Characterization of the Jason Multiagent Platform on Multicore Processors

Pascual Pérez-Carro, Francisco Grimaldo*, Miguel Lozano
and Juan M. Orduña
Departament d'Informàtica, Universitat de València
Av. de la Universitat, s/n, 46100, Burjassot, Spain,
Tel: +34 963544487, Fax: +34 963544768, francisco.grimaldo@uv.es

Abstract

Multiagent platforms need to be evaluated focusing on the underlying computer architecture in order to allow developers to exploit the parallelism available in multicore processors. This paper presents the characterization of Jason, a well-known Java-based multiagent platform, when executed on distributed shared memory architectures. Since this kind of architecture is already present in current multicore processors, this should be the first step for the characterization of this platform on distributed systems. To this end, we propose the execution of a set of benchmarks recently proposed for evaluating multiagent platforms. The results obtained show that Jason can be used to program CPU-intensive multiagent applications without loosing the Java scalability over multicore processors. Though, Jason's performance for communication-intensive applications depends on the traffic pattern generated by the agents, the layout of the cores and the selected execution mode (i.e. synchronous or asynchronous).

Keywords: Multiagent platforms, Performance evaluation, Jason, Agent infrastructure benchmarks

1 Introduction

Java-based multiagent platforms are currently used to develop multiagent systems (MAS), since they mainly provide portability as well as an interesting reduction of the development cost. Although these tools work well with a limited number of agents, they do not manage to provide good performance in large-scale implementations [15, 12, 10]. In fact, the absence of clear results has led many researchers to test the performance of general purpose multiagent platforms [20, 26, 1] during the last years. These studies have been mainly focused either on the services or on the messaging layer offered by multiagent platforms involving two or more hosts, thus trying to fully exploit the parallelism in a distributed system. However, no attention has been paid to the architecture of the processor(s) in each host, preventing the multiagent platforms from exploiting the parallelism currently available in the existing multicore (distributed shared-memory) processors.

Therefore, some key issues present in up-to-date processors have not yet been taken into account such as: multicore processors, multi-threaded operating systems, thread mapping, context switch overheads, etc. Notwithstanding, multiagent platforms should take advantage of these features in order to increase their performance even in their centralized versions (i.e. when they are executed on a single host). Since multicore processors are usually present in current computers, the behavior of multiagent platforms should be characterized (and properly tuned) on these processors, so they can become truly efficient platforms for MAS development.

The purpose of this paper is to evaluate the performance of the Jason platform [5] on multicore processors. Jason is a well-known Java-based multiagent platform that is extensively used both in teaching[1] and in doing research on MAS[2]. In spite of that, to the best of our knowledge no other work has yet

[1] For a non exhaustive list of universities teaching Jason see http://jason.sourceforge.net/wp/teaching/

[2] For instance, it has been used in all editions of the Multi-Agent Programming Contest (http://multiagentcontest.org/), also winning in 2006 and 2012.

described the behavior of Jason in regard to multicore systems, which is a relevant issue for the community of agent-oriented programmers in general and for Jason programmers in particular. We have used a set of specific benchmarks that have been proposed for evaluating both general (multithreaded) Java applications [18] and multiagent platforms [24], in an attempt of establishing a reference in the performance for some standard applications. By tuning the Jason multiagent platform and the Java Virtual Machine (JVM), we show how the performance and scalability of Jason greatly depends on the computation and communication requirements of the application executed. On the one hand, the Jason platform can be used to program CPU-intensive multiagent applications without loosing the Java scalability over multicore processors. On the other hand, communication-intensive applications can obtain different performances depending on agent synchronization and on the traffic pattern generated by the agents. Thus, a list of recommendations is provided for increasing the performance of Jason on multicore systems. Otherwise, the overall performance may be impaired by the inefficient use of each processor.

The rest of the paper is organized as follows. Section 2 covers related work on multiagent platforms and their performance. Section 3 shows the characterization setup of the Jason platform and the benchmarks we use. Next, Section 4 details the performance evaluation results. Finally, Section 5 states the concluding remarks and future work.

2 Related work

Multiagent platforms are software frameworks that aim to allow the implementation and running of MAS. A great number of them exists (e.g. Jade [2], MadKit [17], AgentScape [7], Jason [5], JACK [14], etc.), which are different in terms of scope and objectives. Regardless of these differences, these platforms mostly provide developers with common facilities such as agent languages, organizational models, communication layers, execution infrastructures or monitoring tools, that ease multiagent programming [4].

During the last years, a lot of work has focused on testing the performance of existing multiagent platforms[1, 8, 9, 12, 20, 24, 26]. One of the main performance metrics considered has been the response time (i.e. round-trip delay) for the messages exchanged among agents. A comparison study of four different Java-based multiagent platforms [26] shows that Jade provides the most efficient messaging service, whereas ZEUS [11] would be the worst one. Another comparative evaluation of three multiagent platforms [8], that again focuses on the messaging service, also concludes that Jade provides the best performance because it is built on Java RMI.

Some other work focuses on the different services provided by a given multiagent platform. For instance, some researchers have studied the Jade services of messaging, agent creation and migration [10]. Though, these tests focusing on messaging services only consider eight pairs of agents, which is rather a low load. Other authors have measured the scalability and performance of the Jade messaging service [12], but they have not provided hints about the behavior of these platforms when executed on multicore computer architectures.

More recent proposals have made a deeper comparative study of three open-source multiagent platforms (namely Jade, AgentScape and MadKit), focusing not only on messaging and service directory services, but also on the memory usage and the network bandwidth required by the platforms [20, 1]. These approaches have again concluded that Jade provides the best performance when MAS are to be run on a distributed setting.

Particularly relevant to this paper is the previous work on the characterization and tuning of Jason for running interactive multiagent simulations [15, 16]. This work has described the performance of Jason when using the Jade infrastructure for distributed executions and it has detected some issues related to the JVM that can prevent Java-based multiagent platforms to provide good performance and scalability for the sort of applications considered (e.g. crowd simulations). Nevertheless, the performance evaluation of the Jason platform on multicore processors was yet to be done and it is the main goal of this paper.

3 Characterization Setup

Jason [5] is a Java-based interpreter for an extended version of AgentSpeak [22], a BDI agent-oriented logic programming language. The BDI (Belief-Desire-Intention) model of rational agency has been widely relevant in the context of artificial intelligence and MAS due to its strong philosophical assumptions, such as: the intentional stance [13], the theory of plans, intentions and practical reasoning [6] and the speech acts theory [23]. These three notions of intentionality [19] provide suitable tools to describe agents at an appropriate level of abstraction and, at the level of design, they invite to develop computer programs as if they had a mental state.

According to the agent-based paradigm, MAS are basically composed of multiple intelligent agents interacting within a shared environment, which represent the two main components present in Jason. Combining this paradigm with organisation-oriented programming and environment-oriented programming [3] has been lately proposed for modelling and designing virtual environments that involve dynamic sets of artifacts of different kinds, aside to the models and platforms used to program agents. Although evaluating artifacts falls out of the scope of this article, the characterization of the core Jason platform presented here is the first step towards the evaluation of this new programming paradigm, which remains as future work. Thus, this paper uses version 1.3.10 of the Jason platform, which is the latest stable release available to the community at the moment of writing this paper.

3.1 Jason infrastructures

Jason currently provides two infrastructures to execute MAS: *Centralized* and Jade [2]. Whereas the *Centralized* infrastructure places all the components of the MAS in the same host, it is also possible to distribute these components in several hosts using the Jade technology. As already mentioned, this paper focuses on the *Centralized* infrastructure when executed on multicore processors, as a first necessary step for the characterization of this multiagent platform on distributed infrastructures. For further details on how Jason uses distributed infrastructures see [15].

When using Jason's *Centralized* infrastructure all MAS components (i.e. the agents and the environment) are placed on a single host. Figure 1 shows a scheme of the *Centralized* infrastructure, where different threads are assigned to perform concrete tasks. On the one hand, the environment has its own execution thread and it is provided with a configurable pool of threads (PThE) devoted to executing the actions requested by the agents. By default, this pool is composed of four threads, thus allowing the environment to deal with several agent requests concurrently. On the other hand, each agent owns by default a thread in charge of executing the agent reasoning cycle, which is broken down into the steps of: perceive, reason and act. In this manner, all the agents can also run concurrently within the MAS. As such, this approach could limit the number of agents that can be executed, since the total number of threads would be limited by the JVM heap size. However, Jason offers the option of having a configurable pool of threads (PThA), so that the agents can share the threads in this pool at the cost of reducing the level of concurrency. The number of threads in both PThE and PThA is initialized during the start-up of the MAS and it is not changed along its execution. By default, the PThE holds 4 threads whereas the PThA is disabled, so that each agent will have its own execution thread. Agent-agent as well as agent-operating system communication is performed through event passing in the *Centralized* infrastructure. That implies that messages are not serialized and sequentially sent through a network but their content is cloned within the JVM heap and object references are passed between sender and receiver threads.

Aside from the infrastructure, the Jason platform offers three different execution modes: *asynchronous*, *synchronous* and *debugging* [5]. In the *asynchronous* mode, which is the one used by default, an agent goes to its next reasoning cycle as soon as it has finished its current cycle. Instead, in the *synchronous* mode all agents perform one reasoning cycle at every "global execution step". This mode waits until all agents have finished their reasoning cycles and then sends the "carry on" signal to them. Finally, the *debugging* execution mode is similar to the synchronous mode, except that Jason waits until the user clicks a "Step" console button before sending the "carry on" signal to the agents.

The execution mode is a parameter that can have significant effects on the performance of the multiagent platform. Also, the existence and the size of the PThA pool of threads will play an important role on the performance of the Jason when it is executed on a multicore processor. Therefore the effects of these two parameters will be studied in section 4.

3.2 Tests description

Since multiagent systems can support either computation-intensive or communication-intensive applications, we have considered both for characterization purposes. The purpose of the tests described below is to measure (characterize) the behavior (performance) that can be obtained with the Jason multiagent platform when executing these types of applications. In particular, we want to discover to which extent Jason can take advantage of the multicore architecture that is present in most of the current processors. In order to achieve this goal, we have performed tests that have been accepted as standard benchmarks for multithreaded Java applications and that represent the two extreme points of the computation-communication spectrum, ensuring that any application will fit within the range considered for this characterization study.

3.2.1 Computation Test

In order to evaluate the performance achieved with Jason for computation-intensive tasks, we have adapted a simple multithreaded CPU benchmark that was originally created to evaluate the multicore capacity of general (multithreaded) Java applications [18]. The test simply measures the CPU cycles consumed by each thread when executing arithmetic operations on a 10^9 size data structure. The sum of the total CPU cycles consumed by all the threads should be equal to the total execution time multiplied by the number of cores used. Any difference between these two values is due to the overhead imposed by the configuration of the JVM and/or the operating system kernel.

We have adapted this test to Jason by implementing simple agents that perform the computations corresponding to each thread in the considered test. We have run this test with different number of threads (agents) for population sizes ranging from 1 to 1024 agents. We have repeated the execution for different platforms, containing from 2 to 16 processing cores. We have computed the ratio between the total CPU time of the threads and the total elapsed time of the execution. This factor shows the acceleration (speed-up) provided by the considered multicore configuration (number of cores).

3.2.2 Communication Tests

In order to evaluate the performance achieved with Jason for communication-intensive tasks, we have applied two of the benchmarks proposed in [24] for testing Jason when running on a single host using the *Centralized* infrastructure. These benchmarks consist of a number of agents exchanging echo-like messages by following different communication patterns. Additionally, there is also a controller agent in charge of synchronizing the start and the finalization of the execution. Figure 2 depicts the communication scenario reproduced by each benchmark.

- Communication Test 1 - Number of agents: This test consists of a set of agents as shown in Figure 2a. All agents behave in the same way, and they are labeled as $Agent_1$ up to $Agent_N$ (with N being the number of agents set by the user). The $Agent_i$ starts sending an echo request message to $Agent_{i+1}$ and waits for the echo reply. Then, it proceeds circularly with $Agent_{i+2}$, $Agent_{i+3}$, and so on. Each agent keeps on sending messages in this way until it sends the number of messages chosen by the user. The purpose of this test is to measure the capacity of Jason for sending messages to different agents placed in the same host. The test attempts to determine how the platform behaves when the number of agents is increased. This test involves a lot of message exchanges between every pair of agents in the multiagent system.

- Communication Test 2 - Massive Reception: In this test, there is a set of sender agents and a set of receiver agents (see figure 2b). Both the number of sender agents (N) and receiver agents (M) are parameters to be set by the user. However, each sender agent exclusively sends messages to its corresponding receiver agent. On the contrary, each receiver agent echoes the arriving messages to the subset of senders from which it receives echo request messages. Therefore, this test can model scenarios where there are one or many hot-spot traffic patterns generated by one or many overloaded agent(s) (e.g. service provider(s)). This test evaluates Jason response in massive reception configurations. When $M = 1$, the purpose of this test is to measure the platform throughput when an agent has a high rate of message exchanges. When setting $N = M$, this test checks the scalability and efficiency of Jason, thus showing how the performance of Jason evolves in large-scale systems for balanced traffic patterns. Finally, for assesing the coherence of the test results, when fixing the value for N and ranging the value of M from 2 to N the results should show symmetric behavior with the ones in the test of $M = 1$.

The performance evaluation carried out in section 4 measures the average round trip time (RTT) of the messages exchanged amongst the agents, that is, the time elapsed between an agent sending an echo request and receiving the echo reply from the receiver agent. The final RTT value of a test execution will then be the average of the RTT obtained by each agent for each message sent. The number of messages sent per agent is a parameter that the final user of the test can configure in order to achieve the needed accuracy level. Regarding the number of agents, in this paper we show values ranging from 8 to 256 agents (or pairs of agents). The performance of Jason when less than 8 agents are present showed no significant differences and it is considered irrelevant for the characterization purposes of this paper, as it implies a very low system load.

The other two benchmarks proposed in [24] consider different traffic distributions for a distributed infrastructure. However, when using a Centralized infrastructure all these cases are represented by the Test 2 explained above.

3.3 Processor architecture and Java tuning

The tests described in section 3.2 have been executed on a distributed shared memory 16-core processor (AMD Opteron 8218, 1.0 GHz) with 32 GB of RAM, running a 64-bit version of Linux and the Sun's HotSpot^{TM}JVM release 1.7.0_25. Concretely, the hardware platform used in these tests is a Sun Fire X4600. The computer architecture of this server consists of eight dual-core processors, interconnected among them as shown in Figure 3. According to the manufacturer [25], average hop distance increases when building eight-socket systems when compared to that of four-socket systems. Minimizing the number of hops is ideal, and Sun Fire X4600 servers accomplish minimum hop distance by enhancing the ladder and twisted ladder topologies as shown in Figure 3. Nevertheless, different hop distances will be present when communicating 2, 4, 8 or 16 cores, depeding on the assignments of threads to processor cores. These differences can have signifficant performance effects, as shown in section 4.

From version 1.5, the JVM incorporates a technology that tunes itself and is referred to as Ergonomics. Even though Ergonomics significantly improves the performance of many applications, optimal results often require manual tuning to better fit the application as well as the underlying hardware [21]. Following the general Java tuning recommendations, obtained from previous work for running multiagent simulations over Jason [16], we have performed the following tuning:

- We have enlarged the JVM heap size up to 2 GB (greater values have demonstrated not to provide better results) instead of using the default values, which would have been 1/64th of the machine's physical memory for the initial heap size and 1/4th of the machine's physical memory for the maximum heap size. In addition, we have set minimum and maximum heap sizes equal for a faster startup. To set the minimum and maximum heap size we use the `-Xms2048M` and `-Xmx2048M` JVM command-line options.

- We have parallelized garbage collection by using the *throughput* Garbage Collector (GC) in order to reduce GC pause times. We have used the JVM command-line option `-XX:+UseParallelGC`, thus setting the number of collector threads equal to the number of processor cores, that is, we use 16 collector threads.

- We have increased the size of the *young* generation up to 90% of the JVM heap size, thus reducing the size of the *tenured* generation responsible for slow major collections. The reason behind this tune relies on how Jason represents agent's beliefs and actions. Both are implemented as objects that are discarded and created again whenever there is a change in a belief or a new action is requested to the agent environment. Due to the huge amount of objects that "die young", enlarging the *young* generation will benefit garbage collection. To do it we use the `-XX:NewSize=1842` JVM command-line option.

Summing up, the Jason multiagent platform benefits from the following JVM setup: (i) the JVM heap size should be enlarged up to 2 GB and both the minimum and the maximum heap sizes should be set to this value for a faster startup; (ii) the *throughput* Garbage Collector should be selected, with a number of collector threads equal to the number of processor cores, so to parallelize garbage collection and reduce GC pause times; and, (iii) the size of the *young* generation of the JVM heap size should be increased up to 90%, thus fastening garbage collection by reducing the number slow major collections.

4 Performance Evaluation

In this section, we focus on the performance of the centralized infrastructure of Jason when it is executed on a multicore processor. The results discussed here have been obtained by running the tests [3] explained in section 3.2 using Jason (version 1.3.10), the Java HotSpot^{TM}JVM (version 1.7.0_25) and the processor architecture described in section 3.3.

Since we are considering the two extreme points of the computation-communication spectrum any MAS application is located in, the evaluation involves several tests. First, we compare the performance obtained for a computation-intensive test by both Java and Jason. Second, we present the results for the communication-intensive tests explained in section 3.2. In this case, we have tested the two execution modes provided by Jason (i.e. the *synchronous* and the *asynchronous* mode) and we have also studied the effects of the pool of agent threads (PThA) present in the asynchronous mode.

All the values in the figures shown in this section have been computed as the average value of ten different executions of the same configuration.

4.1 Computation Test

Figure 4 plots the speed-up obtained by two implementations (Java vs Jason) of the computation test described in section 3.2.1 when executed with different numbers of cores. This figure shows the achieved speed-up on the Y-axis, and it shows the number of simulated agents on the X-axis. The results shown in this figure demonstrate that for populations up to sixteen agents, the speed-up obtained proportionally increases with the number of existing cores, and the Jason implementation provides higher speed-up rates than the Java implementation. From that point up, the speed-up remains more or less constant in the

[3] All tests can be downloaded from http://www.uv.es/grimo/publications/JasonTests.zip in order to reproduce the experiments.

Java implementation and even decreases for the Jason implementation. However, the differences between the speed-up provided by both implementations for the worst case (the largest population size) is not greater that $Log_2(N)$, where N is the number of existing cores. These results show that Jason can be used to program CPU-intensive multiagent applications without significantly loosing Java scalability over multicore processors, since the overhead added with respect to the Java implementation is logarithmic with the number of cores in the processor.

4.2 Synchronous mode

Once we have analized the behavior of Jason with computation-intensive applications, we now study its behavior with communication-intensive ones. Figure 5 shows the evaluation results for the synchronous execution mode in terms of response times (Round Trip Times, RTTs) when increasing the number of agents in the simulation.

Figures 5 (a1) and (a2) plot the results for Test 1, that measures the capacity of Jason for sending messages to different agents placed in the same host. The figure on the left shows the performances obtained for Test 1 when increasing the number of cores from 1 up to 16. The figure on the right omits the plot corresponding to 1 core in order to let us see in more detail the differences among the plots referring to 2, 4, 8 and 16 cores. As shown by the figure on the left, when executed on a single core the system reaches saturation with only 32 agents and, thereafter, RTTs hugely increase as more agents are added to the simulation. When using two or more cores, though, the system can support up to 256 agents without reaching saturation. Note that the values plotted in the figure on the right present little or no slope and that all of them fall into a narrow range varying from around 6 to 16 milliseconds. It is also worth mentioning that no significant improvements are made when moving from 8 to 16 cores. The reason for the behavior is that using more than 8 cores implies employing more than 4 dual-core processors from the platform shown in Figure 3, which in turn increases the number of hops that are necessary to communicate. For instance, the maximum number of hops to connect the four processors in the middle of Figure 3 is 2 while it goes up to 3 as soon we add another processor. These results show that for the synchronous mode and a communication pattern like the one in Test 1,the best hardware platform is the one composed of 8 cores.

Figure 5 (b) shows the results for Test 2 when $M = 1$, that is, when a number of agents send messages to a single receiver agent. In this case, the figure shows that the system reaches saturation with 64 senders, regardless of the number of cores. Effectively, the slope in the plots shows a monotonic increasing slope as more than 64 agents are considered (that is, the saturation point in this case is a workload of 64 agents). The reason for this behavior is that the single receiver becomes the system bottleneck. From 64 senders up, the effect of the number of cores is to reduce the latency achieved in deep saturation[4], since each plot shows increasing RTT values as the number of cores used is lower.

Figure 5 (c) shows the results for Test 2 when $M = N$, that is, when each sender has a receiver as its counterpart. Therefore, the X-axis shows the number of pairs of sender-receiver agents. Since the traffic pattern in this test is much more balanced than in the prior case, the absolute RTT values are much lower in this case. However, the same general behavior applies: as the number of cores increases, the RTT values provided in deep saturation (64 agents or more) are lower. Also, it is worth noting the different behavior for the case of 1 core, that is much worse than the rest of the cases. This behavior shows that the use of any multicore hardware platform will significantly improve the performance of the application running on the multiagent platform.

Finally, Figure 5 (d) shows the results for Test 2 when $N = 256$, that is, when there is a fixed number of 256 sender agents and we increment the number of receivers from 2 up to N. Hence, in this case the X-axis shows the number of receiver agents. As it could be expected, the highest RTT values are obtained for the lower number of receivers, since they become the system bottleneck.

[4] Deep saturation refers to a situation where the workload supported by the system is far beyond the workload at the saturation point.

Again, the plots show that the higher the number of cores, the lower the RTT values obained in deep saturation (2-32 receivers). Also, the plot for a single core shows much higher RTT values, as it could be expected.

These results show that for the Test1 the most efficient platform is composed by 8 cores, and the effects of the number of cores do not depend on the system workload (the number of agents in the system). For the Test 2, the significant effects of the number of cores only appear when the system is in deep saturation. In that situation, the maximum number of cores (16 in our case) obtains the best results.

These results show that for the Test1 the most efficient platform is composed by 8 cores, and the effects of the number of cores do not depend on the system workload (the number of agents in the system). For the Test 2, the significant effects of the number of cores only appear when the system is in deep saturation. In that situation, the maximum number of cores (16 in our case) obtains the best results.

4.3 Asynchronous mode

In this subsection we present the results for the asynchronous execution mode in its default configuration. Therefore, the following results have been obtained with the PThA pool disabled, being each agent handled by its own Java thread. In this way, potential synchronization barriers are avoided so as to achieve a truly asynchronous execution. Figure 6 shows the evaluation results for this asynchronous execution mode in terms of response times (Round Trip Times, RTTs).

Concretely, Figure 6 (a) shows the results for Test 1. The plots in this figure show that the most efficient hardware platform for workloads up to 16 agents is the one based on four cores, since it is the minimum platform that provides RTTs close to 3 milliseconds. For workloads greater or equal to 32 agents, the most efficient platform is the one based on 8 cores, since that plot shows no significant differences with the plot corresponding to 16 cores.

Figure 6 (b) shows the results for the Test 2 with $M = 1$. This figure also shows how RTTs decrease as the number of cores used for the execution increases, although in this case there are no significant differences among 4, 8, and 16 cores.

Figure 6 (c) shows the results for Test 2 with $M = N$. This figure shows how the RTT values provided when the system enters saturation (workloads of 32 agents or more) decrease as the number of cores increases, except for the plot corresponding to 16 cores, where no significant gain is obtained.

The reason for the behavior shown in these three figures is again that when increasing the number of cores, the distance (in hops) for the messages exchanged among threads also increases. Since the number of hops between a given pair of threads depends on the thread mapping and the underlying processor architecture, we have checked the number of hops when using a different number of cores and a sequential thread assignment (thread 0 to core number 0 and so on), and we have verified that when using 16 cores instead of 8, the core layout of the AMD Opteron 8218 (shown in figure 3) imposes one additional hop for communicating half of the cores with the other half[5]. This constraint limits the system performance for communication intensive benchmarks.

Finally, Figure 6 (d) shows the results for Test 2 with $N = 256$. This figure shows that significant differences in the RTT values are obtained as the number of cores increases, except for the case of 16 cores, whose plot shows no significant differences with the one corrresponding to 8 cores. Also, it is worth mentioning that all the plots are straight lines with no significant slopes.

These results show that by configuring Jason to have each agent handled by its own Java thread, the system is able to exploit the computation power of the different cores. However this situation ends with 8 cores, and no benefit is obtained beyond this number of cores. Again, the reason for this behavior is the core interconnection layout (shown in Figure 3). The benefits of adding more

[5] We have also verified this fact when running our benchmarks on a different hardware platform, namely an Intel i7-3930K with six cores and 12 threads. In this case, using 8 threads outperformed the situation in which the 12 threads were used.

cores is hidden by the additional hops required in the communication of these cores with the rest of the cores.

4.4 Asynchronous mode with Pool of Threads (PThA)

Jason's agent pool of threads (PThA) is a specific parameter that can affect the performance of the asynchronous execution mode. In this section we study the effect of the PThA pool of threads on the performance achieved by multi-core processors for communication intensive-applications and the asynchronous execution mode. Since the available hardware platform for this characterization study was a 16-core processor, we have varied the PThA pool size from 1 to 16 threads. Also, we have tested the PThA size for platforms with 2, 4, and 8 cores. Although the graphic results are not shown here for the sake of shortness, we have found a "rule of thumb" for tuning the PThA pool size to the number of existing cores. It must be noticed that apart from the PThA size used, when using the Centralized Jason infrastructure, the system also runs the environment threads (1 + 4, from the PThE), as well as a reduced number of scheduling and management threads. Therefore, when using a PThA size of 16 threads in the case of 16 cores, the number of Java threads exceeds the number of processor cores, and the context switch overhead significantly affects performance. Similar results were obtained for other number of existing cores in the computer platform. As a result, and considering the complete set of experiments performed to obtain the best values for PThA, we have selected 2, 4, 6 and 10 as the best PThA values for 2, 4, 8 and 16 cores, respectively.

Figure 7 shows the RTTs measured when executing the communication tests for different number of processor cores. Figure 7 (a) shows the results for Test 1. The figure shows how the number of threads used in the pool directly affects the performance obtained since, when the load is lower than 64, the best results are obtained with 4 and 8 cores (using 4 and 6 threads respectively). For higher loads the differences between both configurations start to increase. We notice that for 16 cores the times obtained are always slightly worst, which is again due to the increase in the hop distances that must traverse the messages exchanged among the threads involved.

Figure 7 (b) shows the results for Test 2 with $M = 1$. The plots in this figure show lower RTT values than the equivalent ones in figure 6 (b), showing a reduction in the time required. This difference shows that the number of active threads in the system is also another important factor. The PThA parameter should be enabled with a number of threads close (but lower than) the number of processing cores.

Figure 7 (b) shows the results for Test 2 with $M = 1$. The plots in this figure show lower RTT values than the equivalent ones in Figure 6 (b), showing a reduction in the time required. This difference shows that the number of active threads in the system is also another important factor. Thus, the PThA parameter should be enabled with a number of threads close (but lower than) the number of processing cores.

Figure 7 (c) shows the results for Test 2 with $M = N$. The plots in this figure also show lower RTT values than the equivalent ones in Figure 6 (c), showing a more moderate growth for the same workload. However, the shape of the plots and the relative performance achieved with each number of cores are similar.

Finally, Figure 7 d) shows the results for Test 2 with $N = 256$. In this case, it can be seen that ther is a significant reduction of the RTT values in all the plots from 8 to 32 receivers, ans a much lower reduction from 32 to 256 receiver agents. Comparing this figure with Figure 6 (d) it can be seen that in the latter case the plots seems flat lines for all the considered number of receivers, while Figure 7 d) shows a huge reduction of the RTT values as the number of receivers increases. These results show that the Pool of Threads allows to exploit the potencial of the existing cores for this traffic pattern.

Finally, Figure 7 (d) shows the results for Test 2 with $N = 256$. In this case, it can be seen that there is a significant reduction of the RTT values in all the plots from 2 to 32 receivers, and a much lower reduction from 32 to 256 receiver agents. Comparing this figure with Figure 6 (d) it can be seen that in the asynchonous execution mode with the PThA disabled the plots seem flat lines for all the considered number of receivers, while Figure 7 (d) shows a huge reduction of the RTT values as the number of receivers increases. These results show that the pool of threads for the agents (PThA) allows to exploit the potencial of the existing cores for this traffic pattern.

4.5 Analysis of the speed-up

In order to provide the reader with a clear understanding of the effective improvement that the results shown in the previous sections represent, in this section we show the speed-up achieved when adding new processing cores to the execution of communication-intensive applications. Thus, we have computed the speed-up as the ratio between the total time obtained for a single core with respect to that obtained when increasing the number of cores. It must be noticed, though, that by computing the speed-up in this way, the maximum achievable speed-up value is not limited by the number of cores considered (like in the standard speed-up parameter considered for measuring the performance of parallel and/or distributed systems or applications). The reason is that the single core system enters saturation and, therefore, the execution times are no longer linear with the number of cores.

Figure 8 shows the speed-ups obtained when running the tests using the synchronous execution mode. Figure 8 (a) shows huge speed-up values (e.g. nearly 70 for 8 and 16 cores). These huge values are due to the fact that the system is in deep saturation when executing this test in a single core for 256 agents, providing huge execution times.

Figures 8 (b), (c) and (d) show that the best speed-up is always obtained by the plot corresponding to 16 cores, and it ranges from 6 (in Figure 8 (c)) to 9 (in Figure 8 (d)). It is also worth mentioning that in Figures 8 (c) and (d) the best speed-up is obtained for the case of 32 pairs of agents, just twice the number of cores. These results show that the traffic patterns where Jason fully exploits the parallelism offered by multicore architectures are the balanced communication traffic patterns. The reason for this behavior is that the synchronous mode acts as a barrier, since agents must wait till the rest of the agents have completed the current cycle. This waiting time prevents the multicore processors from exploiting the parallelism of multiple agents handled by independent threads, unless all the threads support similar workloads (like in balanced communication patterns).

Figure 9 shows the speed-ups obtained when running the communication tests using the asynchronous execution mode with the PThA disabled. Figure 9 (a) shows that the plots for 2 and 4 cores are flat lines, while the plots for 8 and 16 cores are straight lines with a little positive slope. The speed-up values achieved are far from being linear to the number of cores, and even the speed-up values achieved with 8 cores are higher than the ones achieved with 16 cores. As indicated when analyzing the RTT values, the reason for this behavior is that the layout of the cores in the AMD Opteron 8218 processor, together with the sequential thread assignment, impose additional hops for communicating the existing cores, particularly when 16 cores are used.

Figures 9 (b), (c) and (d) show speed-up values that are proportional (but not linear) to the number of cores. The speed-up values in Figures 9 (c) and (d) are huge, due to the fact that the configuration with a single core is in deep saturation and it provides huge RTT values. It is also worth mentioning that the slope for 16 cores showed in Figure 9 (c) is constant, since we have communication between pairs of agents, but this slope dissappears in Figure 9 (d), since we have a fixed number of senders for all the considered numbers of receivers.

Finally, Figure 10 shows the speed-up obtained when running the communication tests using the asynchronous execution mode with the PThA enabled. In this case the number of running threads is being limited by the pool size. Figures 10 (a) and (b) show very similar results to the ones shown in Figures 9 (a) and (b). Figure 10 (c), however, shows quite different plot shapes than the ones in Figure 9 (c). The reason for this behavior is that limiting the number of threads prevents the multicore configurations to obtain speed-up values similar to the ones obtained with a single thread per agent. Finally, Figure 10 (d) shows similar plot shapes than the ones in Figure 9 (d), although the values on the Y-axis are lower in Figure 10 (d) due to the same reason we have just stated.

Therefore, we can conclude that enabling of the PThA has significant effects when executing applications in the asynchronous execution mode, as it limits the speed-up obtained with the number of cores and increases the execution time of the application. However, the average RTTs obtained for the messages exchanged in Test 2 are much lower when the PThA parameter is enabled and its value is slightly lower than number of existing cores (Figures 7 (b), (c) and (d) show much lower values on the Y-axis than Figures 6 (b), (c) and (d). Thus, the best configuration of the Jason platform will depend on the application. If the critical parameter to be optimized is the latency of the messages (interactive applications), then the PThA parameter should be enabled. Otherwise, this parameter should be disabled.

5 Conclusions and Future Work

In this paper, we have proposed the characterization of the Jason platform on multicore processors by using a set of benchmarks that have been recently proposed for Java-based multiagent platforms. Unlike other proposals, we have focused on the Jason's *Centralized* infrastructure and on the underlying processor architecture, thus tuning Jason in order to exploit the parallelism currently available in the existing multicore (distributed shared-memory) processors.

The results obtained show that the performance and scalability of Jason over a multicore processor greatly depend on the type of application being executed. On the one hand, Jason can be used to program CPU-intensive multiagent applications without loosing the Java scalability over multicore processors, because the overhead added with respect to a pure Java implementation is logarithmic with the number of cores. On the other hand, Jason provides different performances for communication-intensive applications, that depend on the traffic pattern generated by the agents and the selected Jason execution mode (i.e. synchronous or asynchronous). For applications showing an uniform traffic pattern (Test 1), the layout of the cores in the processor can limit the best number of cores to be used, regardless of the execution mode. For asymmetric traffic patterns, instead, the better performance is achieved (in terms of speed-up) as more cores are used, since the effects of longer paths between some cores are hidden by the unbalanced communication pattern. Regarding the execution modes, both of them efficiently exploit the avaliable number of cores. Nevertheless, in the case of applications using the asynchronous execution mode, enabling the PThA significantly reduces the average latency of messages. Thus, for applications requiring interactivity, the best configuration for exploiting the number of cores in the processor is to enable the PThA and to adjuts its size so as the total number of running threads does not exceed the number of processing cores. For applications that do not require message latency constraints, though, the best total execution times are obtained when the PTha parameter is disabled.

As future work, we plan to investigate the performance of Jason for applications that differently balance computation and communication, thus covering the gap in between the two extreme types of applications considered in this paper. Moreover, evaluating the effects of adding Cartago artifacts and distributed virtual environments to the performance of Jason remains as a open issue to be studied.

Acknowledgment

This work has been jointly supported by the Spanish MICINN and the European Commission FEDER funds, under grant TIN2009-14475-C04.

References

[1] J. M. Alberola, J. M. Such, A. Garcia-Fornes, A. Espinosa, and V. Botti. A performance evaluation of three multiagent platforms. *Artif. Intell. Rev.*, 34:145–176, August 2010.

[2] F. Bellifemine, G. Caire, A. Pogg, and G. Rimassa. JADE - a white paper. Technical Report 3, Telecom Italia Lab, EXP Online, 2003.

[3] O. Boissier, R. H. Bordini, J. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, June 2013. Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni. *Multi-agent programming : languages, platforms and applications.*, volume 15 of *Multiagent systems, artificial societies, and simulated organizations.* Springer, New York, 2005.

[5] R. H. Bordini, J. F. Hübner, and M. Wooldrige. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley, 2007.

[6] M. E. Bratman. *Intention, Plans, and Practical Reason.* Cambridge University Press, Mar. 1999.

[7] F. Brazier, D. Mobach, B. Overeinder, S. Van Splunter, M. Van Steen, and N. Wijngaards. Agentscape: middleware, resource management, and services. In *Proceedings of the 3rd international SANE conference*, pages 403–404, 2002.

[8] K. Burbeck, D. Garpe, and S. Nadjm-Tehrani. Scale-up and performance studies of three agent platforms. 2004.

[9] K. Chmiel, M. Gawinecki, P. Kaczmarek, M. Szymczak, and M. Paprzycki. Efficiency of JADE agent platform. *Sci. Program.*, 13(2):159–172, 2005.

[10] K. Chmiel, D. Tomiak, M. Gawinecki, P. Karczmarek, M. Szymczak, and M. Paprzycki. Testing the efficiency of JADE agent platform. In *Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, ISPDC '04, pages 49–56, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. C. Collis, D. T. Ndumu, H. S. Nwana, and L. C. Lee. The ZEUS Agent Building Tool-kit. *BT Technology Journal*, 16:60–68, 1998. 10.1023/A:1009673714049.

[12] E. Cortese, F. Quarta, and G. Vitaglione. Scalability and performance of JADE message transport system. In *AAMAS Workshop on AgentCities*, 2002.

[13] D. C. Dennett. *The Intentional Stance (Bradford Books).* The MIT Press, Cambridge, MA, reprint edition, Mar. 1987.

[14] R. Evertsz, M. Fletcher, R. Jones, J. Jarvis, J. Brusey, and S. Dance. Implementing Industrial Multi-agent Systems Using JACK. pages 18–48. 2004.

[15] V. Fernández, F. Grimaldo, M. Lozano, and J. M. Orduña. Evaluating Jason for distributed crowd simulations. In *Proc. of the 2nd. International Conference on Agents and Artificial Intelligence*, volume 2, pages 206–211, 2010.

[16] V. Fernández, F. Grimaldo, M. Lozano, and J. M. Orduña. Tuning Java to run interactive multiagent simulations over Jason. In *Proc. of the 23rd Australasian Joint Conference on Artificial Intelligence (AI 2010)*, pages 354–363. Springer-Verlag, 2010.

[17] O. Gutknecht and J. Ferber. The MadKit agent platform architecture. In *Revised Papers from the International Workshop on Infrastructure for*

Multi-Agent Systems: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, pages 48–55, London, UK, 2001. Springer-Verlag.

[18] H. M. Kabutz. Are you really multi-core?, 2012. Available at http://www.javaspecialists.eu/archive/Issue135.html.

[19] W. Lyons. *Approaches to Intentionality.* Oxford University Press, Oxford, 1997 1997.

[20] L. Mulet, J. M. Such, and J. M. Alberola. Performance evaluation of open-source multiagent platforms. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1107–1109, New York, NY, USA, 2006. ACM.

[21] Oracle Sun Depeloper Network. Java Tuning White Paper, 2010. Available at http://java.sun.com /performance/reference/whitepapers/tuning.html.

[22] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In S. Verlag, editor, *Proc. of MAAMAW'96*, number 1038 in LNAI, pages 42–55, 1996.

[23] J. Searle. *Expression and Meaning: Studies in the Theory of Speech Acts.* Cambridge University Press, 1979.

[24] J. M. Such, J. M. Alberola, L. Mulet, A. Espinosa, A. Garcia-Fornes, and V. Botti. Large-scale multiagent platform benchmarks. In *LAnguages, methodologies and Development tools for multi-agent systemS (LADS 2007). Proceedings of the Multi-Agent Logics, Languages, and Organisations - Federated Workshops*, LADS'07, pages 192–204, 2007.

[25] Sun fire x4600 m2 server architecture. Technical report, Sun Microsystems, Inc, 2008.

[26] P. Vrba. Java-based agent platform evaluation. In V. Mark, D. McFarlane, and P. Valckenaers, editors, *Holonic and Multi-Agent Systems for*

Manufacturing, volume 2744 of *Lecture Notes in Computer Science*, pages 1086–1087. Springer Berlin Heidelberg, 2004.

Figure captions

Fig 1. Overview of the *Centralized* infrastructure.

Fig 2. Schema of the benchmarks implemented in Jason.

Fig 3. Twisted ladder topology of the processing cores in Sun Fire X4600 server.

Fig 4. Speed-up achieved with the computation test executed on multicore processors.

Fig 5. RTT obtained for the synchronous execution mode.

Fig 6. RTT obtained for the asynchronous execution mode with PThA disabled..

Fig 7. RTT obtained for the asynchronous execution mode with PThA enabled..

Fig 8. Speed-up obtained for the synchronous execution mode.

Fig 9. Speed-up obtained for the asynchronous execution mode with PThA disabled.

Fig 10. Speed-up obtained for the asynchronous execution mode with PThA enabled..

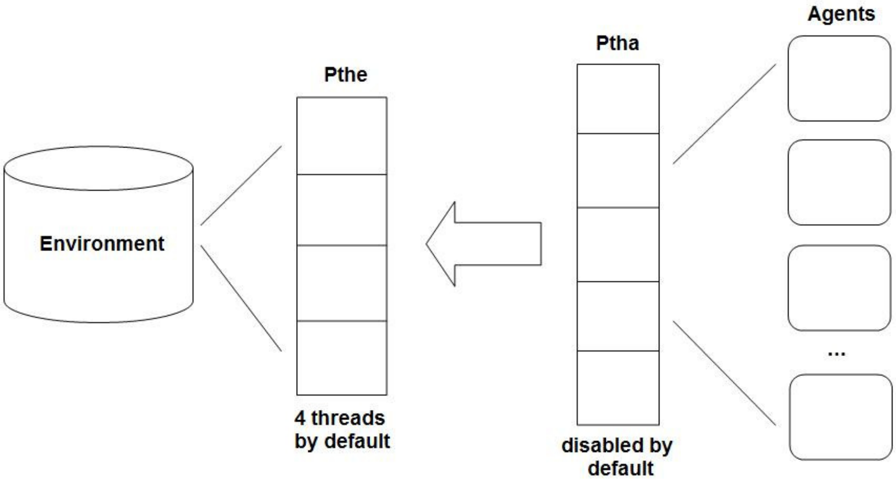


Figure 1: Overview of the *Centralized* infrastructure.

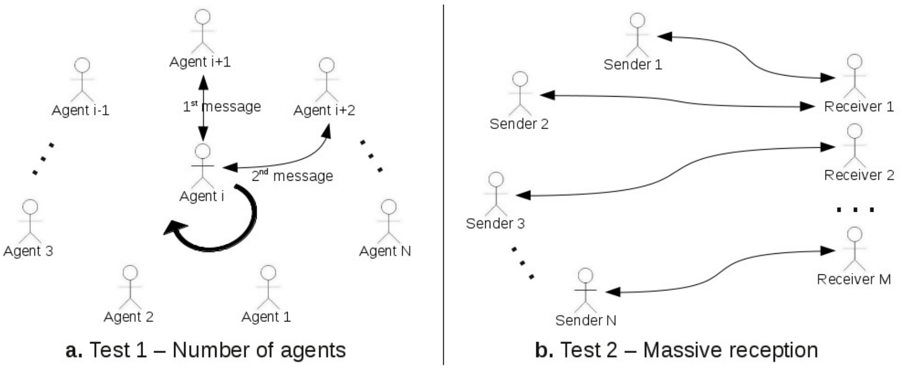


Figure 2: Schema of the benchmarks implemented in Jason.

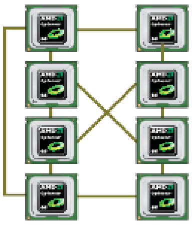


Figure 3: Twisted ladder topology of the processing cores in Sun Fire X4600 server.

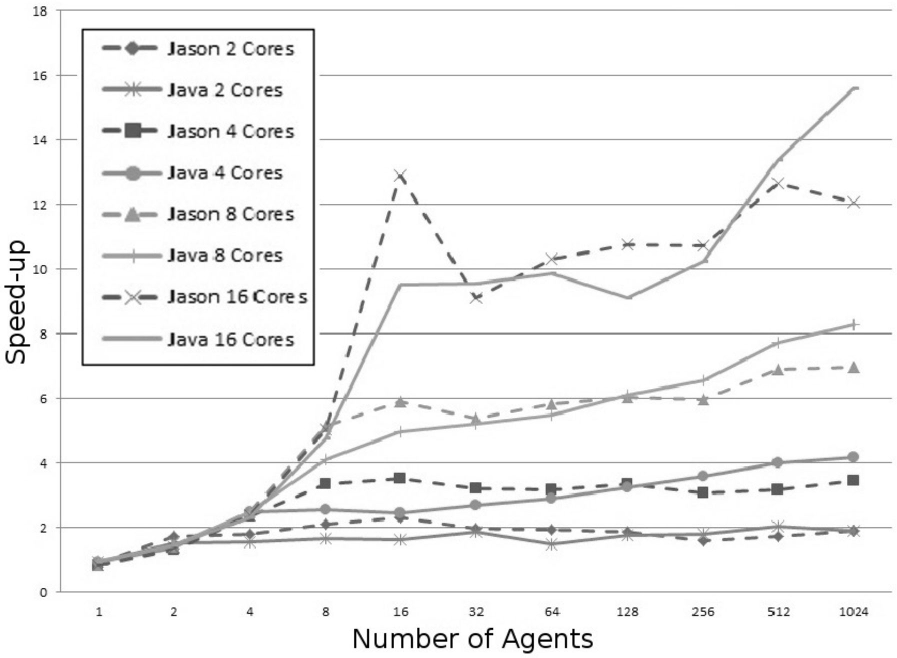


Figure 4: Speed-up achieved with the computation test executed on multicore processors.

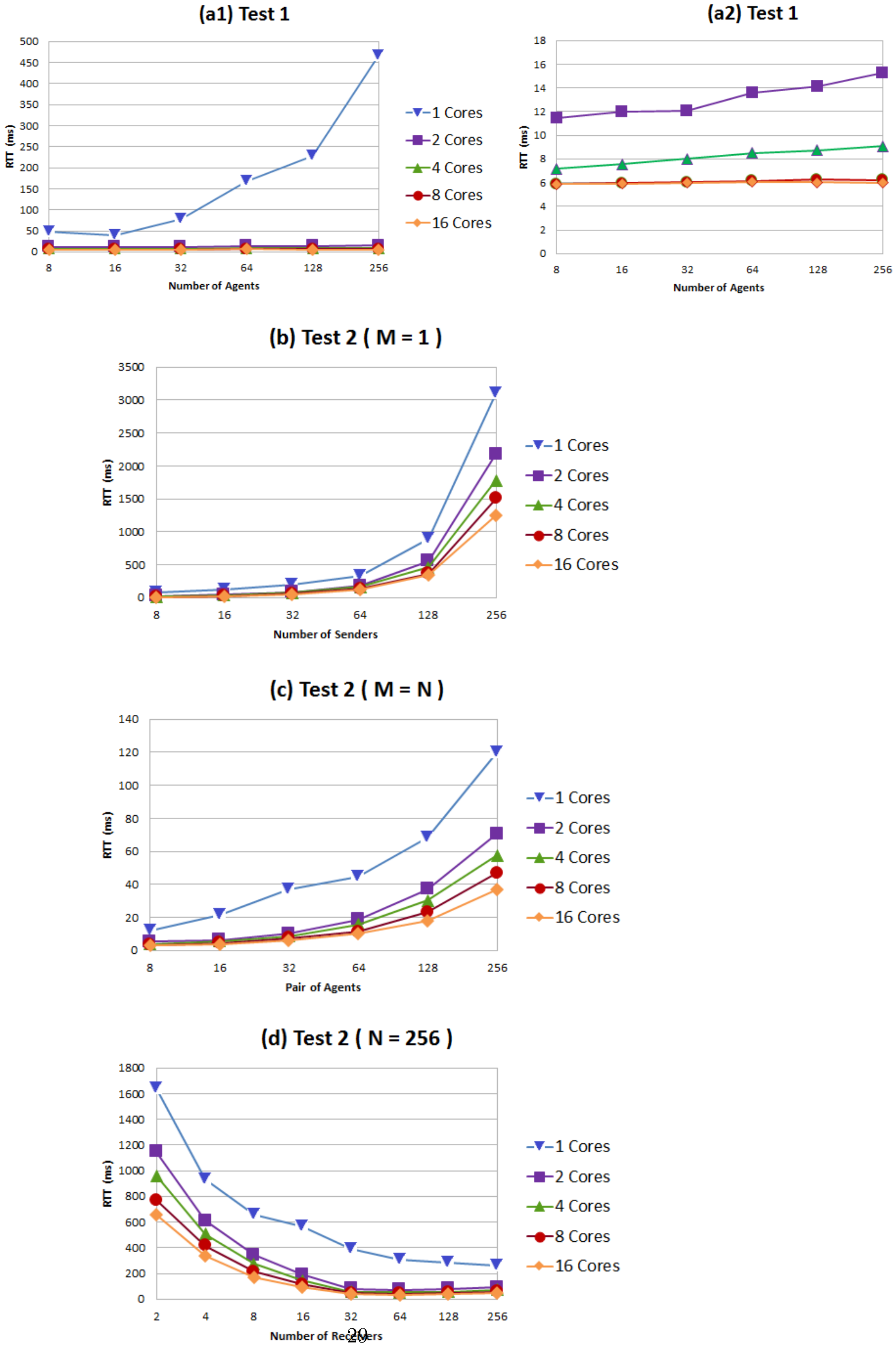


Figure 5: RTT obtained for the synchronous execution mode.

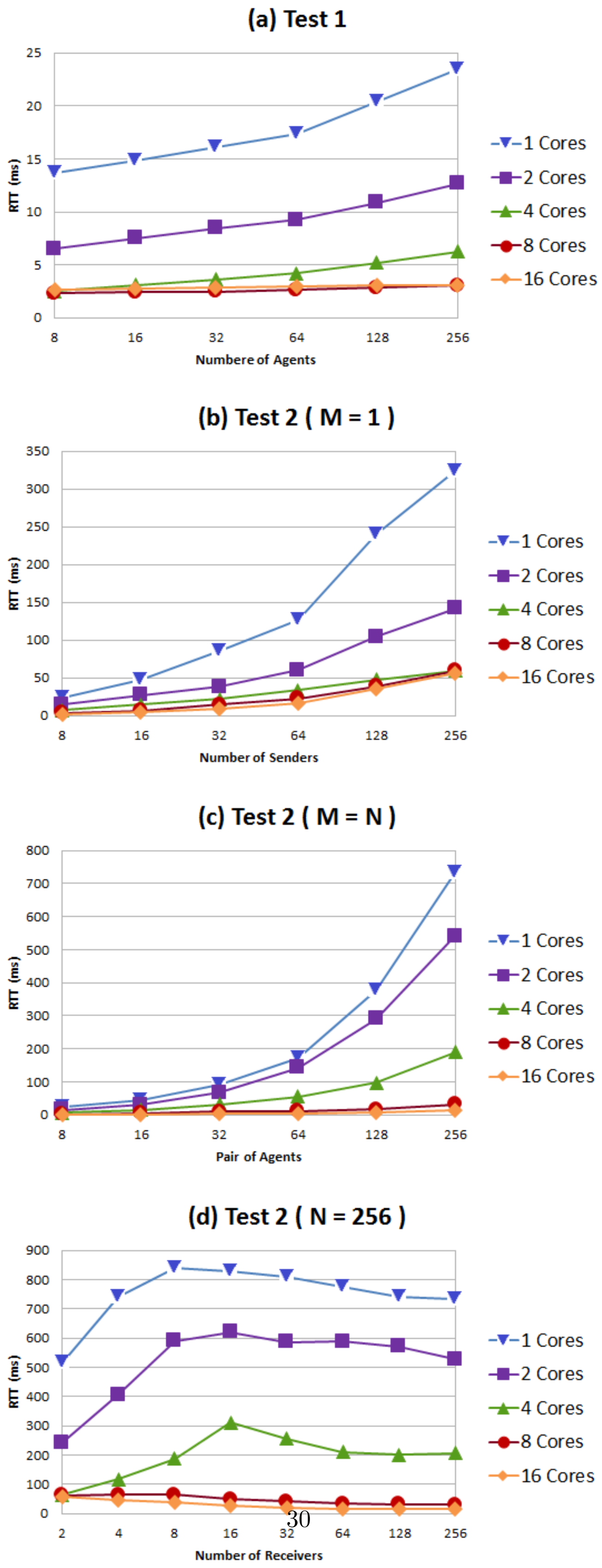


Figure 6: RTT obtained for the asynchronous execution mode with PThA disabled.

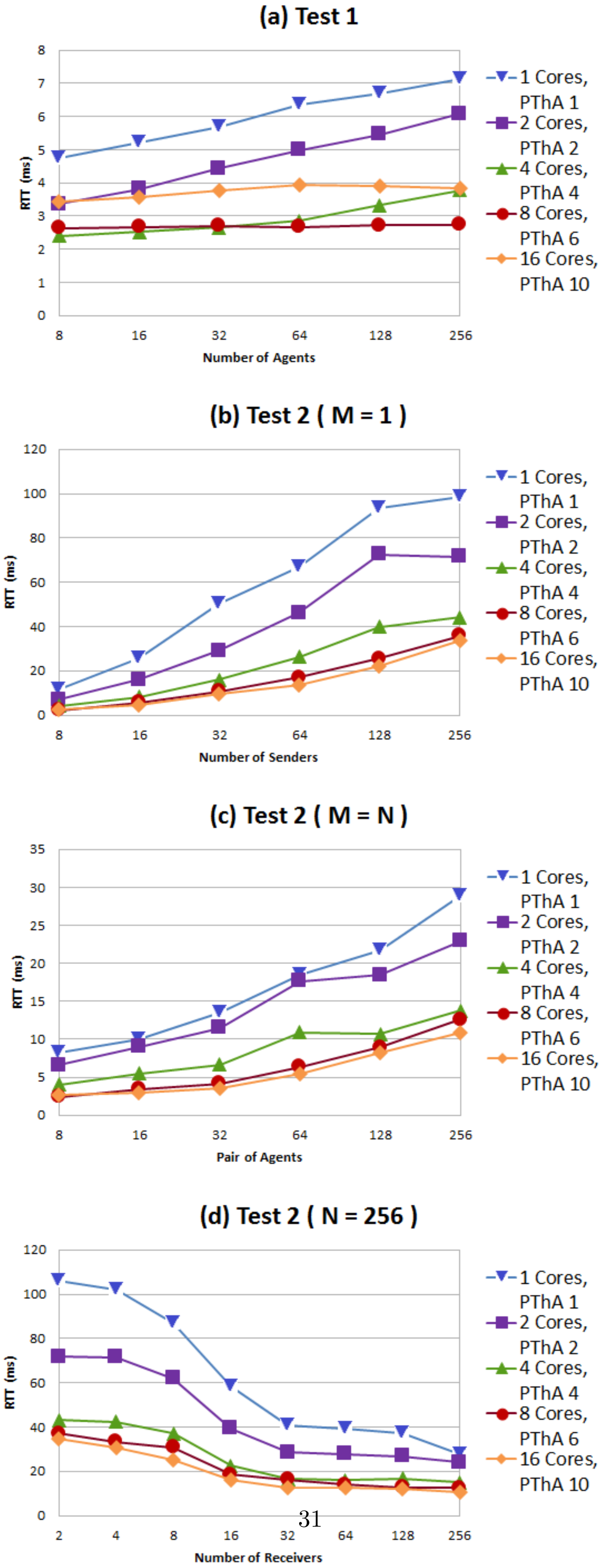
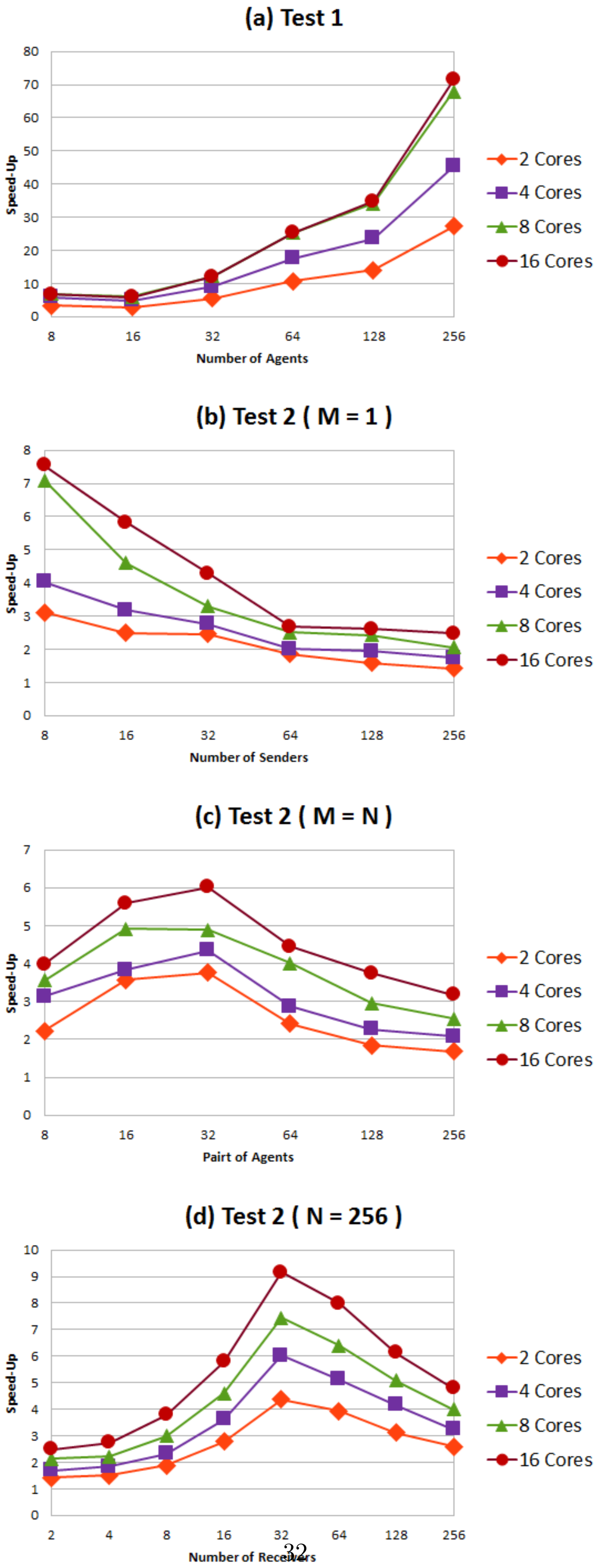


Figure 7: RTT obtained for the asynchronous execution mode with PThA enabled.


(a) Test 1

(b) Test 2 (M = 1)

(c) Test 2 (M = N)

(d) Test 2 (N = 256)

Figure 8: Speed-up obtained for the synchronous execution mode.

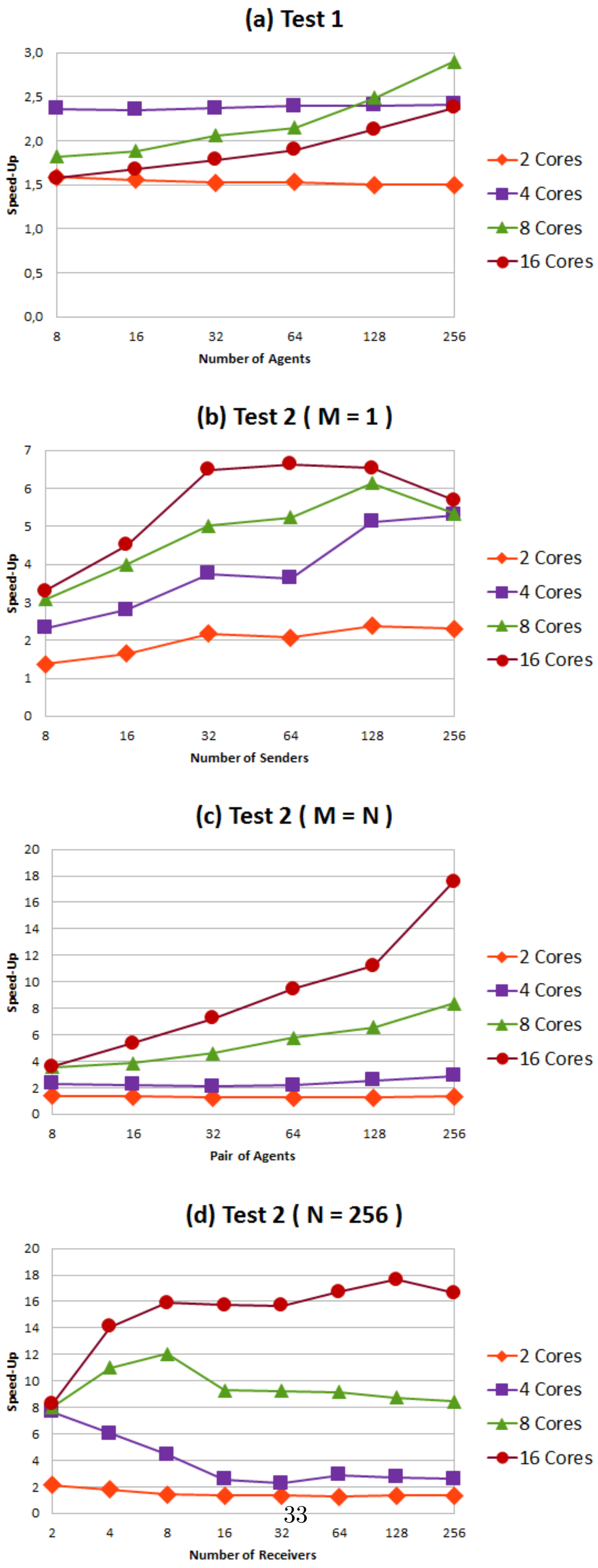


Figure 9: Speed-up obtained for the asynchronous execution mode with PThA disabled.

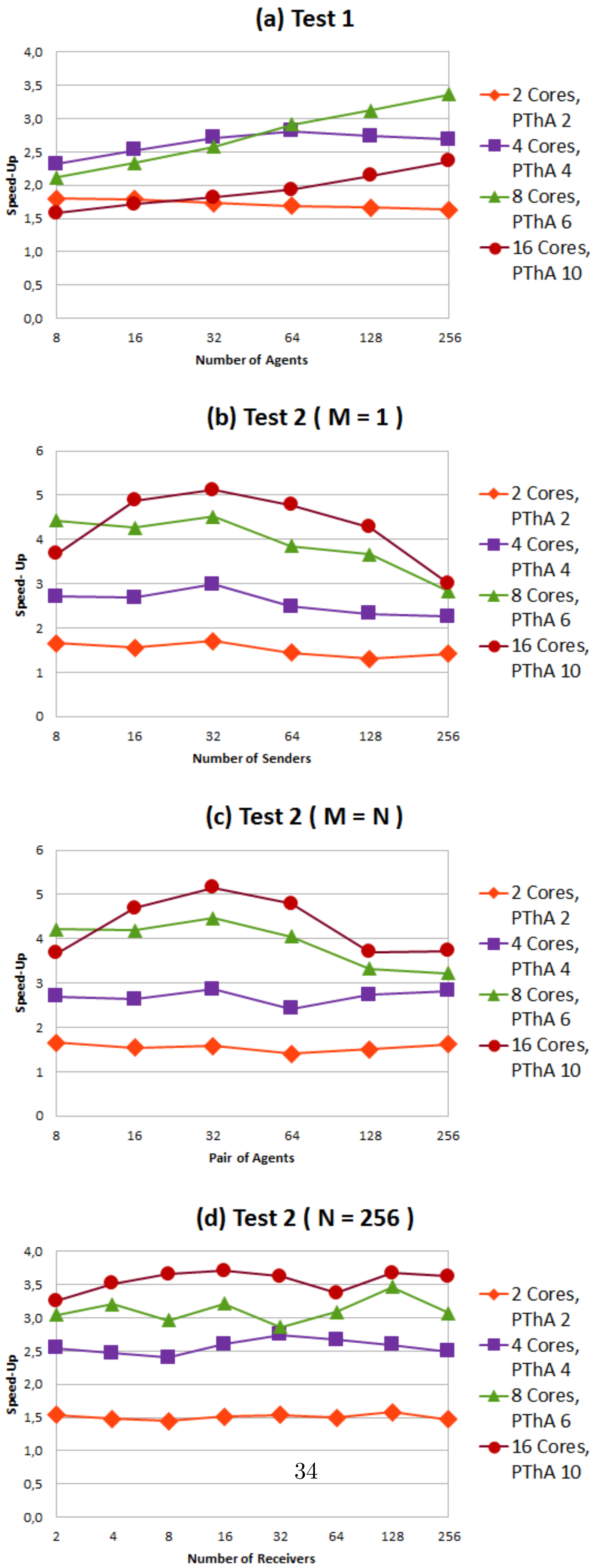


Figure 10: Speed-up of obtained for the asynchronous execution mode whith PThA enabled.