# Performance Evaluation of Thread Pool Configurations in the Run-time Systems of Integration Platforms

**Article** *in* International Journal of Business Process Integration and Management · September 2021

DOI: 10.1504/IJBPIM.2021.10048664



**6 authors**, including:

Daniela Lopes Freire
University of São Paulo
**26** PUBLICATIONS **66** CITATIONS
SEE PROFILE

Angela Mazzonetto
**7** PUBLICATIONS **2** CITATIONS
SEE PROFILE

Rafael Z. Frantz
Universidade Regional do Noroeste do Estado Estado do Rio Grande do Sul (UNIJUÍ)
**118** PUBLICATIONS **504** CITATIONS
SEE PROFILE

Fabricia Roos-Frantz
Universidade Regional do Noroeste do Estado Estado do Rio Grande do Sul (UNIJUÍ)
**91** PUBLICATIONS **424** CITATIONS
SEE PROFILE

# Performance Evaluation of Thread Pool Configurations in the Run-time Systems of Integration Platforms

**Daniela L. Freire***

Institute of Mathematics and Computer Sciences, University of São Paulo at São Carlos, São Paulo, Brazil E-mail: danielalfreire@icmc.usp.br

**Angela Mazzonetto, Rafael Z. Frantz, Fabricia Roos-Frantz, Sandro Sawicki**

Department of Exact Sciences and Engineering, Unijuí University Ijuí-RS, Brazil E-mail: {angela.mazzonetto@sou.unijui.edu.br}, {rzfrantz, frfrantz}@unijui.edu.br

**Vitor Basto-Fernandes**

Instituto Universitário de Lisboa (ISCTE-IUL) ISTAR-IUL Lisbon, Portugal E-mail: vitor.basto.fernandes@iscte-iul.pt

**Abstract:** Companies' software ecosystem - composed of local applications and cloud computing services - is made up by the connection of integration platforms and applications. Run-time systems are arguably the most considerable components for integration platforms performance. Our literature review has identified that most integration run-time systems adopt a global pool as configuration for threads. However, it is possible to configure local thread pools to increase the performance of run-time systems. This article brings a comparison between two configurations of thread pools simulating the execution of a real integration problem. Results show that the execution performance through the local pool configuration exceeds the performance through the global pool in high workload scenarios. These results were reviewed by rigorous statistical analysis.

**Keywords:** application integration; multi-thread; optimisation; run-time system; simulation.

**Biographical notes:** Daniela L. Freire received her PhD in Mathematical Modelling at Unijuí University.

Angela Mazzonetto is a PhD student in Mathematical and Computational Modelling at Unijuí University.

Rafael Z. Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of Unijuí University and leads the Applied Computing Research Group since 2013. He was awarded a PhD degree in Software Engineering by the University of Seville, Spain.

Fabricia Roos-Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of Unijuí University. She received her PhD in Software Engineering from the University of Seville, Spain.

Sandro Sawicki is a Professor and researcher at the Post Graduation Program on Mathematical Modeling at Unijuí University. He received his PhD and M. Sc.

in Computer Science from Federal University of Rio Grande do Sul, UFRGS, Brazil, B.Sc. in Computer Science at the Unijuí University, Brazil.
Vitor Basto-Fernandes works with the University Institute of Lisbon (Portugal) where he is currently assistant professor at the Department of Information Science and Technology.

## 1 Introduction

With the advent of cloud computing, many applications that used to operate on-premises in companies have been offered as cloud services. In this scenario, software ecosystems have become even more heterogeneous by increasing the need for integration amongst them, so that they work synchronously and support business processes However, many of these applications still need to be adapted to operate in a cloud computing context, so they can keep or even improve the same performance they once achieved by running locally (Harman et al., 2013; Linthicum, 2017; Ritter et al., 2017). The main user requirement in the software is the performance. It is probably the most considered property in the software quality field (Bogado et al., 2014).

The integration Platform as a Service (iPaaS) is an example of software provided through cloud service. Adopting iPaaS lowers the cost of maintenance and operations in local integration platforms and it is a good solution for small and mid-sized businesses who need to integrate their processes without increasing costs (Brahmi and Gharbi, 2014). Integration platforms are specific tools behind the design, execution and monitoring of integration processes. Such processes orchestrate applications and services such that data can be synchronised and new functionalities are developed on top of what currently exists, with minimal impact (Frantz et al., 2016). Integration platforms frequently supply a domain-specific language, a development toolkit, a testing environment, a monitoring tool, and a run-time system. DSL' level of abstraction makes the understanding of a problem a whole lot easier, besides creating conceptual models for the integration process. The development toolkit is nothing but a set of tools for the process implementation, as in the conversion of the conceptual model into an executable code. The testing environment favors performing tests in the entire integration process, to mitigate or eliminate possible inadequacies in the implementation. The monitoring tool checks the integration process operation and detects run-time errors. In order to execute such integration processes, all the necessary support must be provided by the run-time system. An integration process performs a workflow composed by different atomic tasks, which will process encapsulated data in messages flowing through the process. Threads, which are grouped in thread pools in the run-time system, execute tasks of the integration system.

Our literature review has found that most integration run-time systems adopt a single thread pool, which is usually efficient with a number of messages reduced. Nonetheless, when such number increases, the total execution time of the processes also increases (van der Weij et al., 2009). The addition of threads is the most common approach, but this alternative escalates financial costs of enterprises, especially when it comes to cloud computing - whose billing system is proportional to the consumption of computational resources. Besides, there is a saturation point from which the increase in the number of threads will not bring any benefit, and may even degrade the run-time system performance (Suleman et al., 2008; Lee et al., 2010; Lorenzon et al., 2016).

Recent research proposes an optimal configuration to local thread pools by using Particle Swarm Optimisation (PSO) (Freire et al., 2019). However, this study did not compare the traditional configuration pool with the proposal configuration - the former employs a global thread, while the latter local thread pools. Our article extends this previous work (Freire et al., 2019) to fill this gap and help researchers and practitioners with arguments, which will allow them to choose the more adequate thread pool configuration model, considering different scenarios of message processing. We simulated the behavior of a real integration process problem submitted to high workloads and executed both at global and local thread pools. According to results, local thread pools optimized provided lower average time to perform messages when compared to the global thread pool, that difference being directly propositional to workload. For that, we used rigorous statistical techniques to validate the results.

The rest of this article is organized as follows: Section 2 discusses related work; Section 3 provides background information on the run-time system and the thread pool configuration models; Section 4 formulates the performance metric and objective function; Section 5 describes a study case; Section 6 reports our simulation; and, Section 7 presents our conclusions.

## 2 Related Work

In this section, we discuss related work that simulates and reports experiences with performance analysis and thread pool configuration. It is possible to divide these works according to their goals, for instance, performance of power transmitting systems, performance of web-based software systems, load balancing and fault detection.

Braun and Krus (2016) suggested an automatic approach for parallel continuous-time simulation of tightly coupled power transmitting systems, more commonly addressed to solve industrial problems. They came up with an automated algorithm for partitioning the distributed system models for multi-core processors with proper load balancing; not only this, but also a synchronisation algorithm which runs several simulation threads at the same time. Ágnes Bogárdi-Mészöly and Rövid (2016) proposed mathematical models which consisted in difference equations by subspace identification. Such equations could predict the performance of web-based software systems, through behaviour simulation from thread pools and queued requests. Pasha et al. (2017) presented a simulation framework for code-level energy estimation. The framework has an instruction-level power estimator module that anticipates average power consumption from individual machine instructions. It also has a high-level energy estimation module that parses the written assembly code for a target processor, and estimates the energy spent while considers control dependencies and inherent data. Ahmad et al. (2018) came up with an approach to test the performance of web applications with the worst sequence path of user interactions, based on a workload model which demands high resource utilization on the system under test. Through an analytical and empirical analysis of its performance, they presented an exact and an approximate method for detecting the less suitable path in the workload model. Altmann et al. (2018) scrutinised the value creation of users and providers' software service platforms at different interoperability levels, in an economic point of view. Three questions were raised from their quest: how investments in interoperability and portability could impact costs; how to enable a cost-effective service integration; and how would they create value and design new ways to optimize investments for platform providers. Bahadur et al. (2018)

presented a distributed framework that tunes thread pool systems based on request arrival rate, and balances the load between nodes of distributed systems. Besides, the framework adjusts the thread pool size when its overload control mechanism detects throughput fall caused by an increase on requests. Through the use of hierarchical discrete-event models, Tarvo and Reiss (2018) exhibited their approach for design performance models: each tier of the model would simulate an impact factor in the programs performance. Plus, the interaction among model tiers would simulate the mutual influence of those factors on performance. Jeon and Jung (2018) suggested another path: the optimisation of thread pool management by curbing the number of received packets in the handler thread pool. In order to increase speed performance in the requests processing, they adjusted the number of threads and throughput according to the number of packets, to the queue capacity, and the time of retransmission for each IoT networks packet. Sriraman and Wenisch (2018) introduced a system named uTune, which has two features: a structure that abstracts the implementation of the application code threading model and an automatic load adaptation system that restricts the micro-service's tail latency by exploring latency trade-offs. They studied uTune for Modern On-Line Data Intensive applications and stated that there was an up to 1.9x improvement in tail latency compared to state-of-the-art static threading options and adaptation techniques. Stetsenko and Dyfuchyna (2019) devised a multithread algorithm model through Petri-object simulation technology coming from the stochastic Petri net and object-oriented approach. The model was designed out of a thread pool, just to show the relation among algorithm complexity, computing resources and thread pool parameters. Casini et al. (2019) propounded a model for parallel real-time tasks carried out by thread pools. As to meet precedence constraints, this model has blocking synchronisation mechanisms. By comparing the analysis approaches with earlier works, the work evaluates possibilities of scheduling as a result of concurrency reduction. Berned et al. (2020) exposed a generic methodology to tune the right number of threads with the right application, in which they suggest the use of learning algorithms in static strategies and the inference from the execution behaviour of parallel applications. By using the number of threads found by the dynamic methodology learning algorithm, it was possible to compare both performance and energy consumption of the execution in applications.

The purpose of our article differs from others for it aims to evaluate performance of run-time systems by using two distinct thread pool configuration models: the global and the local thread pools. In Table 1, we summarized all works related to performance analysis and thread pool configuration.

## 3 Background

We present a brief summary of integration processes and run-time systems from integration platforms. After that, we describe the main thread pool configuration models: the global and the local one.

### *3.1 Integration Process*

An integration process is a computational program which allows the exchange of data and functionalities amongst a set of applications $A = A_1, A_2, ..A_k$. Its conceptual model is a workflow, composed by tasks $T = T_1, T_2, ..T_n$ connected by «communication channels». Data, wrapped as «messages», flow through the workflow. A message has a header and a

**Table 1** Summary of the features approached by the related work

| Research work | Research area | Objective |
|---|---|---|
| Braun and Krus (2016) | Power systems | Load balancing and synchronisation |
| Ágnes Bogárdi-Mészöly and Rövid (2016) | Software system | Prediction performance |
| Pasha et al. (2017) | Power systems | Prediction of power consumption |
| Ahmad et al. (2018) | Software system | Prediction performance |
| Altmann et al. (2018) | Software system | Interoperability |
| Bahadur et al. (2018) | Distributed system | Load balancing |
| Tarvo and Reiss (2018) | Software system | Prediction performance |
| Jeon and Jung (2018) | IoT networks | Increase the performance |
| Sriraman and Wenisch (2018) | Software system | Increase the performance |
| Stetsenko and Dyfuchyna (2019) | Software system | Prediction performance. |
| Casini et al. (2019) | Software system | Evaluation of schedulability |
| Berned et al. (2020) | Software system | Energy consumption |
| [Our proposal] | EAI | Evaluation of performance |

body. The former carries data custom properties, and the latter, payload data. A message is transformed into one or more messages in its processing in the workflow. A message usually can follow more than one path into a workflow. Figure 1 depicts a conceptual model from an integration process. Small rectangles (identified by the letter «T») represent tasks. Arrows linking tasks represent communication channels. Rectangles outside the integration process (identified by the letter «A») represent applications. Dark small rectangles and arrows highlight one possible path to the aforementioned process.

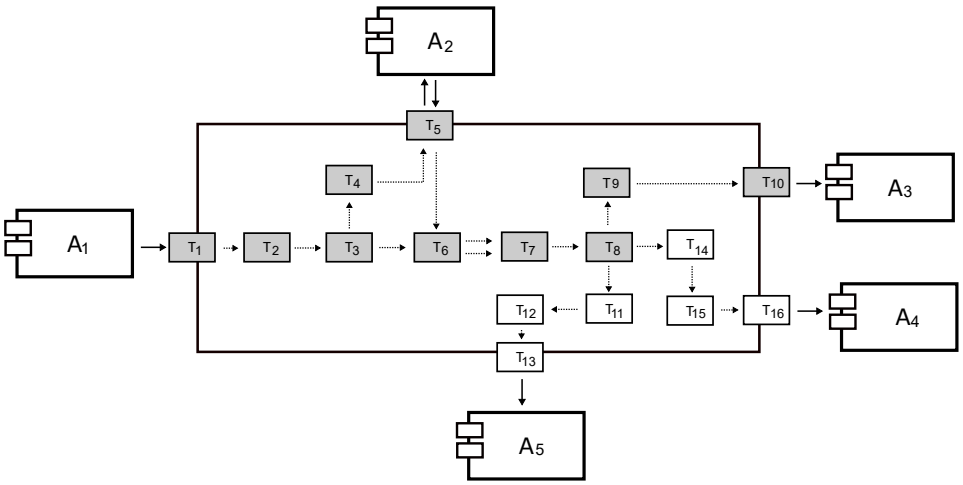



**Figure 1** Integration process conceptual model.

Each task implements an integration pattern, which represents an atomic operation that transforms, filters, splits, joins and forwards messages. As messages arrive into a task by one or more inputs, they likewise leave according to the operation that they implement. Tasks are arranged in order of dependence to be executed. Therefore, a message can only be processed by a task after it has already been processed by all the predecessor tasks of a path. As soon the message is processed by the task, it is written to the communication channel that connects this very task with the next one on the path. The workflow usually

has parts that can be executed in parallel, obeying the order of dependence in an integration process.

### *3.2 Run-time System*

The run-time system is in charge of an integration process execution, and it is often composed by a scheduler, a task queue, a thread pool, and monitors. The schedule is the key element, once it guides and conducts activities from other elements. The task queue function is to keep tasks waiting for threads, and then execute them. While thread pools are task-performing thread groups, monitors control the logging system and frequency, so they can notify warnings and errors.

The execution model settles the way to execute integration processes, sequencing tasks and allocated threads. We address through our article the execution task-based model responsible for task instances, i.e. tasks assigned to execution by a thread. In this model, a task is considered ready to be executed when there are messages in all their inputs. However, when there is no available thread, the task waits in a queue. When there is an available thread, it iteratively polls the task queue and selects a task instance to execute it. The thread pool configuration model of a run-time system determines the way threads are grouped. We approached both global and local thread pools.

In the configuration model of the global thread pool, there is a single task queue and a single thread pool to all tasks. All threads are responsible for executing the instances of the first task; they process incoming messages in the first task input channel. Only when all messages finish their processing they begin to be processed by the subsequent task.

There is a task queue and also a thread pool for every task In the configuration model of the local pool. In the approach of Freire et al. (2019), the number of threads is determined by an optimization algorithm in order to minimize the processing time of incoming messages per unit time. Each thread pool is responsible for the execution of instances of a single task. In this way, the initial load of the message is arriving in the channel from the first task and it is processed in its pool. Therefore, when a message finishes being processed by a task, it can be processed in the next one, since there are threads available for the dedicated pool to execute the next task.

## 4 Objective Function

Execution and processing time are generally used as performance metrics for systems. In this section, we describe mathematical models for the metrics of message processing in an integration process. After that, we define the objective function used to minimize such times. The processing time $TP_{t_i}$ of a task $t_i$ of the integration process is calculated by the sum of its execution time $TE_{t_i}$ with its waiting time in the queue $TF_{t_i}$, c.f. Equation 1.

$$TP_{t_i} = TE_{t_i} + TF_{t_i} \tag{1}$$

Makespan is the most extended time interval between the start and the end of a message in an integration solution (Ali et al., 2003; Blythe et al., 2005). We propose an analytical model for makespan of integration processes, in each model from the thread pool configuration, considering the non-variation of tasks processing time. For the global thread pool model, makespan can be calculated by Equation 2. The makespan using global thread

pool model can be calculated by Equation 3, where $to, t_{messages}$ represents the total number of messages, $tot_{threads}$ is the total number of threads, $Tot_{tasks}$ is the total number of tasks, and $\alpha$ is a constant equals one, when the mod of the integer division of the total number of message by the total number of threads is greater than zero, and, otherwise, $\alpha$ equals zero.

$$makespan_{global\ pool} = \left(\frac{tot_{messages}}{tot_{threads}} + \alpha\right) \cdot \sum_{i=1}^{tot_{tasks}} TP_{t_i} \qquad \alpha = \begin{cases} 1, & \text{if } \frac{tot_{messages}}{tot_{threads}} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

$$makespan_{local\ pool} = \sum_{i=1}^{tot_{tasks}} \left(\frac{tot_{messages}}{tot_{threads_{(t_i)}}} + \alpha_i\right) \cdot TP_{t_i} \qquad \alpha_i \begin{cases} 1, & \text{if } \frac{tot_{messages}}{tot_{threads_{(t_i)}}} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

We define the makespan average as the division of the current makespan by the total number of messages, cf. Equation 4.

$$\overline{makespan} = \frac{makespan}{tot_{messages}} \tag{4}$$

The objective function seeks to minimize makespan average and this formulation is represented by Equation 5:

$$Minimize\ \{\overline{makespan}\} \tag{5}$$

## 5 Study Case

In this section, we describe the case study used in our experiment. It involves a real-world integration problem whose objective is to improve the call center application at Unijuí University, by the automation of expenses of personal phone calls through university's phones. Every call has an access code that identifies who the employee is. The call is registered to future debt in the employees' paychecks. Workers can be notified about their calls and subsequent charges both by e-mail and short message service (SMS).

### *5.1 Software Ecosystem*

The software ecosystem consists of five applications designed with no concerns regards integration; this ecosystem's data layer carries out the interaction with applications. Figure 2 shows the integration process and the following applications: Call Centre, Human Resources System, Payroll System, Mail Server, and SMS Notifier. Phone calls made by each employee from the university phone is recorded by the Call Centre. The code can also correlate phone calls with information from the Human Resources System and the Payroll System. The HR System supplies personal data regarding staff members, while the Payroll System computes their salaries. The Mail Server and the SMS Notifier notify employees about their expenses. The former provides e-mail service and the latter offers short message system services. There are 16 tasks identified with $T_i$, where $i$ ranges from 1 to 16. The $T_1$ is an input task, in which messages are firstly processed. The $T_5$ is a task that solicits and receives data from the Human Resources System application. The $T_10$, $T_1$ and $T_16$ are output tasks, in which messages are lastly processed.

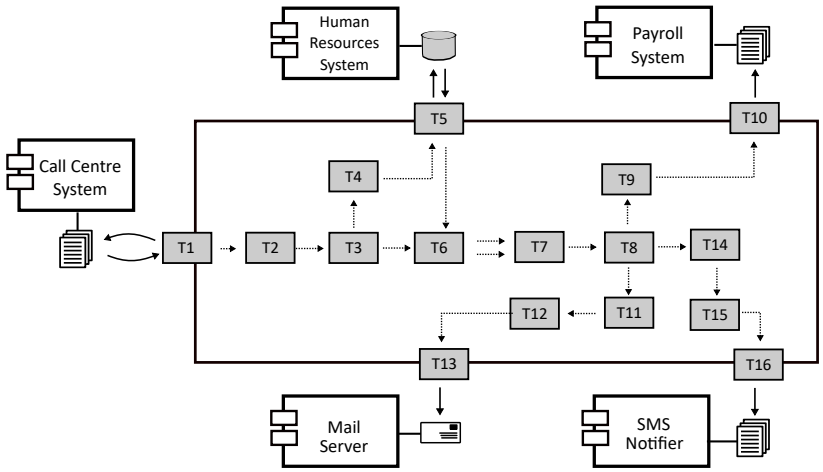



**Figure 2** Real-world integration process.

### *5.2 Computational Model of Simulation*

To simulate the execution with both thread pool configurations (global and local), we implemented two computer programs, whose source codes are publicly available for download [a]. The first program simulates the execution of tasks through a global thread pool and the second one simulates the execution of the same tasks through the optimal local thread pool model. For the global thread pool, we have implemented an algorithm that calculates makespan through Equation 2, afterwards, we calculated makespan average with Equation 4. For optimal local thread pool model, we implemented the algorithm inspired by the PSO metaheuristic, which finds the best thread configuration for local pools (Freire et al., 2019), calculates makespan through Equation 3 and the makespan average with Equation 4. In both cases, the number of threads is the same, i.e., the number of threads in the global thread pool is equal to the total sum of threads in local pools. Input parameters for the two programs are: (1) total number of threads, (2) total number of incoming messages (workload), and (3) the vector which contains the tasks processing times. In this last input parameter (3), the number of columns in the vector defines the number of tasks to be performed, and the column index denotes the order of execution.
Programs generate an array, which displays the final processing times of each message to every task of the flow. The array row index represents the executed task, and the column index represents the message. Makespan average is calculated by the arithmetic average from the final message times in the last flow task. The program graphically shows the sequence of message processing in the tasks and delivers makespan average. In the case of the optimal local thread pool model, it also returns the pools optimal configuration.

## 6 Experimental Results

We will present in this section the experiment, which compares two implementations of algorithms concerning the execution performance in an integration process, by both global

[a] https://github.com/gca-research-group/simulation-thread-pool-config

and local thread pool configuration models. We first describe the environment where the experiment was carried out to then present the scenarios used, and the observed variables. We end up reporting and discussing the results obtained by each model.

Simulation in multithreaded architectures allows the evaluation of multithreading techniques and their influence on performance as well as the relationship between efficiency and the characteristics of different numbers of messages (Vlassov et al., 1997). We used procedure protocols for controlled experiments in the field of engineering studies as indicated by (Jedlitschka and Pfahl, 2005; Wohlin et al., 2012; Basili et al., 2007). Besides, we followed guidelines from the literature, which claim that statistical theory must be used to validate results from experiments on performance (Georges et al., 2007). The statistic suitably deals with non-determinism factors present in run-time systems (Frantz et al., 2011). Thus, we utilized ANOVA (Wilkinson and Rogers, 1973) and Scott & Knott (Scott and Knott, 1974) statistical techniques for the validation of our proposal. Every step of the experiment and its validation will be detailed in the following sections.

### *6.1 Hypothesis and Research Questions*

Our experiment answers the following research question:

**RQ:** In a great number of messages, is execution performance of integration processes better by the usage of the local thread pool configuration, instead of the global model?

Our hypothesis shall be confirmed or refuted by the experiment, respectively:

**H:** In a great number of messages, execution performance of integration processes by the use of an optimal local thread pool model is better than using the global model.

### *6.2 Experimental Environment*

Experiments were conducted on a machine containing 16 processors Intel Xeon CPU E5-4610 V4, 1.8 GHz, 32GB of RAM, and Windows operating system Server 2016 Datacenter 64-bits. The software Matlab (Leonard and Levine, 1995), version R2018, was the one chosen to create and execute the algorithms. The software Genes (Cruz, 2006) version 2015.5.0 was responsible to process the descriptive statistics, while both ANOVA and Scott & Knott techniques were employed to measure makespan.

### *6.3 Variables*

These are the independent variables:

`Number of threads`. Represents the total of threads that can be distributed through thread pools. That variable value was 100 threads.

`Number of messages`. The total of incoming messages (workload). Values for this variable were $10^2$, $10^3$, $10^4$, $10^5$, and $10^6$.

`Thread pool models`. Models for thread pool configuration. Values for this variable were global and optimal local.

Dependent variables are:

`Makespan average`. Corresponds to the average time consumed by a message when going through the longest path of the integration process.

### 6.4 Data Configuration and Execution

The execution of the algorithms was carried out through the longest path in the integration process, as shown in Figure 2. It aims to evaluate how processing performs under an increased number of messages triggered by the worst-case scenario. Table 2 presents the result of the times for each task in milliseconds (ms). Execution was achieved from the actual implementation of the integration process, supported by the literature (Haugg et al., 2019).

**Table 2** Tasks time information.

| Task | Execution Time | Waiting Time | Processing Time |
|---|---|---|---|
| $t_1$, $t_9$ | 0.005 | 2 | 2.005 |
| $t_2$, $t_{11}$, $t_{14}$ | 0.003 | 1 | 1.003 |
| $t_3$, $t_8$ | 0.553 | 1 | 1.553 |
| $t_4$, $t_7$, $t_{10}$, $t_{12}$, $t_{13}$, $t_{15}$, $t_{16}$ | 0.005 | 1 | 1.005 |
| $t_5$ | 0.005 | 4 | 4.005 |
| $t_6$ | 0.004 | 1 | 1.004 |

The literature labelled the experiment a termination simulation, in which the experiment output is expressed as a function from initial conditions. The repetitions method (a number ranging from 20 to 30) is usually adopted for the statistical analysis of results. Since it considers distributions with more extreme values than a regular distribution, this number of repetitions can generate a population mean (Sargent, 2013). We used 200 distinct scenarios, which are summarized according to Table 3. We executed the algorithms by setting input parameters:

- Total number of threads: 100.
- Total number of incoming messages: $10^2$, $10^3$, $10^4$, $10^5$,and $10^6$.
- Processing time vector: [2.005; 1.003; 1.553; 1.005; 4.005; 1.004; 1.005; 1.553; 2.005; 1.005; 1.003; 1.005; 1.005; 1.003; 1.005; 1.005]

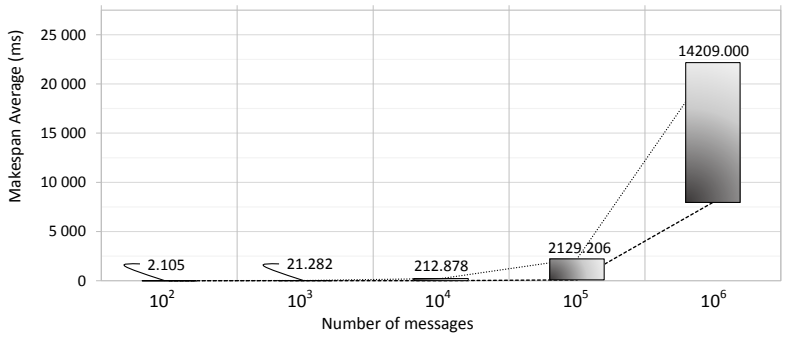
We considered the execution algorithm and the number of tasks in the process. Thus, 10 is considered a reasonable number to the amount of solutions tested by the optimization algorithm from the optimal local thread pool model. For every message, the execution of each algorithm was repeated 20 times. One by one, we collected the makespan average processing time and we measured how long each algorithm took to execution. We exported the data we collected to a spreadsheet application, where we analysed them to build the mathematical model and charts.

**Table 3** Scenarios tested.

| Number of threads: | Number of messages: | Thread pool models: | Number of: repetitions: |
|---|---|---|---|
| «100» | «$10^2$, $10^3$, $10^4$, $10^5$, $10^6$» | «global and local» | «1, 2, ..., 20» |
| 1 | 5 | 2 | 20 |
| **Total of scenarios:** | 1 x 5 x 2 x 20 = | | **200** |

### *6.5 Results*

In this section, we present and discuss the results of our experiment. Line charts show the average of makespan in 20 executions from each algorithm, as shown in Figure 3. The x-axis represents the number of messages, while the y-axis represents makespan average in milliseconds (ms). The rectangles represent the difference in makespan between global and local thread pool configuration. This difference was 2.105 ms for $10^2$ messages, 21.282 ms for $10^3$ messages, 212.878 ms for $10^4$ messages, 2129.206 ms for $10^5$ messages, and 14209 ms for $10^6$ messages. The table below Figure 3 shows the average of makespan measured in each number of messages. In the first line it is possible to see the average of makespan when the global thread pool configuration is employed, while in the second one makespan averages were measured through the optimized local thread pool configuration.




| | | | | | |
|---|---|---|---|---|---|
| Global | 2.216 | 22.169 | 221.69 | 2216.9 | 22169 |
| Local | 0.111 | 0.886 | 8.81 | 87.6 | 7960 |

**Figure 3** Makespan average.

The variance analysis statistical technique allows the evaluation the variations difference in makespan average, which derived from thread pool configuration models, and the ones caused by other random factors, named error. Table 4 presents the variance analysis of makespan average.

From the multiplication of the number of repetitions, by the number of configuration models, we got the results total (as seen in the Equation 6). In our case, $\mathsf{Total\ results} = 20 \cdot 2 = 40$. The total of results minus 1 calculates the total freedom degree, as in Equation 7. In our case, $df_{total} = 40 - 1 = 39$. The degree of freedom of the configuration models is calculated by the number of possible values of configuration models tested subtracting 1, cf. Equation 8. In our case, $df_{models} = 2 - 1 = 1$. We obtained the degree of freedom by calculating the difference between the total freedom degree and the models freedom degree, as in Equation 9. In our case, $df_{error} = 39 - 1 = 38$.

$$\mathsf{Total\ results} = \mathsf{repetitions} \cdot \mathsf{number\ of\ models} \tag{6}$$

$$df_{total} = \mathsf{total\ results} - 1 \tag{7}$$

$$df_{models} = \mathsf{number\ of\ models} - 1 \tag{8}$$

$$df_{error} = df_{total} - df_{models} \tag{9}$$

The variance analysis from makespan average shows the average square of 4412.04 for the thread pool configurations models and 200.83 for error. While the coefficient of variation (cv) was 35.43%, the overall average was equal to 39.99 seconds. Variation coefficient is a standardized measure that shows us the dispersion of a probability or frequency distribution, by displaying the spread of data variability relative to the mean. This measure is capable of comparing results from different works involving the same variable-response, by quantifying the precision of the research (Kalil, 1977). Some parameters can influence the experimental error, such as the number of repetitions and the experimental design. However, whether parameters from experiments are the same, the experiment that has the lower coefficient of variation is the more accurate one (Garcia, 1989).

**Table 4** Variance analysis of the makespan average.

| **Sources of variation** | **Degree of freedom** | **Average square** *Number of messages* | | | | |
|---|---|---|---|---|---|---|
| | | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| Thread pool model | 1 | 44.31 † | 4529.33 † | 453170.98 † | 45335191.48 † | 2018956810 † |
| Error | 38 | 0.0001 | 0.0182 | 1.4809 | 183.96 | 2375092.10 |
| Total | 39 | | | | | |
| Overall average | | 1.16 | 11.52 | 115.25 | 1152.29 | 15064.50 |
| Coefficient of variation (%) | | 0.85 | 1.17 | 1.05 | 1.17 | 10.23 |

† *significant statistical considered by Fisher-Snedecor's with an error of 5%.*

The technique of Scott & Knoot is widely adopted in performance experiments for its simplicity. Nonetheless, literature considers such technique rigorous because there are only relevant distinctions between experimented alternatives - as our case here, the two models of thread pool configuration. The Level of significance is a parameter used in the Scott & Knott algorithms to create groups of averages, which default value is 0.05. This parameter helps to measure the relevance from the difference between the alternatives (Jelihovschi and Faria, 2014). This technique is usually used when the analysis of variance indicates significant statistical between the dependent variable. Such technique also allowed the identification of the effect in the use of every model of thread pool configuration in the makespan average, as in Table 5. For every number of messages, there are two columns: «Average» and «Group». «Average» presents makespan average in the 20 repetitions. «Group» ranks makespan averages, where groups from different letters are statistically different one another. There were two groups «a» and «b». The thread pool model with the highest makespan average was found in group «a» while the thread pool model with the lowest makespan average was found in group «b».

**Table 5** Average of makespan by Scott & Knott technique.

| **Thread pool model** | **Makespan average & Group** *Number of messages* | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $10^2$ | | $10^3$ | | $10^4$ | | $10^5$ | | $10^6$ | |
| Global | 2.2169 | a | 22.169 | a | 221.69 | a | 2216.90 | a | 22169 | a |
| Local | 0.119 | b | 0.8868 | b | 8.8119 | b | 87.69 | b | 7960 | b |

*Error level of 5% by the Scott & Knoot technique.*

### *6.6 Discussion and Comparison*

By analyzing all numbers from the messages, we realized that the makespan average achieved through the local thread pool configuration optimized model is lower than the one obtained through the global thread pool. Figure 3 shows that the greater the number of messages, the greater such difference between makespan averages is. Figure 4 shows the difference of makespan average due to the number of messages and an approximation found by linear regression (represented by a dotted line). This approximation is an exponential function, described by Equation 10, where $tot_{messages}$ is the total number of messages. This equation indicates that this difference exponentially grows in proportion the number of messages. The correlation coefficient, also known as $R^2$, helps to determine the variables degree of linear correlation by considering regression analysis. Thus, the closer $R^2$ is from 1, the exacter the mathematical model will be. For Equation 10, $R^2$ equals 0.9986 indicates that the mathematical model is a reliable approximation.

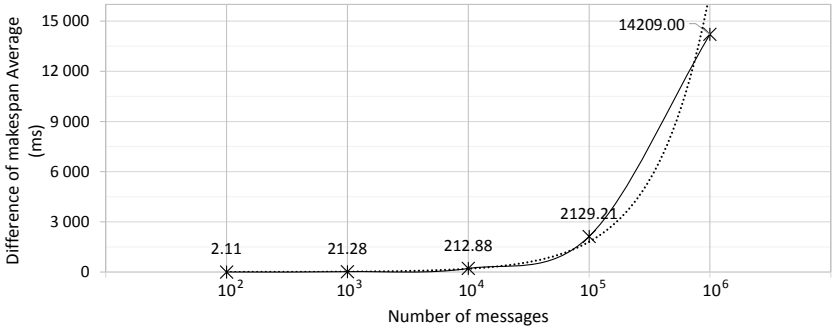



**Figure 4** Difference in makespan average.

$$\overline{makespan} = 0.248\, e^{2.224 \cdot tot_{messages}} \tag{10}$$

The analysis of variance confirmed that the optimized local thread pool generates a significant improvement in makespan average to every tested scenario. Low values for variation coefficients indicated the adequacy and reliability of the experiment. Through Scott & Knott averages comparison technique, the lowest averages of makespan occurred with the optimized local thread pool in every scenario tested, as in Table 5. The largest difference in makespan was 14209 ms when the number of messages was $10^6$. In this table, groups of different letters indicate that there is a statistical difference between them. Makespan average from the global thread pool (group «a») is statistically different from makespan average out of the optimized local thread pool (group «b»).

### *6.7 Threats to Validity*

According to Cruzes and ben Othman (2017), validity threats can happen to any empirical research. We sought to identify possible causes of disturbance in the experiment in order to mitigate such threats. We discussed validity threats by separating them into constructor, internal validity, external validity, and conclusion.

### 6.7.1 Constructor Validity

In order to tackle constructor validity threats, we learned from previous articles and mirrored our experiment on procedures from empirical software engineering, as in the work by Jedlitschka and Pfahl (2005); Wohlin et al. (2012); Basili et al. (2007). All information from the execution environment, plus supporting tools, variables, the execution and data collection was previously collected. After that, the experiment was developed with 200 different scenarios and we employed statistical techniques to validate experimental results.

### 6.7.2 Conclusion Validity

According to Feldt and Magazinius (2010), conclusion validity goes to show that the treatment used in the experiment is the actual cause of the outcome observed. We resorted statistical techniques to get this assurance, and we saw that performance measures observed in our experiment are associated with the number of messages and with the algorithms that were employed.

### 6.7.3 Internal Validity

Internal validity deals with uncertain, unmeasured factors, so that their effects may not impact the outcome of the treatment (Feldt and Magazinius, 2010). Since we performed the experiment with the machine settings on security mode, some factors in the execution time were reduced. We were also disconnected from the Internet during executions, and we used minimal features.

### 6.7.4 External Validity

External validity refers to expand the results beyond the scope of our study (Feldt and Magazinius, 2010). The experiment shall be useful whenever different scenarios are compared with other integration processes, and with various numbers of messages. In order to generalize results in future works, we shall perform our experiment with an extensive data set.

## 7 Conclusion

Enterprises have taken advantage of some services offered by cloud computing, such as the Integration as a Service Platform (iPaaS) - a tool which allows to keep data consistency and synchronism in all applications. The iPaaS represents a new option for small and mid-sized businesses who need to integrate their business processes without increasing their costs Brahmi and Gharbi (2014). However, integration platforms need to be adapted to tackle environments with large volume, velocity, and variety of data from several different sources and types of devices Ritter et al. (2016). Since run-time systems are responsible for running integration processes, they are directly related to integration platforms performance.

The efficiency of a run-time system in integration platforms is directly related to the tasks scheduling algorithm and to the allocation of threads to execute them. An inefficient algorithm leads to an increase in execution time, thus degrading the execution performance of integration processes. This article aims to analyse the behavior of thread pool configuration models at high workloads. Our experiment compared two models of thread

pool configuration: global and local (optimized). First, there is a single thread pool for all tasks, while in the latter, a single thread pool for each task. Not only this, the algorithm also finds the best distribution of threads to local pools, taking into account the task processing time. We used a case study with a real-world problem, in scenarios that considered the variation in the number of messages.

The results showed that in high workload scenarios, the optimized local thread pool model performs better than the global one. This assessment can help save costs for companies, especially for those who use cloud services and the pay-as-you-go charging model, where companies pay for the usage time of services. Whereas we validated the results through ANOVA and Scott & Knott statistical techniques, the models were evaluated by the average of makespan. We summarized the main conclusions from results found in our experiment. We also responded to the research questions in order to validate our hypothesis, coming to the conclusion that:

- When using global thread pool configuration, makespan average can be analytically calculated by Equation 2.
- When using local thread pool configuration, makespan average can be analytically calculated by Equation 3.
- The optimized local thread pool configuration generated lower makespan average than the global thread pool configuration.
- An exponential equation represents the difference of makespan average between two models of thread pool configuration, due to the number of messages. Therefore, makespan average tends to be infinite when the number of messages is huge.
- The results from the experiment are valid, according to statistical techniques from ANOVA and Scott & Knoot.

Results of the experiment confirmed our hypothesis, and we answered our research question as following:

- **RQ:** The average of makespan from the optimized local thread pool configuration model in run-time systems was 2.8 times lower than the average of makespan from a global thread pool, with a workload of $10^6$ messages. Thus, the execution performance of integration processes is better when using the optimized local thread pool.

## 8 Acknowledgements

This work was supported by the Brazilian Co-ordination Board for the Improvement of University Personnel (CAPES), by the National Council for Scientific and Technological Development (CNPq) under grant 309315/2020-4, and the Research Support Foundation of Rio Grande do Sul (FAPERGS) under grant 17/2551-0001206-2.

## References

Ahmad T, Truscan D, Porres I (2018) Identifying worst-case user scenarios for performance testing of web applications using markov-chain workload models. Future Generation Computer Systems 87:910–920

Ali S, Maciejewski AA, Siegel HJ, Kim JK (2003) Definition of a robustness metric for resource allocation. In: Parallel and Distributed Processing Symposium (PDPS), pp 10–21

Altmann J, Ángel Bañares J, Petri I (2018) Economics of computing services: A literature survey about technologies for an economy of fungible cloud services. Future Generation Computer Systems 87:828 – 830

Bahadur F, Umar AI, Khurshid F (2018) Dynamic tuning and overload management of thread pool system. International Journal of Advanced Computer Science and Applications 9:444–450

Basili VR, Rombach D, Kitchenham KSB, Selby D, Pfahl RW (2007) Empirical Software Engineering Issues. Springer Berlin/Heidelberg

Berned G, Rossi FD, Luizelli MC, Beck ACS, Lorenzon AF (2020) Decreasing the learning cost of offline parallel application optimization strategies. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp 144–151

Blythe J, Jain S, Deelman E, Gil Y, Vahi K, Mandal A, Kennedy K (2005) Task scheduling strategies for workflow-based applications in grids. In: Cluster Computing and the Grid (CCGrid), vol 2, pp 759–767

Bogado V, Gonnet S, Leone H (2014) Modeling and simulation of software architecture in discrete event system specification for quality evaluation. Simulation 90:290–319

Ágnes Bogárdi-Mészöly, Rövid A (2016) Performance modeling of web-based software systems with subspace identification. Acta Polytechnica Hungarica 13:27–41

Brahmi Z, Gharbi C (2014) Temporal reconfiguration-based orchestration engine in the cloud computing. In: International Conference on Business Information Systems (ICBIS), pp 73–85

Braun R, Krus P (2016) Multi-threaded distributed system simulations using the transmission line element method. Simulation 92:921–930

Casini D, Biondi A, Buttazzo G (2019) Analyzing parallel real-time tasks implemented with thread pools. In: ACM/IEEE Design Automation Conference (DAC), pp 1–6

Cruz CD (2006) Programa Genes - Estatística Experimental e Matrizes. Editora Universidade Federal de Viçosa

Cruzes DS, ben Othman L (2017) Threats to validity in empirical software security research. In: Empirical Research for Sof. Security, pp 295–320

Feldt R, Magazinius A (2010) Validity threats in empirical software engineering research-an initial survey. In: Int. Conf. on Software Engineering and Knowledge Engineering (SEKE), pp 374–379

Frantz RZ, Corchuelo R, Arjona JL (2011) An efficient orchestration engine for the cloud. In: Int. Conf. on Cloud Computing Technology and Science (CloudCom), pp 711–716

Frantz RZ, Corchuelo R, Roos-Frantz F (2016) On the design of a maintainable software development kit to implement integration solutions. Journal of Systems and Software 111:89–104

Freire DL, Frantz RZ, Roos-Frantz F (2019) Towards optimal thread pool configuration for run-time systems of integration platforms. International Journal of Computer Applications in Technology 60:1–18

Garcia CH (1989) Tabelas para classificação do coeficiente de variação. IPEF

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. ACM SIGPLAN Notices 42:57–76

Harman M, Lakhotia K, Singer J, White DR, Yoo S (2013) Cloud engineering is search based software engineering too. Journal of Systems and Software 86:2225–2241

Haugg IG, Frantz RZ, Roos-Frantz F, Sawicki S, Zucolotto B (2019) Towards optimisation of the number of threads in the integration platform engines using simulation models based on queueing theory. Revista Brasileira de Computação Aplicada 11:48–58

Jedlitschka A, Pfahl D (2005) Reporting guidelines for controlled experiments in software engineering. In: Int. Sym. Empirical Soft. Engineering (ESEM), pp 95–104

Jelihovschi E, Faria J (2014) Scottknott: A package for performing the scott-knott clustering algorithm in r. Tendências em Matemática Aplicada e Computacional 15(1):3–17

Jeon S, Jung I (2018) Experimental evaluation of improved IoT middleware for flexible performance and efficient connectivity. Ad Hoc Networks 70:61–72

Kalil EB (1977) Princípios de técnica experimental com animais. ESALQ

Lee J, Wu H, Ravichandran M, Clark N (2010) Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. ACM SIGARCH Computer Architecture News 38:270–279

Leonard NE, Levine WS (1995) Using MATLAB to analyze and design Control Systems. Benjamin-Cummings Publishing Company

Linthicum DS (2017) Cloud computing changes data integration forever: What's needed right now. IEEE Cloud Computing 4:50–53

Lorenzon AF, Cera MC, Beck ACS (2016) Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. Journal of Parallel and Distributed Computing 95:107–123

Pasha MA, Gul U, Mushahar M, Masud S (2017) A simulation framework for code-level energy estimation of embedded soft-core processors. Simulation 93:809–823

Ritter D, May N, Sachs K, Rinderle-Ma S (2016) Benchmarking integration pattern implementations. In: International Conference on Distributed and Event-based Systems (DEBS), pp 125–136

Ritter D, Dann J, May N, Rinderle-Ma S (2017) Hardware accelerated application integration processing: Industry paper. In: International Conference on Distributed and Event-based Systems (DEBS), pp 215–226

Sargent RG (2013) Verification and validation of simulation models. J of simulation 7:12–24

Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics 30(3):507–512

Sriraman A, Wenisch TF (2018) Tune: Auto-tuned threading for OLDI microservices. In: Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association, Carlsbad, CA, pp 177–194

Stetsenko IV, Dyfuchyna O (2019) Thread pool parameters tuning using simulation. In: International Conference on Computer Science, Engineering and Education Applications (ICCSEEA), pp 78–89

Suleman MA, Qureshi MK, Patt YN (2008) Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. ACM Sigplan Notices 43:277–286

Tarvo A, Reiss SP (2018) Automatic performance prediction of multithreaded programs: a simulation approach. Automated Software Engineering 25:101–155

Vlassov V, Ayani R, Thorelli LE (1997) Modeling and simulation of multithreaded architectures. Simulation 68:219–230

van der Weij W, Bhulai S, van der Mei R (2009) Dynamic thread assignment in web server performance optimization. Performance Evaluation 66:301–310

Wilkinson GN, Rogers CE (1973) Symbolic description of factorial models for analysis of variance. Journal of the Royal Statistical Society Series C 22(3):392–399

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media