TU Delft

Delft University of Technology

Self-adaptive Executors for Big Data Processing

Omranian Khorasani, Sobhan; Rellermeyer, Jan S.; Epema, Dick

DOI
10.1145/3361525.3361545

Publication date
2019

Document Version
Final published version

Published in
Middleware 2019 - Proceedings of the 2019 20th International Middleware Conference

Citation (APA)
Omranian Khorasani, S., Rellermeyer, J. S., & Epema, D. (2019). Self-adaptive Executors for Big Data Processing. In *Middleware 2019 - Proceedings of the 2019 20th International Middleware Conference: Proceedings of the 20th International Middleware Conference* (pp. 176-188). (Middleware 2019 - Proceedings of the 2019 20th International Middleware Conference). Association for Computing Machinery (ACM). https://doi.org/10.1145/3361525.3361545

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' – Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.

Self-adaptive Executors for Big Data Processing

Sobhan Omranian Khorasani
Delft University of Technology
S.OmranianKhorasani@tudelft.nl

Jan S. Rellermeyer
Delft University of Technology
J.S.Rellermeyer@tudelft.nl

Dick Epema
Delft University of Technology
D.H.J.Epema@tudelft.nl

Abstract

The demand for additional performance due to the rapid increase in the size and importance of data-intensive applications has considerably elevated the complexity of computer architecture. In response, systems offer pre-determined behaviors based on heuristics and then expose a large number of configuration parameters for operators to adjust them to their particular infrastructure. Unfortunately, in practice this leads to a substantial manual tuning effort. In this work, we focus on one of the most impactful tuning decisions in big data systems: the number of executor threads. We first show the impact of I/O contention on the runtime of workloads and a simple static solution to reduce the number of threads for I/O-bound phases. We then present a more elaborate solution in the form of self-adaptive executors which are able to continuously monitor the underlying system resources and detect contentions. This enables the executors to tune their thread pool size dynamically at runtime in order to achieve the best performance. Our experimental results show that being adaptive can significantly reduce the execution time especially in I/O intensive applications such as Terasort and PageRank which see a 34% and 54% reduction in runtime.

CCS Concepts • **Software and its engineering → Multithreading**; **Software performance**.

Keywords Self-Adaptive Executors, Big Data, Apache Spark

ACM Reference Format:
Sobhan Omranian Khorasani, Jan S. Rellermeyer, and Dick Epema. 2019. Self-adaptive Executors for Big Data Processing. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361545

1 Introduction

Growing problem sizes and an increasing appetite for incorporating big data analytics into decision processes has pushed the demand for additional performance out of the computer hardware. With the looming sunset of Moore's Law [11], architects are no longer able to deliver performance improvements in a completely transparent way by increasing the clock frequency. Instead, they were forced to add new features such as multiple compute cores per CPU (which requires explicit parallelization efforts to leverage) and on-chip memory controllers (which leads to non-uniform memory access). This trend, however, has not only increased the complexity of computer systems but also made the landscape more heterogeneous. System software has responded by providing a default behavior based on heuristics or empirical evidence, and then exposing a large and increasing number of tuning knobs for operators to adjust these implicit assumptions and tailor the system to the concrete hardware.

In the latest version of Apache Spark (2.4.2), there are a total of 117 functional parameters (summarized in Table 1), the majority of them directly affecting the performance of the system [6]. Trying to find the suitable configuration parameters is not a trivial task for users since it requires a deep understanding of both the hardware and software, and how they interact. We are not the first to realize this problem and several remedies have been proposed ranging from using local search techniques [7] to online services that use artificial intelligence for determining optimal configurations [21, 26].

In this work, we focus on providing a thorough solution to one configuration aspect of particular importance for the resulting performance: *the problem of threading*. The state of the art in big data processing for the parallel portion of the execution is to probe the number of physical CPU cores as the default and instantiate a thread pool of equivalent size. Indeed, this setting is typically tunable through a launch-time configuration parameter. However, the implicit underlying assumption behind this design is that big data processing is primarily and uniformly CPU-bound, an assumption that we refute by experimental evidence in Section 3. Instead, our results show that different phases of big data workloads experience different limitations and therefore warrant a more differentiated approach to threading, beyond the scope of a single static configuration parameter. We present a drop-in replacement for the Spark Executor that is able to adjust the number of threads based on two sources of reflection. First, we infer structural properties of the workload to identify phases that are likely not CPU-bound (in our concrete example I/O bound instead). In Section 4 we present an initial model that distinguishes between generic, probably CPU-bound stages of the pipeline and likely I/O-bound

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware '19, December 8–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361545

Category	#Parameters
Shuffle	19
Compression and Serialization	16
Memory Management	14
Execution Behavior	14
Network	13
Scheduling	32
Dynamic Allocation	9
Total	117

Table 1. Number of functional parameters in Spark

stages. With this simple approach, we are able to improve the runtime of workloads like Terasort by almost 40% without changing the program. In the second step (Section 5), we develop a controller that dynamically adjusts the number of threads, to avoid contention, by monitoring I/O wait time and throughput. As we show in the evaluation (Section 6), this solution is able to perform more fine-grained dynamic adaptations and, for problems like PageRank where the static solution only delivers marginal improvements, reduces the execution time by more than 50%.

2 Problem Statement and Related Work

The ongoing trend towards higher CPU core counts with only marginal improvement in clock speed for new processor generations has increased the pressure on users to effectively parallelize and schedule their workloads. Fortunately, commonly used Big Data programming models and frameworks take care of the parallelization into tasks and scheduling them to machines in the computer cluster. The question, however, remains how to adjust the number of threads per computer to achieve the best system utilization and workload throughput.

The different means to enable concurrency have been a long and contentious topic in the design of large-scale and performance-critical systems. Before the multi-processor and multi-core era, much of the discussion revolved around the question of either using multiple concurrent threads [17], typically in a timesharing setup [15], or handling all processing in a single, continuously running thread [4]. While the second model avoids costly context switches and scheduler overhead, it requires the application to actively manage the different states associated with the concurrent workflows and essentially perform soft context switches itself.

As soon as the hardware supports true parallelism, the single-threaded design is no longer a favorable option except for very specific designs (e.g., the main event loop in Node.js [20]) that prioritize scalability and low latency over throughput. Instead, multi-threading is now the dominant approach in modern systems. Servers (or services) formally implement a producer-consumer pattern with the server socket *producing* work by accepting incoming connections worker threads *consuming* the work by performing the necessary operations to serve the request. However, having an unbounded number of threads (e.g., by instantiating a new thread per incoming connection) is not an option in view of limited compute resources since this leads to an unbound wait queue and therefore unbounded latency. As a result, systems typically instantiate an either fixed-sized or limited-sized pool of worker threads that continuously handle tasks. Re-using the same threads through a pool eliminates the static cost of repeatedly allocating new threads.

Finding the right thread pool size to maximize performance is an involved task. In the idealized case of entirely CPU-bound tasks and no further processes computing for CPU time, choosing as many threads as there are hardware execution contexts available is the logical choice. However, as soon as the workload involves I/O operations, threads can no longer utilize their entire time slice and the wait time for the I/O operation to complete leads to an underutilization of the CPU. In this case, common knowledge suggests to use more threads than physical cores, with concrete numbers ranging up to twice the core count or more (e.g., for GNU Make [19, 22]). Using too many threads, however, can also hurt the performance by creating I/O contention [18]. The optimal number of threads that saturates but not overwhelms the I/O subsystem highly depends on the workload and even the environment (e.g., concurrently running processes that compete for resources like CPU, cache lines, memory, network, or storage) and in practice needs to be determined through a tedious, manual, experimentation-driven process.

The problem is severe enough in practice that some authors demanded to take the decision of threading out of the hands of programmers (or operators if the choice is exposed as a configuration parameter) and centralizing the authority over concurrency within the operating system. Von Behren at al. were among the first to criticize the virtual processor model of threading [24]. With Cappricio [25], the authors presented a user-level threading library that features a central resource-aware scheduler that is able to adapt the number of threads to the global system utilization.

Apple introduced a task-based abstraction for concurrency with OS X in Grand Central Dispatch [16]. This central facility allows developers to submit tasks and it maintains a single system-wide thread pool to execute them, thereby relieving the developer or operator from any tuning efforts and ensuring fairness across multiple applications.

For the broader problem of parameter tuning, several systems were proposed to externalize the tuning parameters and allow a central controller to make global decisions as, e.g., in ActiveHarmony [2]. Domain-specific solutions exist for Java enterprise servers [28] or Oracle databases [5]. Karcher and Pankratius presented Pepetuum [9], an automatic system-wide performance tuner for multi-core applications that they embedded into Linux. The system is able to dynamically adjust the number of threads but requires

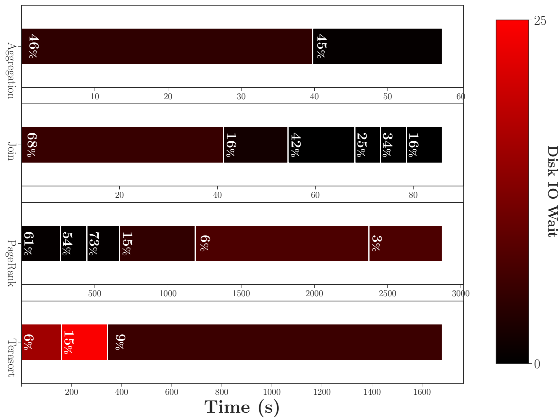


Figure 1. I/O wait and CPU usage of different stages of applications

Application	Input Size	I/O Activity	Diff.
Aggregation	17.87 GiB	37.44 GiB	**+ 109%**
Bayes	3.50 GiB	9.80 GiB	**+ 180%**
Join	17.87 GiB	21.06 GiB	**+ 18%**
LDA	0.63 GiB	3.83 GiB	**+ 508%**
NWeight	0.28 GiB	10.23 GiB	**+ 3553%**
PageRank	18.56 GiB	128.3 GiB	**+ 591%**
Scan	17.87 GiB	112.56 GiB	**+ 530%**
Terasort	111.75 GiB	429.35 GiB	**+ 284%**
SVM	107.29 GiB	203.92 GiB	**+ 90%**

Table 2. I/O activity of Spark applications relative to their input size

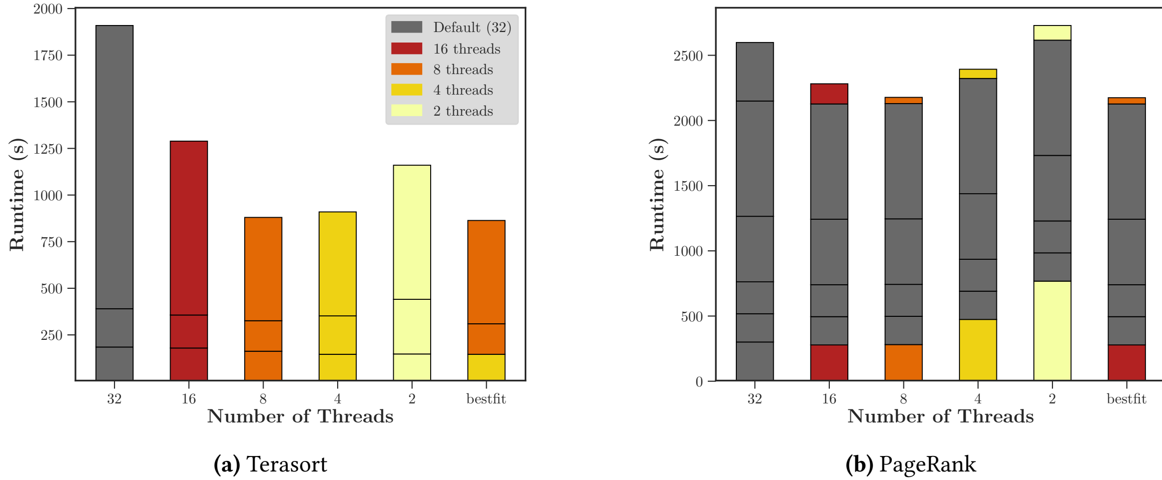
changes to all participating applications in order to expose this tuning parameter and allow for performance measurements. In addition to these white-box approaches, several systems for black or gray-box tuning were proposed. Those systems do not require changes to the application but, as a consequence of not understanding the semantics of tuning parameters, are restricted to more or less elaborate searches within the parameter space. One example in the big-data domain is MRonline [13] for automatic performance tuning of Hadoop.

In the context of big data processing, the subtle interaction between threads and I/O is of particular importance since large amounts of data need to be moved through the system in order to derive insights. Traditional solutions for full or semi-automatic performance tuning have limitations. Offline tuners require multiple full runs of the system in order to approach the momentarily optimal solution. However, since the workload characteristics of big data platforms is primarily determined by the user program, these results are then not applicable to other workloads. Online solutions often fail to react quickly enough to adapt to the different phases within a data processing pipeline. These shortcomings motivated our research into improving the adaptivity of threading inside big data processing frameworks like Apache Spark and leveraging the knowledge of the workload structure, rather than relying on generic external approaches that are agnostic to the workload.

3 Adaptive Executors

The responsible entities for task execution in Spark (Executors) use thread pools whose size is by default the number of available virtual cores based on the implicit assumption that most tasks are primarily CPU-bound. This setting can be explicitly overridden by the operator of the system as application performance can be bound by a different resource such as disk or memory rather than only CPU. For instance, multiple sources report that executors on large simultaneous multithreading (SMT) machines running with more than a certain number of concurrent tasks may lead to poor HDFS I/O throughput [3, 14]. Figure 1 shows the average CPU usage of various applications in every stage of their execution. The `mpstat` command line tool in Linux was used to collect this information on each node and the results were averaged across the cluster. There are two main observations. (1) We can see that almost in all cases the CPU is not fully utilized. For example, in Terasort stages, the CPU usage is 6, 15, and 9% respectively. (2) This also suggests that each stage of execution might be dominated by a different system resource. This would make the (static) decision of determining the number of threads based on a single resource (e.g., CPU) not optimal. Additionally, the color of bars in Figure 1 represent the average percentage of time that CPU has waited for disk I/O. This also suggests that some stages are more I/O-bound than the others which corroborates the need of having a dynamic solution. Similar to many other configuration properties in Spark, the deciding parameter for determining the number of threads cannot be changed once the application has started and must be set in advance. This is problematic since in practice big data applications often have multiple stages of execution, each of which could have distinct characteristics that benefit from different set of parameters. Our adaptive executors address this behavior by differentiating between tasks that are likely to be compute-bound and for which thus the current approach of using as many threads as physical cores is feasible, from those tasks are that are almost certainly not. For the latter class of tasks, we focus on those that are likely to cause a high amount of I/O activity.

The typical pipeline for a big data application consists of an initial phase in which the data is read (i.e., data ingestion), then some transformation are applied to the dataset, and ultimately the final output is written back to the disk. The first generation of frameworks such as Hadoop stored all the intermediate results on the local disk [12]. While in modern in-memory processing systems like Spark the transformations should not directly contribute to I/O activity, there are



(a) Terasort

(b) PageRank

Figure 2. The runtime effect of static solution on Terasort and PageRank

still cases in which they do. For instance, shuffle maps are spilled out to disk for fault-tolerance or to reduce memory pressure. As a result, big data processing frameworks produce a surprisingly high amount of I/O activity relative to their input size. Table 2 shows the disk activity of various applications in Spark. As we can see, with the exception of the join workload, the ratio between input size and intermediate I/O activity ranges between factors of 2x and more than 30x. This further motivates our approach of examining the interplay between thread count and I/O throughput.

4 Static Solution

As the first step toward having flexible parameters for different execution contexts, we modified Spark to identify the I/O intensive stages and employ a user-defined value as the number of threads in those stages. The I/O stages are considered to be the ones that read from or write to the disk regardless of their input/output size. Since all the transformations and actions in Spark happen at the level of RDDs [27], we modified them to let the executors know whether the current stage should be considered as I/O. Transformations such as `textFile()` and actions such as `saveAsTextFile()` and `saveAsHadoopFile()` would all mark the stage as I/O. For instance, the Terasort application consists of three stages, all of which are considered to be I/O intensive since the first two read from the disk and the last one writes the results, whereas in the PageRank application, out of the total 5 stages, only the first and the last stages use I/O operations while the remaining stages primarily shuffle data.

Figure 2 shows how having different numbers of threads for I/O stages affects the runtime of Terasort and PageRank on 4 nodes. The x and y axis show the number of threads and runtime respectively and the stages are separated by the black lines. The non-I/O stages still use the default number of threads (indicated by the gray color). It is clear that it is not always efficient to use all the available cores in the system. Additionally, the results suggest that different stages could benefit from a different number of threads. By treating I/O stages differently, compared to the default version, the static solution is able to reduce the runtime of Terasort and PageRank in the best case (8 threads) by 39.35% and 19.02% respectively. It is important to note that these performance gains do not require any modifications to the workload since the solution is fully transparent to the user program. However, the approach comes with several limitations which is mainly due to having to make static decisions. In Section 5, we present a more elaborate dynamic solution that is capable of eliminating each one of the following limitations while still producing significant performance gains, sometimes outperforming the static solution in practice.

L1: Parameters are fixed for all I/O stages The last bar (i.e, BestFit) in Figure 2 shows the (hypothetical) best combination of threads for each stage. For example, the first stage of Terasort (Figure 2a) exhibits the best performance when the number of threads is set to 4 whereas it is 8 for the other two stages. Similarly, in Figure 2b, PageRank benefits the most from having 16 and 8 threads in the first and last stage respectively. This behavior greatly illustrates one of the limitations of the static approach. Although we are able to change the number of threads for the I/O stages, it is not possible to differentiate between the different I/O stages. In essence, we are not able to reach the optimal "BestFit" performance with the static solution.

L2: Unable to identify every I/O stage Only considering typical I/O operations (e.g., read and write) is not ideal since there might exist other stages that use the disk but do not explicitly use I/O actions. In case of Spark, shuffle stages use the disk for storing intermediate data or any stage could use the disk for spilling the cached data in memory. For example,

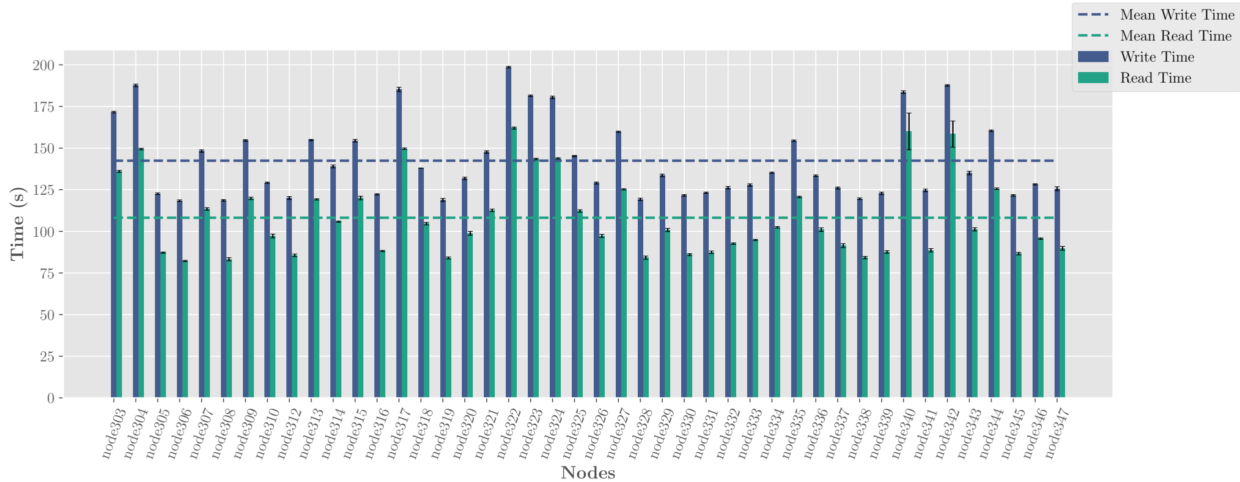
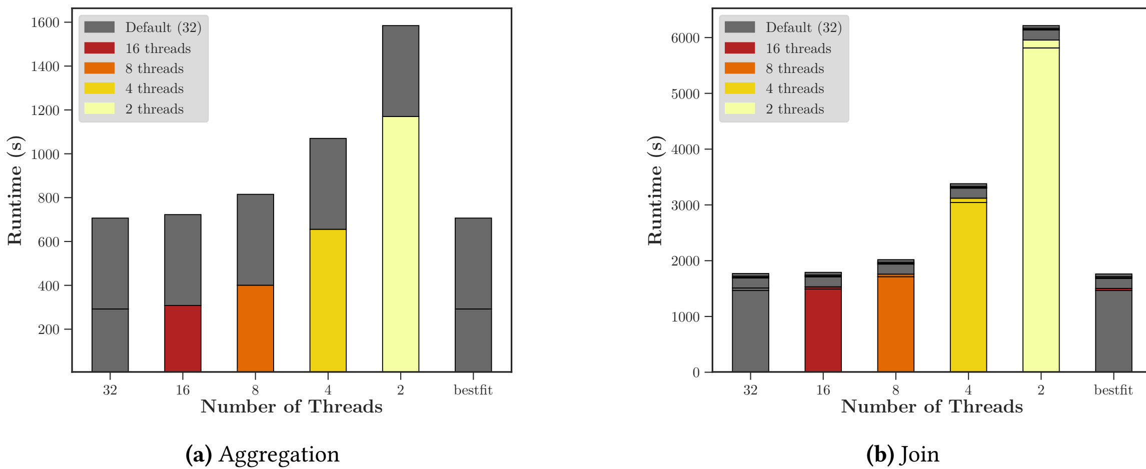


Figure 3. I/O performance variability in the DAS-5 cluster



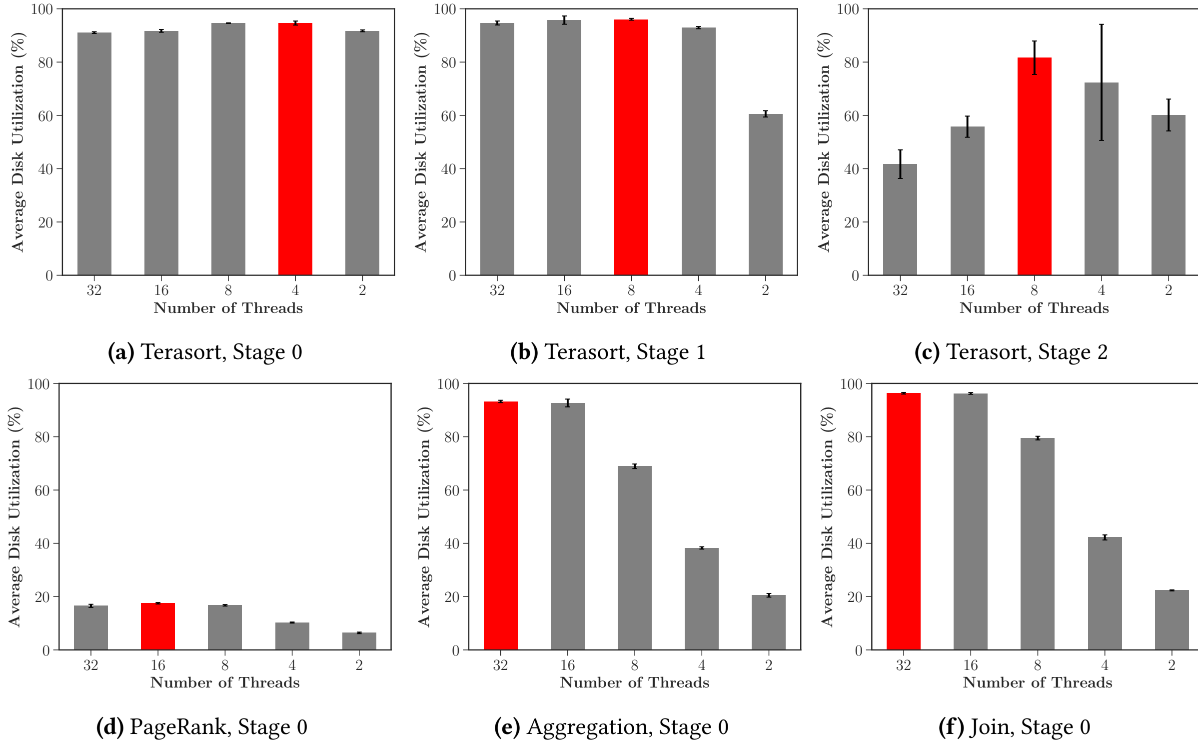
(a) Aggregation

(b) Join

Figure 4. The runtime effect of static solution on SQL applications

the shuffle stages in PageRank (stages 1 to 4), read 65.5GB and write 59.4 GB of data which means they should also be considered for tuning, even though they do not express that they are I/O.

L3: Agnostic against the workload characteristics The third limitation stems from the fact that the static solution does not take the workload characteristics such as input/output size into account. While in the Sort application, both input and output size are the same (e.g., 120GB), there are other applications that read or write only small amounts of data. In that case, there is not enough I/O activity to justify using fewer threads since the maximum throughput of the disk is never reached. Furthermore, the static solution is not always able to decrease the runtime. For example in SQL applications such as Aggregation (Figure 4a) and Join (Figure 4b), even for the I/O stages (i.e., first stage), the default number of threads performs best. The underlying reason can be explained by analyzing the difference in average disk utilization in I/O stages of these applications (Figure 5). For applications which benefit from the static solution such as Terasort (Figure 5a), the average disk utilization in the I/O stages is the highest (indicated in red) when 4, 8, and 8 threads are used respectively which is exactly equal to the output of static BestFit (Figure 2a) and corroborates the reduction in runtime. PageRank (Figure 5d) follows the same pattern in its data ingestion phase (stage 0) by having the highest disk utilization at 16 threads which is also complies with the static BestFit result shown in Figure 2b. However, for Aggregation and Join workloads which do not follow the same pattern, disk utilization in the read stage is significantly lower when fewer threads are used, most probably due to the additional transformations in that stage. In fact the average CPU utilization for the first stage of Join and Aggregation



(a) Terasort, Stage 0 **(b)** Terasort, Stage 1 **(c)** Terasort, Stage 2

(d) PageRank, Stage 0 **(e)** Aggregation, Stage 0 **(f)** Join, Stage 0

Figure 5. Average disk utilization across all nodes in the I/O stage of different applications. The red bar represents the highest average disk utilization

is 68% and 46% respectively whereas it is only 6% for the Terasort. This means that the amount of computation done in I/O stages differ between applications which reveals yet another important limitation for the static solution. Since the performance of I/O stages heavily depends on different properties such as the input size and other transformations in the same stage, it is not always guaranteed that using different number of threads is beneficial. Therefore, any static decision could lead to a sub-optimal performance due to the lack of information about the workload characteristics.

L4: Agnostic against inherent performance variability The next limitation comes from the practical experience in running actual workloads on large computer clusters. Figure 3 shows the difference in reading and writing 30 GB of data in our nation-wide DAS-5 cluster [1]. Although the machines have an identical setup, there is still a significant gap in their actual performance. Having to manually identify these differences and assess their impact could be a very tedious tasks for the users. Even the heterogeneous and dynamic environment in which applications run in (e.g., Cloud) could play an important role since an ideal state at one time is not guaranteed to be the same at another [23].

L5: The solution still requires manual tuning The last and perhaps the most prominent limitation of the static solution is that the users still need to provide the parameter values, which turns into a substantial manual tuning effort. In other words, although the users are now able to select different numbers of threads for the I/O stages, it is still their responsibility to find the most suitable configuration.

Even though the static approach is a step forward from the default behavior of Spark, its limitations hinders us from having a true adaptive solution. In the next section, a dynamic solution is presented which builds upon the static approach and removes its limitations by monitoring the underlying I/O infrastructure and dynamically changing the number of threads to achieve the best performance.

5 Self-adaptive Executors

The aforementioned limitations motivate the effort in seeking an autonomic, self-adaptive solution based on observing the runtime characteristics of the workload instead of putting the burden on the users to find the suitable parameters. It is now the framework's responsibility to dynamically tune its parameters for optimal performance.

The dynamic solution aims to address all the aforementioned limitations of the static approach. Concretely, the first limitation (L1) is addressed by tuning each individual stage of execution which enables the algorithm to find different optimal thread number settings for different stages. Figure 6 shows the decisions by the dynamic solution for different stages of Terasort. We see that the algorithm has decided

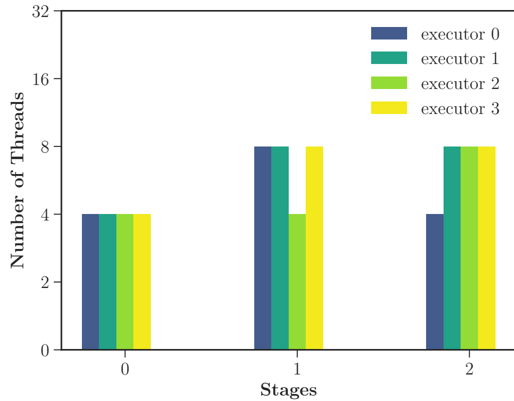


Figure 6. Selecting the thread number by the dynamic solution in different stages of Terasort for every executor

different number of threads for different stages which is the desired behavior. Secondly, by considering every stage, it has a chance to find a better configuration for stages which do not explicitly perform I/O actions such as shuffle stages. (addresses L2). The third limitation (L3) is addressed by monitoring suitable performance metrics which are used by the algorithm to infer whether using a certain number of threads results in a better performance. In this case, if the input/output size or the disk utilization is too low to justify using fewer threads, then the performance metrics would be able to capture this information and let the tuner decide accordingly. Furthermore, by tuning each executor individually in each stage, it can potentially find different settings for different executors based on their capabilities and removes the need for manual intervention by the users (addresses L4). In Figure 6, we see that the algorithm can potentially decide a different setting for each executor in a stage. For example in stage 1, executor 2 uses 4 threads whereas other executors all use 8. This behavior would be most effective when there is a large discrepancy between the (I/O) performance of the machines.

In the realm of self-adaptive systems, having explicit feedback loops is a common trend with the aim of decoupling system management activities from software development cycles. A major breakthrough in making feedback loops explicit came with IBM's autonomic computing initiative with its emphasis on engineering self-managing systems. One of the key findings of this research is the blueprint for building autonomic systems using MAPE-K (monitor-analyze-plan-execute over a knowledge base) feedback loops [10]. The managed element in this case is the thread pool whose performance is *monitored* by various tools (i.e, sensors), *analyzed* based on which a decision is *planned* and ultimately *executed* in order to adjust its size. The feedback behavior of a self-adaptive system which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modeling, design, implementation, validation, and operation.

5.1 [M]onitor

The monitor senses the managed process (e.g., thread pool) and its context, filters the accumulated data, and stores relevant events in the knowledge base for future reference. As we saw in Section 3, the significance of disk I/O in most big data applications which stems from multiple sources such as long data ingestion and writing phases and the persistence of intermediate results makes it a viable metric for monitoring. In order to measure the disk performance, the following metrics are monitored:

1. Epoll wait time (ε): This system call waits for events on a file descriptor.
2. I/O throughput (μ) : The overall read/write throughput of the tasks.

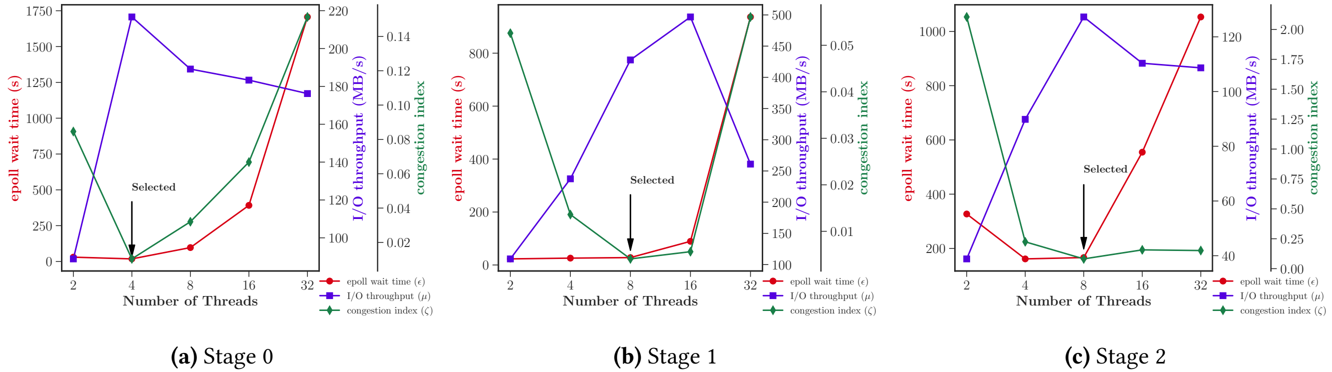
The `strace` tool in Linux is used to measure the epoll wait time for an interval. For the I/O throughput, we use a sampling approach in which each second the throughput of the running tasks (reported by the Spark metric system) is measured and then the total average represents the throughput for a given interval. The first metric represents the time spent waiting for a read or write request to complete and the second indicates the overall read and write (either disk or shuffle data) throughput of the tasks. Since epoll wait and throughput contribute to the I/O performance in different ways, in order to have a single value which incorporates both metrics and represents the amount of I/O congestion, we divided the epoll wait time by the throughput of the tasks and then select the configuration where this value is minimized ($min(\zeta)$):

$$\zeta_j = \frac{\varepsilon_j}{\mu_j} \mid c_{min} \leqslant j \leqslant c_{max} \tag{1}$$

where c_{min} and c_{max} are the minimum and maximum number of threads and ζ_j is the I/O congestion index, ε_j is the accumulated epoll wait time and μ_j is the I/O throughput for the interval in which the number of threads is set to j. An interval (I_j) is finished once j tasks have completed, inside of which the performance of the current number of threads (ζ_j) is monitored. For instance, the interval for 16 threads (I_{16}) starts by setting the thread pool size to 16 and then monitors the performance of 16 concurrent tasks and finally finishes as soon as they are all complete. At the end of each interval, the control is passed to the next component to analyze the gathered data.

5.2 [A]nalyze

The analyzer performs complex data analysis and reasoning on the symptoms provided by the monitoring function. In concrete terms, the analyzer is responsible for finding out which parameter values cause more contention on the



(a) Stage 0

(b) Stage 1

(c) Stage 2

Figure 7. Effect of having different number of threads on epoll wait time and throughput in one of the executors for different stages of Terasort

underlying I/O infrastructure and potentially degrade the performance.

Contention at the level of the disk can have implications on both provided metrics whereby reading or writing beyond the saturation level of a disk increases the epoll wait time and diminishes its throughput. In order to demonstrate this effect, Figure 7 shows the impact of having different numbers of threads on these metrics for different stages of Terasort on one of the executors. In stage 0 (Figure 7a), which is essentially a read stage, the epoll wait time expectedly grows as the number of threads increases. More interestingly, the throughput is the highest when the executor uses only 4 threads. Similarly , stage 1 and 2 (Fig 7b and 7c) both see a similar trend in epoll wait time, however throughput is maximized at 8 threads. Recall that in Figure 2a, the (hypothetical) best combination of threads was 4, 8 and 8 for the three stages of Terasort respectively which is equal to the minimum point of the congestion index (green line) in every stage. Therefore, the dynamic solution selects the configuration where the I/O congestion is minimized (indicated by the arrow).

One could argue that average disk utilization shown in Figure 5 is another good metric since it also gives a hint as to which configuration is performing better. The first reason why we chose the combination of epoll wait time and throughput over other metrics is that in some cases disk utilization is very similar between different settings. For example in the first stage of Terasort (Figure 5a), all core numbers achieve 91.13% disk utilization or higher and the difference between the minimum and maximum is less than 6%. In such case, it is difficult to find out which configuration has indeed performed better since they are all relatively high. That is why we are combining two metrics in order to get a more accurate view of the I/O performance. The second reason is that although these metrics are primarily used for disk I/O, unlike average disk utilization, they would also work for network I/O since: a) epoll wait time tracks the time spent waiting for events on any file descriptor which in the case of a network I/O is a network socket, and b) the I/O throughput considers both disk and shuffle data. Therefore, the gathered monitoring values are still meaningful for network operations such as shuffle and remote reads/writes.

In order to traverse through the problem space and evaluate different configurations, the analyzer employs a hill-climbing algorithm in which it attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. In other words, the algorithm always starts from the minimum number of threads (c_{min}) in each stage and doubles the number of threads (in the interest of keeping the settling time low) until it reaches the maximum thread threshold (c_{max}). While these parameters are configurable, in this work, c_{min} is set to 2 and not 1 since it is almost impossible that a single-thread outperforms multiple ones and c_{max} is set to the number of virtual cores as it is often the upper limit At the end of each interval, the analyzer compares the performance between the current (ζ_j) and the previous ($\zeta_{j/2}$) interval and in case of a lower performance, it rolls back to the core size of the previous interval and executes the subsequent tasks without adjusting the number of threads until the current stage finishes. The reason behind the rollback, apart from being the essence of hill-climbing algorithms, is because if a specific number of threads (c_j) perform worse than half of its size ($c_{j/2}$), then most probably increasing the number of threads (c_{j*2}) would only cause more contention and thus lead to a degraded performance.

Additionally, the reason for starting from the bottom and ascending rather than from the top and descending is twofold. Firstly, it is due to the scheduling mechanism in Spark which assigns a new task to an executor as soon as it completes one. This means if we start from a higher number of threads (e.g., 32) and halve each time, then by the time the first interval is finished, 32 new tasks have already been

assigned to be executed and thus halving the thread number would cause tasks to be queued which is in contradiction with the scheduling mechanism of Spark. Secondly, we have observed that most of the time, if maximum number of threads indeed worsens the performance, then starting from there can significantly affect the runtime in which case starting from the bottom gives us a quicker route to finding the optimal thread count.

5.3 [P]lan

As soon as the analyzer makes a decision, the planner is notified whose job is to devise the procedure to enact a desired alteration in the managed resource. The plan function can take on many forms, ranging from a single command to a complex workflow. In the proposed system, the only requested alteration is adjusting the number of threads in the thread pool which is as simple as calling the appropriate method on the thread pool object. However, the internal mechanisms of Spark do not expect this parameter to change. Namely, the Spark scheduler keeps track of all the executors, how many cores they have been launched with and more importantly, their current number of free cores which controls how many new tasks should be assigned to each executor. This quickly unravels that changing something inside one component such as the executor is not necessarily cascaded through other components in the system, causing undesired behavior. As a result, it is crucial that the planner is aware of all the consequences a change might have and therefore be able to select the appropriate courses of action which preserve the system integrity.

5.4 [E]xecute

The final component in the control loop is the execute function which ultimately changes the behavior of the managed resource using effectors based on the actions recommended by the plan function. These actions include first changing the number of threads for an executor and then notifying the scheduler for updating its internal registries. For the first case, the thread pool object in Java conveniently exposes a method (`setMaximumPoolSize()`) which can be used to adjust its size. For the second case, since this was not supported by default in Spark, we had to extend the messaging protocol to facilitate a mechanism for executors to notify the scheduler about any changes in the size of their thread pool to make sure that the scheduler has the same view on the number of threads each executor is currently using.

6 Evaluation

The previous experiments motivated our main design choices for self-adaptive executors. We have enhanced Apache Spark to employ our self-adaptive executors and show their performance benefits in the following section using several community benchmarks.

Name	Type	Size
Terasort	micro	120 GiB
Join	sql	bigdata
Aggregation	sql	bigdata
Page Rank	web search	gigantic

Table 3. Spark applications and problem sizes used in the experiments

6.1 Experimental Setup

In order to evaluate the performance of self-adaptive executors, we run Spark applications on the DAS-5 cluster, using 4 nodes, each with 56 GB of memory and 32 virtual (16 with HyperThreading) cores. The storage system consists of a HDD with 7'200 rpm and a SATA 6.0 Gbit/s interface. We used the HiBench benchmarking suite [8] to conduct the experiments.

Table 3 shows the various applications used in the experiments as well as their input sizes. The input data is read from HDFS (Hadoop version 2.9.1) with the replication factor equal to the number of nodes (i.e., 4) to make sure all executors achieve maximum locality during the read stages. We compare the runtime of various applications against two methods:

- Default Spark: which uses all the available virtual cores in the system.
- Static BestFit: which represents the theoretical optimum derived by combining the per-stage best setting determined by the results of the static solution.

6.2 Performance Results

Figure 8 compares the performance between the proposed method and the default behavior of Spark as well as the static BestFit solution. The numbers inside each stage represent the number of used threads out of the total available cores across the machines (omitted for the very short stages due to space limitations). In all cases, both solutions are able to reduce the runtime compared to the default version.

In case of Terasort (Figure 8a), the static BestFit and dynamic executor reduce the runtime by 47.5% and 34.4% respectively. The reason why the BestFit is able to outperform the dynamic approach is that all three stages in Terasort are considered I/O intensive and while the dynamic algorithm needs to explore and evaluate the performance of all the possible core settings, the static solution starts from an optimized number of threads and therefore is able to finish faster.

However, in stages where the static solution fails to identify them as sensitive to I/O activity (see Section 4), the dynamic approach is able to outperform the static BestFit. For example, in PageRank (Figure 8b) the static approach employs different number of threads only for the first (read) and last (write) stages which reduces the overall runtime

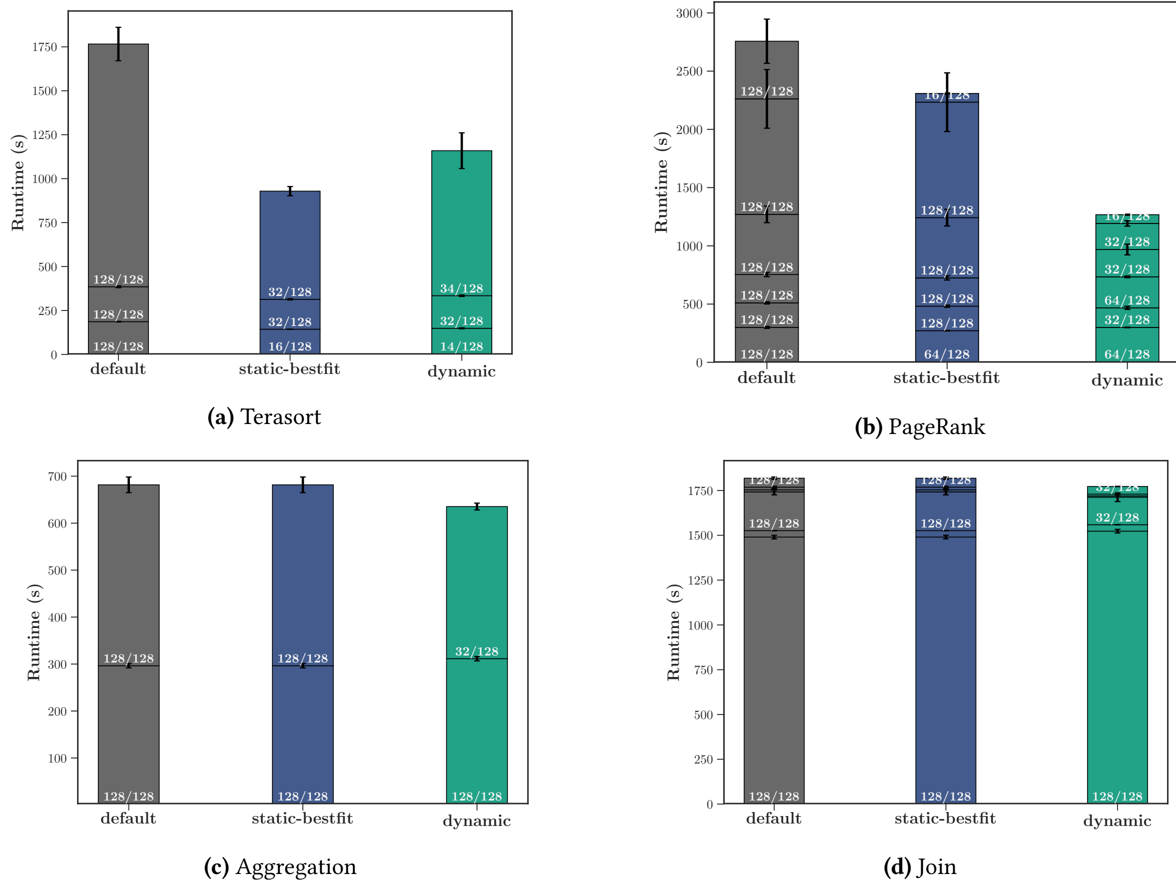


Figure 8. The performance of the dynamic solution compared to the default and the static BestFit. The numbers in each stage represent the number of used threads out of the total available cores across the machines.

by 16.28%, whereas the dynamic approach does it for all the stages and by doing so it is able to achieve a significant 54.08% and 45.15% reduced runtime compared to the default and the static versions, respectively.

For SQL applications such as Aggregation and Join, which did not benefit from the static approach (see Section 4) due to the additional computations performed in their I/O stages, the dynamic approach is able to reduce the runtime by adjusting the number of threads in the other stages. Specifically, the dynamic solution increases the performance by 6.83% and 2.54% for Aggregation and Join respectively. The diminishing effect in the runtime reduction for these particular type of applications suggests that the self-adaptive executors perform better in applications which have pure I/O stages, potentially causing contention on the disk.

The scalability of the dynamic solution in terms of cluster size should not be limited since every nodes makes a local decision on the optimal thread count. Figure 9 confirms this by juxtaposing the 4 node Terasort results with the results from a 16 nodes setup for which the input size has been scaled up proportionally. Most notably, it can be observed that the default settings do not scale (execution time is significantly higher in the 16 node experiment despite constant resources to problem size ratio) while both the static and dynamic solution achieve nearly the same execution time.

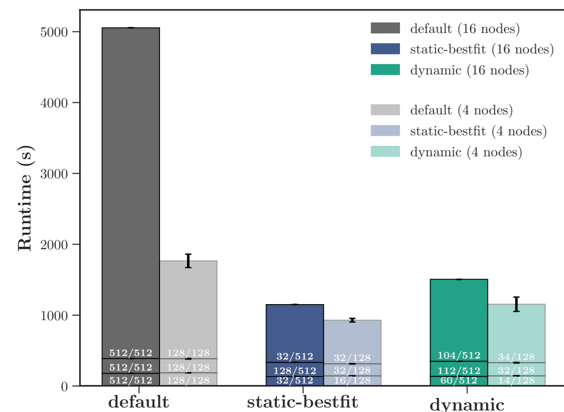
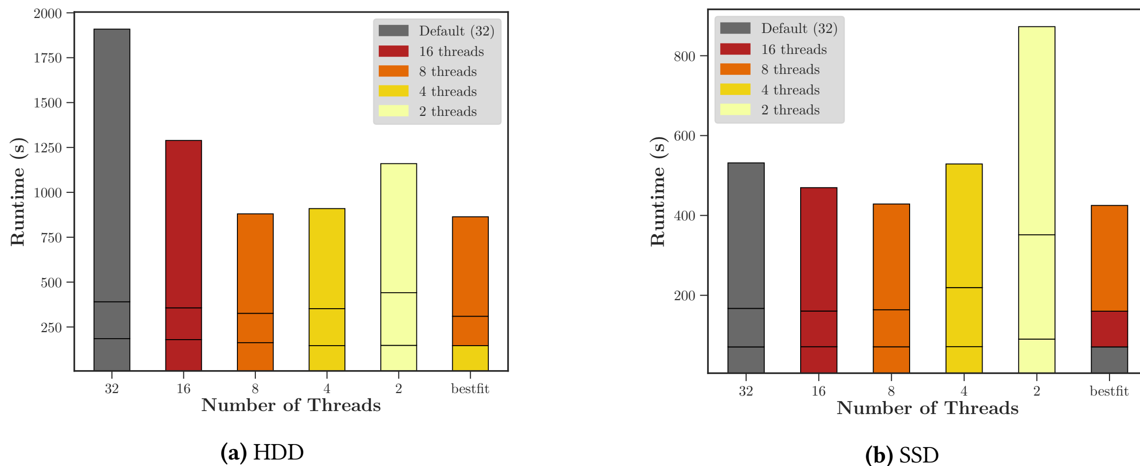


Figure 9. Assessing the scalability of the dynamic solution using Terasort on 16 nodes



(a) HDD

(b) SSD

Figure 10. The effect of HDDs and SSDs on the performance of the static solution for Terasort

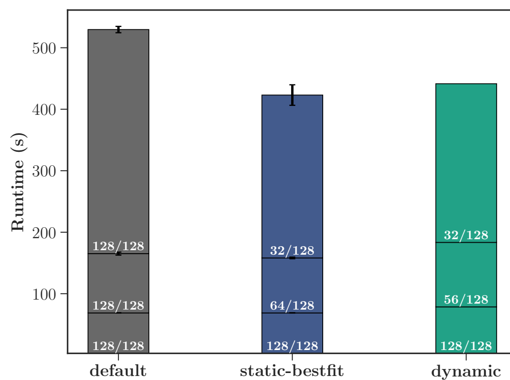


Figure 11. The effect of SSDs on the dynamic solution for Terasort

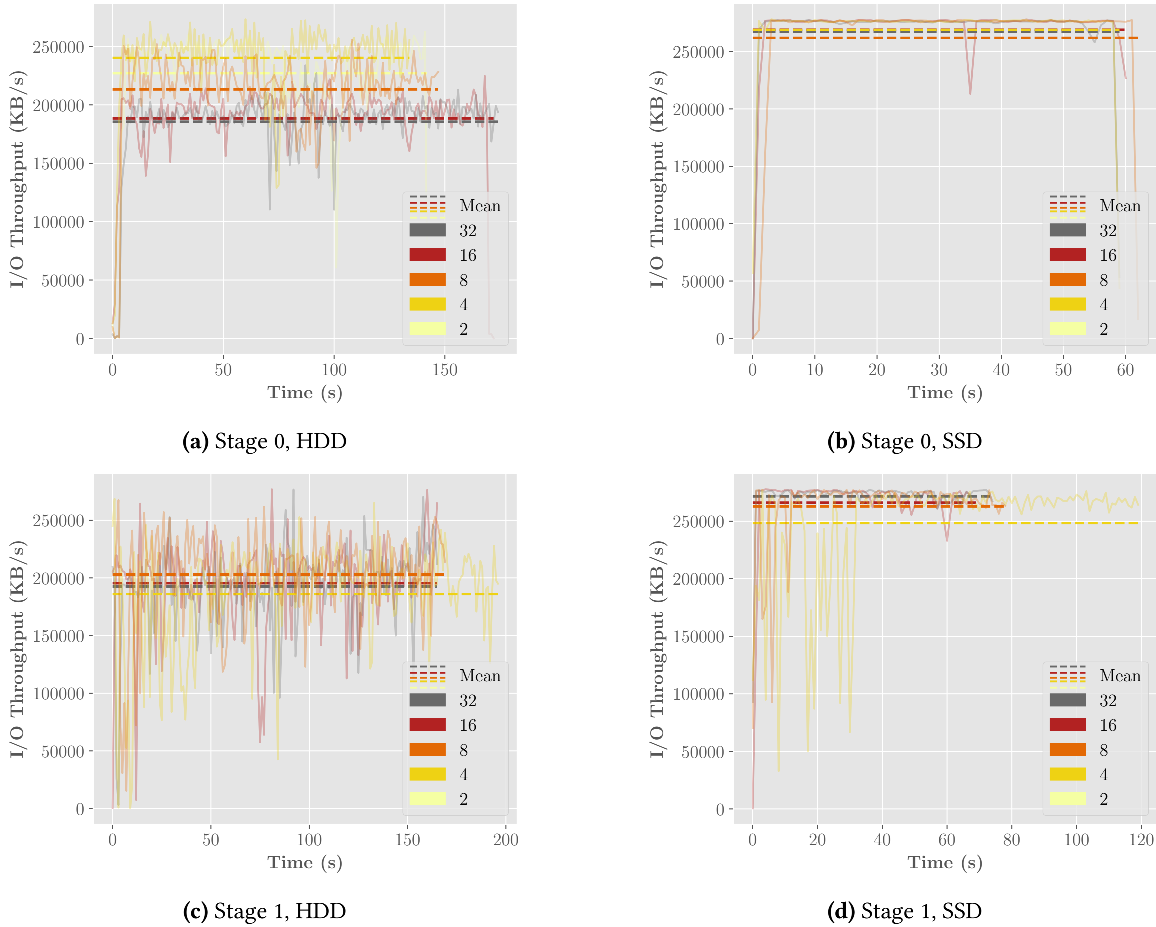
6.3 SSD vs HDD

The effect of the number of threads on I/O performance is highly dependent not only on the nature of the applications but also the underlying I/O infrastructure. If the disk is able to provide a high throughput, then it might be able to service more concurrent I/O requests without becoming a source of bottleneck. Modern high-end servers widely use Solid State Drives (SSDs) as their storage device which are typically more expensive than Hard Disk Drives (HDDs) when measured by cost per Gigabyte of storage but in return provide a higher I/O Operations Per Second (IOPS). In order to analyze the effect of SSDs on the behavior of self-adaptive executors, we ran Terasort on an identical setup (see Section 6.1) with the exception of using SSDs as the storage device. Figure 10 compares the results for the previously shown static solution for HDDs versus the SSD version in different stages.

The first stage which consists of pure read operation, the default number of threads (32) performs best for SSD unlike the HDD version which shows 4. This difference can be explained by considering how HDDs and SSDs work under the hood. In order to read data, HDDs have to wait for their mechanical head to move to the appropriate position which is exacerbated by having more number of concurrent threads, while it is not the case for SSDs. These devices support full random access at a uniform latency, resulting in a much higher read throughput. Figure 12a and 12b show the difference in I/O throughput in the first stage when ran with different number of threads on HDD and SSD respectively. As we can see, with HDD the mean throughput varies quite significantly between different settings with 4 being the maximum whereas in the case of SSD, it is more uniform. The performance degradation in HDDs with higher number of threads is most likely due to the additional head movement, which does not exist in SSDs.

The second stage, consisting of a read operation and a shuffle map which writes intermediate shuffle data, has also changed where SSD benefits the most from 16 threads compared to 8 in HDD. The write speed in SSDs is slower than the read since the data cannot be written to a page unless it is first erased. However, only the entire blocks are erasable which means in order to write a single bit of data to a page, it is necessary to copy all the pages in the block to a staging area, erase the entire block and then write all the pages and the new data back to the erased block. This overhead would explain why the maximum number of threads in this stage does not yield the best performance. If we compare the I/O throughput of this stage between HDD (Figure 12c) and SSD (Figure 12d), we can see that not only SSDs provide a higher throughput as expected, but also perform better when more threads are used.

The last stage which includes shuffling the data (network I/O) and writing the output (disk I/O) is the same between both settings. This is expected as network I/O is not heavily influenced by the underlying disk storage.



(a) Stage 0, HDD

(b) Stage 0, SSD

(c) Stage 1, HDD

(d) Stage 1, SSD

Figure 12. I/O throughput of Terasort with HDDs and SSDs

Regardless of the concrete storage device, both static and dynamic solutions are able to reduce the Terasort execution time, despite to a lesser extent for SSDs (20.23% vs. 47.48% for static and 16.73% vs. 34.4% for dynamic) since these devices are less susceptible to thread contention for stages involving heavy disk I/O.

The performance discrepancy between HDD and SSD is another reminiscent of how tuning decisions can differ amongst different setups. While these results were obtained on two particular configurations, they could potentially be entirely different on another cluster which once again emphasizes the importance of having a dynamic mechanism for deciding the suitable parameter values.

7 Conclusions and Outlook

Tuning the number of threads for big data processing frameworks like Spark is a tedious but necessary task in order to gain best performance. As we have shown in this paper, the difference in runtime between the default setting of using one executor thread per physical core and the optimal setting can easily be 2x. Much of this effect can be attributed to I/O contention. We have presented a static solution for thread pool tuning that only differentiates between phases likely to be I/O-bound and the other phases for which no structural evidence for I/O activity exists. This solution is able to produce good speedup for workloads like Terasort which have many I/O-bound phases but falls short for workloads like PageRank. Using a dynamic approach in the form of self-adaptive executors that employ a MAPE-K style control loop to measure system metrics and adjust the thread pool size accordingly, even such complex workloads can be accelerated substantially, all without any changes to the workload itself. Our implementation serves as a drop-in replacement for the Spark executor and has been released under the Apache-2 license on GitHub[1]. Despite shown for Spark, we envision this approach to be highly applicable to a broad range of different big data processing frameworks and even consider it a blueprint for the design of novel frameworks designed with the goal of liberating the user from the difficult task of manually tuning the worker thread count.

[1] https://github.com/SobhanOmranian/spark-dca

References

[1] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer* 49, 5 (2016), 54–63.

[2] I-H Chung and Jeffrey K Hollingsworth. 2004. Automated cluster-based web service performance tuning. In *Proceedings. 13th IEEE International Symposium on High performance Distributed Computing, 2004.* IEEE, 36–44.

[3] Cloudera Blog. 2015. How to Tune your Apache Spark Jobs. https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Accessed: 2019-05-03.

[4] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. 2002. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 186–189.

[5] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIDR*. 84–94.

[6] Anastasios Gounaris and Jordi Torres. 2018. A Methodology for Spark Parameter Tuning. *Big Data Research* 11 (March 2018), 22–32. https://doi.org/10.1016/j.bdr.2017.05.001

[7] Holger H Hoos. 2011. Automated algorithm configuration and parameter tuning. In *Autonomous search*. Springer, 37–71.

[8] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. 2010. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*. 41–51.

[9] Thomas Karcher and Victor Pankratius. 2011. Run-time automatic performance tuning for multicore applications. In *European Conference on Parallel Processing*. Springer, 3–14.

[10] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 1 (2003), 41–50.

[11] Laszlo B Kish. 2002. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A* 305, 3-4 (2002), 144–149.

[12] Woo-Hyun Lee, Hee-Gook Jun, and Hyoung-Joo Kim. 2015. Hadoop Mapreduce Performance Enhancement Using In-Node Combiners. *International Journal of Computer Science and Information Technology* 7, 5 (Oct. 2015), 1–17. https://doi.org/10.5121/ijcsit.2015.7501

[13] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 165–176.

[14] REA Group. 2017. How We Optimise Apache Spark Jobs. https://www.rea-group.com/blog/how-we-optimize-apache-spark-apps/. Accessed: 2019-05-03.

[15] Dennis M Ritchie and Ken Thompson. 1978. The UNIX time-sharing system. *Bell System Technical Journal* 57, 6 (1978), 1905–1929.

[16] Kazuki Sakamoto and Tomohiko Furumoto. 2012. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 139–145.

[17] Jerome Howard Saltzer. 1966. *Traffic control in a multiplexed computer system.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[18] Charles E. Skinner and Jonathan R. Asher. 1969. Effects of storage contention on system performance. *IBM Systems Journal* 8, 4 (1969), 319–333.

[19] StackOverflow. 2010. GNU make: should the number of jobs equal the number of CPU cores in a system? https://stackoverflow.com/questions/2499070/gnu-make-should-the-number-of-jobs-equal-the-number-of-cpu-cores-in-a-system.

[20] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[21] TuneUp.ai. [n. d.]. Performance Tuning as a Service. https://tuneup.ai.

[22] Unix StackExchange. 2015. How to determine the maximum number to pass to make -j option? https://unix.stackexchange.com/questions/208568/how-to-determine-the-maximum-number-to-pass-to-make-j-option.

[23] Alexandru Uta and Harry Obaseki. 2018. A Performance Study of Big Data Workloads in Cloud Datacenters with Network Variability. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 113–118.

[24] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. 2003. Why Events Are a Bad Idea (for High-Concurrency Servers).. In *HotOS*. 19–24.

[25] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 268–281.

[26] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. 2013. Towards Machine Learning-Based Auto-tuning of MapReduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE. https://doi.org/10.1109/mascots.2013.9

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[28] Yan Zhang, Wei Qu, and Anna Liu. 2005. Automatic performance tuning for j2ee application server systems. In *International Conference on Web Information Systems Engineering*. Springer, 520–527.