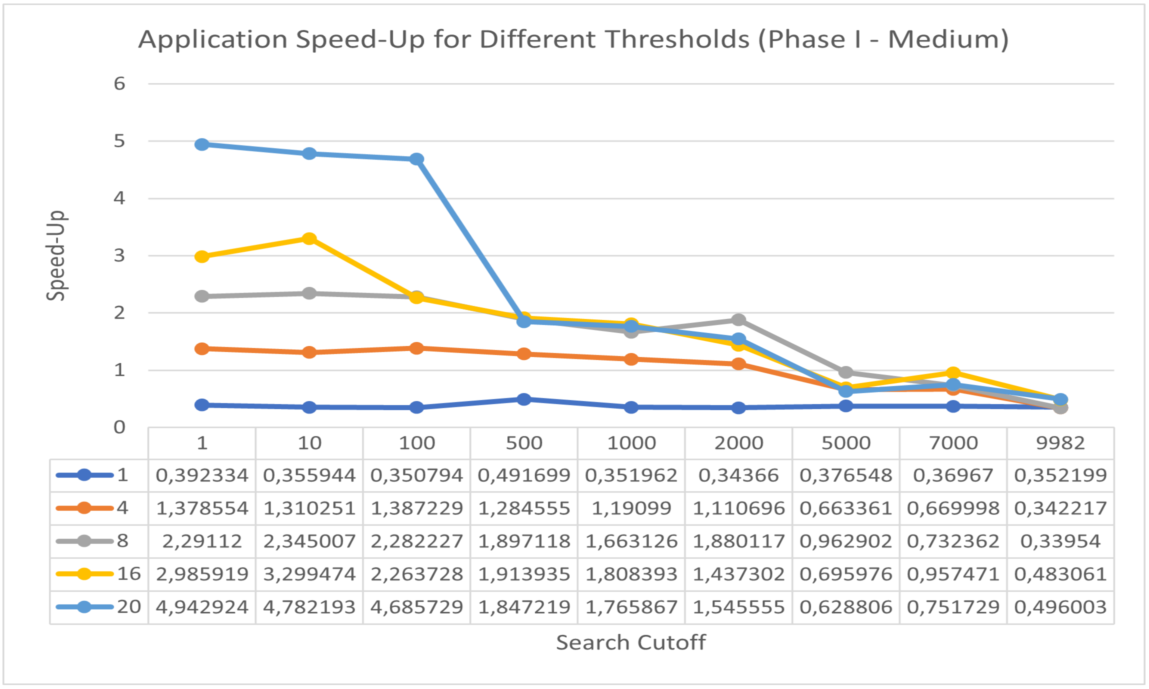
(a) Application speed-ups for the small dataset with T2 = $+\infty$



	1	10	50	100	500	1000	5000	7484
1	0,4582152	0,629654	0,690346	0,4496597	0,6664725	0,4409728	0,6768806	0,4595184
4	1,2237292	1,5949657	1,6269667	0,6445839	1,5971727	1,682716	1,6632525	0,6317622
8	1,4260496	1,8019465	1,7684196	0,5548449	0,7368203	1,6206282	1,5034088	0,5459992
16	1,4782227	1,7686881	1,6769456	1,6227835	0,5443129	1,5957603	0,4830108	0,5553643
20	1,508862	1,7879651	0,5478108	1,6711711	0,5736917	1,6828708	0,5460947	0,6300194

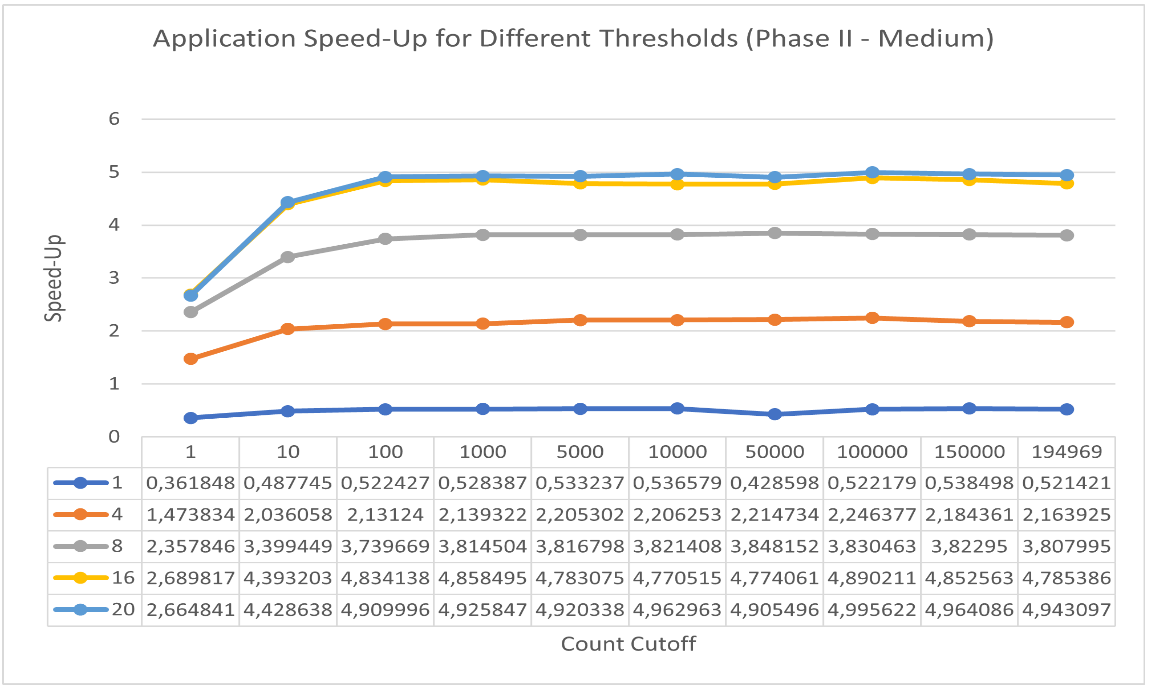
(b) Application speed-ups for the small dataset with T = 1

Figure 6: Application speed-ups for small dataset and different thresholds



	1	10	100	500	1000	2000	5000	7000	9982
1	0,392334	0,355944	0,350794	0,491699	0,351962	0,34366	0,376548	0,36967	0,352199
4	1,378554	1,310251	1,387229	1,284555	1,19099	1,110696	0,663361	0,669998	0,342217
8	2,29112	2,345007	2,282227	1,897118	1,663126	1,880117	0,962902	0,732362	0,33954
16	2,985919	3,299474	2,263728	1,913935	1,808393	1,437302	0,695976	0,957471	0,483061
20	4,942924	4,782193	4,685729	1,847219	1,765867	1,545555	0,628806	0,751729	0,496003

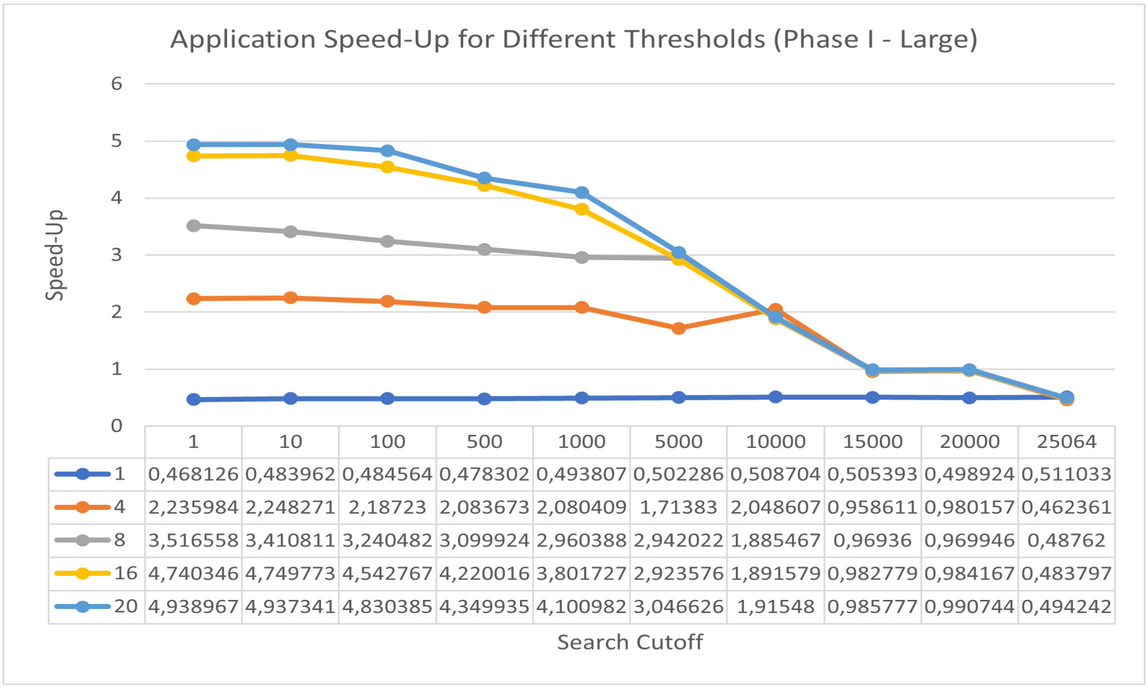
(a) Application speed-ups for the medium dataset with T2 = $+\infty$



	1	10	100	1000	5000	10000	50000	100000	150000	194969
1	0,361848	0,487745	0,522427	0,528387	0,533237	0,536579	0,428598	0,522179	0,538498	0,521421
4	1,473834	2,036058	2,13124	2,139322	2,205302	2,206253	2,214734	2,246377	2,184361	2,163925
8	2,357846	3,399449	3,739669	3,814504	3,816798	3,821408	3,848152	3,830463	3,82295	3,807995
16	2,689817	4,393203	4,834138	4,858495	4,783075	4,770515	4,774061	4,890211	4,852563	4,785386
20	2,664841	4,428638	4,909996	4,925847	4,920338	4,962963	4,905496	4,995622	4,964086	4,943097

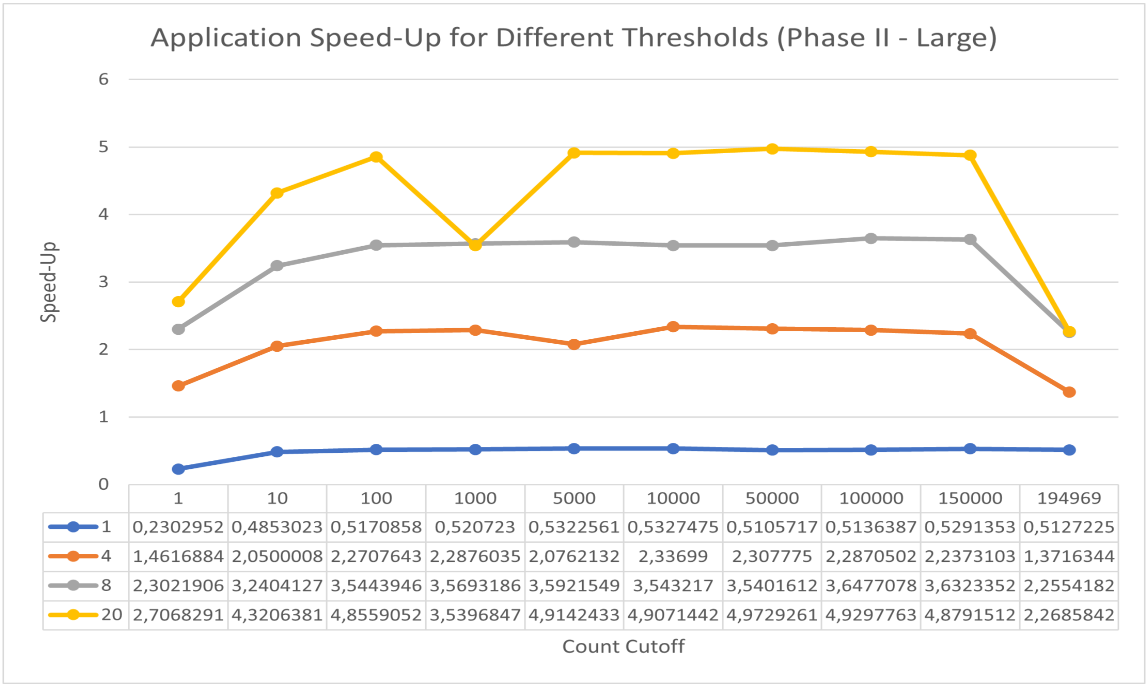
(b) Application speed-ups for the medium dataset with T = 1

Figure 7: Application speed-ups for medium dataset and different thresholds



	1	10	100	500	1000	5000	10000	15000	20000	25064
1	0,468126	0,483962	0,484564	0,478302	0,493807	0,502286	0,508704	0,505393	0,498924	0,511033
4	2,235984	2,248271	2,18723	2,083673	2,080409	1,71383	2,048607	0,958611	0,980157	0,462361
8	3,516558	3,410811	3,240482	3,099924	2,960388	2,942022	1,885467	0,96936	0,969946	0,48762
16	4,740346	4,749773	4,542767	4,220016	3,801727	2,923576	1,891579	0,982779	0,984167	0,483797
20	4,938967	4,937341	4,830385	4,349935	4,100982	3,046626	1,91548	0,985777	0,990744	0,494242

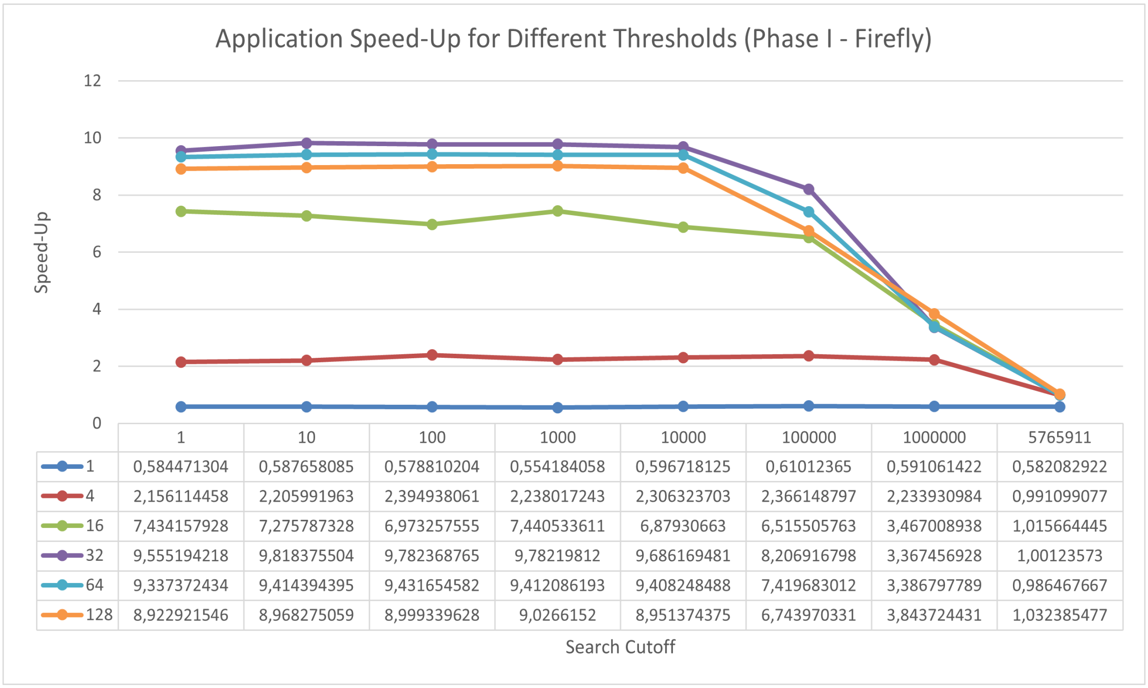
(a) Application speed-ups for the large dataset with T2 = $+\infty$



	1	10	100	1000	5000	10000	50000	100000	150000	194969
1	0,2302952	0,4853023	0,5170858	0,520723	0,5322561	0,5327475	0,5105717	0,5136387	0,5291353	0,5127225
4	1,4616884	2,0500008	2,2707643	2,2876035	2,0762132	2,33699	2,307775	2,2870502	2,2373103	1,3716344
8	2,3021906	3,2404127	3,5443946	3,5693186	3,5921549	3,543217	3,5401612	3,6477078	3,6323352	2,2554182
20	2,7068291	4,3206381	4,8559052	3,5396847	4,9142433	4,9071442	4,9729261	4,9297763	4,8791512	2,2685842

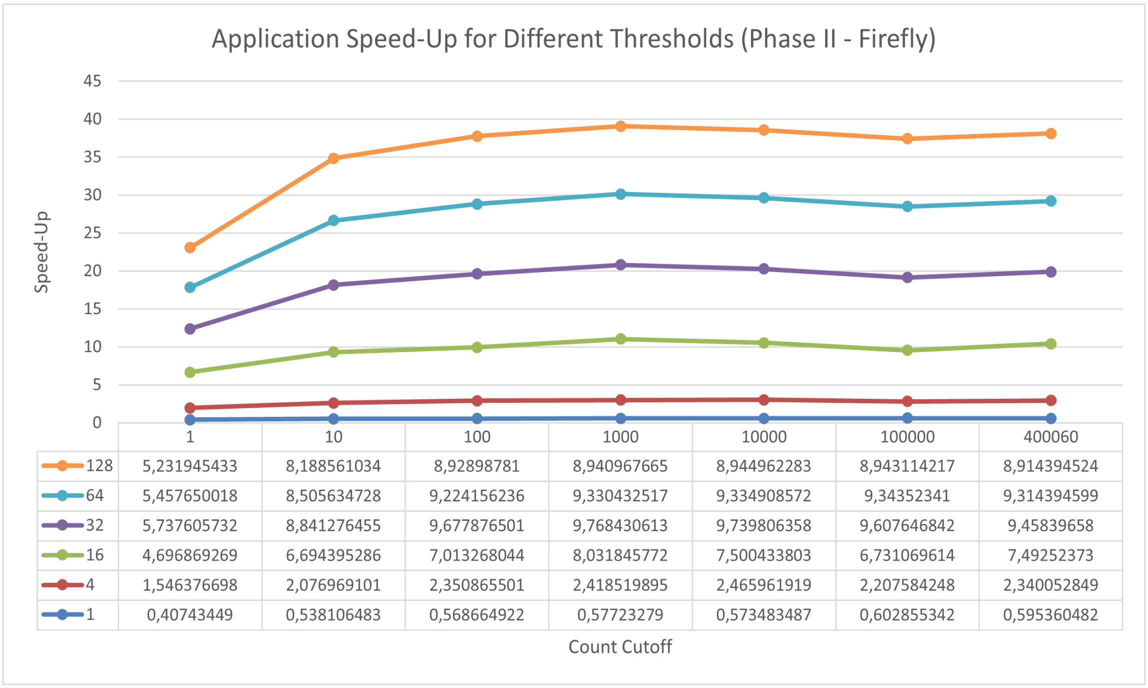
(b) Application speed-ups for the large dataset with T = 1

Figure 8: Application speed-ups for large dataset and different thresholds



	1	10	100	1000	10000	100000	1000000	5765911
1	0,584471304	0,587658085	0,578810204	0,554184058	0,596718125	0,61012365	0,591061422	0,582082922
4	2,156114458	2,205991963	2,394938061	2,238017243	2,306323703	2,366148797	2,233930984	0,991099077
16	7,434157928	7,275787328	6,973257555	7,440533611	6,87930663	6,515505763	3,467008938	1,015664445
32	9,555194218	9,818375504	9,782368765	9,78219812	9,686169481	8,206916798	3,367456928	1,00123573
64	9,337372434	9,414394395	9,431654582	9,412086193	9,408248488	7,419683012	3,386797789	0,986467667
128	8,922921546	8,968275059	8,999339628	9,0266152	8,951374375	6,743970331	3,843724431	1,032385477

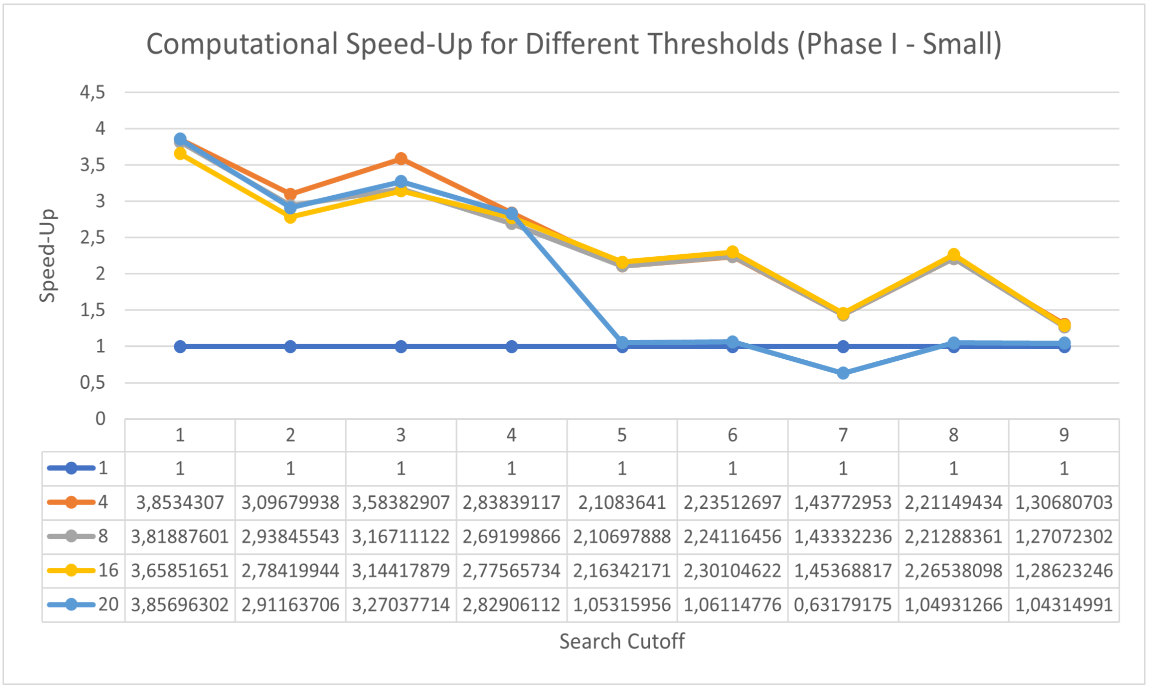
(a) Application speed-ups for the firefly dataset with T2 = $+\infty$



	1	10	100	1000	10000	100000	400060
128	5,231945433	8,188561034	8,92898781	8,940967665	8,944962283	8,943114217	8,914394524
64	5,457650018	8,505634728	9,224156236	9,330432517	9,334908572	9,34352341	9,314394599
32	5,737605732	8,841276455	9,677876501	9,768430613	9,739806358	9,607646842	9,45839658
16	4,696869269	6,694395286	7,013268044	8,031845772	7,500433803	6,731069614	7,49252373
4	1,546376698	2,076969101	2,350865501	2,418519895	2,465961919	2,207584248	2,340052849
1	0,40743449	0,538106483	0,568664922	0,57723279	0,573483487	0,602855342	0,595360482

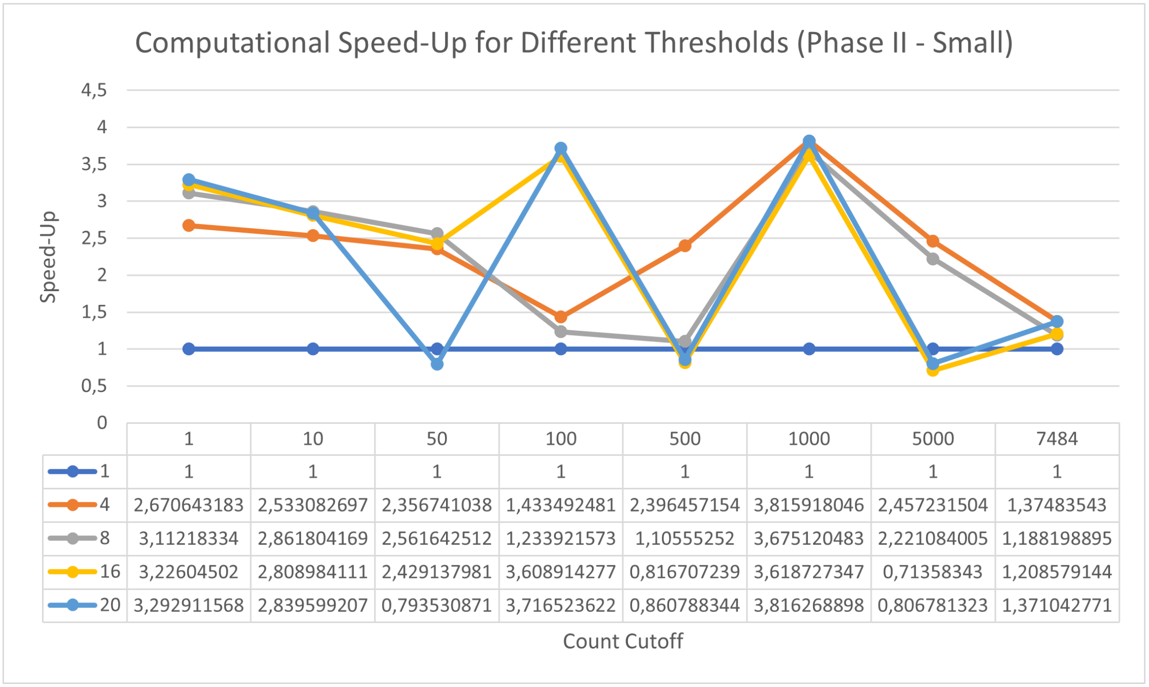
(b) Application speed-ups for the firefly dataset with T = 1

Figure 9: Application speed-ups for firefly dataset and different thresholds



	1	2	3	4	5	6	7	8	9
● 1	1	1	1	1	1	1	1	1	1
● 4	3,8534307	3,09679938	3,58382907	2,83839117	2,1083641	2,23512697	1,43772953	2,21149434	1,30680703
● 8	3,81887601	2,93845543	3,16711122	2,69199866	2,10697888	2,24116456	1,43332236	2,21288361	1,27072302
● 16	3,65851651	2,78419944	3,14417879	2,77565734	2,16342171	2,30104622	1,45368817	2,26538098	1,28623246
● 20	3,85696302	2,91163706	3,27037714	2,82906112	1,05315956	1,06114776	0,63179175	1,04931266	1,04314991

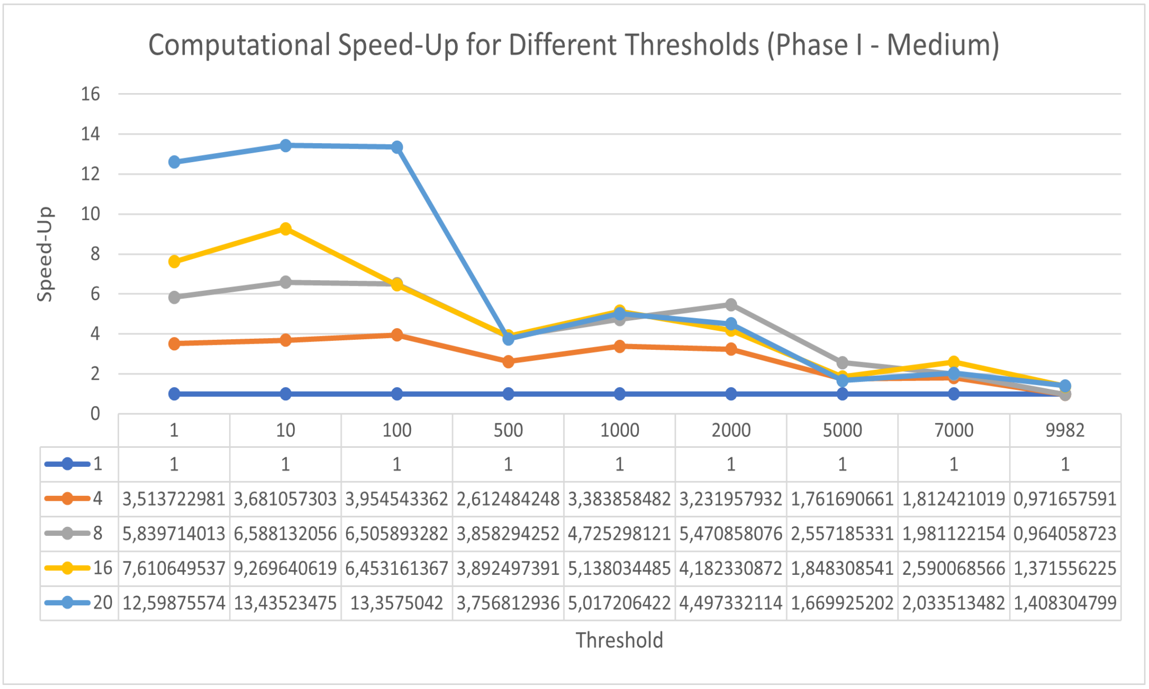
(a) Computational speed-ups for the small dataset with T2 = $+\infty$



	1	10	50	100	500	1000	5000	7484
● 1	1	1	1	1	1	1	1	1
● 4	2,670643183	2,533082697	2,356741038	1,433492481	2,396457154	3,815918046	2,457231504	1,37483543
● 8	3,11218334	2,861804169	2,561642512	1,233921573	1,10555252	3,675120483	2,221084005	1,188198895
● 16	3,22604502	2,808984111	2,429137981	3,608914277	0,816707239	3,618727347	0,71358343	1,208579144
● 20	3,292911568	2,839599207	0,793530871	3,716523622	0,860788344	3,816268898	0,806781323	1,371042771

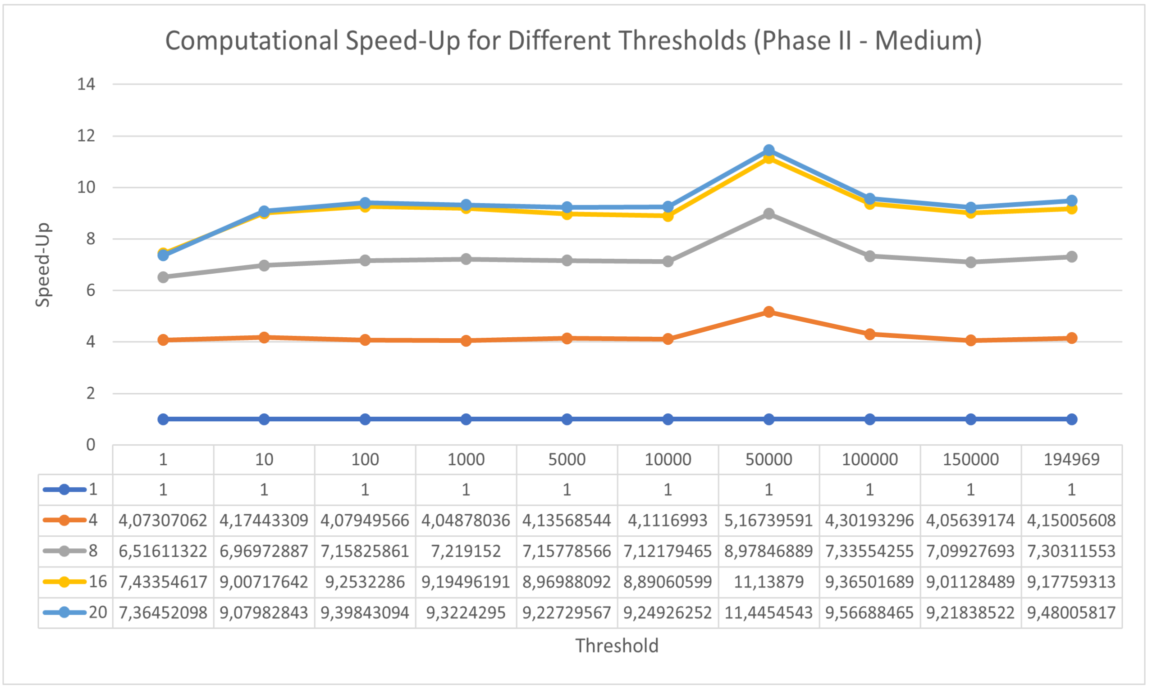
(b) Computational speed-ups for the small dataset with T = 1

Figure 10: Computational speed-ups for small dataset and different thresholds



	1	10	100	500	1000	2000	5000	7000	9982
● 1	1	1	1	1	1	1	1	1	1
● 4	3,513722981	3,681057303	3,954543362	2,612484248	3,383858482	3,231957932	1,761690661	1,812421019	0,971657591
● 8	5,839714013	6,588132056	6,505893282	3,858294252	4,725298121	5,470858076	2,557185331	1,981122154	0,964058723
● 16	7,610649537	9,269640619	6,453161367	3,892497391	5,138034485	4,182330872	1,848308541	2,590068566	1,371556225
● 20	12,59875574	13,43523475	13,3575042	3,756812936	5,017206422	4,497332114	1,669925202	2,033513482	1,408304799

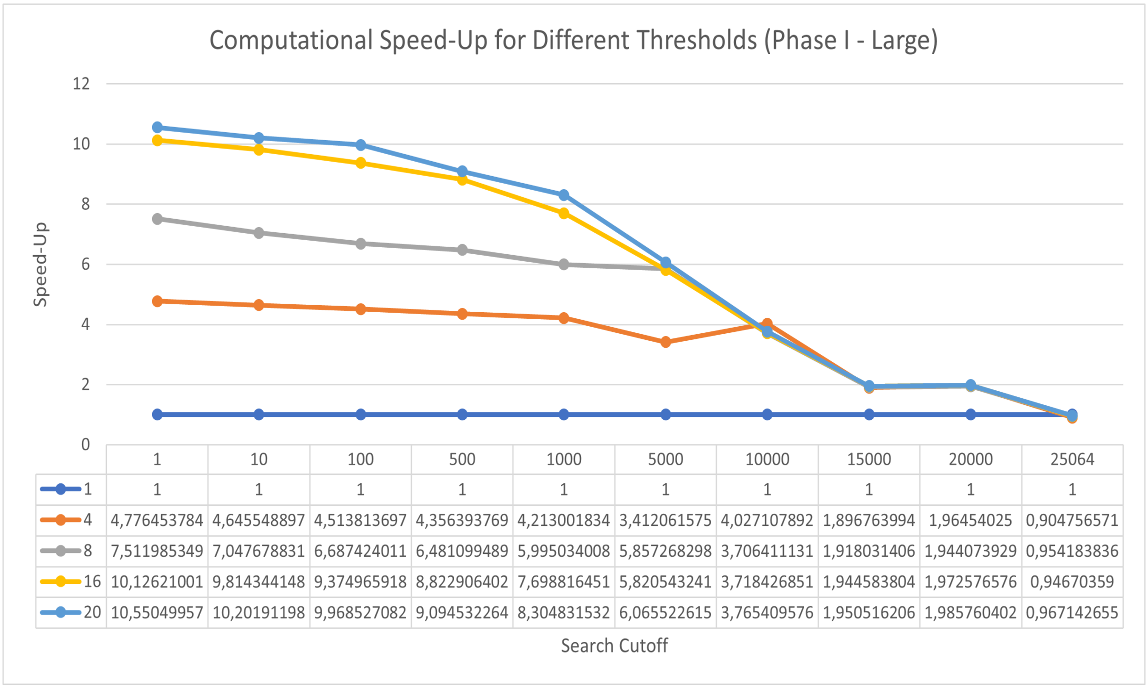
(a) Computational speed-ups for the medium dataset with T2 = +∞



	1	10	100	1000	5000	10000	50000	100000	150000	194969
● 1	1	1	1	1	1	1	1	1	1	1
● 4	4,07307062	4,17443309	4,07949566	4,04878036	4,13568544	4,1116993	5,16739591	4,30193296	4,05639174	4,15005608
● 8	6,51611322	6,96972887	7,15825861	7,219152	7,15778566	7,12179465	8,97846889	7,33554255	7,09927693	7,30311553
● 16	7,43354617	9,00717642	9,2532286	9,19496191	8,96988092	8,89060599	11,13879	9,36501689	9,01128489	9,17759313
● 20	7,36452098	9,07982843	9,39843094	9,3224295	9,22729567	9,24926252	11,4454543	9,56688465	9,21838522	9,48005817

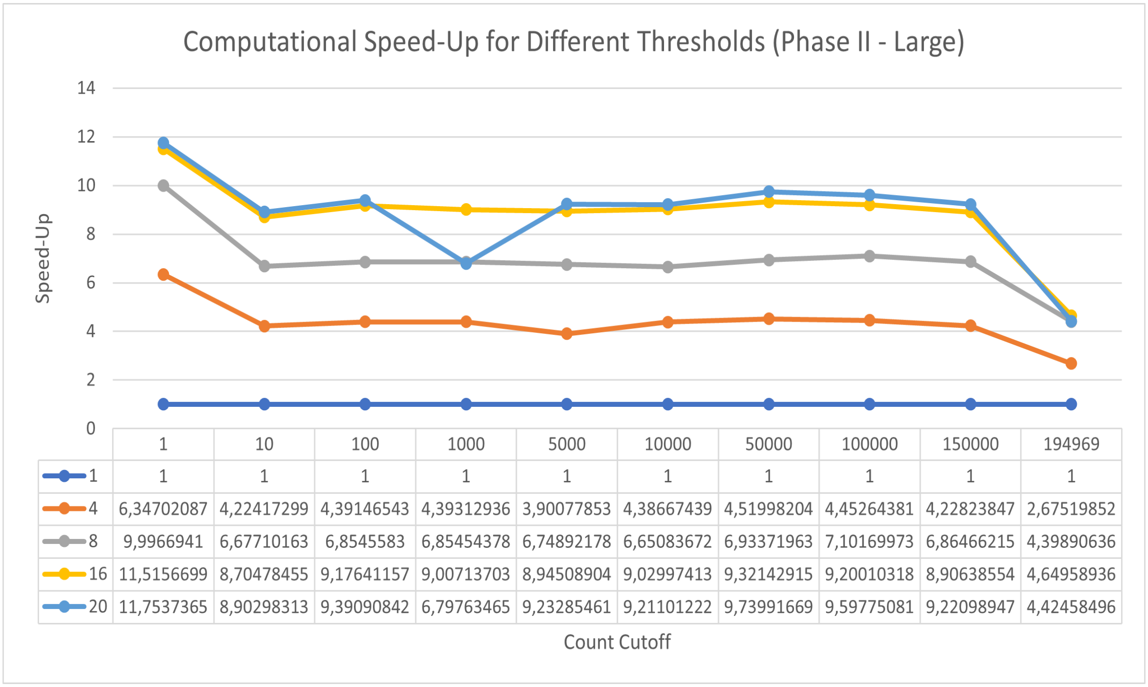
(b) Computational speed-ups for the medium dataset with T = 1

Figure 11: Computational speed-ups for medium dataset and different thresholds



	1	10	100	500	1000	5000	10000	15000	20000	25064
● 1	1	1	1	1	1	1	1	1	1	1
● 4	4,776453784	4,645548897	4,513813697	4,356393769	4,213001834	3,412061575	4,027107892	1,896763994	1,96454025	0,904756571
● 8	7,511985349	7,047678831	6,687424011	6,481099489	5,995034008	5,857268298	3,706411131	1,918031406	1,944073929	0,954183836
● 16	10,12621001	9,814344148	9,374965918	8,822906402	7,698816451	5,820543241	3,718426851	1,944583804	1,972576576	0,94670359
● 20	10,55049957	10,20191198	9,968527082	9,094532264	8,304831532	6,065522615	3,765409576	1,950516206	1,985760402	0,967142655

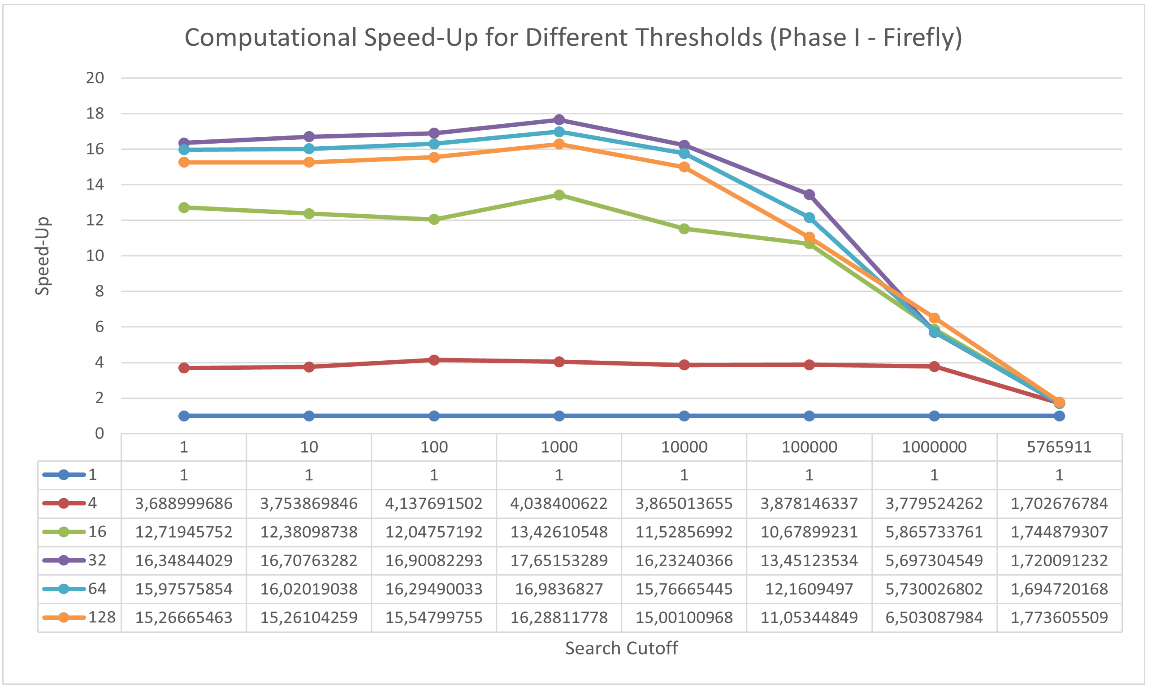
(a) Computational speed-ups for the large dataset with T2 $= +\infty$



	1	10	100	1000	5000	10000	50000	100000	150000	194969
● 1	1	1	1	1	1	1	1	1	1	1
● 4	6,34702087	4,22417299	4,39146543	4,39312936	3,90077853	4,38667439	4,51998204	4,45264381	4,22823847	2,67519852
● 8	9,9966941	6,67710163	6,8545583	6,85454378	6,74892178	6,65083672	6,93371963	7,10169973	6,86466215	4,39890636
● 16	11,5156699	8,70478455	9,17641157	9,00713703	8,94508904	9,02997413	9,32142915	9,20010318	8,90638554	4,64958936
● 20	11,7537365	8,90298313	9,39090842	6,79763465	9,23285461	9,21101222	9,73991669	9,59775081	9,22098947	4,42458496

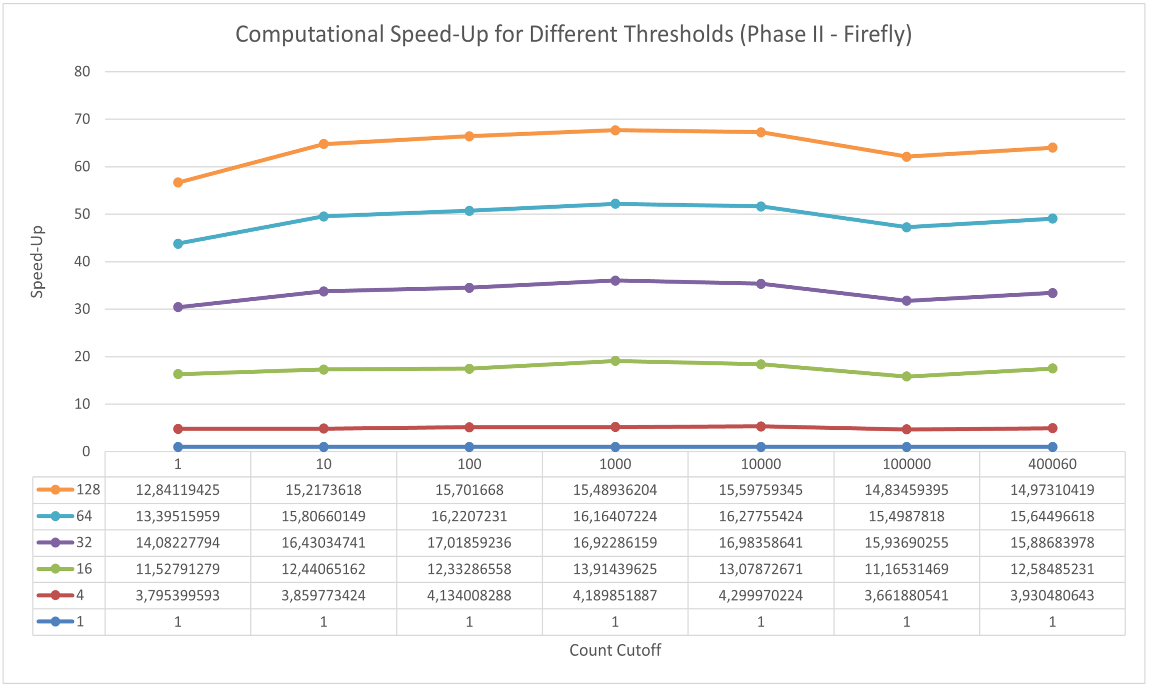
(b) Computational speed-ups for the large dataset with T $= 1$

Figure 12: Computational speed-ups for large dataset and different thresholds



	1	10	100	1000	10000	100000	1000000	5765911
1	1	1	1	1	1	1	1	1
4	3,688999686	3,753869846	4,137691502	4,038400622	3,865013655	3,878146337	3,779524262	1,702676784
16	12,71945752	12,38098738	12,04757192	13,42610548	11,52856992	10,67899231	5,865733761	1,744879307
32	16,34844029	16,70763282	16,90082293	17,65153289	16,23240366	13,45123534	5,697304549	1,720091232
64	15,97575854	16,02019038	16,29490033	16,9836827	15,76665445	12,1609497	5,730026802	1,694720168
128	15,26665463	15,26104259	15,54799755	16,28811778	15,00100968	11,05344849	6,503087984	1,773605509

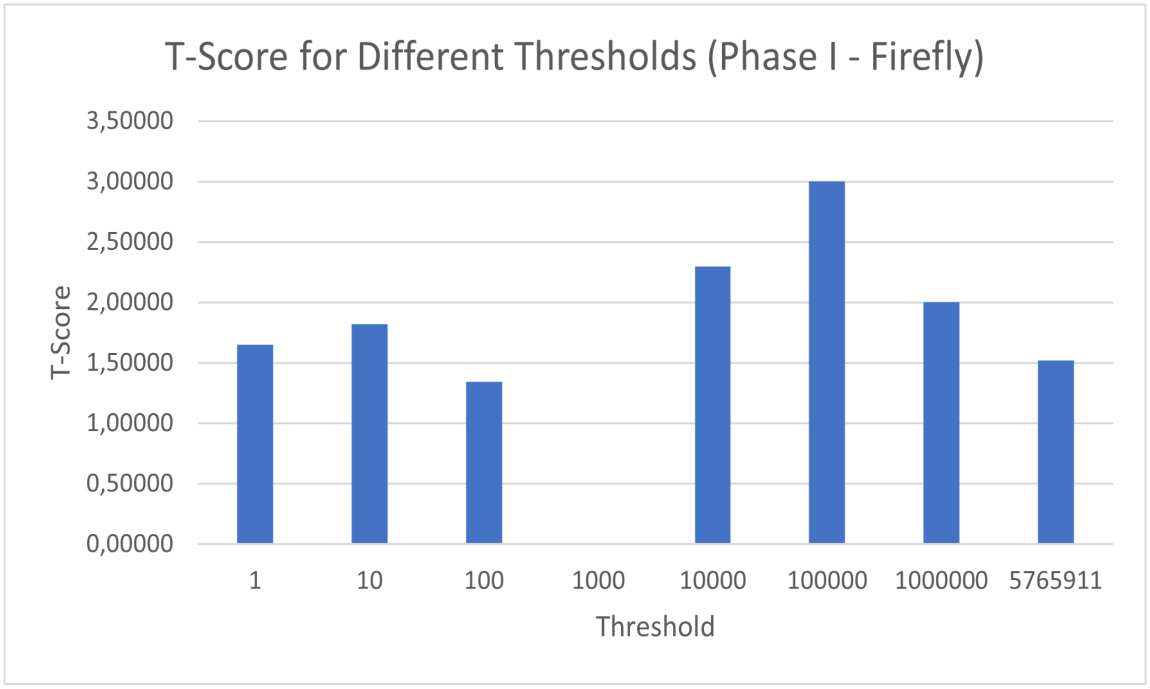
(a) Computational speed-ups for the firelfy dataset with T2 $= +\infty$



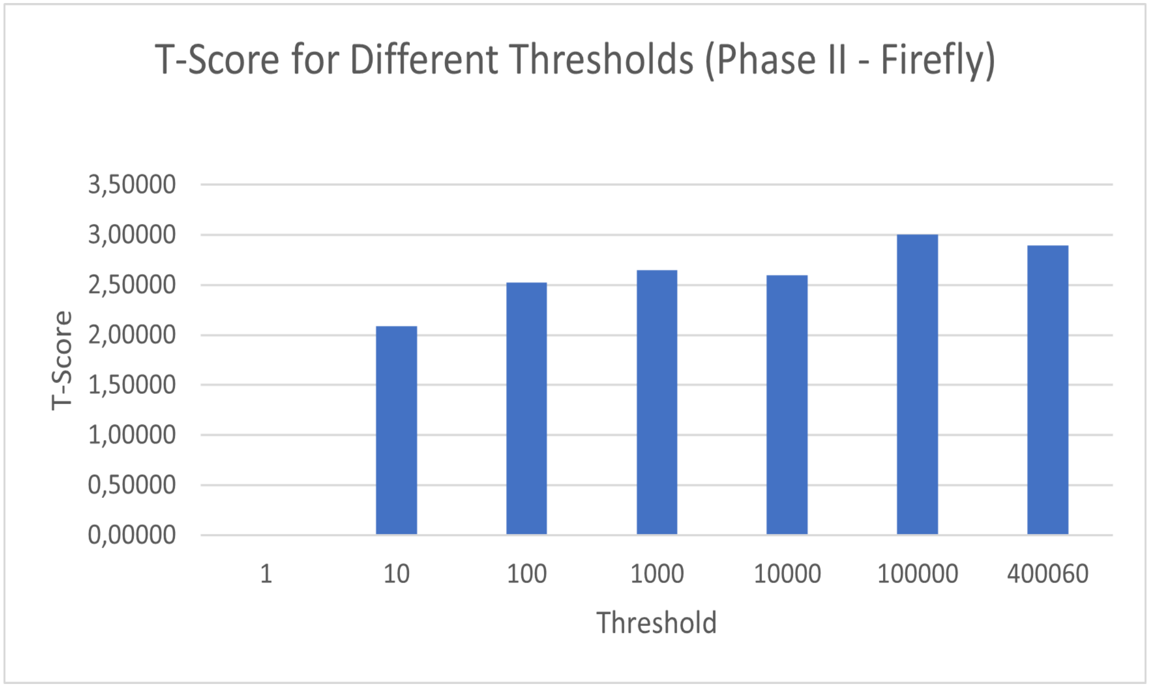
	1	10	100	1000	10000	100000	400060
128	12,84119425	15,2173618	15,701668	15,48936204	15,59759345	14,83459395	14,97310419
64	13,39515959	15,80660149	16,2207231	16,16407224	16,27755424	15,4987818	15,64496618
32	14,08227794	16,43034741	17,01859236	16,92286159	16,98358641	15,93690255	15,88683978
16	11,52791279	12,44065162	12,33286558	13,91439625	13,07872671	11,16531469	12,58485231
4	3,795399593	3,859773424	4,134008288	4,189851887	4,299970224	3,661880541	3,930480643
1	1	1	1	1	1	1	1

(b) Computational speed-ups for the firelfy dataset with T $= 1$

Figure 13: Computational speed-ups for firelfy dataset and different thresholds



(a) T-Scores for the firelfy dataset with T2 = $+\infty$



(b) T-Scores for the firelfy dataset with T = 1

Figure 14: T-Scores for firelfy dataset and different thresholds