POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Electronics and Computer Engineering
SPECJALNOŚĆ:

PRACA DYPLOMOWA INŻYNIERSKA

Performance Analysis of Microservices Based Architectures Using Parallel Processing Benchmarks

Autor:
Luís Alves de Sousa Rêgo

PROWADZĄCY PRACĘ:
dr inż. Dariusz Caban, K30W04D03

WROCŁAW 2021

Table of contents

Introduction

Developing applications in a microservices architecture has many benefits[7] such as development agility, faster time-to-market and higher developer productivity, making adopting this architecture a viable choice for many different situations even when good scalability is not a requirement. One of these use cases can be parallel processing applications with worker services running side by side. Developing parallel processing applications with this technique may have the upsides that come with using microservices but comes with a potential cost of communication overhead compared to using another solution for parallel processing such as multithreading.

This project aims to compare the performance between multithreading and microservices implementation running a single machine. To perform this comparison, a parallel processing benchmark is implemented for both techniques and performance between the two is analysed. It is expected to see less efficiency using microservices but it is still interesting to know how significant this difference is.

Benchmark

A benchmark is a test used to compare performance on similar objects. These objects can be hardware pieces such CPUs or GPUs, software tools like compilers or database management systems, and even same purpose software techniques. In this case, because the subject is parallel processing, the interest is in benchmarks for parallel processing techniques. Performance is compared by implementing a specific task in both techniques and analysing the execution times. Benchmarks are designed to mimic a particular type of workload so it produces results more similar to real world usage.

Some common parallel processing benchmarks are The NAS Parallel Benchmarks[3] consisting of seven parallel benchmarks designed for highly parallel computation. Others are described in A Benchmark Suite for Evaluating Parallel Programming Models[4] and include image processing tasks such as colour conversion and image rotation, cryptographic tasks like MD5 hash calculations or even video computer graphics tasks for example ray tracing among others.

This project is comparing between multithreading and microservices. In a microservices architecture, services are dependent on each other making communication essential for an application to work. This importance in communication between services makes the case to choose a benchmark designed to test performance on a very high communication use case. For this reason The big benchmark described in Benchmarking Parallelism and Concurrency in the Encore Programming Language[5] was chosen.

Big Benchmark

The big benchmark consists of numerous message exchanges between workers. Workers are divided into neighbours, 10 in this case, and are only able to communicate with other workers in the same neighbourhood. There are two different types of message being sent: ping and pong messages. Ping messages are sent from one worker to another random one inside its neighbourhood. Pong messages are sent as a reply when a ping message is received. Each worker will send ping messages and reply with pong messages until a specified ping limit is met.

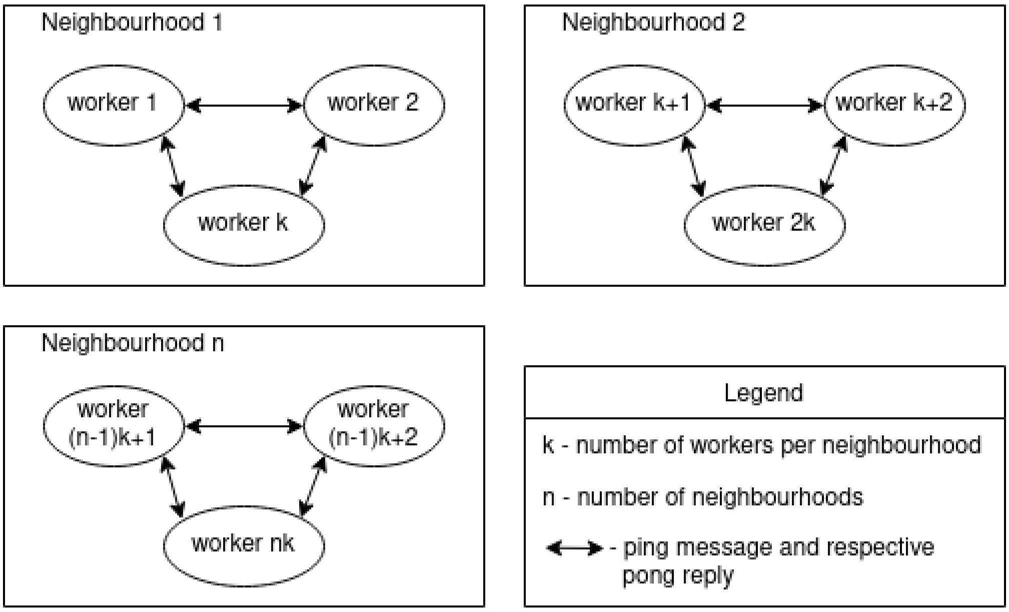

Fig. 1 - Big benchmark visualization

To measure how different loads perform, two different parameters are used: the amount of pings each worker sends and the total number of workers expressed as the number of workers per neighbourhood.

Implementations

All the code in this project was developed in the Go[6] language, version go1.16.3 linux/amd64.

Multithreading

Multithreading in Go is achieved using goroutines[2] which are lightweight threads managed by the Go runtime. These goroutines can easily communicate with each other by sending messages through channels. The big benchmark is really heavy on communication so this implementation takes advantage of Go's channels[1] to effectively achieve message passing from one worker to the next.
The program begins inside the main function that spawns each worker in a new goroutine. Workers then repeatedly ping each other through channels. Once the ping limit is reached they signal their status to the main thread using a WaitGroup and stop sending ping messages.

The main function starts by creating all the neighbourhoods implemented as arrays of workers where each worker is a struct with an id corresponding to its index in the neighbourhood, and two channels: the ping and the pong channels for reading the respective messages sent by the other workers.
After the neighbourhoods are created, each worker is initialized in a separate goroutine by calling the function *work* and passing it its corresponding worker and neighbourhood.
Finally all workers start communicating with each other after a signal from the main thread in the form of a pong message.

The work function expects as inputs a worker w, the worker's neighbourhood as a pointer to an array of workers, and a pointer to a WaitGroup. It then initializes a ping counter as 0 and starts the work loop, in every cycle tries to read the pong and ping channels in that order. In the case that there is a message to be read in the pong channel, the ping counter is checked to verify if the worker already reached the communication limit and should signal the main thread and stop sending ping messages. If not, a ping message is sent to a random worker in its neighbourhood and the ping counter incremented. In case there is a message to be read in the ping channel, the worker immediately replies to the sender with a pong message.

The ping message content is the id of the worker sending the message. When the receiving worker reads from the ping channel, it uses this id as the index in the neighbourhood to find the sender worker and send back a message to the pong channel associated with it.
The pong message content is irrelevant, just the act of sending a message is sufficient. In this case the string ‘pong’ is being sent for clarity but any other message would be equally valid.

The elapsed time is counted from the moment the program starts until the moment the last worker signals the main thread before terminating.

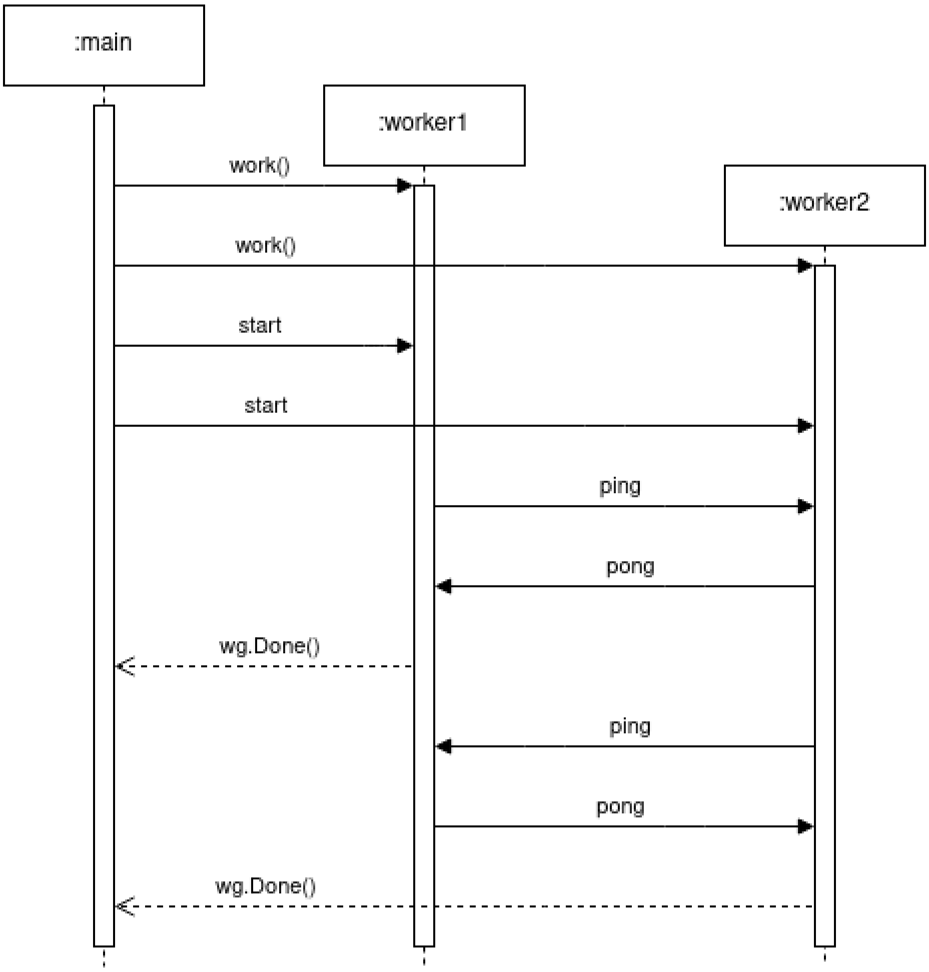

Fig. 2 - Multithreading sequence diagram

Microservices

First approach

For this microservices implementation two different types of services were developed: the core service, responsible for coordinating and delegating the workers task, and the worker service that sends and receives the ping and pong messages. All services exchange messages in the form of http requests with the content of the message encoded in json.

Each service runs in a different process in the same machine, this means that the core runs on its own process and so do each of the workers. The services are started by a bash script that will first compile the core's and worker's code into two different binaries, then start the core service, wait 1 second and start all the workers.

The core service begins by serving an http server listening on port 8090. This server is waiting for requests on two endpoints: /register and /finish.

The /register endpoint expects a RegisterMessage, a json object with a single field "port" containing the port the worker is going to be listening to. When a request to /register is made its body is validated and the port value is appended to a list, implemented as a slice, that holds all the workers ports. Requests made to http servers in Go are processed in different goroutines, so to avoid race conditions when writing to the shared memory, in this case the workers list, the sensitive operations are inside a critical zone guarded by a mutex.

Once all the workers have been registered, the startWorkers() function is called. In this function the neighbourhoods are created by splitting the workers list into multiple arrays, each one representing one neighbourhood. Then the neighbourhoods are cycled through and a request is made to each worker signaling it to start its task. This request has the form of a StartMessage, a json object with one field "pingLimit" containing an integer with the number of pings the worker should execute and another field "neighbourhood" containing that worker's neighbourhood array.

The /finish endpoint simply increments a counter for each request received and when it reaches the number of workers, the elapsed time is printed and the service terminates. Once again to avoid race conditions regarding the counter the code access is limited by a mutex.

The worker service starts by listening on an unspecified port attributed by the operating system, however unlike the core service, it doesn't immediately start serving an http server. First it gets its own port and registers it to core by sending the RegisterMessage as an http request to core’s port and /register endpoint. Only after that starts serving the http server. This server is expecting requests on two endpoints: /start and /ping.

The /start endpoint expects and validates a StartMessage containing the ping limit and neighbourhood needed to run the service’s task. If the validation is successful, the service starts a new goroutine that simply picks a random neighbourhood and sends an http request to its /ping endpoint in a loop until the ping limit is reached. When this happens an http request is made to core’s /finish endpoint signaling that the task is over and the process terminates successfully.

The /ping endpoint behaviour is very simple, handles the http request by sending an http response with the string “pong” as the body.

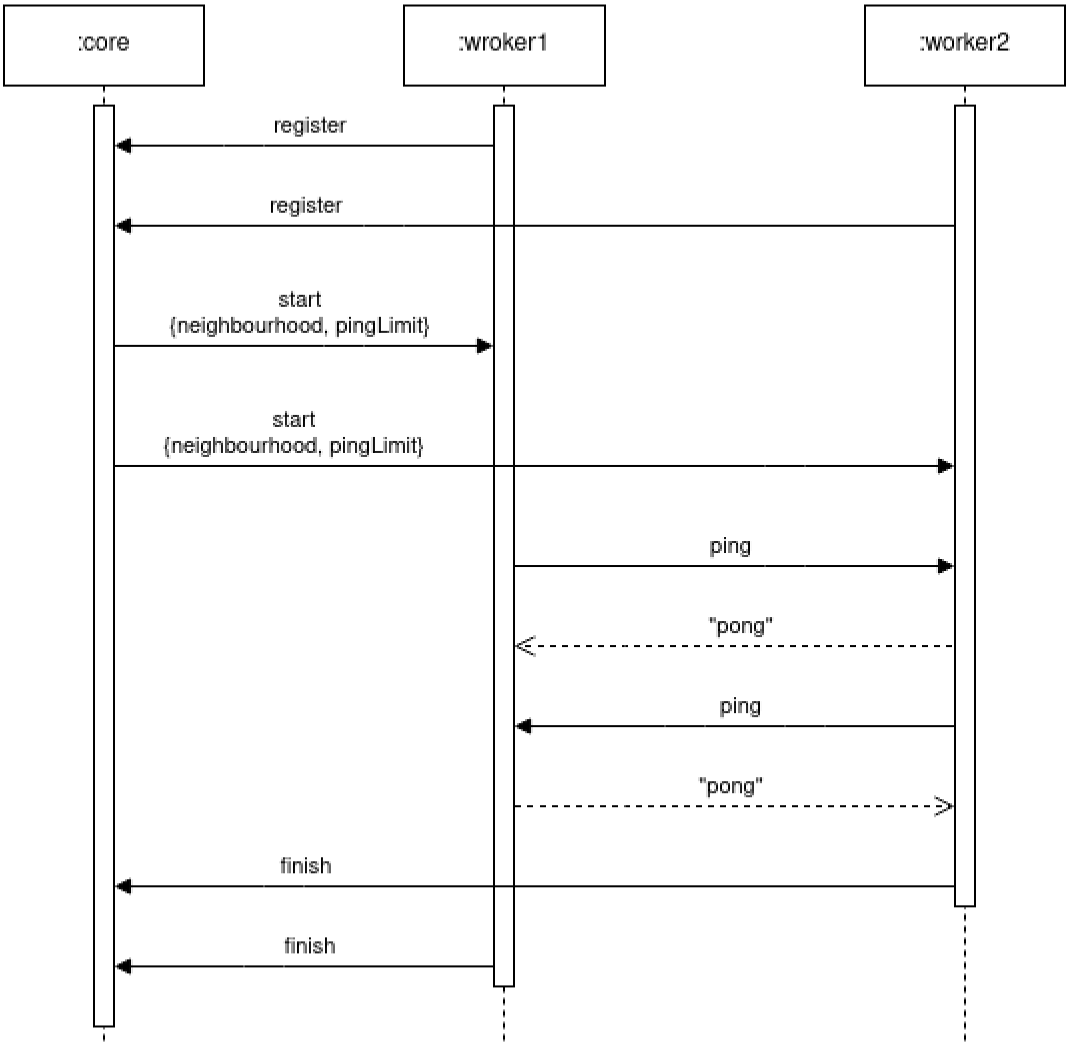

Fig. 3 - Microservices first approach sequence diagram

In Fig. 3, the solid arrows represent an http request, the name on top of it is the endpoint path and the content inside the curly braces is the json object sent in the body when applicable. The dashed arrow represents the http response of the previous request and the string on top of it is its body. Some of the requests have no corresponding response arrow on the diagram, this means that the code is ignoring the response as long as the response is successful.

When analysing the results, this implementation proved to be quite inefficient. A complete http server is being used to handle all communications but also the communication itself is happening over tcp. Because this benchmark assumes everything is running in the same machine, there is the possibility to use Unix Domain Sockets[8] that are more lightweight and have their interaction operated directly in the kernel which should make the communication faster.

Looking back at the code it was also possible to see that there are two instances where some multithreading is happening in the microservices implementation. The first instance is in the http server that serves each request in a separate goroutine, and the second one is happening when the worker starts and pings all the other workers also in a separate goroutine. This means that the incoming requests are being handled concurrently with each other and also the ping requests are being created concurrently with the incoming requests.

In order to resolve both of these issues: improving the performance and removing the concurrent processing from the microservices implementation, a new microservices implementation was made with a slightly different approach.

Second approach

The major optimizations done in this implementation are:

The replacement of all http over tcp communications with communication through unix domain sockets, uds for short. Because uds doesn’t have the network overhead that tcp does, this will hopefully increase performance.

Replacing the http server for direct communication between sockets. The code becomes more verbose and complex but the performance should increase.

The removal of all concurrent processing from this implementation, making each service completely single threaded. This will probably decrease performance but makes a case for a better comparison against the purely multithreading technique.

Every message sent in this implementation follows these steps: a new communication is created over uds to the remote address; the new communication is accepted and the message is transmitted; the communication is closed. For this message exchange to work in a single thread both the core and workers will block waiting for a new communication, once it is received, handle the communication and perform the needed processing. After that the cycle restarts and they block again waiting for a new communication.

Messages sent in this implementation are encoded as a json with two fields: “op” that indicates the type of message as a string, for example “register”, ”finish”, or “pong”. And “content” that is optional and brings more information for some types of message, for example the NeighbourhoodMessage will have an array of address in the “content” field.

Core starts by listening and accepting communications in it’s address, in this case “/tmp/core.sock”. It expects two types of messages: a RegisterMessage sent by a worker to signal core that it is ready to start the task, and a FinishMessage also sent by a worker to signal core that it finished its task.

RegisterMessages contain information about the workers address and when one is received, core will add that address to an array that holds all registered workers. When the expected number of workers is reached, core will signal all the workers to start the task.

Before starting the workers, core creates the neighbourhoods by separating the workers array into smaller arrays of the same size and sends to each worker the respective neighbourhood array as a NeighbourhoodMessage. After all workers are informed of their neighbourhood, core starts counting the execution time and sends a PongMessage to every worker in order to trigger them to start the task.

When a FinishMessage is received, core increments a counter for finished workers and when that counter reaches the total number of workers the elapsed time is recorded and the process exits successfully.

Worker service starts by listening on its address, then registering itself in core by sending a RegisterMessage with its address and finally waiting for new communications. It expects three different types of messages: a NeighbourhoodMessage, a PingMessage and a PongMessage.

The NeighbourhoodMessage just informs this worker which other workers it should communicate with, when receiving this information it saves it as an array.

The PingMessage contains the address of the sender, and when a worker receives it, it immediately sends a pong message to that address.

When a PongMessage is received, a ping counter that accounts how many pings were sent gets incremented and in case that the number of pings hasn’t reached the ping limit, the worker will choose another random worker and ping it. In the case that the ping limit was in fact reached, the worker signals core with a FinishMessage indicating that it finished the task.

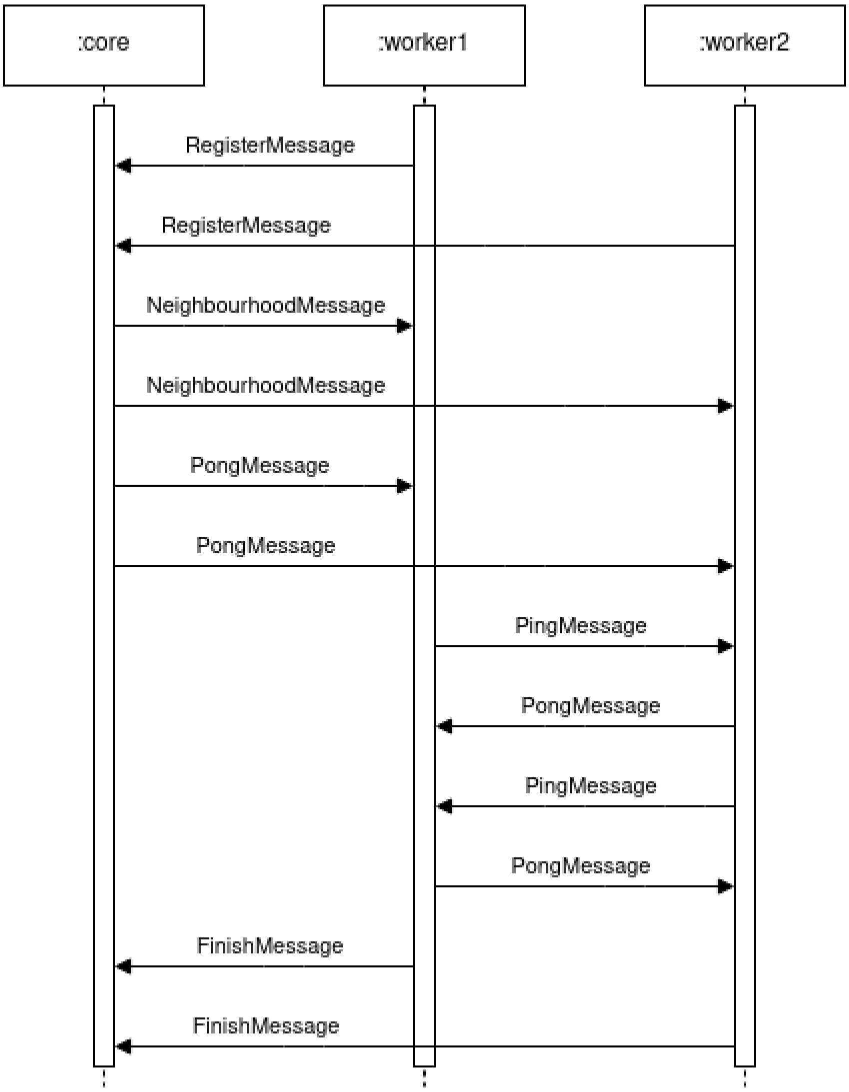

Fig. 4 - Microservices second approach sequence diagram

In Fig. 4 each arrow represents a single message being sent over an uds socket. The name on top of the arrow indicates the type of message.

Although this approach performs better, as is shown further in the document, there is still room for improvement while keeping it single threaded. For example making all the communication faster by keeping connections between workers through the whole benchmark instead of opening and closing a new one for every message.

Experimental results

The experimental values were obtained by running the benchmark on a AMD Ryzen 7 4800H processor, capable of running 16 threads in parallel, with the Pop!_OS 20.04 LTS operating system, linux kernel version 5.11.0-7614-generic.

Each different set of parameters was run 5 times both using the multithreading and the microservices approach. Two parameters were varied taking the measures: the workers per neighbourhood and the pings per worker. The number of neighbourhoods was always fixed at 10. First the pings per worker were fixed at 50000 and the number of neighbourhoods was incremented from 10 to 50 at steps of 10. Then the pings per worker incremented from 50000 to 250000 at steps of 50000.

For the multithreading implementation, times are shown in milliseconds (ms) and for microservices, times are shown in seconds (s). The following table contains all raw measures taken:

Table 1 - All time measures taken for each implementation and respective parameters used

Parameter Identifier	Neighbourhoods	Workers per neighbourhood	Pings per worker	Multithreading (ms)	Microservices - first approach (s)	Microservices - second approach (s)
s1	10	10	50000	125.131	114.685	31.131
				113.895	117.531	32.803
				131.661	116.764	33.026
				126.592	123.710	33.167
				118.633	128.898	33.193
s2	10	20	50000	188.835	266.543	73.522
				180.796	273.415	75.472
				180.714	256.985	75.084
				194.794	256.842	76.788
				222.690	259.998	78.642
s3	10	30	50000	269.663	371.745	117.047
				264.608	387.221	123.328
				272.512	396.879	122.052
				272.350	393.354	119.446
				263.730	390.172	118.398
s4	10	40	50000	346.970	532.610	161.898
				342.865	527.695	161.959
				337.515	529.190	162.459

				351.390	534.117	162.714
				334.845	528.756	161.579
s5	10	50	50000	410.599	641.731	208.578
				419.604	658.760	222.302
				418.722	667.732	225.284
				398.458	670.459	224.765
				384.779	672.396	222.673
s6	10	10	100000	228.080	228.550	74.887
				201.598	256.466	75.147
				268.376	258.278	75.335
				257.335	255.600	78.332
				214.346	247.733	82.168
s7	10	10	150000	390.591	362.137	117.417
				265.001	369.380	117.950
				288.706	369.734	117.442
				342.432	366.942	118.185
				407.695	366.928	117.576
s8	10	10	200000	452.979	492.071	165.622
				397.709	498.036	165.584
				447.942	490.015	165.567
				536.922	493.626	153.519
				414.872	493.740	148.996
s9	10	10	250000	539.135	601.440	206.375
				565.724	614.787	207.504
				494.103	611.088	206.638
				510.980	619.260	199.389
				511.751	629.985	189.139

Results analysis

Arithmetic mean

In order to compare the performance between the different results, some more analysable values are needed, the raw measures alone are not very good to work with directly. So the arithmetic mean of execution times is calculated for each set of parameters.

$$A = \frac{1}{n}\sum_{i=1}^{n} a_i$$

Where n is the number of measures taken and a_i the measures themselves. Applying this to all sets of parameters results on the following table:

Table 2 - Arithmetic mean of time measures for each set of parameters

Identifier	Neighbourhoods	Workers per neighbourhood	Pings per worker	$A_{multithreading}$ (ms)	$A_{microservices}$ first approach (s)	$A_{microservices}$ second approach (s)
s1	10	10	50000	123.183	120.318	32.664
s2	10	20	50000	193.566	262.757	75.902
s3	10	30	50000	268.572	387.874	120.054
s4	10	40	50000	342.717	530.473	162.122
s5	10	50	50000	406.432	662.215	220.721
s6	10	10	100000	233.947	249.325	77.174
s7	10	10	150000	338.885	367.024	117.714
s8	10	10	200000	450.085	493.498	159.857
s9	10	10	250000	524.339	615.312	201.809

Once again the values for multithreading are in milliseconds (ms) and microservices in seconds (s). It is immediate to understand that, as expected, multithreading execution times are always much faster than microservices.

Comparing microservices approaches

Before comparing the two techniques it is important to understand how much the second approach to implementing microservices improves compared to the first one.

Looking directly at the calculated arithmetic means it is already possible to see that the second approach is indeed more performant than the first one since its execution time is faster for any given set of parameters.

It is established that there is a performance improvement for the second technique, however, it is also important to see how much is the improvement. For this the ratio between the first and second approaches is calculated.

Microservices approaches ratio

This ratio is calculated using the following expression:

$$R_{microservices} = \frac{A_{microservices\ first\ approach}}{A_{microservices\ second\ approach}}$$

The following table shows the calculated values:

Table 3 - Microservices ratio

Identifier	$R_{microservices}$
s1	3.684
s2	3.462
s3	3.231
s4	3.272
s5	3.000
s6	3.231
s7	3.118
s8	3.087
s9	3.049

The results seem fairly consistent to all the different sets of parameters. So all the ratios are averaged in order to get one representative value of the increase in performance.
Calculating the average returns the following result:

$$A_{R_{microservices}} = 3.237$$

This means that improvement in performance gained by the optimizations developed in the second approaches by a factor of 3, in other words: the first approach was on average 3.237 times slower than the second one.

Comparing multithreading and microservices

Once again looking at the average values in Table 2 it is easy to see that the multithreading technique is faster than both microservices implementation for any given set of parameters.

Ratio

Comparing the arithmetic means already shows that microservices architecture performs slower than multithreading on a single machine, although it doesn't give a lot of information on how much. To help with that the ratio between both microservices approaches and multithreading, was calculated in the following manner:

$$R_i = \frac{A_{microservices\ approach\ i}}{A_{multithreading}}$$

Where i takes the value 1 and 2 representing which microservices implementation the ratio refers to. This way it is possible to understand better by what factor the difference in performance is. The following table has the calculated ratios for all sets of parameters:

Table 4 - Ratio for both microservices approaches

Identifier	R_1	R_2
s1	976.741	265.165
s2	1357.453	392.123
s3	1444.207	447.009
s4	1547.846	473.049
s5	1629.337	543.068
s6	1065.735	329.878
s7	1083.033	347.357
s8	1096.455	355.172
s9	1173.502	384.883

An interpretation for these values is how many times more does the microservices implementation take in comparison to the multithreading. For example for the first set of parameters "s1" the ratio R_1 is 976.741 and R_2 is 265.165, this means that the microservices implementations took on average 976.74 and 265.165 times more time to run than the multithreading implementation. These are very significant values. The values are considerably smaller on the second approach, which once again shows the performance gained, but the ratio is still very significant.

It may be interesting to see how these ratios change with the different parameters, for that the following graph was graph was created:

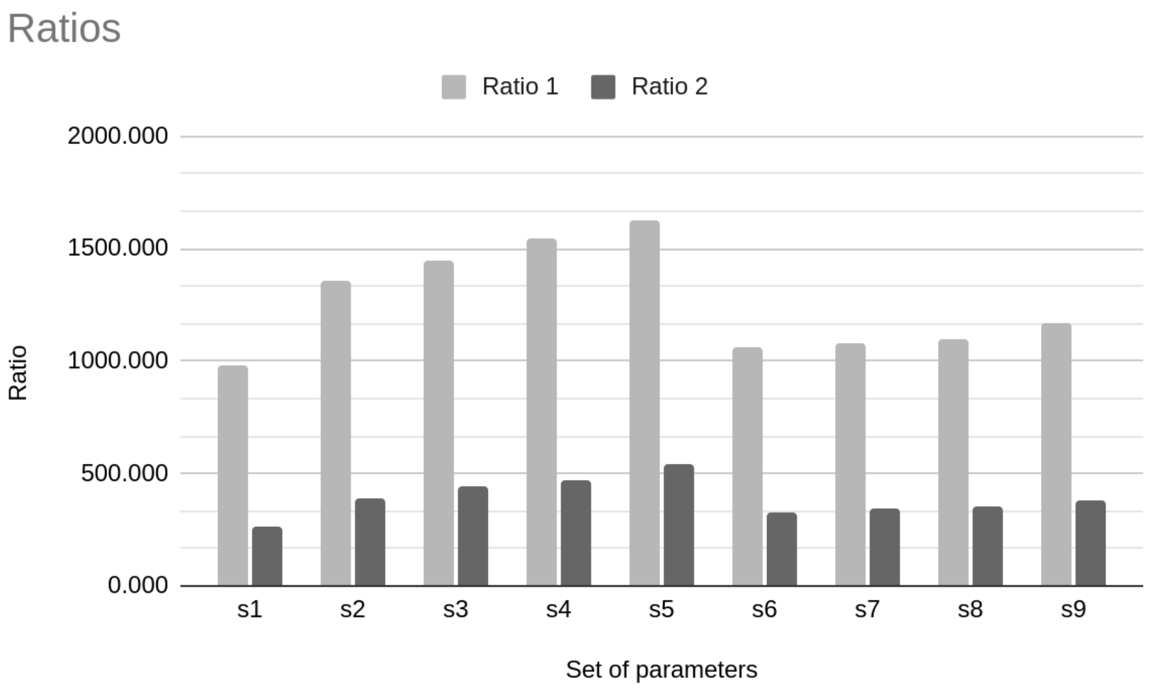

Fig. 5 - Bar graph with calculated ratio for each set of parameters

In Fig. 5 it is possible to see the ratios from each microservices approach plotted side by side for each set of parameters. Looking at the graph, values for both ratios tend to grow as the parameter identifier increases, except on s6 where they abruptly but continue growing after, just at a different pace. To interpret these results it's important to understand exactly what is happening with the parameters. From s1 to s5, the number of pings per worker is fixed and the number of neighbours per neighbourhood is increasing 10 at a time. And from s6 to s9, the number of workers per neighbourhood goes back to its first value and is fixed there, while the number of pings per worker is increasing. The big drop in s6 followed by a differently paced growth also makes sense since it represents a “reset” of the parameters followed by a different parameter being changed.

This growth of the ratios suggests that the multithreading implementation scales at a different pace than both microservices implementations, otherwise the ratio would be constant.

Scaling

To further investigate how each technique scales, the execution time mean was plotted in the y-axis against the parameter being changed on the x-axis. This gives 4 different graphs: multithreading implementation changing the number of workers per neighbourhood, microservices implementations varying the number of workers per neighbourhood, multithreading incrementing the pings per worker, and microservices also incrementing the pings per worker.

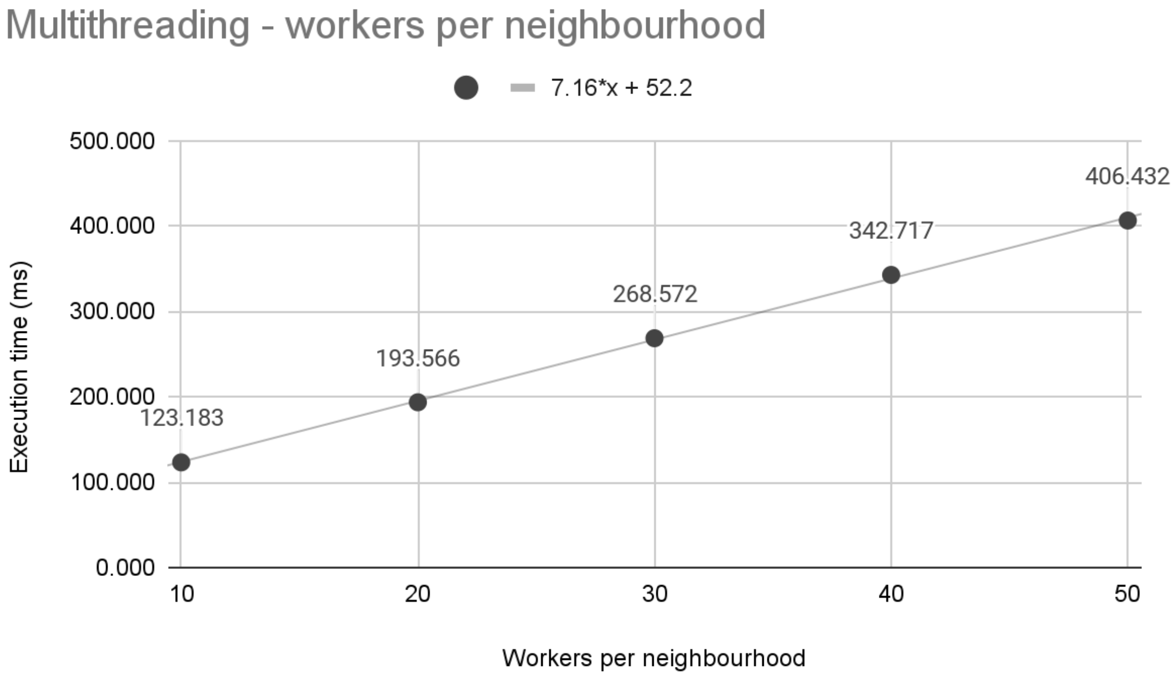

Fig. 6 - Plot of execution time for different number of workers per neighbourhood - multithreading implementation

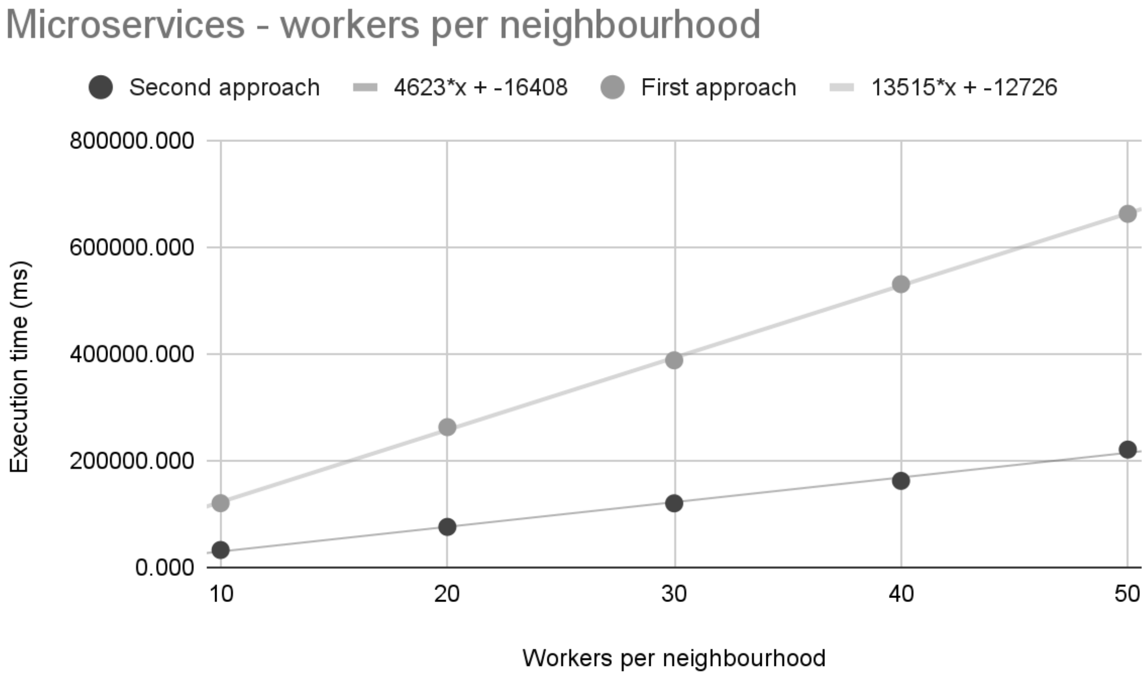

Fig. 7 - Plot of execution time for different number of workers per neighbourhood - second approach to microservices implementations

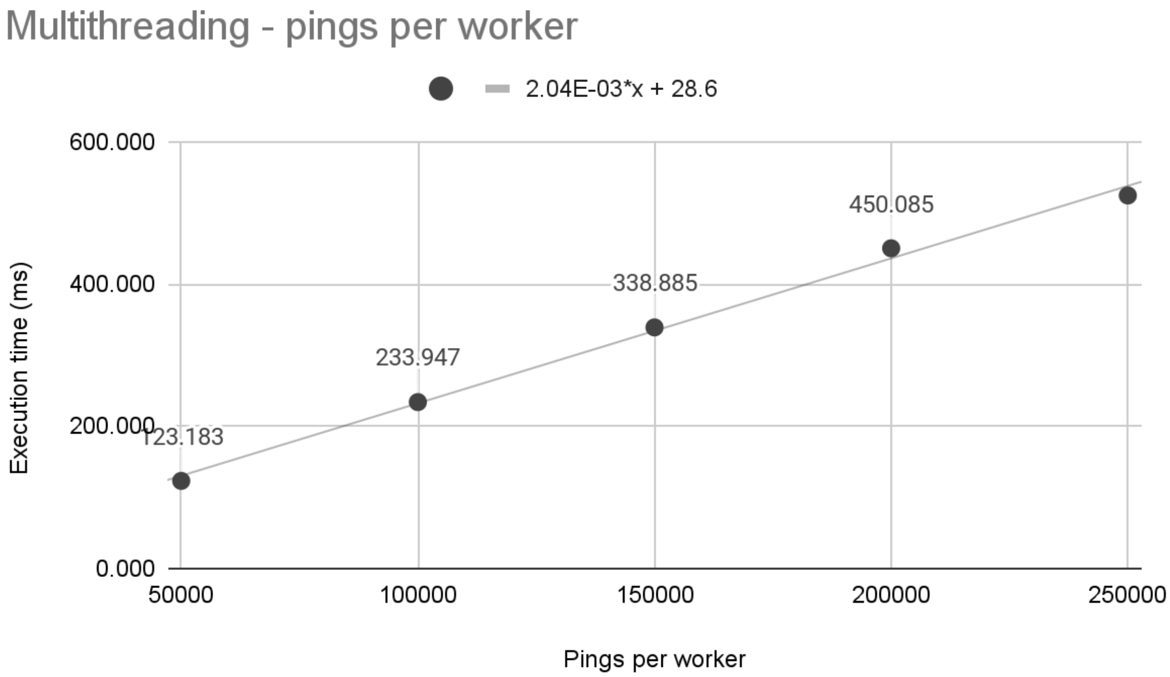

Fig. 8 - Plot of execution time for different number pings per worker - multithreading implementation

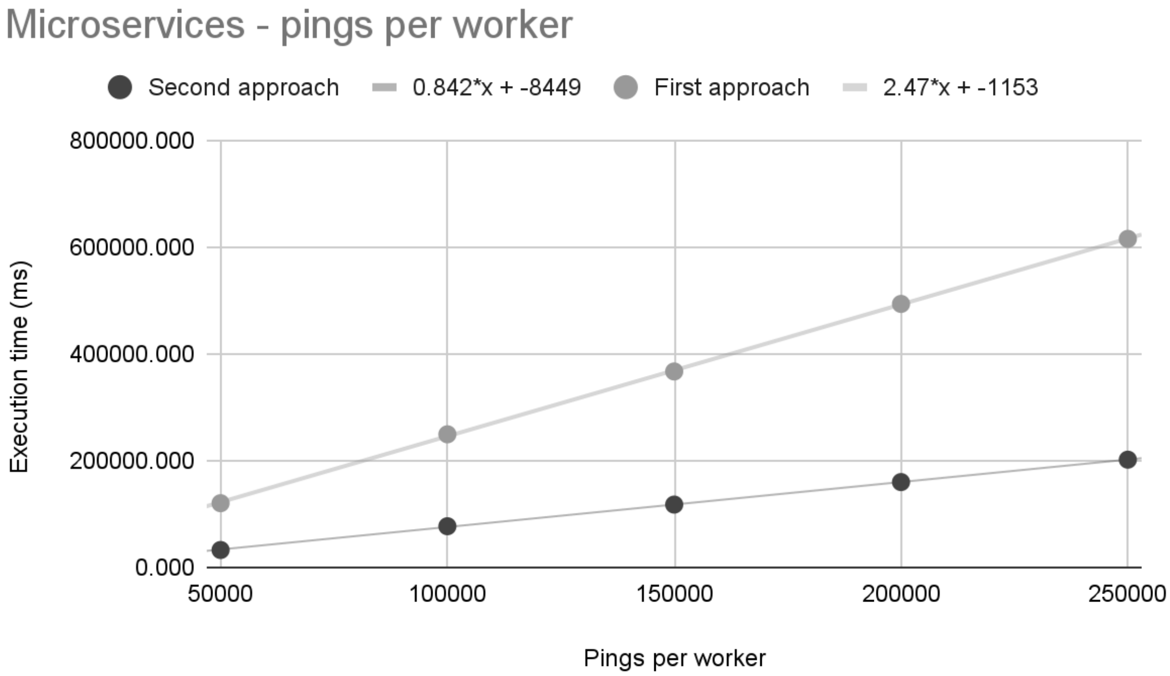

Fig. 9 - Plot of execution time for different number pings per worker - second approach to microservices implementations

All the graphs suggest that the execution time scales linearly with the parameter being changed. This is not surprising given that the benchmark algorithm's time complexity is O(n). Even though both techniques seem to scale very similarly it is possible to compare the slopes of the linear functions. With the help of Google Spreadsheets, the trendlines for the linear functions were calculated. The following list shows the functions obtained by parameter:

- Workers per neighbourhood
  - Multithreading:$y_1 = 7.16x + 52.2$
  - Microservices first approach:$y_2 = 13515x - 12726$
  - Microservices second approach: $y_3 = 4623x - 16404$
- Pings per worker
  - Multithreading: $y_4 = 2.04 * 10^{-3}x + 28.6$
  - Microservices first approach:$y_5 = 2.47x - 1153$
  - Microservices second approach: $y_6 = 0.842x - 8449$

For the first parameter, the slope values are 7.16 for multithreading and 13515 and 4623 for the first and second approach to microservices respectively. For the second parameter, the slopes are 0.00204 for multithreading and 2.47 and 0.842 for microservices. The slope is always much steeper for microservices. This means that even though the scaling happens linearly, multithreading execution times grow a lot slower with increased workloads.

Limitations

There are some limitations associated with the results. One of them is the small number of measures taken for each set of parameters, a bigger sample size would probably give more accurate results. Another one is the unpredictability of running the benchmark in a single computer with an operating system managing a lot of other processes at the same time, making it difficult to achieve controlled consistent performance. Finally, this is a very biased benchmark since the optimized second approach to microservices has no parallel processing at all, just communication and synchronization among services, which is probably not representative of most microservices scenarios.

Conclusion

The results obtained on this experiment all show that in this benchmark, running in a single machine, performs significantly better implemented in a multithreading architecture than in a microservices architecture. Not only the execution times were remarkably slower on multithreading but how these execution times scale with the parameters was also notably better.

It is also possible to note that microservices architecture can be easily implemented with serious inefficiencies that can be found and substituted by a more optimized solution.

In conclusion, running this benchmark with very high communication in a microservices architecture, can perform and scale significantly worse compared to using multithreading instead.

References

1. Channels in GoLang, https://golangdocs.com/channels-in-golang, June 2021
2. Goroutines in GoLang, https://golangdocs.com/goroutines-in-golang, June 2021
3. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. o. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, The NAS Parallel Benchmarks - Summary and Preliminary Results, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5348941, 1991
4. Michael Andersch, Ben Juurlink, Chi Ching Chi, A Benchmark Suite for Evaluating Parallel Programming Models, https://depositonce.tu-berlin.de/bitstream/11303/8009/3/andersch_etal_2011.pdf, 2011
5. Mikael Östlund, Benchmarking Parallelism and Concurrency in the Encore Programming Language, https://www.diva-portal.org/smash/get/diva2:1043738/FULLTEXT02, October 2016
6. The Go Programming Language, https://golang.org, June 2021
7. The State of Microservices, https://www.redhat.com/en/blog/state-microservices, 11 January 2018
8. UNIX domain sockets, https://www.ibm.com/docs/en/ztpf/2020?topic=considerations-unix-domain-sockets, June 2021