Towards optimal thread pool configuration for run-time systems of integration platforms

Daniela L. Freire, Rafael Z. Frantz* and Fabricia Roos-Frantz

Department of Exact Sciences and Engineering,
Unijuí University,
3000 - Universitário, Ijuí, Brazil
Email: dsellaro@unijui.edu.br
Email: rzfrantz@unijui.edu.br
Email: frfrantz@unijui.edu.br
*Corresponding author

Abstract: Companies seek technological alternatives to increase competitiveness, an example, are the integration platforms, that develop integration processes in order to connect functionalities and data from applications that compose software ecosystems. Threads are computational resources of the platforms, responsible for integration processes execution. Thus, the configuration of threads has a direct influence on the performance of platforms. However, this is a challenge faced by software engineers, who do this configuration empirically. Our scientific and technical literature review did not identify a systematic approach to find the ideal configuration, which depends on factors such as workload, hardware and integration process. Thus, it is appropriate to seek alternatives for configuration that provide a positive impact on the performance of the run-time system, increase productivity, and reduce costs. Inspired by the Particle Swarm Optimisation meta-heuristic, this article proposes an algorithm that finds the ideal configuration for local thread pool, minimising the total average processing time to improve the execution of integration platforms. The algorithm was implemented and tested using a real-life integration process and its performance measures show the feasibility and efficiency of our proposal, supported by a rigorous statistical analysis of results.

Keywords: enterprise application integration; optimisation; PSO; particle swarm optimisation; meta-heuristics; multi-thread; makespan; workflow; integration patterns.

Reference to this paper should be made as follows: Freire, D.L., Frantz, R.Z. and Roos-Frantz, F. (2020) 'Towards optimal thread pool configuration for run-time systems of integration platforms', *Int. J. Computer Applications in Technology*, Vol. 62, No. 2, pp.129–147.

Biographical notes: Daniela L. Freire researches on the field of enterprise application integration, meta-heuristics, and runtime systems. She has got a master in software engineering at Center for Studies and Advanced Systems of Recife, in 2013, in Recife (Brazil). She also has experience in the area of interactive digital books, requirements engineering and domain analysis for a family of software. She has more than 17 years of experience in systems development, where she worked as a programmer, systems analyst and project manager.

Rafael Z. Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the Unijui University, Brazil, and leads the Applied Computing Research Group since 2013. He was awarded a PhD degree in Software Engineering by the University of Seville, Spain. His current research interests focus on the integration of enterprise applications and search-based software engineering.

Fabricia Roos-Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the UNIJUÍ University, Brazil. She received her PhD in Software Engineering from the University of Seville, Spain. Her current research interests include software product lines and search-based software engineering.

1 Introduction

Business processes of enterprises are supported by a set of applications that make up their software ecosystem. Application development technologies have been transformed over time and new software services available on the internet have also been incorporated into software ecosystems, making ecosystems more heterogeneous (Manikas, 2016). Such applications need to work together to provide efficient responses for business processes.

Enterprise Application Integration (EAI) is the research field that provides methodologies, techniques, and tools to construct integration processes, enabling applications to share data and functionality to meet requirements of business processes. Integration platforms are specialised software tools that allow software engineers to design, implement, run, and monitor integration processes (Freire et al., 2019a). An integration process implements a workflow composed of distinct atomic tasks that process messages which flow into the process. Hohpe and Woolf (2004) have documented a set of concep l integration patterns that have inspired the development of open-source integration platforms. Such open-source integration platforms have also followed the Pipes-and-Filters architecture (Alexander et al., 1977). In an integration process, pipes represent message channels and filters represent atomic tasks that implement a concrete integration pattern to process data encapsulated within the messages. Amongst the state-of-the-art open-source integration platforms, which adopt the integration patterns and the Pipes-and-Filters architecture, are: Fuse (Russell and Cohn, 2012a), ServiceMix (Konsek, 2013), Petals (Surhone et al., 2010), Jitterbit (Russell and Cohn, 2012b), WSO2 ESB (Indrasiri, 2016), and Guarana (Frantz et al., 2016). Typically, these platforms provide a domain-specific language, a development toolkit, a run-time system, and monitoring tools. The domain-specific language enables the description of conceptual models for integration processes. The development toolkit is a set of software tools that allow the implementation of integration processes, i.e., transforms a conceptual model into executable code. The run-time system is the component responsible for running the integration processes (Freire et al., 2019b). Monitoring tools are used to detect failures that may occur during the execution of an integration process.

The tasks that compose an integration process are executed by available threads in the run-time system. Threads are the smallest sequence of programmed statements that can be managed by the run-time system. Threads are grouped in thread pools that are generally configured in two ways: Global thread pool and Local thread pool. In the former, there is one single thread pool to execute every task of the workflow. In the latter model, there are multiple thread pools, each of them executing one task of the task flow. The performance that the execution of an integration process is able to achieve, in terms of message processing per unit of time, is directly related to the run-time system.

Typically, in order to achieve the desired performance, software engineers increase the number of threads in the run-time system. This strategy has generally an initial positive impact on execution performance, but it can lead to degradation, causing a lower message processing per unit of time. This degradation is due to the time spent by the operating system to manage the context change amongst threads and their competition over the resources of the system (Suleman et al., 2008; Lorenzon et al., 2016; Liu et al., 2018). Besides, this increased performance is proportional to an increase in financial costs required to purchase hardware or to hire cloud services with greater processing power. In the case of cloud service, the charging model is pay-as-you-go, in which companies pay by the number of computing resources consumed (Buyya et al., 2009). Linthicum (2017) claims that the integration platforms need to be re-engineered to ensure they are suitable for cloud deployment, to take advantage of the scalability provided and to reduce costs by optimising computational resource usage.

In order to provide adequate performance concomi-tantly with the threads number constraint, it is necessary to find its optimum distribution in local thread pools, using the least total number of threads to perform tasks in an integration process. This is a challenge for software engineers who indicate these numbers of threads according to their practical knowledge. If these numbers are high, shared resources, such as cache capacity or memory bandwidth, can quickly saturate, thus degrading performance; in contrast, if these numbers are low, the integration processes execution becomes inefficient (Lee et al., 2010).

Recent researches have tackled task scheduling in other domains. In the distributed systems field, Ghosh and Das (2018) proposed Particle Swarm Optimisation (PSO) meta-heuristic-based algorithm and Touzene et al. (2019) proposed mixed-Integer Linear Program-based algorithm. Zhang et al. (2018) proposed an algorithm to deal with a class of job-shop scheduling optimisation problems. In the cloud computing field, Verma and Kaushal (2015) and Milani and Navin (2015) proposed PSO-based algorithms. However, there is a lack of research in EAI field. In this article, we propose an algorithm based on the PSO, which provides the optimum or near optimal configuration for the thread number in every local thread pool of run-time systems. The proposed algorithm was validated in a real-world integration process. The results show the efficiency of our algorithm to find a configuration of local thread pool that minimises the average total time of message processing, obeying to the restriction of the total threads number. The proposed algorithm contributes to increase the performance of run-time systems, achieving a higher number of processed messages.

The rest of this article is organised as follows: Section 2 discusses related work; Section 3 provides background information on the run-time system and the local thread pool execution model; Section 4 formulates the configuration of local thread pools problem; Section 5 exposes the proposed optimisation algorithm; Section 6 reports our proposal validation; and, Section 7 presents our conclusions.

2 Related work

In this section, we gather works in different research fields, regarding performance optimisation, which have adopted meta-heuristics to deal with the configuration of computational resources in order to increase the performance of applications execution by minimising the makespan. Makespan is a performance metric, defined as the total execution time of an application or process for a given message (Canon and Jeannot, 2007).

Pandey et al. (2010) aimed to minimise the running cost of a single workflow while balancing tasks on available resources to cloud resources, considering computation cost and data transmission cost. Their work focused on scheduling applications to cloud computing resources, whereas this proposal focuses on task execution balance of an integration process on available threads of a run-time system. Wu et al. (2010) performed an experiment with workflow applications, by varying data communication costs and computation costs, c.f. the cloud price model. They used PSO meta-heuristic to minimise data communication and computation costs in cloud; whereas this article used PSO to minimise total average processing time in the integration processes execution. Byun et al. (2011) estimated the necessary optimal number of resources to be allocated in order to minimise the cost of running a workflow. Their work presented an algorithm that estimates the minimum computing resources to execute a workflow within a predefined time span, for the automatic execution of applications on dynamically and elastically provisioned computing resources; whereas this article presents an algorithm that finds the best configuration of threads in local thread pool for a task workflow.

Subashini and Bhuvaneswari (2012) proposed a PSO adaptation to increase the task allocation performance of parallel applications amongst the various processors on a distributed system. Their algorithm obtained a set of optimal allocations with an increased performance level. Their work concerned the task allocation amongst the various processors, whereas this article concerns the allocation of tasks of an integration process amongst the various thread pools. An et al. (2012) proposed a PSO-based algorithm to find a near optimal operation sequence and schedule strategy for production processes. Their algorithm sought the minimal total makespan in its admissible sequence space. Their work applied PSO to schedule tasks in a production process minimising the makespan, whereas this article applies a PSO-based algorithm to schedule task of an integration process amongst local thread pools.

Yassa et al. (2013) proposed an approach for multi-objective workflow scheduling in clouds, and presented the hybrid algorithm, using a method called multi-objective Discrete Particle Swarm Optimisation combined with the Dynamic Voltage and Frequency Scaling technique to optimise the scheduling performance and to minimise energy consumption, while preserving the quality of service preferences of the users. Their approach used PSO to optimise energy consumption; whereas this article uses PSO to minimise makespan in task execution of workflow applications. Sidhu et al. (2013) proposed a load re-balance algorithm using PSO together with the smallest position value technique for task schedule problem. Their work measured the overall task completion time and compared their results with another-based PSO heuristic. Their work aimed to improve applications in commercial computing environments like cloud; whereas, our algorithm seeks to improve the run-time systems performance of integration platforms and measures the total average processing time of message and compares their results with different configuration for thread pool.

Chitra et al. (2014) used PSO to locate a suitable workflow schedule to optimise load balancing, speedup ratio, and makespan in cloud computing environment and their proposed algorithm was experimented and compared with Genetic Algorithms and standard PSO methods; whereas, this article seeks to minimise the makespan of integration processes, our algorithm was experimented and compared with different configuration for thread pool. Pragaladan and Maheswari (2014) presented dynamic and static algorithms for task scheduling and resource provisioning that rely on workflow structure information, such as critical shortest paths and workflow levels, beyond estimates of task run-times in multiple cloud providers. Their experiments compared their algorithms with others based in standard PSO, using four workflows. This article seeks to evaluate the proposed algorithm regarding its ability to find an optimal or near-optimal configuration that results in a lower makespan in run-time systems of integration platforms. Rodriguez and Buyya (2014) presented a PSO-based algorithm, which aimed to minimise the overall execution cost while meeting deadline constraints for scheduling a scientific workflows application in a cloud environment. Their focuses were features of the Infrastructure as a Service, such as the dynamic provisioning and heterogeneity of unlimited computational resources and the performance variation of the virtual machines. Their algorithm produced a schedule defining the mapping of the tasks to the resources, the number and type of virtual machines, the initial and final time of their leases; whereas, our PSO-based algorithm indicates a configuration to thread pool in run-time systems of integration platforms, which objective function is to minimise the makespan. Jian et al. (2014) proposed a PSO-based algorithm to schedule tasks to cloud resource suppliers considering the reliability of these resource providers and of the network data transmission between suppliers. They defined a reliability measure by a mathematical model, used to evaluate the task to run and the reliability degree in data transmission. Their work used PSO algorithm to address reliability to schedule tasks to cloud computing; whereas, this article finds the best threads distribution to the task execution of integration processes. Ramezani et al. (2014) proposed a task-based system to load balancing by migrating tasks from an overloaded virtual machine to another homogeneous virtual machine, instead of migrating the entire overloaded virtual machine. Their work measured the task execution time and task transfer time and compared their results with other traditional methods for load balancing, aiming an energy consumption reduction. In this article, the proposed algorithm balances the number of threads into thread pool, aiming to minimise the makespan of the task execution of integration process.

Verma and Kaushal (2015) extended the previous proposal, called Bi-Criteria Priority-based PSO, which scheduled workflow tasks over the available cloud resources, minimising the execution cost, while considering the constraints of deadline and budget. Their work simulated the proposed algorithm and compared state-of-art algorithms. In a different approach, this article proposes an algorithm that minimises

the makespan in run-time system of integration platforms. Milani and Navin (2015) proposed a PSO-based algorithm, using a multi-objective function, to schedule the tasks in the cloud. They measured the execution time, waiting time and missed tasks and compared it to other task scheduling policies, such as First Come First Served, Shortest Process Next and Highest Response Ratio Next; whereas, this article measures the execution time, total average processing time of messages, time of processing gain, standard deviation and compared using different configurations for thread pool. Aron et al. (2015) proposed PSO-based hyper-heuristic method that minimises the time and cost along with optimised utilisation of the resources in the grid environment, without violating the security norms. Their authors simulated the proposed algorithm in order to evaluate its performance and compared with other existing common heuristic-based scheduling algorithms; whereas, this article presents the algorithm based on PSO, which finds a configuration to thread pool and also presents a makespan evaluation of the execution of a real-world integration process.

Ghosh et al. (2017) proposed a Cuckoo Search adaptation optimisation method for scheduling user-jobs to available resources for grid computing, in order to optimise performance metrics in terms of makespan and completion time. They measured makespan, standard deviation and completion times for the proposed algorithm and compared with simulated annealing and cuckoo search algorithm. Ghosh and Das (2018) proposed a hybrid algorithm, combining Extreme Optimisation and PSO. Their algorithm aimed, simultaneously, to minimise makespan, processing cost and job failure rate, and maximise resource utilisation of computational grid systems. Their works approach concerned task schedule in grid computing; whereas our concern focus on the task schedule in integration platforms runtime systems. Zhang et al. (2018) proposed an algorithm to deal with a class of job-shop scheduling optimisation problems. Based on the job-shop process model, their algorithm proposed active schedules encoding and decoding approaches for production processes scheduling. Their works approach concerned job-shop process schedule; whereas our concern is the integration processes tasks schedule.

Touzene et al. (2019) proposed a service oriented architecture for resource management optimisation based on a mathematical model, on mixed-Integer Linear Program, which selected the best resource allocation for all the smart grid constituencies and generates only the amount of energy needed by the consumers. Their works approach concerned resource allocation in grid computing; whereas our concern is resource allocation in EAI run-time systems. The related works are summarised in Table 1, contemplating the research field, the goal, and if the approach is PSO based or not.

Table 1 Related works summary

Work	*Research field*	*Goal*	*PSO*
(Pandey et al., 2010)	Cloud computing	Minimise computation cost and data transmission cost.	×
(Wu et al., 2010)	Cloud computing	Minimise data communication and computation costs.	✓
(Byun et al., 2011)	Cloud computing	Minimise computing resources use.	×
(Subashini and Bhuvaneswari, 2012)	Distributed system	Find optimal resource allocations.	✓
(An et al., 2012)	Production processes	Find optimal operation sequence and schedule.	✓
(Yassa et al., 2013)	Cloud computing	Minimise energy consumption.	✓
(Sidhu et al., 2013)	Cloud computing	Minimise overall task completion time.	✓
(Chitra et al., 2014)	Cloud computing	Optimise load balancing, speedup ratio, and makespan.	✓
(Pragaladan and Maheswari, 2014)	Cloud computing	Minimise task run-times in multiple cloud providers.	✓
(Rodriguez and Buyya, 2014)	Cloud computing	Minimise the overall execution cost while meeting deadline constraints.	✓
(Jian et al., 2014)	Cloud computing	Optimise reliability of resource provider tasks in network data transmission.	✓
(Ramezani et al., 2014)	Distributed system	Load balancing by migrating tasks amongst virtual.	×
(Verma and Kaushal, 2015)	Cloud computing	Minimise execution cost, while considering constraints of deadline and budget.	✓
(Milani and Navin, 2015)	Cloud computing	Minimise the execution time, waiting time and missed.	✓
(Aron et al., 2015)	Distributed system	Minimise the time and cost along with optimised utilisation of the resources, without violating the security norms.	✓
(Ghosh et al., 2017)	Distributed system	Minimise makespan and completion time.	×
(Ghosh and Das, 2018)	Distributed system	Minimise makespan, processing cost and job failure rate, and maximise resource utilisation.	✓
(Zhang et al., 2018)	Production processes	Find optimal operation sequence and schedule.	×
(Touzene et al., 2019)	Distributed system	Optimise the profits and resource utilisation over the smart grids.	×
[Our proposal]	EAI	Minimise makespan while considering the constraints	✓

3 Background

In this section, we describe the main elements of a run-time system of integration platforms and present the execution model that uses multiple local thread pool. Figure 1 presents a conceptual map introducing concepts discussed in this section, starting from the concept of root that is the run-time system.

The run-time system is the component that supplies fundamental services to a language or a library and applications implemented on top of them (Appel, 1990). It implements an execution model that determines how an integration process must be executed and provides resources that support this execution. The execution model determines the behaviour of the run-time system during the execution of an integration process. The main elements of run-time systems are: scheduler, task queue, thread pool, and monitors. The scheduler is the central element because it manages and orchestrates the activities of the task queue, the threads, and the monitors. It usually has a configuration file that contains information, such as the maximal number of threads, names and paths of files for data generated by monitors, monitors running frequency and logging system to notify about warnings and errors. The task queue maintains the tasks ready to be executed.

A task is considered ready to be executed when a message arrives in its input. The execution of a task starts when the execution time, for which it was scheduled, expires and there are available threads to execute it. The thread pool owns threads that, when available, recurrently poll the queue for tasks and execute the tasks. The monitors provide data about computational resources, such as the percentage of memory usage, time consumed for execution, task queue size, and total number of tasks processed. Monitors are executed by specific threads, in which collect and store the information. A message is a packed data that the messaging system can transmit through a message channel. So, data must be converted into one or more messages in order to flow in an integration process. A task has one or more inputs, and one or more outputs, depending on the implemented integration pattern. Every integration pattern represents an atomic operation to construct, route, transform or to manage messages. The messages must be processed by every task, following the order of dependence, that is, a message only can be processing in a task, when this message was already processed in every task that precedes the current task. Additionally, every task instance, i.e., every task that processes a message, needs an available thread to execute it and the execution of a task instance cannot be divided, given that every task is atomic. The execution model is presented in Figure 2.

Figure 1 Conceptual map of a run-time system

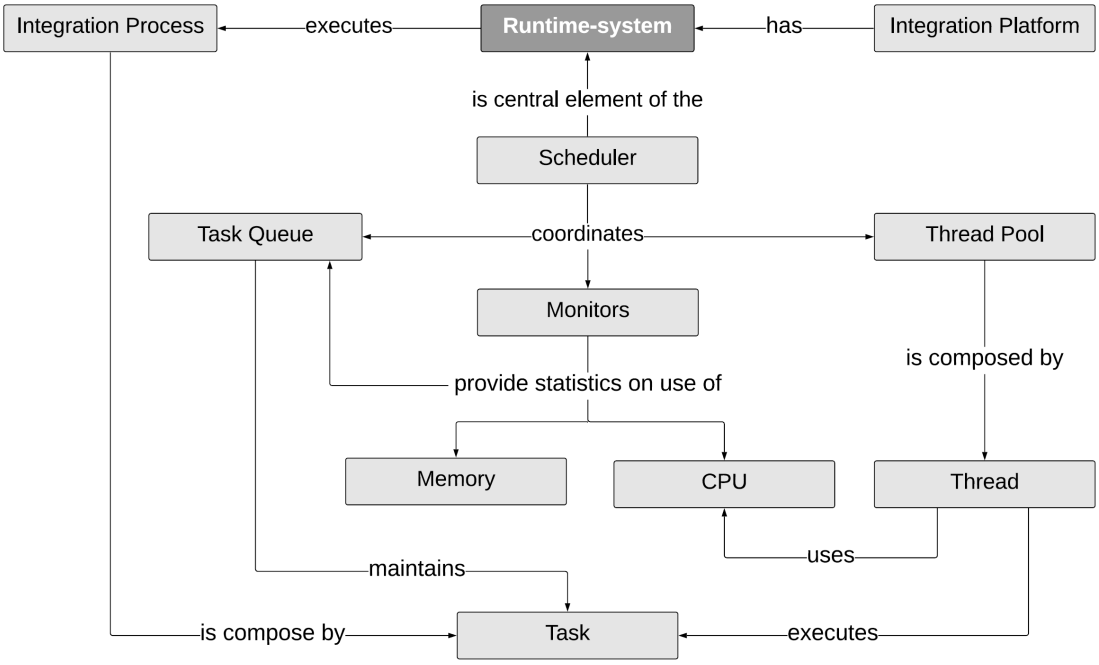
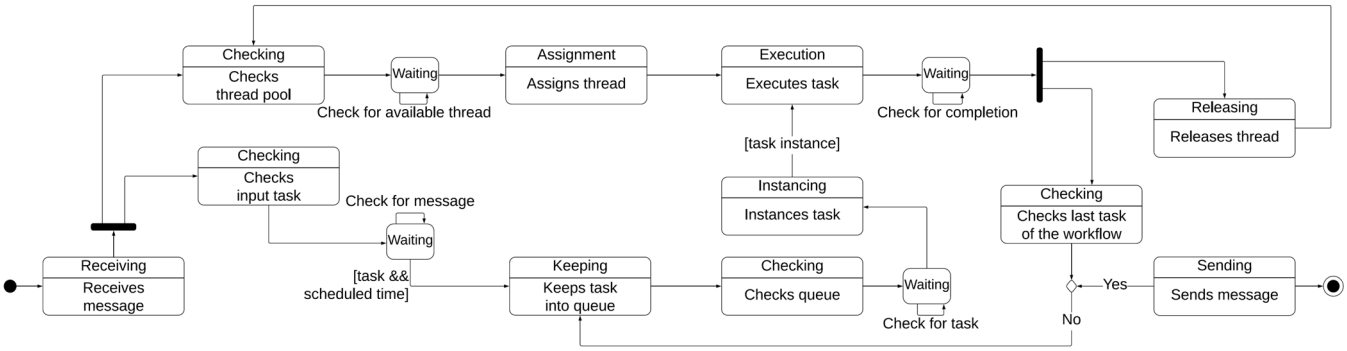


Figure 2 State diagram of the run-time system execution



In local thread pool execution model, there is a thread pool and a queue of tasks ready for every task of the integration process. A thread pool configuration determines the number of threads that every local thread pool must have. When a message arrives in the input of a task, then, it is maintained in its specific task queue while waiting for an available thread of its specific thread pool. If many messages arrive at the task input, then many instances of this task can be simultaneously executed if there are available threads in the thread pool dedicated to this task; and the available threads of a local thread pool recurrently check the task queue dedicated to the task, obeying the First-In-First-Out (FIFO) policy.

4 Problem formulation

In this section, we formulate the research problem. Firstly, we describe the software ecosystem and an integration process of a real-world problem. Then, we introduce the problem definition and objective function. The former is the modelling and codification of the problem and the latter measures the adequacy of a thread pool configuration in order to minimise the makespan.

4.1 The software ecosystem

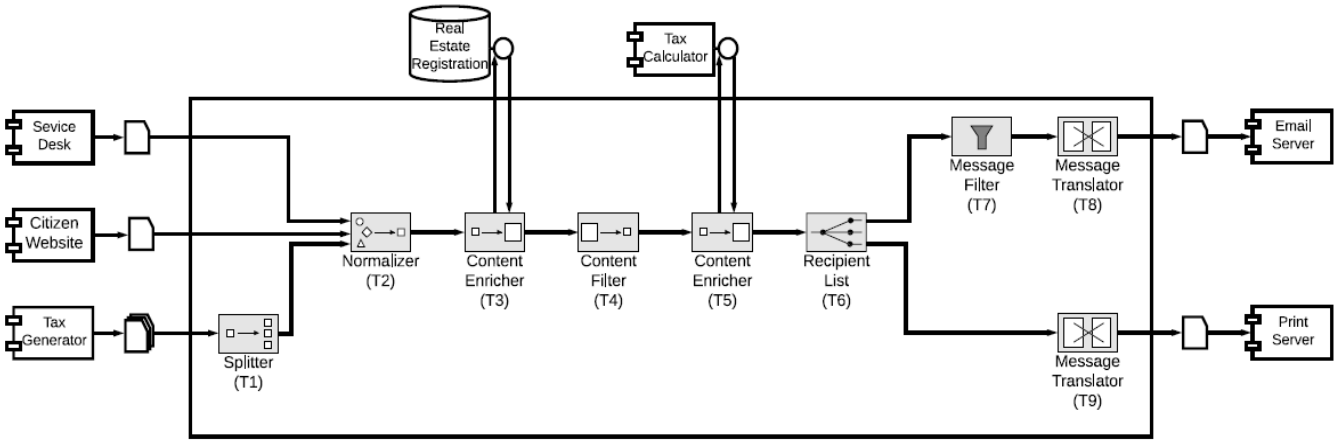
In order to demonstrate the feasibility of our proposal, we introduce a real-world software ecosystem that approaches a real estate tax management system used in Ijui City, Brazil. This ecosystem supports a business process that calculates the annual tax of registered real estates, generates the respective billing documents, and sends them to the real estate taxpayers. In 2017, Ijui had an estimated population of 83,173 inhabitants, 38,723 registered real estates and the referred tax raised 15,701,504 Brazilian Real (R$) collected by the city, according to Brazilian Institute of Geography and Statistic (Instituto Brasileiro de Geografia e Es- tatistica – IBGE) (IBGE, 2017). Seven applications from the software ecosystem of the city hall, are involved in the integration process, namely: Service Desk, Citizen Website, Tax Generator, Real Estate Registration, Tax Calculator, Email Server and, Print Server. When these applications were designed, their integration, in order to work together and collaborate with each other, was not the main focus of project.

Service Desk standalone renders the first citizen service, providing information regarding their taxes. Citizen Website is a web application that allows the issuance of new tax payment documents with the updated payment amount. Tax Generator requests the generation of the tax payment document to the registered real estates, which is done by an operator at the end of the year, issuing all the documents of the subsequent year. Real Estate Registration is a database containing taxpayer and real estate data used to calculate the tax. Tax Calculator calculates the amount of tax and generates the payment document. Email Server manages the requisitions to taxpayers and sends the payment document to their electronic address. Finally, the Print Server prints the tax payment documents that will be later sent to the physical address of the taxpayers.

4.2 The integration process

The integration process conceptual model integrates the applications involved in the software ecosystem, as Figure 3 shows. Splitter T1, Normaliser T2, Content En-richer T3, Content Filter T4, Content Enricher T5, Recipient List T6, Message Filter T7, Message Translator T8 and, Message Translator T9 are the internal tasks and are inside the rectangle. Service Desk, Citizen Website, Tax Generator are applications that provide inputs to the integration process; Real Estate Registration, Tax Calculator are applications that provide information, which is used to compute the tax and the payment documents; Email Server, Print Server are applications that receive the outputs to the integration process. The applications are outside the rectangle.

Figure 3 Conceptual model of the real estate tax management system

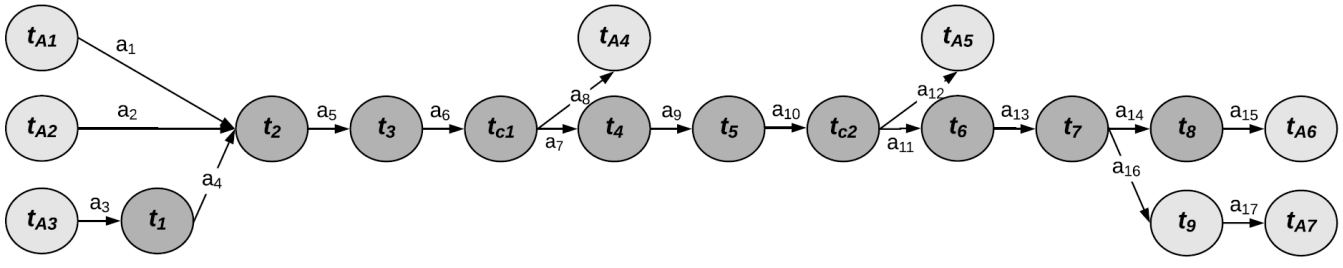


The process starts when any of the Service Desk, Citizen Website or Tax Generator application generates requests to the integration process. Service Desk and Citizen Website generate requests regarding a single real estate, while Tax Generator generates a list of requests regarding every registered real estate in the city. Splitter T1 breaks requests from Tax Generator into several output messages, each one of them corresponds to a single real estate register. Different formats of requests generated by the applications are sent to the Normaliser T2; T2 normalises them, producing an output message in a canonical format. Content Enricher T3 enriches the information of the real estates with taxpayer and real estate data obtained from the Real Estate Registration database. Content Filter T4 filters the messages, eliminating the unnecessary ones, such as phone number or personal document numbers. Content Enricher T5 enriches the information of the real estate's payment document with the tax amount, calculated by Tax Calculator. Recipient List T6 makes a copy of the message that contains the payment document and forwards it to the Message Filter T7 and to the Message Translator T9. Message Filter T7 filters the messages eliminating those that do not have a registered electronic address and forwards those that have email to Message Translator T8; T8 translates the message to an e-mail server compatible format and forwards the transformed message to Email Server, so that the payment document is sent by email to the taxpayer. Similarly, Message Translator T9 translates the message to a print server compatible format and forwards the transformed message to Print Server so that the payment document is printed and sent to the physical address of the taxpayer.

4.3 Problem definition

An integration process can be represented by a workflow composed by many paths of interdependent tasks linked by communication channels that desynchronise one task from another. A path is a set of sequentially arranged tasks in a flow, where every task has to be executed in a predefined order. Figure 4 shows an Integration Pattern Typed Graph (IPTG) (Ritter et al., 2018) that represents the integration process of Figure 3, in which each node represents a task and each edge represents a communication channel. Every edge has a weight, which represents the waiting time of the task in the queue. The set of tasks is composed by the 18 nodes, $T = \{ t_1, t_2, t_3, t_{c1}, t_4, t_5, t_{c2}, t_6, t_7, t_8, t_9, t_{A1}, t_{A2}, t_{A3}, t_{A4}, t_{A5}, t_{A6}, t_{A7}\}$ and the set of directed edges is composed by the 17 edges $E = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}, a_{17}\}$. An edge a_k of the form (t_i, t_j) exists, if there is dependence between t_i and t_j, in which t_j is the t_j predecessor task and t_j is the t_i successor task. Therefore, a successor task cannot be performed until all of its predecessor tasks are completed. IPTG can be separated into several paths formed by tasks that must be executed sequentially in a predefined order. The critical path is where a message spends longer time to be processed (James and Kelley, 1961). In IPTG of Figure 4, the critical path is highlighted with darker nodes, considering the execution time of the internal tasks of this path and their waiting time.

Figure 4 Integration pattern typed graph example



Ritter et al. (2018) define IPTG as a direct graph with set of nodes T and a set of edge $E \subseteq T \times T$ add to the function *type*: $T \rightarrow F$, where F = {*start, end, message processor, fork, structural join, condition, merge, external call*}. For a node $t \in T$, $\cdot t = \{t' \in T \mid (t' \cdot t) \in E\}$ for the set of direct predecessors of t, and $t \cdot = \{t'' \in T \mid (t \cdot \leq t'') \in E\}$ for the set of direct successors of t. An IPTG (T, E, *type*) is *correct* if the following condition apply:

- $\exists t_1, t_2 \in T$ with type (t_i) = *start* and *type* (t_2) = *end*;
- if *type* (t) ∈ [*fork*, *condition*} then | $\cdot t$| = 1 and |$t \cdot$| = n;
- if *type* (t) ∈ [*join*} then | $\cdot$ t | = n and |t
- if type (t) ∈ [*message processor, merge*} then |$\cdot t$| = 1 and |$t \cdot$| = 1;
- if type (t) G [*external call*} then |-t| = 1 and |$t \cdot$| = 2;
- The graph (T, E) is connected and acyclic.

The function *type* records what type of task each node represents. The first correctness condition claims that an integration pattern has at least one input and one output; the second indicates the cardinality of the involved tasks, i.e., the in-degrees and out-degrees of a node. The last condition states, «the graph (T, E) is connected and acyclic», indicates that a graph represents only a task and its relation with its predecessor and successor tasks and that messages do not loop back to previous tasks.

Table 2 classifies the tasks that compose IPTG of Figure 4, by their *types,* c.f. Ritter et al. (2018), where the tasks represented by t_{Ai} correspond to applications; the tasks represented by t_{ci} connect with applications; and, the tasks represented by t_i carry out internal operations into an integration process.

Table 2 Classification of the tasks of the IPTG

Type	*Tasks*
Stark	t_{A1}, t_{A2}, t_{A3}
Fork	t_7
Join	t_2
Message processor	$t_1, t_3, t_4, t_5, t_6, t_8, t_9$
External call	t_{c1}, t_{c2}
End	t_{A6}, t_{A7}

In the integration process of Figure 3, tasks T_3 and T_5 execute two operations: message enrichment and communication with external resources. In the IPTG of Figure 4, the task T_3 is represented by t_3 and t_{c1} and the task T5 by t_5 and t_{c2}. The task types t_3 and t_5 is *message processor* and their operation implements a message enrichment; and the task types t_{c1} and t_{c2} is *external call* and their operation implements a communication with external resources.

4.4 Objective function

In the approached execution model, there is a local thread pool to execute every internal tasks into the path of an IPTG. An internal task carries out internal operations into an integration process or makes connections with applications. In our mathematical model, we considered that:

- there is a constraint for run-time systems that limits the total number of threads to be distributed amongst local thread pools;
- the execution time of an internal task is estimated by the average number of instructions executed for each clock cycle of a processor (Abraham et al., 2016);
- the processing time of an internal task is the total time that a message spends to be processed into a task, i.e., the sum of the execution time and the waiting time in the task queue;
- a local thread pool must have an appropriate number of threads to execute instances of tasks in order to process messages at the shortest possible time.

Some researchers presented mathematical modelling for time metrics of applications. Rodriguez and Buyya (2014) presented equation (1) for total processing time of a task in a virtual machine (VM) in the scientific workflows scheduling in a cloud environment. $TE_{t_i}^{VM_j}$ is the task execution time t_i in a VM of type VM_j ; $TT_{e_{ij}}$ is the time it takes to transfer data between a task t_i and its successor t_j; k is the number of edges in which t_i is the predecessor task and s_k is 0 whenever t_i and t_j run on the same VM or 1 otherwise.

$$TP_{t_i}^{VM_j} = TE_{t_i}^{VM_j} + \left(\sum_{1}^{k} TT_{e_{ij}} * s_k \right) \quad (1)$$

According to Chirkin et al. (2017), the processing time in a workflow can be represented by equation (2), where TE is the task execution time, T_R is the resource preparation time, T_Q is the queuing time, T_D is the data transfer time, T_O is the system overhead time, such as time spent in analysing the task structure, selecting the resource, amongst others.

$$TP = TE + \left(T_R + T_Q + T_D + T_O \right) \quad (2)$$

For Shishido et al. (2018), the total processing time of a task t_i in a VM of type vm_s^k is the sum of the task execution time $TE\left(t_i, vm_s^k\right)$, transfer time $TT(t_i)$, and security services overhead of the SC (t_i) as defined in equation (3).

$$TP\left(t_i vm_s^k\right) = TE\left(t_i, vm_s^k\right) + \left[TT\left(t_i\right) + SC\left(t_i\right) \right] \quad (3)$$

Many researchers use the makespan arithmetic average as a performance metric for scheduling algorithms (Abdulhamid et al., 2014; Chhabra and Oshin, 2018; Lin and Ying, 2019). According to Canon and Jeannot (2007), makespan is computed by instantiating every computation and communication duration according to random variables, i.e. it is the end-time of the processing last task. For Abdulhamid et al. (2014), the lower the makespan, the better the processing efficiency, meaning less processing time. They defined

equation (3) where makespan is the maximum time needed to complete processing max C_i'.

$$Makespan = \{\max C_i'\} = \max\{C_1', C_2', \ldots, C_n'\} \quad (4)$$

We define an internal task total processing time in a thread pool as computed by equation (5), where TP_{t_i} is the total processing time; TP_{t_i} is a task execution time in a thread into its respective thread pool; $TQ_{t_{ij}}$ is the task waiting time in its task queue; and, k is the number of edges to which the current task is its successor.

$$TP_{t_i} = TE_{t_i} + \sum_{1}^{k} TQ_{t_{ij}} \quad (5)$$

The total task processing time that connects an application is estimated by the throughput time, which considers the time of sending and receiving of a message from a task to the external application. Equation (6) calculates the total processing time, where $TP_{t_{ci}}$ is the total processing time; $TSend_{t_{ci}}$ is the sending time of a message from a task to an application; and, $T\,Receive_{t_{ci}}$ is the receiving time of a message task from an application to a task.

$$TP_{t_{ci}} = TSend_{t_{ci}} + T\,Receive_{t_{ci}} \quad (6)$$

In integration processes, we define makespan as the message processing total time in a workflow and it is calculated by the task processing times of the critical path of the IPTG. Equation (7) calculates the makespan, where n is the total number of tasks that carry out internal operations; and m is the total number of tasks that connect applications.

$$Makespan = \sum_{1}^{n} TP_{t_i} + \sum_{1}^{m} TP_{t_{ci}} \quad (7)$$

We defined that the Total Average Processing Time (TAPT) of a given number of messages is calculated by the division of the makespan by the total number of messages. Equation (8) calculates the TATP, where TM is the total number of messages.

$$TAPT = \frac{Makespan}{TM} \quad (8)$$

Time optimisation becomes fundamental in situations where the process execution duration must meet certain constraints or deadlines, because, their violations increase the business processes costs (Pereira and Varajão, 2019). Thus, the problem can be formulated as:

> *find the optimal or near optimal number of threads for every local thread pool, in order to process a given number of messages through the critical path of an integration process, spending the lowest total average processing time and subjected to a constraint of the total number of threads that can be distributed.*

This formulation is represented by the objective function of equation (9), in which TR is the total number of threads and δ_r is a constraint of number of threads that can be distributed.

$$Minimise\ \{TAPT\} \quad (9)$$

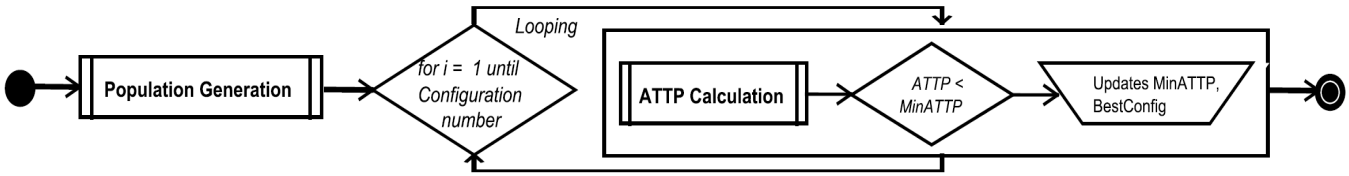
subject to $TR \leq \delta_r$

5 Our proposal

In this section, we model the previously formulated problem as a PSO problem and propose an algorithm to resolve it. The algorithm provides an optimum or near optimal configuration for the local thread pool that executes tasks of a given path of the integration process workflow. The thread pool configuration produced by the algorithm is a vector composed of n elements, where n is the number of tasks of the path. The element order on this vector corresponds to the task order in the path and the value of each element is the number of threads in the local thread pool for each task.

Three algorithms were implemented: «Population Generation», «TAPT Calculation», and «Best Configuration». «Population Generation» generates a population composed of possible configurations of the thread pool for the critical path of an integration process, i.e., a population of possible solutions. «TAPT Calculation» calculates the total average processing time of a given number of messages into tasks of a workflow path. «Best Configuration» combines the previous two algorithms to find the configuration that minimises the total average processing time amongst the possible solutions. The «Best Configuration» generates the several thread pool configurations by «Population Generation»; for every configuration, it calculates the *TAPT* by «TAPT Calculation» ; and updates the minimum *TAPT,* if the *TAPT* calculated is lower than the current minimum *TAPT*, c.f. flowchart shown in Figure 5.

Figure 5 Best configuration flowchart



5.1 Population generation

The population of thread pool configurations is obtained by the Algorithm 1. It receives as input: the constraint of the number of possible solutions, the constraint of the total thread number, and the number of local thread pools.

Algorithm 1 *Population Generation*

Input: Number of configurations σ_s - stop criterion
Input: Constraint of the total number of threads δ_r
Input: Number of thread pool ζ_{Pool}
Output: Matrix of configurations $R_{initial}$

1: $R_{initial} \leftarrow \emptyset$ ▷ Initialises the configurations for thread pool population
2: $\alpha_r \leftarrow \emptyset$ ▷ Initialises the set of available threads
3: $\alpha_{Pool} \leftarrow \zeta_{Pool} - 1$ ▷ Initialises thread pool number
4: $S_r \leftarrow \emptyset$ ▷ Initialises the set of the used threads sum
▷ Initialises thread number
5: **for** $[i] = 1$ until σ_s **do**
6: $R_{initial}[i,1] \leftarrow Random[1, \delta_r - (\zeta_{Pool} - 1)]$
7: $S_r[i,1] \leftarrow R_{initial}[i,1]$
8: $\alpha_r[i,1] \leftarrow \delta_r - S_r[i,1]$
9: **end for**
▷ Generates of threads number
10: **for** $[i] = 1$ until σ_s **do**
11: **for** $[j] = 2$ until ζ_{Pool} **do**
12: $R_{initial}[i,j] \leftarrow Random[1, \alpha_r[i,1] - (\alpha_{Pool} - 1)]$
13: $S_r[i,1] \leftarrow S_r[i,1] + R_{initial}[i,j]$
14: $\alpha_r[i,1] \leftarrow \delta_r - S_r[i,1]$
15: $\alpha_{Pool} \leftarrow \alpha_{Pool} - 1$
16: **end for**
17: $\alpha_{Pool} \leftarrow \zeta_{Pool} - 1$
18: **end for**
19: **return** $R_{initial}$

The algorithm starts with a set of initial configurations for thread pool, and three auxiliary variables, a vector with the number of threads available, the available thread pool number, and a vector for the sum of used threads. From lines 5 to 9, the algorithm initialises the number of threads with a random number for every local thread pool of the initial configurations set. Then, it updates the vector for the sum of the threads and the vector for the number of available threads. From lines 10 to 18, the algorithm generates the remaining elements for the initial configurations set, which corresponds to the number of threads from the second thread pool to the last pool thread. Then, the algorithm updates the used threads sum vector; the available thread number vector; and, the used threads sum and the available thread pool number. At line 19, the algorithm returns the initial configuration set for thread pools.

5.2 TAPT calculation

Algorithm 2 calculates the value of the total average processing time, which is used to test the objective function. It receives as input the following parameters: processing time vector, pool configuration vector, and the total message number. The parameter «processing time vector» represents a path, in which each element represents the value of a task processing time. The parameter «pool configuration vector» is a local thread pool configuration. The parameter «total message number» represents the total message number to process, i.e., a workload in a given moment.

Algorithm 2 *TAPT Calculation*

Input: Processing time vector TP_t
Input: Configuration vector R
Input: Total of messages τ_m
Output: $TAPT$

1: $\zeta_{Pool} \leftarrow size(TP_t)$ ▷ Calculates thread pool number
2: $TF_m \leftarrow \emptyset$ ▷ Initialises final processing time matrix
3: $TT_m \leftarrow \emptyset$ ▷ Initialises vector processing time
4: $S_{TT} \leftarrow 0$ ▷ Initialises the total processing time
5: $r \leftarrow R_{[1]}$ ▷ Initialises the number of threads
6: $[j] \leftarrow 1; [aux] \leftarrow 1$
▷ Updates final processing time in the first task
7: **while** $[i] \leq \tau_m$ **do**
8: **for** $[j] = 1$ until r **do**
9: $TF_m[1,i] \leftarrow [aux] \times TP_t[1]$
10: [i]++
11: **end for**
12: **if** $[i] > \tau_m$ **then**
13: *break*
14: **end if**
15: [aux]++
16: **end while**
17: $[i] \leftarrow 1; [aux] \leftarrow 1; [k] \leftarrow 1$
▷ Updates final proc. time of the first r in tasks
18: **for** $[i] = 2$ until ζ_{Pool} **do**
19: $r \leftarrow R[i]$
20: **for** $[j] = 1$ until r **do**
21: $TF_m[i,j] \leftarrow TF_m[i-1,j] + TP_t[i]$
22: **end for**
23: **end for**
▷ Updates final processing time from *nth*-message in tasks
24: **for** $[i] = 2$ until ζ_{Pool} **do**
25: $r \leftarrow R[i]$
26: **for** $[j] = r$ until τ_m **do**
27: **if** $r < R[i-1]$ **then**
28: $delay = TF_m[i, last_m]$
29: **else**
30: $delay = TF_m[i, first_m]$
31: **end if**
32: **if** $delay < TF_m[i-1,j]$ **then**
33: $delay \leftarrow TF_m[i-1,j]$
34: **end if**
35: $TF_m[i,j] \leftarrow delay + TP_t[i]$
36: **end for**
37: **end for**
▷ Updates total processing time
38: **for** $[i] = 1$ until τ_m **do**
39: $TT_m[i] \leftarrow TF_m[\zeta_{Pool}, i]$
40: $S_{TT} \leftarrow S_{TT} + TT_m[i]$
41: **end for**
42: $TAPT \leftarrow S_{TT} \div \tau_m$
43: **return** $TAPT$

The algorithm initialises the number of thread pool, a matrix to keep the final processing time of each message, a vector to keep the total processing time of each message; a variable to keep the accumulated value of the processing time for all the messages, and an auxiliary vector to keep the number of threads of each thread pool.

The algorithm calculates from lines 7 to 16, for each message, the first task final processing time for the first thread pool. The number of messages simultaneously processed is equal to the number of threads in the pool. The final processing time of the messages in the first task is equal to the number of threads in first local thread pool times the processing time of the first task. From lines 18 to 23, the algorithm calculates the final processing time of the second until the last task, for the first message until the *n*-th message, where the *n*-th message corresponds to the number of threads of the pool dedicated to the task that is being processed. Since a task has to wait for messages from its predecessor to process, the final processing time of these messages is equal to the sum of two processing times: the final processing time of the message in the predecessor task and the processing time of the current task.

From lines 24 to 37, the algorithm calculates the final processing time from the second task until the last task in every thread pool, for the *n*-th message until the last message. There are two conditions to start the message processing in a task. First, the current message processing in the predecessor task has to be finished. Second, the predecessor message processing in a current task has to be finished. The processing time that finishes later between these two will be considered in the computation of the final processing time, called *delay*. Then, the final processing time of the messages is equal to the sum of two processing times: the *delay* and the processing time of the current task. From lines 38 to 41, the algorithm updates the final processing time of each message and the sum of these final processing times. At line 42, the algorithm calculates the total average processing time by the division of the sum of the final processing times by the total message number, and at line 43 it returns the total average processing time.

5.3 Best configuration

Algorithm 3 aggregates algorithms for «Population Generation» and for calculation of the total average processing time, based on the original PSO implementation. It provides the configuration for the local thread pool that results in the lowest total average processing time. The «Best Configuration» algorithm receives as inputs: the constraint of possible configurations, the constraint for the total number of threads, the total message number, and the processing time vector. Algorithm 1 is invoiced at line 5 to generate the initial population of configurations to thread pool. At line 10, the algorithm tests the stop condition, which is the constraint for the number of possible solutions. At line 12, the algorithm calculates the objective function by the invocation of Algorithm 2. At line 13, the algorithm compares the *TAPT* of the current configuration with the lower *TAPT* found until that moment. At line 18, the algorithm returns the minimal TAPT and the best configuration for the thread pool amongst the solutions of the generated population.

Algorithm 3 *Best Configuration*

Input: Stop criterion, i.e, configurations numberσ_s
Input: Constraint of for the total threads numberδ_r
Input: Total messages numberτ_m
Input: Processing Time vector TP_t
Output: Best Configuration vector R_{best}
Output: Minimum TAPT Min_TAPT

1: $\zeta_{Pool} \leftarrow size(TP_t)$ ▷ Calc. the n. of thread pool
2: $R_{best} \leftarrow \emptyset$ ▷ Initial set of configurations
3: $TF_m \leftarrow \emptyset$ ▷ Initialises final processing time matrix
4: $TT_m \leftarrow \emptyset$ ▷ Initialises total processing time vector
5: $PoolConf \leftarrow Population\_Generation(\sigma_s, \zeta_{Pool}, \delta_r)$ ▷ Initialises Minimum TAPT
6: $Min\_TAPT \leftarrow 0$
7: **for** $[i] = 1$ until ζ_{Pool} **do**
8: $Min\_TAPT \leftarrow Min\_TAPT + \tau_m \times TP_t[i]$
9: **end for** ▷ Calculates TAPT
10: **for** $[i] = 1$ until σ_s **do**
11: $poolconf \leftarrow PoolConf[i, :]$
12: $TAPT[i, 1] \leftarrow TAPT\_Calculation(TP, pool, \tau_m)$ ▷ Compares TAPT with TAPT minimum
13: **if** $TAPT \leftrightarrow Min\_TAPT$ **then**
14: $Min\_TAPT \leftarrow TAPT$
15: $R_{best} \leftarrow PoolConf$
16: **end if**
17: **end for**
18: **return** Min_TAPT; R_{best}

6 Validation

In this section, we describe the experiment that performed using the real-world integration process, introduced in Section 4. The applied protocol was based on the works of Jedlitschka and Pfahl (2005), Wohlin et al. (2012) and Basili et al. (2007) which provide procedures for controlled experiments in the engineering studies field. Then, we collected performance metrics from algorithm executions and used ANOVA and Scott & Knott statistical techniques to evaluate the results. In the following sections, we detailed the steps of the experiment and its validation.

6.1 Research questions and hypothesis

To validate the proposed algorithm in this article, our experiment answers the following research questions:

*RQ*1: Is it possible to obtain the optimal or near optimal local thread pool configurations for run-time systems of integration platforms, which minimises the total average processing time?

*RQ*2: Is it possible to obtain a mathematical model for the total average processing time as a function of the total thread number used in local thread pools?

For each research question, we provide a hypothesis that has to be confirmed or refuted by the experiment, respectively:

*H*1: An optimal or near optimal thread pool configuration for run-time systems of integration platforms can be found by an algorithm based on optimisation techniques, in which objective function is to minimise the average total time of message processing in integration processes.

*H*2: A mathematical model for the total average processing time as a function of the thread number can be found by statistical techniques.

6.2 Environment and support tools

The experiments were carried out on a machine equipped with 2 processors Intel Core i5-5200U, 2.20 GHz, 4GB of RAM, and the operating system Microsoft Windows 10 Education 64-bits. The Matlab (Leonard and Levine, 1995) software, version R2013, was used to create and execute the algorithms. The Genes (Cruz, 2006) software, version 2015.5.0, was used to statistically analyse measured data in this experiment. The source code developed and used in this experiment is publicly available for download.[1]

6.3 Variables

Independent variables are:

- *Solutions number*: The initial population of thread pool configurations tested by the optimisation algorithm. The tested value for this variable was 10 solutions.
- *Threads number*: The number of threads that can be distributed amongst the thread pools. The tested values for this variable were: 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 threads.
- *Messages number*: The messages number that is processed, i.e., the integration process workload. The tested value for this variable was 38,723 messages. This value is the number of registered real estates in Ijuí town, in Brazil, where the integration process is used.

Dependent variables are:

- *Total average processing time*: The meantime a message takes to be processed by all tasks that compose the critical path of the integration process.
- *Execution time*: The time that the algorithm spends to conclude an execution.

6.4 Execution and data collection

The execution of the algorithm was conducted using the critical path of the integration process highlighted in the integration pattern type graph shown in Figure 4. Table 3 shows the times, in milliseconds (ms), for every internal task of the critical path, obtained from the execution of the actual implementation of the integration process.

Table 3 Processing times of tasks

Task	*Execution time*	*Waiting time*	*Sending time*	*Receiving time*	*Processing time*
t_1	0.531	2	–	–	2.531
t_2	0.303	1	–	–	1.303
t_3	0.005	–	2	2	4.005
t_{c1}	0.005	1	–	–	1.005
t_4	0.003	1	–	–	1.003
t_5	0.005	–	2	2	4.005
t_{c2}	0.005	1	–	–	1.005
1 t_6	0.531	1	–	–	1.531
t_7	0.003	1	–	–	1.003
t_8	0.001	1	–	–	1.001
Total	1.392	9	4	4	18.392

The experiment is classified in the literature as a termination simulation, in which the experiment output is express as a function of the initial conditions. In these cases, the results are usually analysed statistically by the method of the repetitions, where a repetitions number between 20 and 30 is sufficient to obtain a population mean, in distributions with more extreme values that a normal distribution (Sargent, 2013). We experimented 200 different scenarios, which are synthesised follows:

Solutions number: 10	×	Threads number: 10, 20,…, 100	×	Messages number: 38,723	×	Executions:	=	Total of scenarios
1	×	10	×	1	×	20	=	200

We executed the algorithm by setting the number solutions, the messages number, and the integration process. The latter is represented by a vector, in which every element represents the processing time of one of the tasks of the critical path of the integration process. Besides this, we varied the threads number. The execution of the algorithm was repeated 20 times for every thread number. In every execution, we collected the minimal total average processing time, the best thread pool configuration, and the execution time of the «Best Configuration» algorithm. The standard deviation and the gain were calculated by the average total times of processing measured in the experiment.

Statistical theory is indicated for the analysis of data from experiments on performance (Georges et al., 2007), since statistical reasoning is an appropriate resource to deal with the non-determinism in computational systems, such as run-time systems (Frantz et al., 2011). We used the ANOVA variance analysis statistical technique to differentiate amongst the variations found in a set of results, which are derived from random factors called error and are influenced by the total number of threads. Because there

was a statistical difference in the results of the variance analysis, we used the Scott & Knott technique to find out how much the number of threads impacted differently the total average processing time. The Scott & Knott technique is considered a more rigorous test because it only considers relevant differences between the alternatives and is adopted in the literature in experiments with performance due to its simplicity.

6.5 Results

In this section, we present the results. Line charts present the optimal configurations of the thread pools and the total average processing time for every one of constraints of total number of threads. A table summarises standard deviations, execution time averages of the algorithm, and the gains in total average processing time. A scatter chart compares the results of minimum average total times processing for every one of constraints. Lastly, tables present the statistical analysis carried out with ANOVA and Scott & Knott techniques.

The results of average total times processing for the configurations of the local thread pool to the tasks of the integration process of Figure 3 are presented in Figure 7. In this last figure, the *x*-axis represents the order of the execution. The *y*-axis represents the total average processing time, in seconds.

A thread pool configuration is represented by a vector, whose elements are the number of threads in each thread pool and the index of the vector corresponds to the order of local thread pool for tasks of a path of the integration process. The black square on the curves shows minimum total average processing time and the thread pool configuration that provides this TAPT.

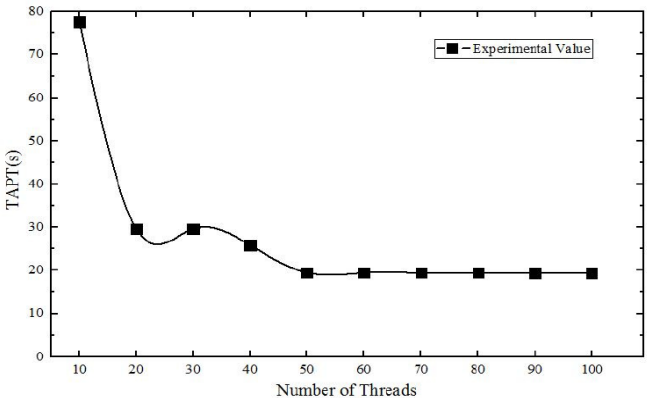
The standard deviation of the total average processing time, the execution time of the algorithm, and the gain in total average processing time are presented in Table 4. The standard deviation is a measure that expresses the degree of dispersion of a data set, i.e., it indicates how uniform a set of data is. Thus, the more homogeneous a data set is, the closer the value of the standard deviation is to zero. The execution time of the algorithm is extracted by the execution tool itself, which calculates the spent time executing in seconds the algorithm by calculating the processing times of each message in the threads, checking for the best thread configuration for the thread pool, and calculating the total average processing time. This metric does not include the time of random generation of the initial population of solutions since it is done only once and is fixed for every variation of the total number of threads and for the variation of the number of messages. The gain in total average processing time, defined like the absolute value of the highest difference between the TAPT and the minimum TAPT obtained with the optimal configuration of thread pool, measured in seconds.

A scatter chart presents the lowest total average processing time of all the executions for every value of the total thread number constraint, cf. Figure 6. The *x*-axis represents the total thread number constraint and the *y*-axis represents the values of the total average processing time.

Table 4 Summary of the calculated results

Total number of threads	*Standard deviation of the TAPT*	*Execution time average of the algorithm*	*Gain of the TAPT*
10	0	102.18	0
20	19.26	108.32	47.89
30	15.93	147.84	47.89
40	18.75	183.26	51.69
50	15.35	227.79	58.08
60	12.22	282.87	58.12
70	12.42	347.45	58.12
80	16.80	404.76	58.16
90	12.84	505.10	58.16
100	6.78	577.32	19.39

Figure 6 Minimal total average processing time



Regression analysis is a method to estimate the relation amongst the dependent variable, *TAPT,* and the independent variables, threads number, *NT*. Let $NT = (NT_1, NT_2,..., NT_p)$ be a vector of the independent variables and *TAPT* a dependent variable, the mathematics function, which relates *TAPT* and *NT,* can be expressed by the regression model, c.f. In equation (10), where β is a vector of unknown parameters, and ε is a disturbance term (Yao and Liu, 2018).

$$TAPT = f(NT \mid \beta) + \varepsilon \tag{10}$$

In regression analysis, the square of Pearson product-moment correlation coefficient is an important parameter to determine the degree of linear correlation of variables. This coefficient is known as the correlation coefficient or, simply, R^2. R^2 is defined by equation (11), where SSE is the sum of squared error and SST is the sum of squared total (Kaytez et al., 2015). Thus, R^2 tends to 1 when $SSE \ll SST$, i.e., the sum of squared error is too small compared to the sum of the squared total.

$$R^2 = 1 - \frac{SSE}{SST} \tag{11}$$

TAPT is represented by a statistical trend line in Figure 6. The trend line is a polynomial equation that describes the behaviour of the total average processing time as a function of the total threads number. The value of R^2 was equal to 0.9947. Analytically, TAPT is represented by equation (12),

where *NT* represents the number of threads and *TAPT* is the value of the minimum total average processing time.

$$TAPT = 0.014(NT)^6 - 0.5003(NT)^5 + 7.053(NT)^4 - 49.946(NT)^3 + 186.03(NT)^2 - 346.3(NT) + 280.99 \quad (12)$$

We used statistical techniques to verify the influence of the total number of threads used in the total average processing time. Table 5 presents the analysis of variance of the dependent variable, *TAPT*. The total of results is calculated upon the multiplication of the number of executions by the number of possible values for the total number of threads. The total freedom degree is calculated by the total of results –1.

Table 5 Variance analysis of TAPT by ANOVA technique

Sources of variation	*Degree of freedom*	*Average square*
Total numbers of threads	9	4412.04†
Error	190	200.83
Total	199	
Overall average	39.99	
Coefficient of variation (%)	35.43	

Note: †significant statistical by Fisher-Snedecor's Probability and error level of 5%.

Table 6 Average of TAPT by Scott & Knott technique

Total number of threads	*Average of the TAPT*	*Group*
10	77.55	a
20	49.55	b
30	41.85	b
40	42.31	b
50	34.70	c
60	32.26	c
70	30.25	c
80	34.04	c
90	29.42	c
100	27.94	c

Note: Error level of 5% by Scott & Knott model.

The degree of freedom of the total number of threads is calculated by the amount of possible values of the total number of threads subtracting 1. The degree of freedom for error, calculated by the difference between the degree of freedom of the total of results and the degree of freedom of the factors. The analysis of variance of TAPT shows the average square of 4412.04 for the total number of threads and 200.83 for error. Overall average was equal to 39.99 seconds and the coefficient of variation was 35.43 %.

The average comparison test by Scott & Knott technique of the dependent variable is presented in Table 6. The constraints on the number of threads are in first column, average of the minimum TAPT in the 20 executions is in second column, and the group of Scott & Knott technique is in second column. This technique groups number of threads that are not the statistically different between themselves. There were three groups: a, b and c. In «a» group are the constraints on the number of threads 10. In «b» group are the constraints on the number of threads 20, 30, and 40. In «c» group are the constraints on the number of threads 50, 60, 70, 80, 90 and 100.

6.6 Discussion and comparison

The algorithm was able to find the minimum total average total processing time, c.f. Figure 7. The total average total processing time reduces with the increase of the thread number, c.f. Figure 8. However, it is possible to infer that this reduction has a limit, after which it, the TAPT stabilise and do not decrease. The minimal number of threads to distribute amongst the 10 thread pools equals 10, one thread for every pool. In this case, the algorithm only can provide one configuration for thread pools and the minimum total average processing time was 77.55 seconds. When the number of threads to distribute is greater than 10, the algorithm can provide different configurations for thread pools, resulting in different values for minimum average total time. When the constraint equals 20 and 30 threads, the minimum TAPT was 29.66 seconds, so there was a 47.89 seconds reduction regarding the constraint equal 10 threads. When the constraint equals 40 threads, the minimum TAPT was 25.86 seconds, so there was a 3.80 seconds reduction regarding the constraint equals 20 or 30 threads. When the constraint equals 50 threads, the minimum TAPT was 19.47 seconds, so there was a 6.39 seconds reduction regarding the constraint 40 threads. When the constraint equals 60, 70, or 80, the minimum TAPT was the same: 19.43 seconds, so there was only 0.04 seconds reduction regarding the constraint equals 50 threads. When the constraint equal 90 threads and 100 threads, the minimum TAPT was the same: 19.39 seconds, so there was only 0.04 seconds reduction regarding the constraint equals 60, 70 and 80 threads.

The values of the standard deviation show that the set of configurations for thread pool was quite heterogeneous. The execution time values of the proposed algorithm increase proportionally to the number of threads, cf. 4. We can infer that it is possible to improve the implementation performance of the algorithm changing the stop criterion, such as establishing a previous value for TAPT and stopping the algorithm execution when a thread configuration reaches a TAPT value lower than this. We also identified that a variation of the total of threads distributed results in a variation of until 50 seconds in the total average processing time. Trend line in Figure 6 shows that the minimum total average processing time decreases with the increase of the number of threads, suggesting that the total average processing time decreases with the addition of threads up to a limit. The trend line allows to predict the TAPT value for a given number of threads in local thread pools. In this figure, the coefficient of determination value equal to 1 indicates that the model is able to explain the observed TAPT in the experiments conditions.

Figure 7 Minimum TAPT in every execution of algorithm

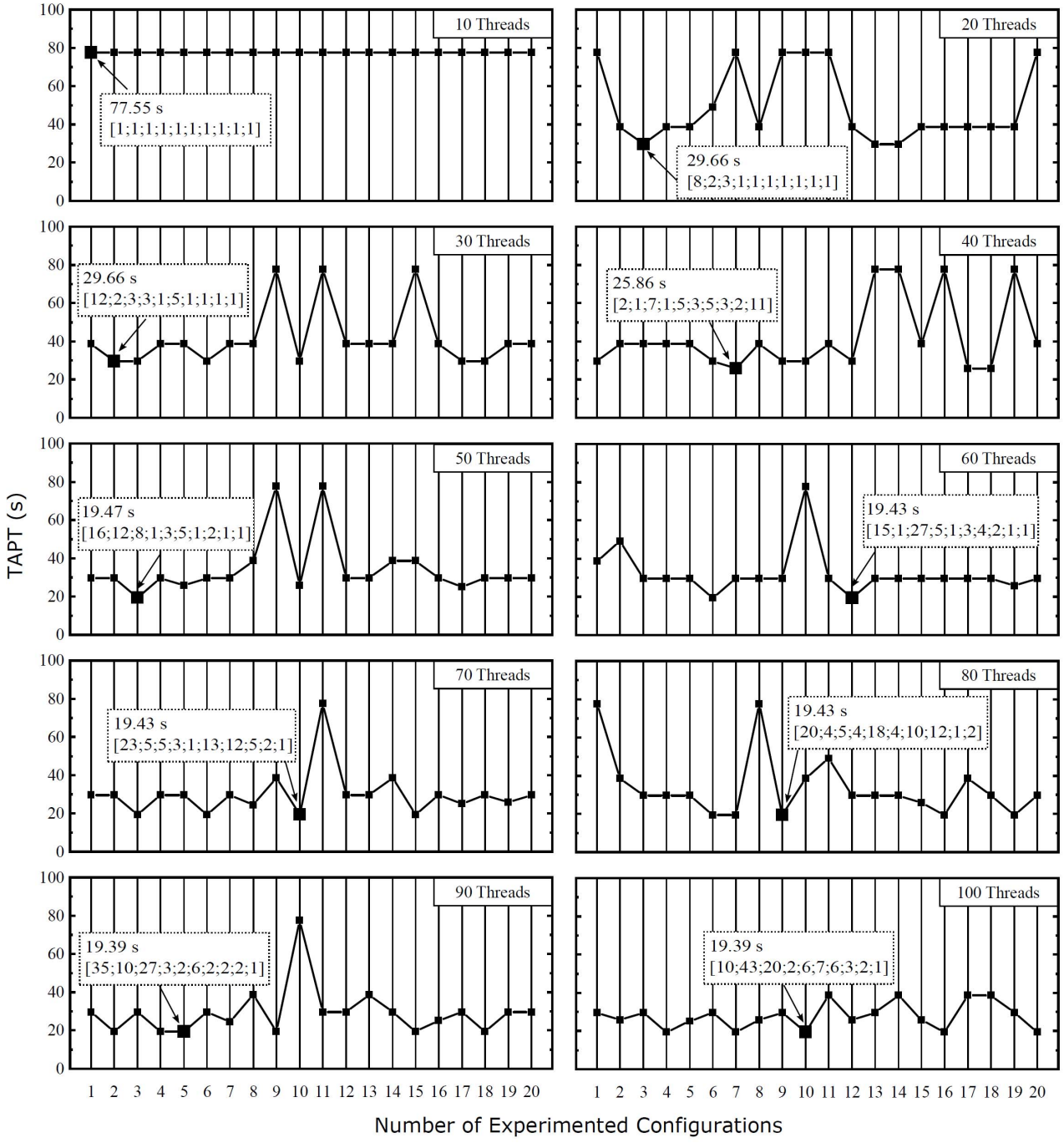
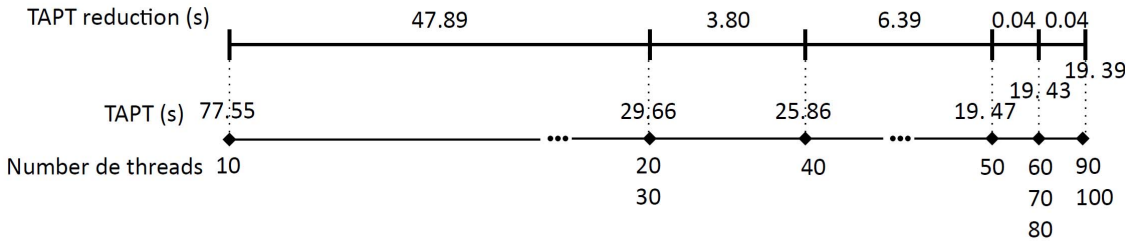


Figure 8 TAPT reduction



We observed that the variation of the total number of threads generates a significant difference on total average processing time, cf. Table 5. The coefficient of variation was reduced, indicating that the experiment is adequate and reliable. In The Scott & Knott averages comparison test, the lowest TAPT was using 100 threads, the lowest TAPT average was 27.94 seconds, cf. Table 6. The largest difference between TAPT averages was between 10 and 100 threads was 49.61 seconds, whereas the difference between 30 and 40 threads was only 0.46 seconds. In the TAPT analysis, same letter groups do not differ statistically amongst themselves. The TAPT averages with 20,30 and 40 threads do not differ statistically, which belong «b» group; from 50 threads there is no statistically significant difference, which belong «c» group; whereas, the use of 10 threads presents a statistically significant difference of the others.

6.7 Summary of results

In this section, we sum up the main conclusions from the results found in our experiment, answered our research questions and validate our hypothesis. Regarding the conclusions:

- In terms of processing time savings, the gain achieved by the thread pool configuration found by the algorithm attests the advantage of using this algorithm rather than empirically choose a configuration.
- The polynomial equation found to represent TAPT as a function of the total number of threads shows that TAPT tends to a constant, that is, from a certain point, TAPT does not change when the thread number increase. Thus, the model finds the least number of threads for a given TAPT value that one wishes to obtain.
- The ANOVA technique proved that our experiment is valid, since there was a significant difference in TAPT with different numbers of threads.
- The Scott & Knott technique showed that thread numbers can be grouped and within each group, these numbers bring similar results from TAPT. Therefore, one can choose the least number of threads in the group that results in the desired TAPT, in order to save computational resources and, consequently, costs for the companies.

Regarding the research questions and hypothesis:

- *RQ*1: Our algorithm obtained a near optimal local thread pool configurations of run-time systems that minimised the total average processing time. Gain about 75% was obtained between TAPT found 19.39 seconds and the worst case with TAPT equals 77.55 seconds, using 90 threads.
- *RQ*2: A trend line found by regression analysis and shown in equation (12), represents the total average processing time as a function of the total number of threads used in the local thread pool execution model.

By the experiment, we confirm our hypothesis for each of the research questions, respectively:

- *H*1: An optimal or near optimal configuration for the thread pools of run-time systems was found by a PSO-based algorithm, which objective function was to minimise the total average processing time messages in an integration process.
- *H*2: A mathematical model for the TAPT as a function of the number of threads was obtained by linear regression after validation of the results by ANOVA and Scott & Knott statistical techniques.

6.8 Threats to validity

As researchers, our goal is to mitigate all possible validity threats, since they are present in any empirical research (Cruzes and Ben Othman, 2017). We evaluated the factors that could influence results of the experiment and tried to mitigate these threats. In the following, we discuss its constructor, conclusion, internal, and external validity.

First, we substantiate our research by previous studies and mathematical base. After, we planned the experiment according to procedures from empirical software engineering presented by Jedlitschka and Pfahl (2005), Wohlin et al. (2012) and Basili et al. (2007). In this planning, we provide information about the execution environment, supporting tools, variables, execution and data collection. Then, we simulate a real-world integration process in two hundred different scenarios and used ANOVA and Scott & Knott statistical techniques to evaluate the results.

Conclusion validity concerns with to ensure that the treatment used in the experiment is really related to the actual outcome observed (Feldt and Magazinius, 2010). We used statistical techniques to assure that the actual outcome observed in our experiment is related to the used threads configurations, and not to factors that we do not control or have not measured, and we verified that there was a significant difference in the outcome.

Internal validity aims to ensure that the treatment actually caused the outcome, mitigating effects of other uncertain factors or not measured (Feldt and Magazinius, 2010). In order to minimise interference in the execution time of the algorithm, the experiment was performed in the same machine, that was set on security mode, using minimal features and the machine was disconnected from the internet during the executions.

External validity focuses on the generalisation the results outside the scope of our study (Feldt and Magazinius, 2010). The steps of the experiment are valid to compare other scenarios with other integration processes, other numbers of messages and other numbers of possible solutions generated by the population algorithm. Thus, as future work, we intend to perform the experiment with a large data set in order to evaluate the generalisation of the results.

7 Conclusions

Integration platforms are tools used by companies to exchange data and share functionality amongst different applications that compose their software ecosystems. These platforms implement and execute integration processes, which can be seen as workflow composed of atomic tasks. The run-time system is the component of the platform responsible for the execution of integration processes. Therefore, it is the most important element when the goal of a companies is performance. The runtime system efficiency is directly related to configuration of computational resources that perform the tasks, the threads. However, this configuration is based only in the experience of software engineers because there is no automatic way to do this. Threads are grouped in local thread pools and every one of them must contain the proper number to achieve the shortest processing time.

This article proposed an algorithm that obtains an optimum or near optimal configuration for local thread pools, which provides a lower average processing time of messages. This approach improves the performance of run-time systems and, consequently, increases productivity and reduces costs for enterprises. The algorithm, which is based on PSO meta-heuristic, was implemented in a programming language and experimented in the execution of a real-world integration process, where the scenarios varied the total number of threads distributed in local thread pools. We also applied statistical techniques to analyse the results and find a mathematical model to describe the behaviour of the total average processing time as a function of the total number of threads. We can point out our main contributions:

- Optimal thread pool configuration found by algorithm obtains the lowest message processing time.
- The ANOVA technique showed a significant difference in relation to the use of different thread numbers to local thread pools, thus it makes sense to find the optimum thread number.
- The Scott & Knott technique found groups of thread numbers, where every group results in the same gain, in statistical terms. Thus, it is possible to select the lowest thread number in every group and so to save costs to obtain the same TAPT.
- A polynomial equation that describes the TAPT can help to estimate it varying the thread numbers or to estimate the thread number to obtain a determined TAPT.
- The experiment can be adopted for other scenarios, varying message numbers, integration processes, thread numbers, and tested configuration numbers.

We answered the research questions and confirm the initial hypotheses:

*RQ*1: Our algorithm allowed to obtain near-optimal local thread pool configurations of run-time systems of integration platforms that minimised the TAPT. Thus, the algorithm is a helpful tool for software engineers to obtain thread pool configurations.

*RQ*2: We obtained a mathematical model for the TAPT, where it is possible to estimate the TAPT to a given number of threads or to choose the number of threads that results in the desired TAPT.

*H*1: Our PSO-based algorithm found a near-optimal configuration for the thread pools of run-time systems of integration platforms, which resulted in the lowest TAPT of the set of configurations tested.

*H*2: From linear regression statistical technique, obtained a mathematical model for the TAPT as a function of the number of threads.

Following, we list the points that intent to carry out in the experiment, as future work, in order to extend the results of this research.

- Variation in messages number, aiming to analyse the algorithm behaviour in big data scenarios.
- Variation in message arrival rates, aiming to analyse the algorithm behaviour in stream processing systems.
- Variation in the total thread number, aiming to analyse the algorithm behaviour in thrashing scenarios.
- Simulation of the other integration processes, aiming to analyse the algorithm behaviour in different integration logic.
- Increase in tested solution number in the initial population of Algorithm 1, aiming to analyse the algorithm general behaviour.

Acknowledgement

This work was supported by CAPES and FAPERGS under grant 17/2551-0001206-2.

References

Abdulhamid, S., Shafie, A.L. and Idris, I. (2014) 'Tasks scheduling technique using league championship algorithm for makespan minimization in IaaS cloud', *Journal of Engineering and Applied Sciences*, Vol. 9, pp.2528–2533.

Abraham, A.A., King, G.M., Rosa, D.V. and Schmidt, D.W. (2016) *Runtime capacity planning in a simultaneous multithreading (SMT) environment*.

Alexander, C., Ishikawa, S. and Silvertein, M (1977) *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press.

An, J., Kang, Q., Wang, L. and Wu, Q. (2012) 'Population- based dynamic scheduling optimisation for complex production process', *International Journal of Computer Applications in Technology* Vol. 43, No. 4, pp.304–310.

Appel, A.W. (1990) 'A runtime system', *LISP and Symbolic Computation*, Vol. 3, No. 4, pp.343–380.

Aron, R., Chana, I. and Abraham, A (2015) 'A hyper-heuristic approach for resource provisioning-based scheduling in grid environment', *The Journal of Supercomputing*, Vol. 71, No. 4, pp.1427–1450.

Basili, V.R., Rombach, D., Kitchenham, K.S.B., Selby, D., Pfahl, R.W. (2007) *Empirical Software Engineering Issues*, Springer Berlin/Heidelberg.

Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J. and Brandic, I. (2009) 'Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility', *Future Generation computer systems*, Vol. 25, No. 6, pp.599–616.

Byun, E.K., Kee, Y.S., Kim, J.S. and Maeng, S. (2011) 'Cost optimized provisioning of elastic resources for application workflows', *Future Generation Computer Systems*, Vol. 27, No. 8, pp.1011–1026.

Canon, L.C. and Jeannot, E. (2007) 'A comparison of robustness metrics for scheduling DAGs on heterogeneous systems', *Proceedings of the International Conference on Cluster Computing (IEEE Cluster)*, pp.558–567.

Chhabra A., and Oshin (2018) 'Hybrid psacga algorithm for job scheduling to minimize makespan in heterogeneous grids', *Proceedings of the Industry Interactive Innovations in Science, Engineering and Technology (I3SET)*, pp.107–120.

Chirkin, A.M., Belloum, A.S., Kovalchuk, S.V., Makkes, M.X., Melnik, M.A., Visheratin, A.A. and Nasonov, D.A. (2017) 'Execution time estimation for workflow scheduling', *Future Generation Computer Systems*, Vol. 75, pp.376–387.

Chitra, S., Madhusudhanan, B., Sakthidharan, G.R. and Saravanan, P. (2014) 'Local minima jump PSO for workflow scheduling in cloud computing environments', *Proceedings of the Advances in Computer Science and its Applications (CSA)*, pp.1225–1234.

Cruz, C.D. (2006) *Programa Genes: Eestatistica Experimental e Matrizes*, Editora Universidade Federal de Viœsa.

Cruzes, D.S. and Ben Othman, L. (2017) 'Threats to validity in empirical software security research', *Proceedings of the Empirical Research for Software Security*, pp.295–320.

Feldt, R. and Magazinius, A. (2010) 'Validity threats in empirical software engineering research-an initial survey', *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp.374–379.

Frantz, R.Z., Corchuelo, R. and Arjona, J.L. (2011) 'An efficient orchestration engine for the cloud', *Proceedings of the International Conference on Cloud Computing Technology and Science (Cloud-Com)*, pp711–716.

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

Freire, D.L., Frantz, R.Z. and Roos-Frantz, F. (2019a) 'Ranking enterprise application integration platforms from a performance perspective: an experience report', *Software: Practice and Experience*, Vol. 49, No. 5, pp.921–941.

Freire, D.L., Frantz, R.Z., Roos-Frantz, F. and Sawicki, S. (2019b) 'Survey on the run-time systems of enterprise application integration platforms focusing on performance', *Software: Practice and Experience*, Vol. 49, No. 3, pp.341–360.

Georges, A., Buytaert, D. and Eeckhout, L. (2007) 'Statistically rigorous java performance evaluation', *ACM SIGPLAN Notices*, Vol. 42, No. 10, pp.57–76.

Ghosh, T.K. and Das, S. (2018) 'Job scheduling in computational grid using a hybrid algorithm based on particle swarm optimization and extremal optimization', *Journal of Information Technology Research*, Vol. 11, No. 4, pp.72–86.

Ghosh, T.K., Das, S., Barman, S. and Goswami, R. (2017) 'Job scheduling in computational grid based on an improved cuckoo search method', *International Journal of Computer Applications in Technology*, Vol. 55, No. 2, pp.138–146.

Hohpe, G. and Woolf, B. (2004) *Enterprise integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional

IBGE (2017) *Brazilian institute of georgraphy and statistics*. Available online at: www.ibge.gov.br

Indrasiri, K. (2016) *Introduction to WSO2 ESB*, Springer

James, E. and Kelley, J. (1961) 'Critical-path planning and scheduling: Mathematical basis', *Operations Research*, Vol. 9, No. 3, pp.296–320.

Jedlitschka, A. and Pfahl, D. (2005) 'Reporting guidelines for controlled experiments in software engineering', *Proceedings of the International Symposium on Empirical Soft. Engineering (ESEM)*, pp.95–104.

Jian, C., Tao, M. and Wang Y (2014) 'A particle swarm optimisation algorithm for cloud-oriented workflow scheduling based on reliability', *International Journal of Computer Applications in Technology*, Vol. 50, Nos. 3/4, pp.220–225.

Kaytez, F., Taplamacioglu, M.C., Cam, E. and Hardalac, F. (2015) 'Forecasting electricity consumption: a comparison of regression analysis, neural networks and least squares support vector machines', *International Journal of Electrical Power and Energy Systems*, Vol. 67, pp.431–438.

Konsek, H (2013) *Instant Apache ServiceMix How-to*, Packt Publishing.

Lee, J., Wu, H., Ravichandran, M. and Clark, N. (2010) 'Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications', *Proceedings of the International Symposium on Computer Architecture (ISCA)*,, Vol. 38, pp.270–279.

Leonard, N.E. and Levine, W.S. (1995) *Using MATLAB to Analyze and Design Control Systems*, Benjamin-Cummings Publishing Company

Lin, S.W. and Ying, K.C. (2019) 'Makespan optimization in a no-wait flowline manufacturing cell with sequence- dependent family setup times', *Computers and Industrial Engineering*, Vol. 128, pp.1–7.

Linthicum, D.S. (2017) 'Cloud computing changes data integration forever: what's needed right now', *IEEE Cloud Computing*, Vol. 4, No. 3, pp.50–53.

Liu, L., Fan, Q. and Fu, D. (2018) 'A survey of resource allocation in the mobile cloud computing environment', *International Journal of Computer Applications in Technology*, Vol. 57, No. 4, pp.281–290.

Lorenzon, A.F., Cera, M.C. and Beck, A.C.S. (2016) 'Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy', *Journal of Parallel and Distributed Computing*, Vol. 95, pp.107–123.

Manikas, K. (2016) 'Revisiting software ecosystems research: A longitudinal literature study', *Journal of Systems and Software*, Vol. 117, pp.84–103.

Milani, F.S. and Navin, A.H. (2015) 'Multi-objective task scheduling in the cloud computing based on the particle swarm optimization', *International Journal of Information Technology and Computer Science*, Vol. 7, No. 5, pp.61–66.

Pandey, S., Wu, L., Guru, S.M. and Buyya, R. (2010) 'A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments', *Proceedings of the International Conference on Advanced inform. networking and applications (AINA)*, pp.400–407.

Pereira, J.L. and Varajão, J. (2019) 'The temporal dimension of business processes: requirements and challenges', *International Journal of Computer Applications in Technology*, Vol. 59, No. 1, pp.74–81.

Pragaladan, R. and Maheswari, R. (2014) 'Improve workflow scheduling technique for novel particle swarm optimization in cloud environment', *International Journal of Engineering Research and General Science*, Vol. 2, No. 5, pp.675–680.

Ramezani, F., Lu, J. and Hussain, F.K. (2014) 'Task-based system load balancing in cloud computing using particle swarm optimization', *International Journal of Parallel Programming*, Vol. 42, No. 5, pp.739–754.

Ritter, D., Forsberg, F.N., Rinderle-Ma, S. (2018) 'Optimization strategies for integration pattern compositions', Proceedings of the *International Conference of Distributed Event-based Systems (DEBS)*, ACM, New York.

Rodriguez, M.A. and Buyya, R. (2014) 'Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds', *Transactions on Cloud Computing*, Vol. 2, No. 2, pp.222–235.

Russell, J. and Cohn, R. (2012b) *Jitterbit Integration Server*, Book on Demand

Russell, J. and Cohn, R. (2012a) *Fuse ESB*, Book on Demand.

Sargent, R.G. (2013) 'Verification and validation of simulation models', *Journal of simulation*, Vol. 7, No. 1, pp.12–24.

Shishido, H.Y., Estrella, J.C., Toledo, C.F.M. and Arantes, M.S. (2018) 'Genetic-based algorithms applied to a workflow scheduling algorithm with security and deadline constraints in clouds', *Computers and Electrical Engineering*, Vol. 69, pp.378–394.

Sidhu, M.S., Thulasiraman, P. and Thulasiram, R.K. (2013) 'A load-rebalance PSO heuristic for task matching in heterogeneous computing systems', *Proceedings of the Symposium on Swarm Intelligence (SIS)*, pp.180–187.

Subashini, G. and Bhuvaneswari, M (2012) 'Task allocation in distributed computing systems using adaptive particle swarm optimisation', *International Journal of Computer Applications in Technology*, Vol. 44, No. 4, pp.293–302.

Suleman, M.A., Qureshi, M.K. and Patt, Y.N. (2008) 'Feedback-driven threading: power-efficient and high performance execution of multi-threaded workloads on CMPS', *ACM Sigplan Notices*, Vol. 43, No. 3, pp.277–286.

Surhone, L.M., Timpledon, M.T. and Marseken, S.F. (2010) *Petals ESB*, Betascript Publishing.

Touzene, A., Yahyai, S.A. and Oukil, A. (2019) 'Smart grid resources optimisation using service oriented middleware', *International Journal of Computer Applications in Technology*, Vol. 59, No. 1, pp.53–63.

Verma, A. and Kaushal, S. (2015) 'Cost minimized PSO based workflow scheduling plan for cloud computing', *International Journal of Information Technology and Computer Science*, Vol. 7, No. 8, pp.37–43.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B. and Wesslen, A. (2012) *Experimentation in Software Engineering*, Springer Science & Business Media.

Wu, Z., Ni, Z., Gu, L. and Liu, X (2010) 'A revised discrete particle swarm optimization for cloud workflow scheduling', Proceedings of the *International Conference on Computational Intelligence and Security (CIS)*, pp.184–188.

Yao, K. and Liu, B. (2018) 'Uncertain regression analysis: an approach for imprecise observations', *Soft Computing-A Fusion of Foundations, Methodologies and Applications* Vol. 22, No. 17, pp.5579–5582.

Yassa, S., Chelouah, R., Kadimaand, H. and Granado, B (2013) 'Multi-objective approach for energy-aware workflow scheduling in cloud computing environments', *The Scientific World Journal*, pp.1–14.

Zhang, J., Chen, J. and Zhang, H. (2018) 'Job-shop schedule modelling and parents-crossover evolutionary optimisation for integrated production schedules', *International Journal of Computer Applications in Technology*, Vol. 58, No. 4, pp.288–295.

Note

1 http://www.gca.unijui.edu.br/publication/data/ijcat-pso-sources.zip