Self-adapting ExecutorService implementation

SOFTENG 751 Project

https://github.com/encryptededdy/ExecutorService21 (Shared with osinnen)

Edward Zhang, Simon Su and Franklin Wang

I. Introduction and Motivation

MULTIPROCESSING is one of the most important paradigms in High Performance Computing today. As modern computers hit the limits of single-processor performance, highly multi-core/multi-processor systems (previously only seen in the territory of supercomputers) have spread to almost all computing devices, from client PCs to smartphones. These modern multi-processor systems allow for much higher performance when compared to single-processor systems of old, however, this comes at the cost of requiring workloads to be efficiently distributed across the processors.

Multithreading on modern Operating Systems allows multiple threads of execution to occur concurrently, thus allowing multiple applications to run simultaneously. Of more interest to those in the High Performance Computing space, however, is the ability for a multi*threaded* OS (and associated applications) operating on a multi*processor* system to exploit thread-level parallelism, allowing a single application to improve its performance by scheduling its tasks on multiple threads, and, by extension, multiple processors.

A major component of achieving efficient multithreading in a given application (and therefore good performance) is the effective allocation of tasks onto threads, and proper management of said threads. In higher level languages such as Java, the management of threads and allocation of tasks can be optionally abstracted away from developers, in classes such as the Java ExecutorService.

Unfortunately, the standard Java implementations of ExecutorService are rather simplistic and aren't necessarily optimised for various types of workloads or system environments. They also lack the ability to adapt to changes in workloads/environments while the application is running. In this report, we investigate possible ways of extending the ExecutorService implementations in Java to better cater to dynamically changing workloads and system environments, in order to deliver the best possible performance and/or user experience.

II. Problem Space

A. What is an ExecutorService?

An Executor is an object which executes "Runnable" tasks. It is generally used so that a developer would not have to explicitly create threads and it can provide a way of decoupling task submission from the details of how each task will be run (e.g. thread use, scheduling). In addition, an ExecutorService is an implementation of an Executor which further allows for better management of resources as it provides methods to handle the lifecycle of the threads as well as enabling the tracking of progress of asynchronous tasks.

B. Existing Java Implementation

Java has 2 main ExecutorService frameworks, the ForkJoinPool and the ThreadPoolExecutor. ForkJoinPool is a framework which provides support for fine-grained, recursive, divide and conquer parallelism. It is specifically designed to execute ForkJoinTasks where each task creates additional tasks and make use of work stealing with multiple queues to achieve dynamic load balancing. However, this is implemented by means of a fixed pool of worker threads, the number of which cannot dynamically change. Therefore, we decided not to pursue ForkJoin-type ExecutorServices as it is task type dependent and the efficiency comes from the suitability of specific algorithms rather than adjusting the thread count.

On the other hand, ThreadPoolExecutor is an ExecutorService that executes submitted tasks using a configurable pool of reusable worker threads. The use of thread pools often provides improved performance by reducing the overhead from managing thread lifecycles. The ThreadPoolExecutor parameters of our interest are corePoolSize, maximumPoolSize and keepAliveTime. These are normally configured using the Executors factory and takes form in the following variations of ExecutorService which we can compare our implementation with.

- **SingleThreadExecutor** uses a single worker thread operating off an unbounded queue. Tasks are guaranteed to execute sequentially and no more than one task will be active at any given time. However, the number of threads cannot be reconfigured at runtime and therefore do not offer a useful comparison with our implementation.
- **FixedThreadPool** uses a fixed pool of threads operating off a shared unbounded queue. At any point, at most n (maximumPoolSize) tasks will be active and additional tasks will wait in the queue until a thread is available. By default, all the threads in the thread pool will only be available after on-demand construction (after n task arrives; if the number of active thread is below corePoolSize, the policy used is thread-per-request) and will exist until the ExecutorService is explicitly shutdown. Furthermore, the size of the thread pool can be dynamically adjusted using the provided setters but the corePoolSize will always be equivalent to the maximumPoolSize in this implementation.
- **CachedThreadPool** uses a thread pool which creates threads as they are needed, but will reuse previously constructed threads if they are available. This will typically improve the performance of programs which executes high frequency but short-lived tasks as threads can be terminated and removed from the cache after a period of inactivity (keepAliveTime). Therefore, pools that remain idle for a long period of time will not consume any resources (note that if a user explicitly sets the corePoolSize to be greater than 0, after on-demand construction, the number of threads will not drop below the corePoolSize).

C. How cheap are Threads?

The following is a non-exhaustive list of factors we should consider when adjusting the number of threads.

- **Thread creation and teardown** is not free and creating a new thread will consume computing resources as some processing needs to be done by the JVM and OS (at least one native thread is executed to prepare memory for the thread). Although the cost varies across different platforms if the tasks performed are frequent and lightweight, using thread-per-request will lead to significant overhead.
- **Context switching** takes place when the OS decides to temporarily pause the execution of a thread and start executing another one. All the registers, CPU and RAM are being tuned to the other thread and then restored later. Usually, the overhead from a few operations is insignificant but this can quickly add up with thousands of threads and the OS will spend most of the time switching between threads as opposed to performing the tasks.
- **Thread data** also needs to be considered as each thread has its own stack, descriptors and ThreadLocal variables (usually 256KB). Therefore, further highlighting the expenses for having too many threads.

III. BENCHMARKER

The Benchmarker is a class which allows us to quickly set up different scenarios for testing between our implementation and existing Java implementations. This is accompanied by the Main class which acts as a simple CLI for providing input parameters.

- **Number of tasks** needs to be carefully considered as an important aspect of benchmarking is having reproducible results. By increasing the number of tasks (to approximately 100) and the run time of each task to a magnitude of seconds, we attempted to minimise any randomness in our implementation and thus, always be able to observe the same behaviours in repeated runs.

- **Task granularity** describes the ratio of computation to communication and is often a key performance attribute of parallel programs. Fine-grained tasks may introduce a considerable parallelisation overhead as the program will spend a significant amount of time coordinating the tasks. Coarse-grained tasks on the hand mean that some tasks will take longer and therefore, spend less time communicating. Without a good load-balancing strategy, the program may not fully utilize the available resources (e.g. CPU cores). We attempted to mirror the two scenarios by introducing tasks which are uniform in size (small/large) and tasks which can vary in size (mostly small tasks with a few large tasks). Furthermore, PaperSize is an enumerated type which we used to scale the size of the tasks to a range which provided more meaningful results (not too small so we can identify effects of changes we make and not too large so the tasks are completed in a reasonable amount of time).
- **Task regularity** describes the frequency at which tasks are submitted or in a real application, the frequency of incoming events. This will test the ability of the ExecutorService to react to different event rates. We tested against tasks which arrived at a fixed rate to mimic the scenario where there is a constant demand for the ExecutorService. We also tested against tasks which arrived at a fixed rate but the rate changes randomly over time and tasks arriving at a sinusoidal rate in attempts to mimic the scenario where there are periods of high activity and periods of lower activity. Other configurations included tasks all arriving at the start and a random rate but those did not produce any useful result regarding our EMA implementation (as described above).

IV. Predicting Thread Arrival

As mentioned earlier, the creation and teardown of threads are relatively intensive, which means excessive creation and teardown of threads can result in reduced system performance. Additionally, having too many idle threads sitting around can also result in reduced performance, due to increased resource consumption. As a result, it's important to maintain only as many threads as we need to satisfy incoming tasks, but avoid "thrashing" (constantly creating and tearing down threads) when the incoming stream of tasks is unstable/highly variable.
Java already offers a simple solution to this problem with the CachedThreadPool (discussed earlier), however, its drawback is that when (thread) demand suddenly increases, all the threads have to suddenly be created at once which can choke the performance of currently active tasks and introduce latency for starting the new tasks.

A. Modified Exponential Moving Average

In order to improve latency (of starting new tasks) and reduce performance impacts when new tasks are added, we need an ExecutorService that is able to predict when tasks are about to arrive and create the threads ahead of time. This can reduce latency as when tasks arrive, the threads are already created for them (and so the tasks can start executing instantly. The threads should also be created gradually in order to avoid significantly affecting other tasks currently running.

After reviewing literature, we found that a modified exponential moving average algorithm (henceforth referred to as EMA) as described in Kang et. al [1] would be a feasible and simple method for attempting to pre-empt task arrivals, thus allowing us to adjust our thread count to match. The equation (Figure 1) shows the standard exponential moving average algorithm (1) which shows that the average number of threads at time n is calculated based on the current active threads and previous EMA, weighted by α, the smoothing factor. However, when this is used in practice, it is found to "lag" behind the actual number of active threads. Therefore, Kang et. al proposes a modified version of the algorithm (2) which adds on the difference between the current EMA and the previous one if and only if the difference is positive. This thus allows the algorithms to pre-empt increases in active threads, as it is assumed that any increasing slope would continue.

B. Implementation

Since we wanted the implementation to be transparent to developers using it, and also to follow good object-oriented programming principles, we had to implement the ExecutorService service. To simplify

$$S_n = \begin{cases} t_1 & \text{if } n = 1 \\ \alpha \times t_n + (1 - \alpha) \times S_{n-1} & \text{if } n > 1 \end{cases} \quad (1)$$

$$S'_{n+1} = \begin{cases} t_n + (t_n - S_{n+1}) & \text{if } S_{n+1} > S_n \\ S_{n-1} & \text{else} \end{cases} \quad (2)$$

Fig. 1. The modified Exponential Moving Average equation

development, we extended the existing ThreadPoolExecutor implementation of ExecutorService, as much of the existing logic for a ThreadPoolExecutor did not need to be changed.

Internally, our implementation works similarly to a CachedThreadPool, as we wanted the arriving tasks that weren't successfully pre-empted by our algorithm to automatically create new threads to handle them. In order to implement the EMA algorithm, our ExecutorService spawns a monitor thread when constructed. This thread occasionally polls the main thread pool at a fixed interval in order to get the current number of active threads, then calculates the EMA based on this and stored previous calculations. It then uses the calculated EMA at time $n + 1$ in order to set the CorePoolSize, which determines the number of threads present in the ThreadPoolExecutor without considering cached threads.

C. Results

1) Prediction Accuracy: The prediction accuracy of the EMA algorithm significantly depends on the nature of the workload. As an example, the EMA algorithm is unable to offer any sort of accurate prediction for a totally random workload, where tasks arrive at random times - in fact, in these cases, the EMA algorithm does nothing but reduces performance by randomly creating and killing threads due to false predictions. Luckily however, most real-life workloads do not involve completely random arrivals of tasks - usually, the task rate of arrival ramps up and down over a period of time. Therefore when tested with task regularities such as Sine or Periodic Random (more details on our benchmarking framework in section III) where the task frequency ramps up and down, the EMA is able to reasonably accurately predict the number of active threads.

Figure 2 shows the EMA prediction results when run using our benchmarker in the "Sine" regularity mode. We can see that the algorithm works as expected, tracking the actual active threads (i.e. doing nothing) when tasks are dropping or relatively steady, but pre-empting increases when active threads are increasing. In fact, EMA manages to predict the next increase of active threads approximately 1 time unit prior to the increase.

Unfortunately, we can also see from the graph that at the beginning of the run, the EMA algorithm was unable to effectively suppress the thrashing that was occurring (constantly creating and destroying threads). In some runs we were able to eliminate this by tweaking the α value in the algorithm, however, this compromised the algorithm's ability to pre-empt tasks. Ultimately we decided to undo the tweaks since the task pre-emption was the primary goal of the algorithm.

2) Latency Analysis: One of the benefits of the EMA algorithm being able to predict increases in active threads is the ability to reduce latency (the time between when a task a submitted and when it actually starts executing), due to not having to wait for a thread to be created. In order to test this, the same workloads as above were run, except with code to track the latency for the 100 tasks that were submitted. The results in Figure 3 show that unfortunately these theoretical benefits of EMA were not realised in our implementation/testing. In fact, the mean and max latency performance of the EMA implementation regressed compared to traditional cached or thread per task implementations, even though thread per task should be regarded as the worst possible in terms of latency (when we only consider thread creation/deletion). This shows that the extra logic required to implement the EMA algorithm actually increased latency more than its ability to decrease it.

3) Performance Benefits: After testing, we were unable to detect any difference in performance (i.e. total running time) between the three implementations, as shown in Figure 3. The only differences noted were in latency discussed earlier.

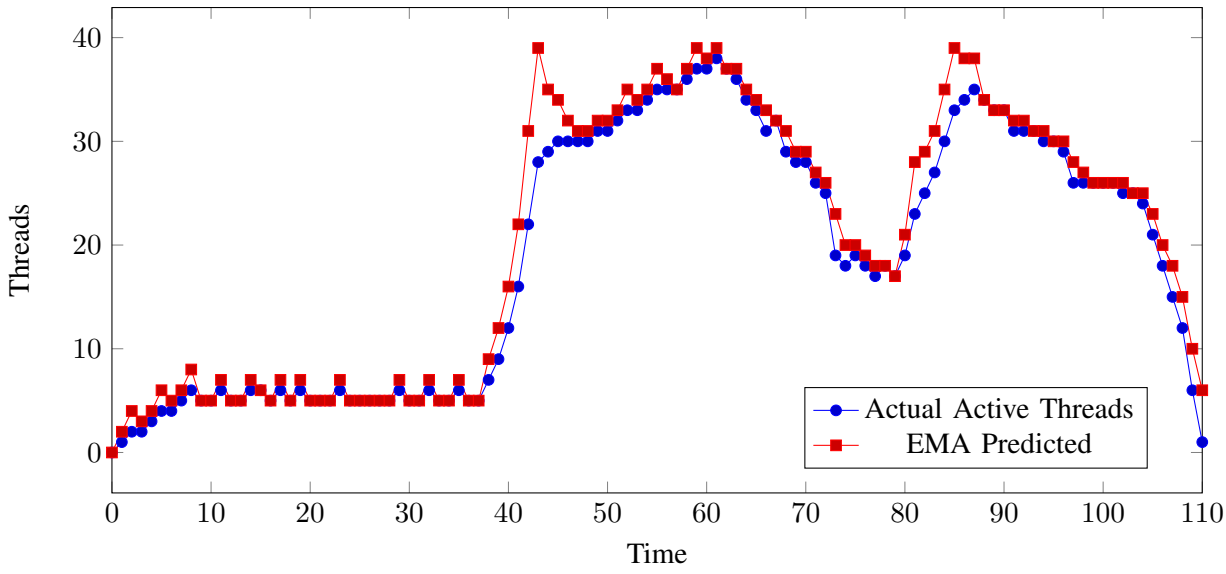

Fig. 2. A plot of predicted vs. actual active threads using EMA. Tested using Sine task regularity with 100 tasks

	EMA	Cached	Thread per Task
Mean	42.5ms	29.8ms	22.5ms
Median	0.08ms	0.08ms	0.08ms
Max	421.1ms	302.8ms	203.4ms
Min	0.05ms	0.05ms	0.05ms
Total Running Time	59.3s	59.2s	59.5s

Fig. 3. Table of latencies and total running time, tested using Sine task regularity with 100 tasks

D. Conclusion and Future Work

As shown by the results, we were unfortunately unable to realise a performance or latency improvement through the use of our EMA implementation due to the high overhead of the algorithm itself.

1) Comparison with Paper: In the paper Kang et. al [1] that we based the EMA algorithm off of, the authors were able to significant ($> 25\%$) improvements in latency, yet we were not able to. While this could be attributed to a faulty implementation on our end, the most likely explanation is the difference in systems used to run the benchmarks. The table in Figure 4 shows the difference between the capabilities of the Ryzen 7 CPU used in our benchmarks and the Pentium III 800Mhz used in the paper (it's unknown why this CPU was chosen for a 2008 paper, given it was released nearly 10 years prior). The much slower performance (100-1000 times slower) and lack of multiprocessing support probably meant that system calls to create or destroy threads on the Pentium III took orders of magnitude longer than on our system, which meant that the pre-empted thread creation of the EMA algorithm would have a much larger impact on overall runtime and latency.

Unfortunately we were unable to find a similar system to the one in the paper in order to run our benchmarks on that was still able to run the modern version of Java (Java 8) that our implementation was written to.

	Memory Bandwidth (GB/s)	Linpack (GFlops)	Processors (incl. SMT)
Ryzen 7 1700	39.74	210	16
Pentium III 800	0.25	0.3	1

Fig. 4. Performance comparison between CPU used for benchmarks here versus in Kang et. al

2) Possible Future Improvements:

- Adjust the polling frequency dynamically - e.g. the frequency could increase if the number of tasks coming in seem to be rapidly increasing. This could affect the algorithm's accuracy however, as it wasn't designed for unequal time steps.
- Instead of relying on the CachedThreadPool backing to handle tasks added when no threads are available, we could instead immediately trigger a calculation of the algorithm.
- Test benchmarks on massive numbers of threads, or on a very slow system (perhaps embedded systems).
- Tweak the algorithm to more effectively suppress small changes that can cause constant creations and deletions of threads.

V. Impact on Graphical User Interfaces

Graphical user interfaces (GUIs) are prominent components of software systems. They are used as a method to portray information to the user and to facilitate interaction between the user and the system. In Java, popular GUI frameworks (such as Android, Swing, and JavaFX) are not thread-safe. As such, the GUI can only be manipulated from a so-called "GUI thread". The GUI thread generally handles the rendering of the GUI, input from the user etc. Therefore, running any computationally expensive tasks on the GUI thread results in an unresponsive GUI as the rendering process is halted. To avoid unresponsive GUIs, systems must be parallelized to avoid clogging up the GUI thread.

Generally, smaller systems can run their computationally expensive tasks on background threads without worrying about affecting the GUI thread. However, in larger systems, tasks may come in larger quantities or require more computational power. As the system demands more computational power, the GUI thread may be affected; once again resulting in a slow or unresponsive GUI.

A. Implementation

In order to maintain a fluid and responsive GUI, a system can maintain a low number of background threads to ensure the GUI thread is not affected. But in order to maximize the usage of the available computational power, the GUI thread can be monitored to determine the optimal number of background threads to maintain before the performance deteriorates.

In order to monitor GUI performance, we decided to monitor the framerate as it's an easy-to-measure metric which captures both GUI fluidity and responsiveness. Framerate can be determined by how frequently the GUI thread invokes the render function on components in the GUI. Therefore, we can add a lightweight and invisible component to the GUI whose sole purpose is to measure the frequency of invocations to the render function. GUI responsiveness is determined by how long it takes the GUI framework to process an input event from the user. In the case of popular Java GUI frameworks, handling user input events is done on the GUI thread, which means that framerate is directly proportional to responsiveness.

Using this information, we can design an ExecutorService that creates just enough threads to maximize the usage of available computational power, while maintaining a target framerate for the GUI. Our ExecutorService implementation quickly ramps up the number of threads until it notices a performance hit on the GUI, then slowly decreases the number of threads until the performance returns to normal. After this, the aforementioned process is repeated but now new threads are created at a much slower rate.

B. Evaluation and Results

To evaluate the effectiveness of our implementation, we compared it against Java's FixedThreadPool and CachedThreadPool implementations. All implementations were all submitted 100 of the same task at once. The task itself computed a mathematical expression repeatedly. When run sequentially, each task completes in roughly 1.8 seconds. In our test, we consider anything below 60 frames per second (FPS) to be a performance hit i.e. once we dip below 60 FPS our implementation will react by reducing the number of threads.

The results in Figure 5 show that the CachedThreadPool performs the best, which is to be expected as GUI performance drops considerably to allow for faster execution of the tasks. Our dynamic implementation completes the 100 tasks slower than CachedThreadPool but maintains a framerate very close to the target FPS. The FixedThreadPool implementation was run with 55 threads as that was the optimal thread count found by our dynamic implementation. Comparatively, our dynamic implementation runs slightly slower than FixedThreadPool as it needs to initially find the optimal amount of threads and continue watching the ExecutorService performance.

	Cached Impl.	**Our Impl.**	**Fixed Impl. 55 Threads**
Trial no. 1	118s	128s	125s
Trial no. 2	117s	125s	124s
Trial no. 3	118s	126s	122s
Average Runtime	117.5s	126.3s	123.7s
Average FPS	32	58	60

Fig. 5. Time to complete 100 tasks in each implementation, along with the average FPS

C. Conclusion

Compared to standard Java ExecutorService implementations, this implementation has benefit of maintaining GUI performance while only incurring a slight cost in task completion performance. In addition, standard Java implementations don't have any mechanisms for dynamically scaling the number of threads in response to other performance factors. Our approach works best for short to medium lived tasks as otherwise, a reduction in thread count may not be apparent until a long-lived task finishes executing.

VI. Thread Underutilization

Not all tasks are created equal. Some tasks are computationally heavy while others may involve idle time due to network requests, reading data off disk, waiting for user input etc. When idle, running threads are underutilizing the available computational power e.g. for a system with 4 cores running 4 threads that sleep half for the time they're executing, only half of the computational power of the 4 cores is being utilized. Again, an issue with the default Java ExecutorService implementations is that none of them pick up on these factors and react accordingly to maximize computational usage.

A. Implementation

To monitor the state of threads, Java provides a class named ThreadMXBean. Of particular interest is a method that allows us to retrieve the thread CPU time i.e. how long a thread has actively used the CPU for. Using this, we can periodically measure the difference in CPU time and determine the actual utilization. For example, if we get the thread CPU time before and after sleeping 300 milliseconds and calculate the difference between the two values, then with full utilization we would expect the difference to equal 300 milliseconds.

We can then apply this method to all of the threads in an ExecutorService implementation to determine the underutilization of the threads. Our ExecutorService implementation takes in a base number of fixed threads. Additional threads then are spawned if the base threads do not fully utilize the core it is executing on i.e. we should expect 400% CPU utilization when 4 base threads are specified.

B. Evaluation and Results

To evaluate our implementation, we created tasks that spend a significant proportion of its execution sleeping. Then 32 of such tasks are submitted to our ExecutorService implementation and Java's FixedThreadPool. During execution, our implementation spawned new threads to fully utilize 4 cores, while FixedThreadPool heavily underutilized 4 cores as the threads were mostly sleeping. As shown in Figure 6, our implementation outperformed Java's FixedThreadPool.

	Our Implementation	**Fixed Implementation**
Trial no. 1	18.6s	35.5s
Trial no. 2	19.7s	37.4s
Trial no. 3	18.8s	35.2s
Average Runtime	19.0s	36.0s

Fig. 6. Time to complete 32 tasks involving sleeping in each implementation

C. Conclusion

Valuable time is lost when a thread is sleeps without allowing other threads to execute over it. Compared to Java's FixedThreadPool, our implementation has the benefit of effectively allowing a "single thread" to fully utilize the core it's running on, even if the task execution involves inactivity. Our approach works best for short to medium lived tasks as otherwise, a change in thread count wont be apparent until a long-lived task finishes executing.

VII. Conclusion

The ExecutorService interface in Java allows for the development of various thread and task management implementations that any Java developer can easily use. In this report, we investigated three such ExecutorService implementations that all adjusted thread counts on the fly based on algorithms that respond to various metrics, whether they be internal (task arrival rate, number of inactive tasks) or external (frame rate). While we were unable to demonstrate a clear performance increase with all of our implementations, we were still able to show the functioning of our algorithms based on the metrics we chose. Perhaps with further work and tweaking, these implementations could become feasible libraries distributed for other Java developers to use in certain use cases, where they could become a better choice than the Java Concurrent library's basic implementations of ExecutorService.

Appendix
Contribution Table

	Report	**Presentation**	**Implementations**			
			Benchmarker	**GUI**	**Thread Util.**	**EMA**
Edward Zhang	40%	33%	20%	-	-	100%
Simon Su	30%	33%	80%	-	-	-
Franklin Wang	30%	33%	-	100%	100%	-

References

[1] D. Kang, S. Han, S. Yoo, and S. Park, "Prediction-based dynamic thread pool scheme for efficient resource usage," in *Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, ser. CITWORKSHOPS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 159–164. [Online]. Available: http://dx.doi.org/10.1109/CIT.2008.Workshops.93