Received: 8 December 2016 | Revised: 6 October 2017 | Accepted: 6 October 2017
DOI: 10.1002/cpe.4381

WILEY

RESEARCH ARTICLE

Evaluating the efficiency of Linda implementations

Vitaly Buravlev | Rocco De Nicola | Claudio Antares Mezzina

IMT School for Advanced Studies Lucca, Piazza S. Francesco, 19, Lucca 55100, Italy

Correspondence
Claudio Antares Mezzina, IMT School for Advanced Studies Lucca, Piazza S. Francesco, 19, Lucca 55100, Italy.
Email: claudio.mezzina@imtlucca.it

Summary

Among the paradigms for parallel and distributed computing, the one popularized with Linda, and based on tuple spaces, is one of the least used, despite the fact of being intuitive, easy to understand, and easy to use. A tuple space is a repository, where processes can add, withdraw, or read tuples by means of atomic operations. Tuples may contain different values, and processes can inspect their content via pattern matching. The lack of a reference implementation for this paradigm has prevented its wide spreading. In this paper, first we perform an extensive analysis of a number of actual implementations of the tuple space paradigm and summarize their main features. Then, we select four such implementations and compare their performances on four different case studies that aim at stressing different aspects of computing, such as communication, data manipulation, and CPU usage. After reasoning on strengths and weaknesses of the four implementations, we conclude with some recommendations for future work towards building an effective implementation of the tuple space paradigm.

KEYWORDS

coordination model, implementation evaluation, tuple space

1 | INTRODUCTION

Distributed computing is getting increasingly pervasive, with demands from various application domains and with highly diverse underlying architectures that range from the multitude of tiny devices to the very large cloud-based systems. Several paradigms for programming parallel and distributed computing have been proposed so far. Among them we can list: distributed shared memory[1] (with shared objects and tuple spaces[2] built on it), remote procedure call (RPC[3]), remote method invocation (RMI[4]), and message passing[5] (with actors[6] and MPI[7] based on it). Nowadays, the most used paradigm seems to be message passing while the least popular one seems to be the one based on tuple spaces that was proposed by David Gelernter for the Linda coordination model.[8]

As the name suggests, message passing permits coordination by allowing exchanges of messages among distributed processes, with message delivery often mediated via brokers. In its simplest incarnation, message passing provides a rather low-level programming abstraction for building distributed systems. Linda instead provides a higher level of abstraction by defining operations for synchronization and exchange of values between different programs that can share information by accessing common repositories named *tuple spaces*. The Linda interaction model provides time and space decoupling,[9] since tuple producers and consumers do not need to know each other.

The key ingredient of Linda is a small number of basic operations which can be embedded into different programming languages to enrich them with communication and synchronization facilities. Three atomic operations are used for writing (`out`), withdrawing (`in`), and reading (`rd`) tuples into/from a tuple space. Another operation `eval` is used to spawn new processes. The operations for reading and withdrawing select tuples via *pattern-matching* and block if the wanted data are not available. Writing is instead performed by asynchronous output of the information for interacting partners. Figure 1 illustrates an example of a tuples space with different, structured, values. For example, tuple ⟨"goofy", 4, 10.4⟩ is produced by a process via the `out`(⟨"goofy", 4, 10.4⟩) operation and is read by the operation `rd`(⟨"goofy", _, _⟩) after pattern-matching: that is, the process reads any tuple of three elements whose first one is exactly the string "goofy." Moreover, tuple ⟨10, ⟨ … ⟩⟩ is consumed (atomically retracted) by operation `in`(⟨10, *x*⟩) which consumes a tuple whose first element is 10 and binds its second element (whatever it is) to the variable *x*. Patterns are sometimes called *templates*.

The simplicity of this coordination model makes it very intuitive and easy to use. Some synchronization primitives, e.g. semaphores or barrier synchronization, can be implemented easily in Linda (cf., Carriero and Gelernter[10, (chapter 3)]). Unfortunately, Linda's implementations of tuple spaces

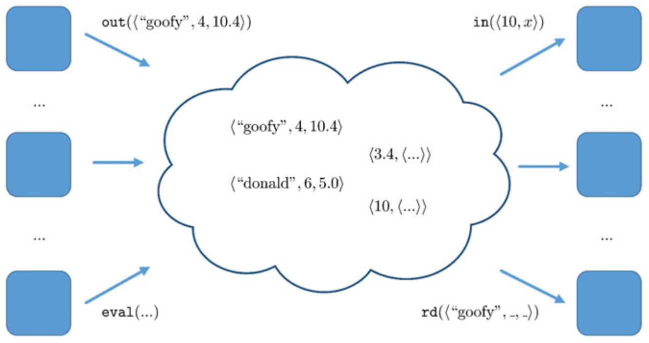

FIGURE 1 A tuple space

have turned out to be quite inefficient, and this has led researchers to opt for different approaches such OpenMP or MPI, which are nowadays offered, as libraries, for many programming languages. When considering distributed applications, the limited use of the Linda coordination model is also due to the need of guaranteeing consistency of different tuple spaces. In fact, in this case, control mechanisms that can significantly affect scalability are needed.[11]

Although time and space decoupling should suggest to have tuples surviving termination of their emitter, even in case of unexpected termination (e.g., due to node crash), the original Linda implementation did not consider this possibility, likewise most of the subsequent ones. Indeed, only few papers have considered the issue of fault tolerance in the context of tuple spaces, exceptions are Patterson et al,[12] Bessani et al,[13] and Barbi et al[14] relying on *data replication* and on *erasure-coding*, respectively. The absence of mechanism for handling faults rendered tuple spaces somehow unpopular in the parallel computing and the distributed systems communities. In this respect, we feel that standard techniques can be employed, and in this work, we will keep ignoring the issue and will focus on implementations closer to the original Linda spirit. In the next section, we will provide a brief description of DepSpace,[13] but we will then evaluate it without taking fault tolerance into account and conclude that it is not much different from one of the implementations on which we perform detailed experiments.

In our view, tuple spaces can be effectively exploited as a basis for the broad range of the distributed applications with different domains (from lightweight applications to large cloud-based systems). However, in order to be effective, we need to take into account that performances of a tuple space system may vary depending on the system architecture and on the type of interaction between its components. Although the concept of tuple spaces is rather simple, the main challenge to face when implementing it is to devise the best data structure to deal with (possibly distributed) *multisets* of tuples, with optimized operations for pattern-matching, insertion, and removal. Moreover, the data structure has to support efficient parallel processing and data distribution. Depending on how these aspects are implemented, performances of an application can be positively or negatively affected.

The aim of this paper is to examine a number of implementations of tuple spaces and to evaluate their strengths and weaknesses. We plan to use the outcome of our analysis for deciding how to build more efficient distributed tuple space.

In the rest of this work, we start by cataloging many existing implementations according to their main features, then we focus on the most recent Linda-based systems that are still maintained while paying specific attention to those offering decentralized tuples space. We compare the performances of the selected systems on four different case studies that aim at stressing different aspects of computing such as communication, data manipulation, and CPU usage. After reasoning on strength and weakness of the four implementations, we conclude with some recommendations for future work towards building an effective implementation of the tuple space paradigm.

The rest of the paper is organized as follows. In Section 2, we survey existing tuple spaces systems and choose some of them for the practical examination. The description of case studies, main principles of their implementation, and the results of the experiments are provided in Section 3. Section 4 concludes the paper by collecting some remarks and highlighting directions for future work. This paper is a revised and extended version of Buravlev et al[15]; it contains an additional case study, the thorough evaluation of a new tuple space system, and more extensive experiments on a much more powerful machine.

2 | TUPLE SPACE SYSTEMS

Since the first publication on Linda,[2] there have been a plenty of implementations of its coordination model in different languages. Our purpose is to review the most significant and recent ones, that are possibly still maintained, avoiding toy implementations or the one shot paper implementations. However, we do not consider tuple space–based middleware specifically devised for ad hoc wireless networks,[16,17] or for agent-based programming.[18]

Based on these considerations, we have chosen JAVASPACES[19] and TSPACES[20] which are two industrial proposals of tuple spaces for Java; GIGASPACES[21] which is a commercial implementation of tuple spaces; TUPLEWARE[22] featuring an adaptive search mechanism based on communication history; GRINDA,[23]BLOSSOM,[24]DTUPLES[25] featuring distributed tuple spaces; LUATS[26] which mixes reactive models with tuple spaces; KLAIM[27] and MOZARTSPACES[28] which are two academic implementations with a good record of research papers based on them.

TABLE 1 Results of the comparison

	JSP	TSP	GSP	TW	GR	BL	DTP	LTS	KL	MS	DS
`eval` operation								✓	✓		✓
Tuple clustering			?	✓				✓			
No domain specificity	✓	✓	✓		✓	✓	✓		✓	✓	✓
Security		✓	✓						✓		✓
Distributed tuple space			?	✓	✓	✓	✓		✓	✓	✓
Decentralized management				✓	✓		✓	✓	✓	✓	
Scalability			✓	✓	✓		✓				

Abbreviations: **BL**, BLOSSOM; **DS**, DEPSPACE; **DTP**, DTUPLES; **GR**, GRINDA; **GSP**, GIGASPACES; **JSP**, JAVASPACES; **KL**, KLAIM; **LTS**, LUATS; **MS**, MOZARTSPACES; **TSP**, TSPACES; **TW**, TUPLEWARE.

In this section, first we review the abovementioned tuple space systems by briefly describing each of them, and single out the main features of their implementations, then we summarize these features in Table 1. Later, we focus on the implementations that enjoy the characteristics we consider important for a tuple space implementation: code mobility, distribution of tuples, and flexible tuples manipulation. All tuple space systems are enumerated in order they were first mentioned in publications.

Blossom. BLOSSOM[24] is a C++ implementation of Linda which was developed to achieve high performance and correctness of the programs using the Linda model. In BLOSSOM, all tuple spaces are homogeneous with a predefined structure that demands less time for type comparison during the tuple lookup. BLOSSOM was designed as a distributed tuple space and can be considered as a distributed hash table. To improve scalability, each tuple can be assigned to a particular place (a machine or a processor) on the basis of its values. The selection of the correspondence of the tuple and the machine is based on the following condition: for every tuple, the field access pattern is defined, that determines which fields always contain value (also for templates); values of these fields can be hashed to obtain a number which determines the place where a tuple has to be stored. Conversely, using the data from the template, it is possible to find the exact place where a required tuple is potentially stored. Prefetching allows a process to send an asynchronous (i.e., non-blocking) request for a tuple and to continue its work while the search is performed. When the requested tuple is needed, if found, it is received without waiting.

TSpaces. TSPACES[20] is an implementation of the Linda model developed at the IBM Almaden Research Center. It combines asynchronous messaging with database features. TSPACES provides a transactional support and a mechanism of tuple aging. Moreover, the embedded mechanism for access control to tuple spaces is based on access permission. It checks whether a client is able to perform specific operations in the specific tuples space. Pattern matching is performed using either standard `equals` method or `compareTo` method. It can use also SQL-like query that allows matching tuples regardless of their structure, e.g., ignoring the order in which fields are stored.

Klaim. KLAIM[27] (A Kernel Language for Agents Interaction and Mobility) is an extension of Linda supporting distribution and process mobility. Processes, like any other data, can be moved from one locality to another and can be executed at any locality. Klava[29] is a Java implementation of KLAIM that supports multiple tuple spaces and permits operating with explicit localities where processes and tuples are allocated. In this way, several tuples can be grouped and stored in one locality. Moreover, all the operations on tuple spaces are parameterized with a locality. The emphasis is put also on access control which is important for mobile applications. For this reason, KLAIM introduces a type system which allows checking whether a process is allowed to perform specific operations at specific localities.

JavaSpaces. JAVASPACES[19] is one of the first implementations of tuple spaces developed by Sun Microsystems. It is based on a number of Java technologies (e.g., Jini and RMI) and has been recently integrated into the Apache River project. Like TSPACES, JAVASPACES supports transactions and a mechanism of tuple aging. A tuple, called entry in JAVASPACES, is an instance of a Java class and its fields are the public properties of the class. This means that tuples are restricted to contain only objects and not primitive values. The tuple space is implemented by using a simple Java collection. Pattern matching is performed on the byte-level, and the byte-level comparison of data supports object-oriented polymorphism.

GigaSpaces. GIGASPACES[21] is a contemporary commercial implementation of tuple spaces. Nowadays, the core of this system is GIGASPACES XAP, a scale-out application server; user applications should interact with the server to create and use their own tuple space. The main areas where GIGASPACES is applied are those concerned with big data analytics. Its main features are linear scalability, optimization of RAM usage, synchronization with databases, and several database-like features such as complex queries, transactions, and replication.

LuaTS. LUATS[26] is a reactive event-driven tuple space system written in Lua. Its main features are the associative mechanism of tuple retrieving, fully asynchronous operations, and the support of code mobility. LUATS provides centralized management of the tuple space which can be logically partitioned into several parts using indexing. LUATS combines the Linda model with the event-driven programming paradigm. This paradigm was chosen to simplify program development since it allows avoiding the use of synchronization mechanisms for tuple retrieval and makes more transparent programming and debugging of multi-thread programs. Tuples can contain any data which can be serialized in Lua. To obtain a more flexible and intelligent search of tuples, processes can send to the server code that once executed returns the matched tuples. The reactive tuple space is implemented as a hash table, in which data are stored along with the information supporting the reactive nature of that tuple space (templates, client addresses, callbacks, and so on).

MozartSpaces. MOZARTSPACES[28] is a Java implementation of the space-based approach.[30] The implementation was initially based on the eXtensible Virtual Shared Memory (XVSM) technology, developed at the Space Based Computing Group, Institute of Computer Languages, Vienna University of Technology.

The basic idea of XVSM is related to the concept of *coordinator*: an object defining how tuples (called entries) are stored. For the retrieval, each coordinator is associated with a *selector*, an object that defines how entries can be fetched. There are several predefined coordinators such as FIFO, LIFO, Label (each tuple is identified by a label, which can be used to retrieve it), Linda (corresponding to the classic tuple matching mechanism), Query (search can be performed via a query-like language), and many others. Along with them, a programmer can define a new coordinator or use a combination of different coordinators (e.g., FIFO and Label). MOZARTSPACES provides also transactional support and a role-based access control model.[31]

DepSpace. DEPSPACE[32] is a Byzantine fault-tolerant coordination service[*] that provides a tuple space abstraction whose main emphasis is fault tolerance and security. DepSpaces uses different replication approaches such as Byzantine fault-tolerant state machine replication[32] and Byzantine quorum systems.[13] Security is achieved by access control mechanism and cryptography. Servers create a dependable tuple space enforcing reliability, availability, integrity and confidentiality of data, and tolerating Byzantine faults.[32] The DepSpaces project is still ongoing, and in a recent work,[33] it has been enhanced with extensible coordination services, e.g., the possibility of adding custom operations to be executed on the server side. These services are somehow similar to the original eval primitive and aim at providing mechanisms to have tuples close to data processing units to guarantee fast data access. Extensions operations can be executed immediately or, reactively, when the state of a coordination service does change. In both cases, extensions can be modelled using eval. However, because of security and fault tolerance requirements, an extension has to satisfy a number of constraints in order to be executed, whereas the eval operation has no specific limitations.

DTuples. DTUPLES[25] is designed for peer-to-peer networks and based on distributed hash tables (DHT), a scalable and efficient approach. Key features of DHT are autonomy and decentralization. There is no central server and each node of the DHT is in charge of storing a part of the hash table and of keeping routing information about other nodes. As the basis of the DTH's implementation, DTUPLES uses FreePastry[†]. DTUPLES supports transactions and guarantees fault tolerance via replication mechanisms. Moreover, it supports multi tuple spaces and allows for two kinds of tuple space: *public* and *subject*. A public tuple space is shared among all the processes, and all of them can perform any operation on it. A subject tuple space is a private space accessible only by the processes that are bound to it. Any subject space can be bound to several processes and can be removed if no process is bound to it. Due to the two types of tuple spaces, pattern matching is specific for each of them. Templates in the subject tuple space can match tuples in the same subject tuple space and in the common tuple space. However, the templates in the common tuple space cannot match the tuple in the subject tuple spaces.

Grinda. GRINDA[23] is a distributed tuple space which was designed for large-scale infrastructures. It combines the Linda coordination model with grid architectures aiming at improving the performance of distributed tuple spaces, especially with a lot of tuples. To boost the search of tuples, GRINDA utilizes spatial indexing schemes (X-Tree, Pyramid) which are usually used in spatial databases and Geographical Information Systems. Distribution of tuple spaces is based on the grid architecture and implemented using structured P2P network (based on Content Addressable Network and tree based).

Tupleware. TUPLEWARE[22] is specially designed for array-based applications in which an array is decomposed into several parts each of which can be processed in parallel. It aims at developing a scalable distributed tuple space with good performances on a computing cluster and provides simple programming facilities to deal with both distributed and centralized tuple space. The tuple space is implemented as a hashtable, containing pairs consisting of a key and a vector of tuples. Since synchronization lock on Java hashtable is done at the level of the hash element, it is possible to access concurrently to several elements of the table. To speed up the search in the distributed tuple space, the system uses an algorithm based on the history of communication. Its main aim is to minimize the number of communications for tuples retrieval. The algorithm uses *success factor*, a real number between 0 and 1, expressing the likelihood of the fact that a node can find a tuple in the tuple space of other nodes. Each instance of TUPLEWARE calculates success factor on the basis of previous attempts and first searches tuples in nodes with greater success factor.

In order to compare the implementations of the different variants of Linda that we have considered so far, we have singled out two groups of criteria.

The first group refers to criteria which we consider fundamental for any tuple space system:

`eval` **operation** This criterion denotes whether the tuple space system has implemented the `eval` operation and, therefore, allows using code mobility. It is worth mentioning that the original `eval` operation was about asynchronous evaluation and not code mobility, but in the scope of a distributed tuple space, it makes programming data manipulation more flexible.

Tuples clustering This criterion determines whether some tuples are grouped by particular parameters that can be used to determine where to store them in the network.

Absence of domain specificity Many implementations have been developed having a particular application domain in mind. On the one hand, this implies that domain-specific implementations outperform the general purpose one, but on the other hand, this can be considered as a limitation if one aims at generality.

Security This criterion specifies whether an implementation has security features or not. For instance, a tuple space can require an authorization and regulate the access to its tuples, for some of them, the access can be limited to performing specific operations (e.g., only writes or read).

[*] Byzantine fault tolerance implies that the architecture with at least 3n+1 servers can tolerate failures of n servers
[†] FreePastry is an open-source implementation of Pastry, a substrate for peer-to-peer applications (http://www.freepastry.org/FreePastry/).

The second group of criteria, gathers features which are desirable for any fully distributed implementation that runs over a computer network, does not rely on a single node of control or management and is scalable. We did not consider fault tolerance as an evaluation criteria because we think this a somewhat orthogonal issue. Indeed, techniques similar to those used by DepSpace to guarantee fault tolerance can be used also in other implementations.

Distributed tuple space This criterion denotes whether tuple spaces are stored in one single node of the distributed network or they are spread across the network.

Decentralized management Distributed systems rely on a node that controls the others or the control is shared among several nodes. Usually, systems with the centralized control have bottlenecks which limit their performance.

Scalability This criterion implies that system based on particular Linda implementation can cope with the increasing amount of data and nodes while maintaining acceptable performance.

Table 1 summarises the result of our comparison: ✓ means that the implementation enjoys the property and ? means that we were not able to provide an answer due to the lack of source code and/or documentation.

We tested DEPSPACE without replicas and experienced performances similar to those of TUPLEWARE: good reading and slow writing times. With replication enabled, we experienced that reading and writing operations become significantly slower.

After considering the results in Table 1, to perform our detailed experiments, we have chosen

- TUPLEWARE because it enjoys most of the wished features;
- KLAIM because it offers distribution and code mobility;
- MOZARTSPACES because it satisfies two important criteria of the second group (fully distribution) and is one of the most recent implementation.
- GIGASPACES because it is the most modern among the commercial systems; we will use it as a yardstick to compare the performance of the others.

We would like to add that DTUPLES has not been considered for the more detailed comparison because we have not been able to obtain its libraries or source code and that GRINDA has been dropped because it seems to be the less maintained one. It is worth noticing that while GIGASPACES and MOZARTSPACES are still maintained, the other implementations are not, and their code is based in the state-of-the-art technology of their last release, which nowadays could result *deprecated*.

In all our implementations of the case studies, we have structured the systems by assigning each process a local tuple space. Since GIGASPACES is a centralized tuple space (e.g., tuples are stored on a server), we have to simulate local spaces on the server: to this end to each process is assigned its own tuple space in the GIGASPACES server.

3 | EXPERIMENTS

In order to compare four different tuple space systems, we consider four different case studies: *Password search*, *Sorting*, *Ocean model*, and *Matrix multiplication*. We describe them below.

3.1 | Introducing case studies

The first case study is of interest since it deals with a large number of tuples and requires performing a huge number of write and read operations. This helps us understand how efficiently an implementation performs operations on local tuple spaces with a large number of tuples. The second case study is computation intensive since each node spends more time for sorting elements than on communicating with the others. This case study has been considered because it needs structured tuples that contain both basic values (with primitive type) and complex data structures that impacts on the speed of the inter-process communication. The third case has been taken into account since it introduces particular dependencies among nodes, which if exploited can improve the application performances. This was considered to check whether adapting a tuple space system to the specific inter-process interaction pattern of a specific class of the applications could lead to significative performance improvements. The last case study is a communication-intensive task, and it requires much reading on local and remote tuple spaces. All case studies are implemented using the master-worker paradigm[10] because among other design patterns (e.g., Pipeline, SPMD, Fork-join),[34] it fits well with all our case studies and allows us to implement them in a similar way. We briefly describe all the case studies in the rest of this subsection.

We would like to stress that, with our case studies, we aim at evaluating advantages and disadvantages of the chosen tuple space implementations. In no way, we aim at providing efficient solutions for our case studies. Indeed, some of them have intentional redundancy that slows down their performance but allows us to highlight the difference among the different implementations. For instance, the code for the *Ocean model* and the one for the *Matrix multiplication problem* can be easily improved by assigning data or jobs to specific workers and making data search unnecessary.

To highlight the key steps of the proposed algorithms, we present the basic Linda code of the master and of the workers for each case study. The full Java code used in the actual experiments has many additional lines that render it less intuitive but allow us to run it on the different tuple space implementations. It can be accessed at Github: www.github.com/IMTAltiStudiLucca/Klava2.

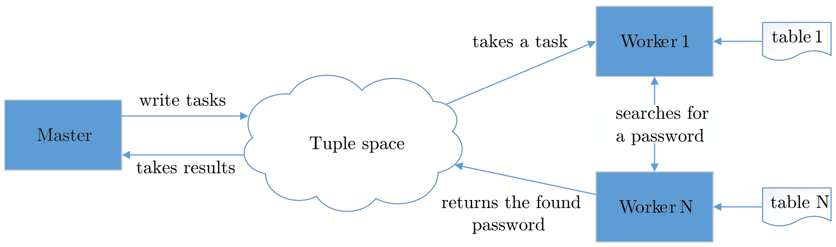

FIGURE 2 Schema of the case study *Password search*

Remark 1. (Local and remote tuple spaces)
It is important to stress the difference between the meaning we assign to the terms *local* and *remote* tuple space. Regardless of the fact that processes run on a single or on different machines, we consider *remote* a tuple space that is instantiated/managed by processes different from the one that performs operations (read, write, withdraw) on it. We instead consider *local* a tuple space that is accessed by the same process that instantiated it.

Password search. The main aim of this application is to find a password using its hashed value in a predefined "database" distributed among processes. Such a database is a set of files containing pairs (password, hashed value). The application creates a master process and several worker processes (Figure 2): the master keeps asking the workers for passwords corresponding to a specific hashed values, by issuing tuples of the form:

$$\langle\text{"search\_task"}\ dd157c03313e452ae4a7a5b72407b3a9\rangle$$

Each worker first loads its portion of the distributed database and then obtains from the master a task to look for the password corresponding to a hash value. Once it has found the password, it sends the result back to the master, with a tuple of the form:

$$\langle\text{"found\_password"}\ dd157c03313e452ae4a7a5b72407b3a9, 7723567\rangle$$

For multiple tuple space implementations, it is necessary to start searching in one local tuple space and then to check the tuple spaces of other workers. The application terminates its execution when all the tasks have been processed and the master has received all required results.

Listing 1 presents the code for *Password Search*. The master writes, in its tuple space, *n* tasks with known hashed values of passwords to be found (lines 2-3), waits for the found passwords (lines 5-6), and informs workers to terminate their work because all tasks have been accomplished (lines 9-10). A worker loads predefined data (lines 15 - 16), takes a task to perform (lines 18, 26), looks for a password in tuple spaces of each worker (line 21-24), and returns the found password to the master (line 25).

Sorting. This program sorts an array of integers. The master performs the initial data loading and collects the sorted data; workers perform the actual sorting. The master, after loading the data, splits them into many parts, stores them in its own tuple space, and then waits for the sorted arrays from the workers. Once the master has collected all sorted sub-arrays, it builds the whole sorted sequence. A worker looks for unsorted data in the tuple space of the master and when it finds a tuple with unsorted data, it sorts it, sends the result to the master, and continues looking for data to sort. An example of sorting is shown in Figure 3.

Listing 1: Password search. Listing of master and worker processes

```
 1  Master():
 2    for i in range(n):
 3      masterTS.out("search_task", hash[i], "not_processed")
 4
 5    for i in range(n):
 6      masterTS.in("found_value", ?hashedValue, ?password)
 7
 8    // poison tuples for all workers
 9    for i in range(workerNumber):
10      masterTS.out("search_task", null, "finished")
11
12
13  Worker():
14    // loads k password
15    for i in range(m):
16      localTS.out("hash_set", db[i].hashedValue, db[i].password)
17
18    masterTS.in("search_task", ?hashedValue, ?status)
19    while status != "finished":
20      // search in other tuple spaces
21      for i in range(workerNumber):
22        res = workerTS[i].rdp("hash_set", hashedValue, ?password)
23        if res == true:
24          break
25      masterTS.out("found_value", hashedValue, password)
26      masterTS.in("search_task", ?hashedValue, ?status)
```

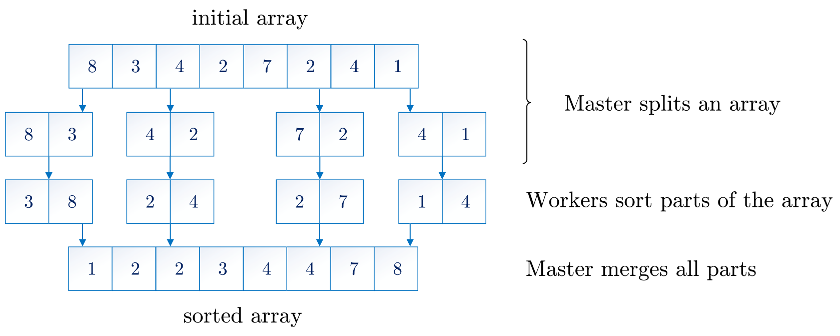

FIGURE 3 Schema of the case study *Sorting*

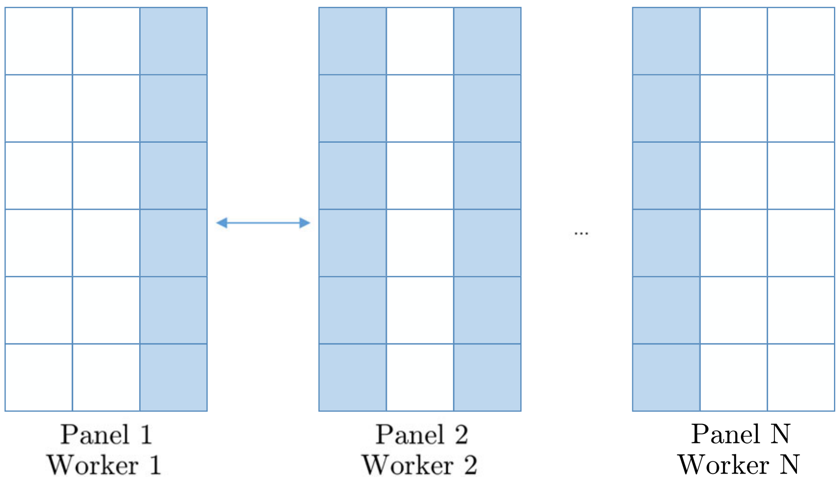

FIGURE 4 Schema of the case study *Ocean model*

Listing 2: Sorting. Listing of master and worker processes

```
1   Master():
2     unsortedParts = splitInitialArray(initialArray, nParts)
3     for i in range(unsortedParts.size):
4       masterTS.out("sort_array", unsortedParts[i], "unsorted")
5
6     while n != initialArray.size:
7       masterTS.in("sorted_part", sortedPart)
8       sortedParts.add(sortedPart)
9       n += sortedPart.size
10    // reconstruct a sorted array
11    sortedArray = reconstructArray(sortedParts)
12
13    // poison tuples for all workers
14    for i in range(workerNumber):
15      masterTS.out("sort_array", null, "finished")
16
17
18  Worker():
19    masterTS.in("sort_array", ?arrayPart, ?status)
20    while status != "finished":
21      sortedArrayPart = sort(arrayPart)
22      masterTS.out("sorted_parts", sortedArrayPart)
23      masterTS.in("sort_array", ?arrayPart, ?status)
```

Listing 2 presents the code for *Sorting*. The master splits an initial array into *nParts* parts (in our tests *nParts* = 200), adds them to its tuple space (lines 3-4), and begins to collect sorted parts (lines 6-9). When all parts are collected (line 6), the master reconstructs the full sorted array (line 11) and notifies all workers to terminate their work (lines 14-15). A worker takes an unsorted array from the master (line 19), sorts it, and sends the sorted array to the master (lines 21-22).

Ocean model. The ocean model is a simulation of the enclosed body of water that was considered in.[22] The two-dimensional (2-D) surface of the water in the model is represented as a 2-D grid and each cell of the grid represents one point of the water. The parameters of the model are current velocity and surface elevation which are based on a given wind velocity and bathymetry. In order to parallelize the computation, the whole grid is divided into vertical panels (Figure 4), and each worker owns one panel and computes its parameters. The parts of the panels, which are located on the border between them are colored. Since the surface of the water is continuous, the state of each point depends on the states of the points close to it. Thus, the information about bordering parts of panels should be taken into account. The aim of the case study is to simulate the body of water during several time-steps. At each time-step, a worker recomputes the state (parameters) of its panel by exploiting parameters of the adjacent panels.

The missions of the master and workers are similar to the previous case studies. In the application the master instantiates the whole grid, divides it into parts and sends them to the workers. When all the iterations are completed, it collects all parts of the grid. Each worker receives its share of the grid and at each iteration it communicates with workers which have adjacent grid parts in order to update and recompute the parameters of its model. When all the iterations are completed, each worker sends its data to the master.

Listing 3 presents the code for *Ocean model*. The master generates a model, splits it into a number of panels and distributes them among workers (lines 2-4). Then it collects the processed panels (lines 6-8) and reconstructs the model (line 9). A worker gets a panel from the master (line 13) and passes it through a number of iterations (lines 16-26); at each iteration it shares a part of its data with other processes (line 18-19) and gets data of adjacent panels (lines 21-24). In the end, the worker returns a panel to the master (line 28).

Matrix multiplication. The case study is designed to multiply two square matrices of the same order. The algorithm of multiplication[35] operates with rows of two matrices A and B and put the result in matrix C. The latter is obtained via subtasks where each row is computed in parallel. At the *j*-th step of a task the *i*-th task, the element, a_{ij}, of A is multiplied by all the elements of the *j*-th row of B; the obtained vector is added to the current *i*-th row of C. The computation stops when all subtasks terminate. Figure 5 shows how the first row of C is computed if A and B are 2×2 matrices. In the first step, the element $a_{1,1}$ is multiplied first by $b_{1,1}$ then by $b_{1,2}$, to obtain the first partial value of the first row. In the second step, the same operation is performed with $a_{1,2}$, $b_{2,1}$, and $b_{2,2}$, and the obtained vector is added to the first row of C thus obtaining its final value.

Listing 3: Ocean model. Listing of master and worker processes

```
1  Master():
2    panels = splitModel(oceanModel)
3    for i in range(workerNumber):
4      masterTS.out("oceanModel", "panel", panels[i])
5    // take back
6    for i in range(workerNumber):
7      masterTS.in("oceanModel", "ready_panel", ?panel)
8      processedPanels.add(panel)
9    oceanModel = reconstructModel(processedPanels)
10
11
12 Worker():
13   masterTS.in("oceanModel", "panel", ?panel)
14
15   // do several iterations
16   while panel.iterationNumber < iterationMax:
17     // for each adjacent panel write required boundary data
18     for borderInfo in panel.borders:
19       localTS.out("oceanModel", "border", borderInfo.name, borderInfo.data)
20
21     for borderInfo in panel.borders:
22       for i in range(workerNumber):
23         workerTS[i].inp("oceanModel", "border", borderInfo.name, ?borderData)
24         panel.borderData.add(borderData)
25     // update its state
26     panel.process()
27
28   masterTS.out("oceanModel", "ready_panel", panel)
```

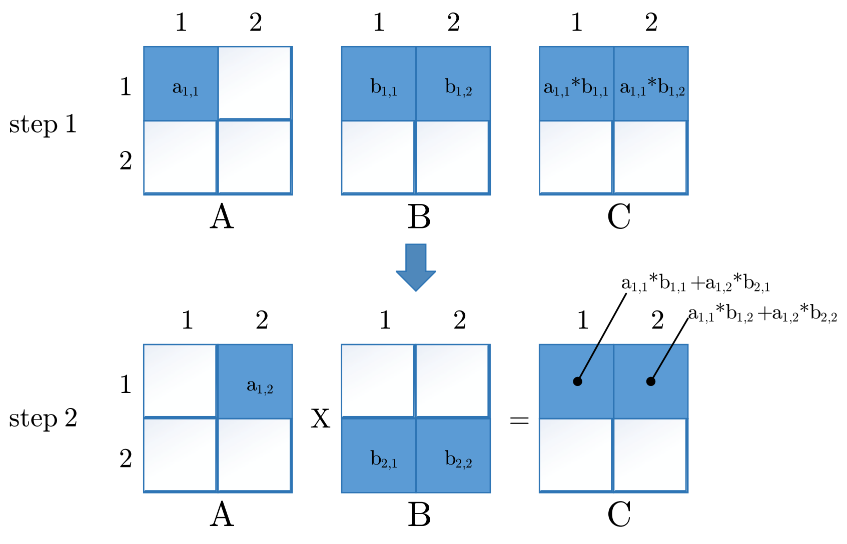

FIGURE 5 Schema of the case study *Matrix multiplication*

Listing 4: Matrix multiplication. Listing of master and worker processes

```
1  Master():
2    for i in range(matrixSize):
3      masterTS.out("matrixA", i, matrixA[i], matrixSize)
4      masterTS.out("matrixB", i, matrixB[i], matrixSize)
5
6    for i in range(matrixSize):
7      masterTS.take("matrixC", i, ?matrixCRow, matrixSize)
8      matrixC.add(i, matrixCRow)
9
10
11 Worker():
12   // rows per each worker
13   rowsPerWorker = matrixSize/workerNumber
14   if workerID < matrixSize % workerNumber:
15     rowsPerWorker++;
16
17   for i in range(rowsPerWorker):
18     masterTS.in("matrixB", ?rowID, ?rowData, ?counter)
19     localTS.out("matrixB", rowID, rowData, counter)
20
21   for i in range(rowsPerWorker):
22     masterTS.in("matrixA", ?rowID, ?rowData, ?counter)
23     tasks.add(rowID, rowData)
24
25   // look at each row lk tasks
26   for i in range(tasks.size()):
27     rowID = tasks[i].id
28     rowFirstOperand = tasks[i].rowData
29
30     // check all rows of matrix B
31     for k in range(matrixSize):
32       // take k-row from matrix B
33       for w in range(workerNumber):
34         if workers[w].rdp("matrixB", i, ?rowB, ?counterB) == true:
35           break
36       rowMatrixC = rowB * rowFirstOperand[k]
37
38       // check matrix C (existence of partial result)
39       if localTS.inp("matrixC", rowID, ?valC, ?counter) == false:
40         localTS.out("matrixC", rowID, rowMatrixC, 1)
41       else
42         rowMatrixC = rowMatrixC + valC
43         counter++
44
45         if counter == matrixSize:
46           masterTS.out("matrixC", rowID, rowMatrixC, counter)
47         else
48           localTS.out("matrixC", rowID, rowMatrixC, counter)
```

Initially, the master distributes the matrices A and B among the workers. In our case study, we have considered two alternatives: (i) the rows of both A and B are spread uniformly and (ii) the rows of A are spread uniformly while B is entirely assigned to a single worker. This helped us in understanding how the behavior of the tuple space and its performances change when only the location of some tuples changes.

Listing 4 presents the code for *Matrix multiplication*. The master, first, makes the matrices (*A* and *B*) to be multiplied available to workers, by putting them into its tuple space (lines 2-4), then it collects the results sent back by the workers (lines 6-8). Each worker, first, loads parts of *A* and *B* (lines 13-23), then it computes rows of the matrix products that correspond to the rows of *A* (lines 26-48). Using the received rows of *A*, a worker looks for the necessary rows of *B* (lines 33-35) and stores intermediate results locally (lines 39-48); finally, it sends to the master the computed rows of matrix product (line 46).

3.2 | Implementing the case studies

All the chosen implementations are Java-based, the most used language (according to TIOBE Programming Community Index[36] and Cass[37]). It guarantees also the possibility of performing better comparisons of the time performances exhibited by the different tuple systems which could significantly depend on the chosen language.

Another key-point of using the same programming language for all the implementations is that case studies can be written as *skeletons*: the code remains the same for all the implementations while only the invocations of different library methods do change. In order to implement the skeleton for the case study, we have developed several wrappers for each of four tuple space systems we have chosen and each wrapper implements interface `ITupleSpace`. This interface defines the basic operations of a tuple space (e.g., initialization/destruction, I/O operations and so on). Since the tuple space systems have a different set of operations on tuple space we have chosen those operations which have the same semantics for all systems and can be unified. It is worth to notice that all I/O operations on a tuple space are wrapped and placed in the class *TupleOperation*.

There is a difference on how the tuple spaces systems implement the search among distributed tuple spaces. TUPLEWARE has a built-in operation with notification mechanism: it searches in local and remote tuple spaces at once (i.e. in broadcast fashion) and then waits for the notification that the desired tuple has been found. Mimicking the behavior of this operation for the other tuple space systems requires to continuously check each tuple space until the required tuple is found.

3.3 | Assessment methodology

All the conducted experiments are parametric with respect to two values. The first one is the number of workers $w, w \in \{1, 5, 10, 15\}$. This parameter is used to test the scalability of the different implementations. The second parameter is application specific, but it aims at testing the implementations when the workload increases.

- *Password search*, we vary the number of the entries in the database (10 000, 100 000, 1 million passwords) where it is necessary to search a password. This parameter directly affects the number of local entries each worker has. Moreover, for this case study, the number of passwords to search was fixed to 100.
- *Sorting* case, we vary the size of the array to be sorted (100 000, 1 million, 10 million elements). In this case, the number of elements does not correspond to the number of tuples because parts of the array are transferred also as arrays of smaller size.
- *Ocean model*, we vary the grid size (300, 600, and 1200) which is related to the computational size of the initial task.
- *Matrix multiplication*, we vary the order of a square matrix (50, 100).

Remark 2. (Execution environment)
Our tests were conducted on a Ubuntu server (version 14.04.5 LTS) with 4 processors Intel Xeon E5-4607, each with 6 cores, 12 M Cache, 2.20 GHz, and with hyper-threading, offering in total 48 threads and 256 GB RAM. All case studies are implemented in Java 8.

Measured metrics. For the measurement of metrics, we have created a profiler which is similar to Clarkware Profiler[‡]. Clarkware Profiler calculates just the average time for the time series, while ours also calculates other statistics (e.g., standard deviation). Moreover, our profiler was designed also for analyzing tests carried out on more than one machine. For that reason, each process writes raw profiling data on a specific file; all files are then collected and used by specific software to calculate required metrics.

We use the manual method of profiling and insert methods `begin(label)` and `end(label)` into program code surrounding parts of the code we are interested in order to begin and stop counting time respectively. For each metrics, the label is different, and it is possible to use several of them simultaneously. This sequence of the actions can be repeated many times, and eventually, all the data are stored on disk for the further analysis.

Each set of experiments has been conducted 10 times with randomly generated input and computed an average value and a standard deviation of each metrics. To extensively compare the different implementations, we have collected the following measures:

Local writing time: time required to write one tuple into a local tuple space.
Local reading time: time required to read one tuple from a local tuple space using a template. This metrics checks how fast pattern matching works.
Remote writing time: time required to communicate with a remote tuple space and to perform write operation on it.
Remote reading time: time required to communicate with a remote tuple space and to perform read operation on it.
Search time: time required to search a tuple in a set of remote tuple spaces.
Total time: total execution time. The time does not include an initialization of tuple spaces.
Number of visited nodes: number of visited tuple spaces before a necessary tuple was found.

3.4 | Experimental results

Please notice that all plots used in the paper report results of our experiments in a logarithmic scale. When describing the outcome, we have only used those plots which are more relevant to evidence the difference between the four tuple space systems.

Password search. In Figures 6-7 is reported the trend of the total execution time as the number of workers and size of considered database increase. In Figure 6, the size of the database is 100 thousand entries, while Figure 7 reports the case in which the database contains 1 million of elements. From the plot, it is evident that GIGASPACES exhibits better performances than the other systems. In the diagrams below, in addition to the results for the four implementations, we will provide measures for a different implementation of the case study with TUPLEWARE. We will refer to the additional experiment by TUPLEWARE(H) and will describe it in some details at the end of this paragraph.

Figure 8 depicts the local writing time for each implementation with different numbers of workers. As we can see, by increasing the number of workers (that implies reducing the amount of local data to consider), the local writing time decreases. This is more evident for TUPLEWARE, which really suffers when a big number of tuples (e.g., 1 million) is stored in a single local tuple space. The writing time of KLAIM is the lowest among other systems and does not change significantly during any variation in the experiments. The local writing time of MOZARTSPACES remains almost the same when the number of workers increases. Nonetheless, its local time is bigger with respect to the other systems, especially when the number of workers is equal or greater than 10.

[‡]The profiler was written by Mike Clark; the source code is available on GitHub: https://github.com/akatkinson/Tupleware/tree/master/src/com/clarkware/profiler

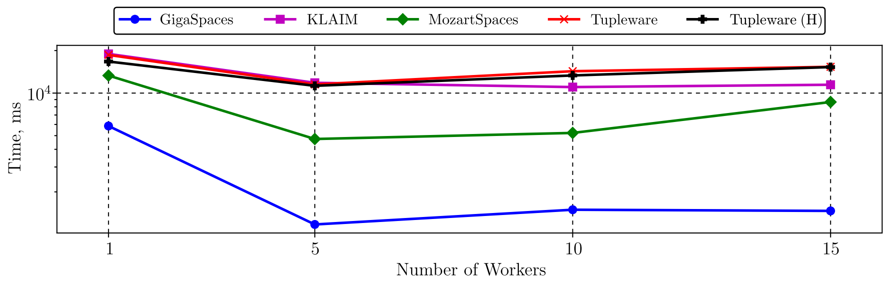

FIGURE 6 Password search. Total time (100 thousand passwords)

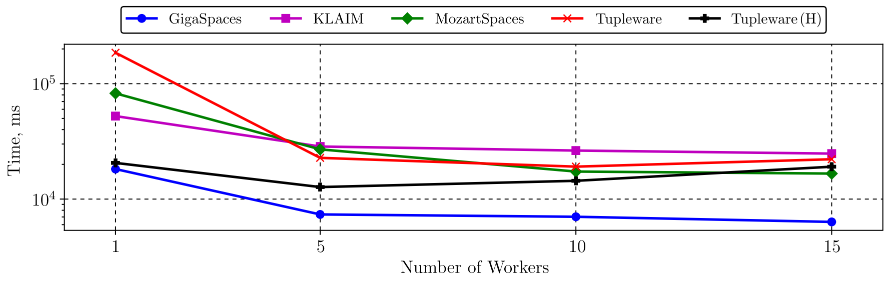

FIGURE 7 Password search. Total time (1 million passwords)

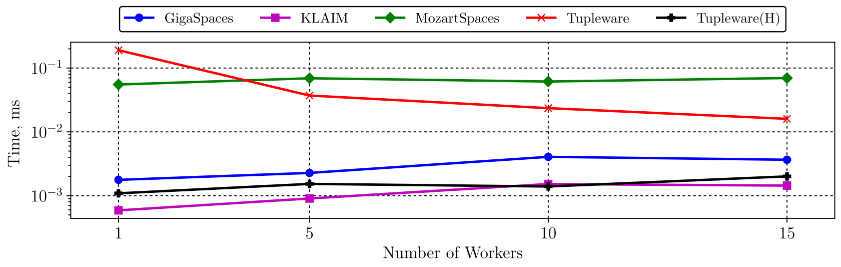

FIGURE 8 Password search. Local writing time (1 million passwords)

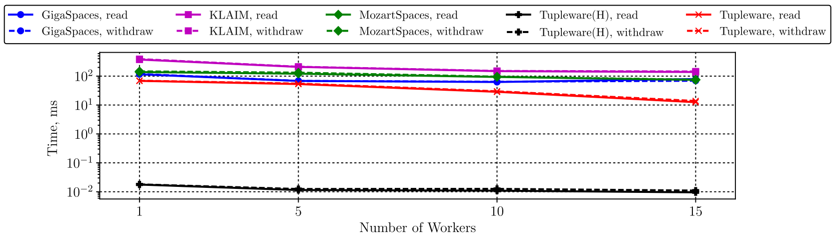

FIGURE 9 Password search. Local reading and withdrawing time (1 million passwords)

In our experiments, we did measure only the writing and reading time which are the most frequently used in the considered case studies. We ignored the withdraw operation; it can be seen as an atomic step consisting of a read followed by a removal action. The latter requires synchronization on (part of) the tuple space, and the way synchronization is guaranteed highly impacts on performances. Thus, we only evaluated the cost of withdrawing for the passwords search case study by considering local searches and allowing withdrawing, and not just reading, passwords. The local reading and withdrawing times for the new program are shown in Figure 9. Figure 9 shows that, as expected, withdrawing time is slightly greater than reading time.

By increasing the number of workers, the difference becomes less evident and approximately equal to MOZARTSPACES time that does not change considerably but always remains much greater than the time of TUPLEWARE and GIGASPACES. Since this case study requires little synchronization among workers, performance improves when the level of parallelism (the number of workers) increases.

To better clarify the behaviors of KLAIM and TUPLEWARE (the only implementations with publicly available code), we can consider their actual implementation of local tuple spaces. KLAIM is based on `Vector` that provides a very fast insertion with the complexity $O(1)$ when it is performed at the end of the vector and a slow lookup with the complexity $O(n)$. TUPLEWARE exploits `Hashtables` as a container for tuples but their efficient use relies on using specific kinds of templates (the first fields should contain values not variables). This is not the case for our skeleton implementation, and thus, TUPLEWARE's potentials are not fully exploited. Indeed, its performances are similar to those of KLAIM.

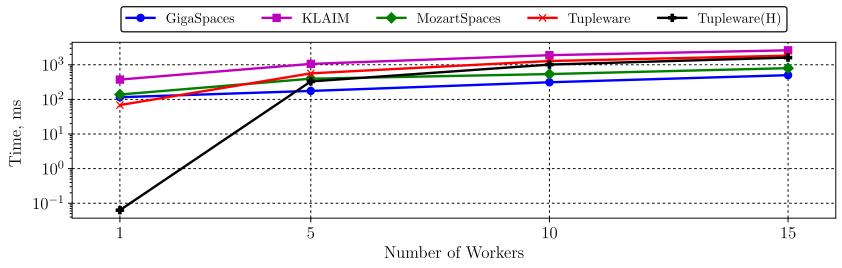

FIGURE 10 Password search. Search time (1 million passwords)

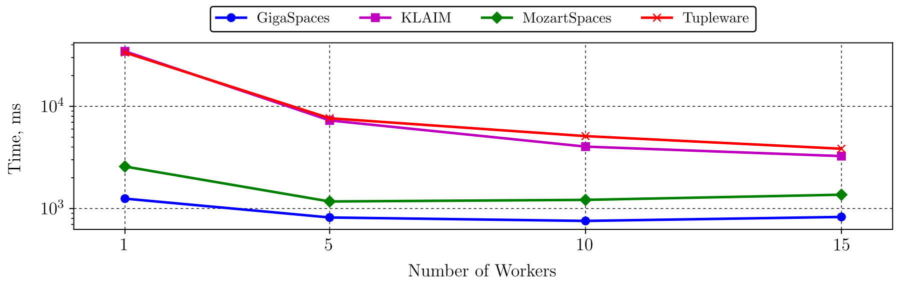

FIGURE 11 Sorting. Total time (1 million elements)

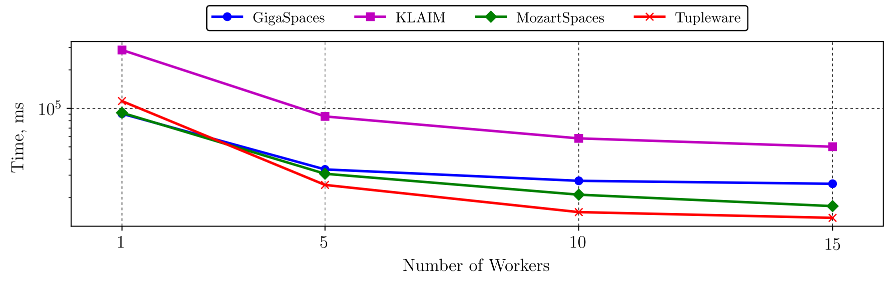

FIGURE 12 Sorting. Total time (10 million elements)

We thus considered an alternative skeleton with tuples that meet TUPLEWARE requirements and noticed that its performances greatly improved while those of the others remained unchanged. In Figures 6 - 10, TUPLEWARE(H) refers to tests with this adapted code. TUPLEWARE(H) demonstrates the writing time that is close to the best one of KLAIM (Figure 8) and the best reading time (Figure 9). TUPLEWARE(H) is especially beneficial when a bigger size of the database is used.

The search time is similar to the local reading time but takes into account searching in remote tuple spaces. When considering just one worker, the search time is the same as the reading time in a local tuple space; however, when the number of workers increases, the search time of TUPLEWARE and KLAIM grows faster than the time of GIGASPACES. Figure 10 shows that GIGASPACES and MOZARTSPACES are more sensitive to the number of tuples than to the number of accesses to the tuple space.

Summing up, we can remark that the local tuple spaces of the four systems exhibit different performances depending on the operation on them: the writing time of KLAIM is always significantly smaller than the others, while the pattern matching mechanism of TUPLEWARE allows for faster local searching. The performance of MOZARTSPACES mostly depends on the number of involved workers: it exhibits average time for local operations when one worker is involved, while it shows the worst time with 15 workers.

Sorting. Figure 11 shows that GIGASPACES exhibits significantly better execution time when the number of elements to sort is 1 million. As shown in Figure 12, when 10 million elements are considered and several workers are involved, TUPLEWARE and MOZARTSPACES exhibit a more efficient parallelization and thus requires less time.

This case study is computation intensive but requires also an exchange of structured data. The benefits of the parallelization is more evident for the tests with an array of 10 million elements where the percentage of the time spent for data communication and transmission is less than the time spent for sorting.

The performance of KLAIM is visibly worse than others even with one worker (see, e.g., Figure 12). As shown in Figure 13, despite the fact that workers do not need to search for unsorted data in a set of tuple spaces, the reading time of KLAIM is considerably higher than the reading time of the other implementations. The reason is that KLAIM spends a considerable amount of time for data transmission that affects the total time. Due to the importance of this problem, we have been working on a new version of KLAIM (described below) and have performed experiments with it as well.

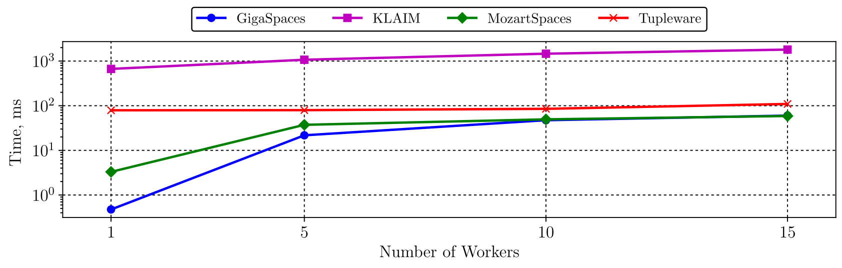

FIGURE 13 Sorting. Remote reading time (10 million elements)

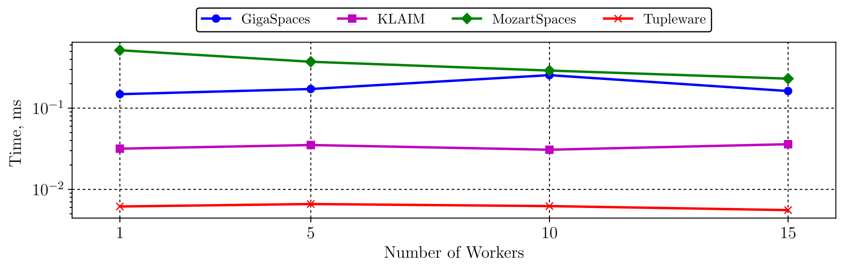

FIGURE 14 Sorting. Local writing time (10 million elements)

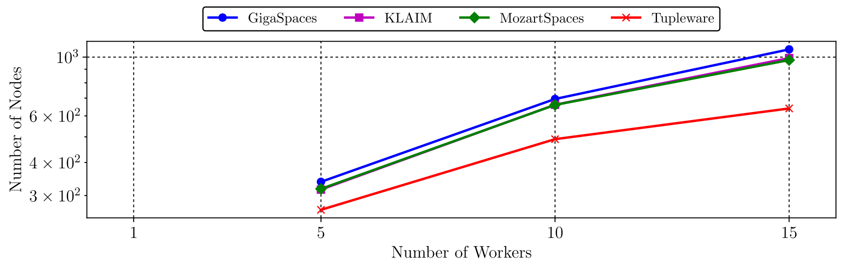

FIGURE 15 Ocean model. Number of visited nodes (the grid size is 1200)

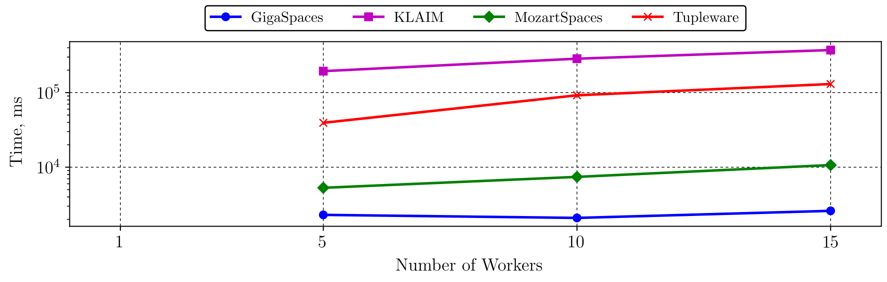

FIGURE 16 Ocean model. Total time (the grid size is 600)

Figure 14 shows that, in all cases, increasing the number of workers does not affect the local writing time. At the same time, by comparing Figures 8 and 14, we see that all tuple space implementation except TUPLEWARE spend more time for writing. That is especially evident for GIGASPACES and KLAIM.

It is worth noting that, during our tests, we experienced some problems with MOZARTSPACES (mozartspaces-dist-2.3-SNAPSHOT-r15239). The problems, related to data loss, occurred when more than 10 workers were present. After analyzing our code, the Vienna group pinpointed the bug and fixed it (mozartspaces-dist-2.3-SNAPSHOT-r21c82ce88e5456). The experiments presented here are based on the revised implementation.

Ocean model. This case study was chosen to examine the behavior of tuple space systems when specific patterns of interactions come into play. Out of the four considered systems, only TUPLEWARE has a method for reducing the number of visited nodes during search operation which helps in lowering search time. Figure 15 depicts the number of visited nodes for different grid sizes and a different number of workers (for this case study, in all figures, we consider only 5, 10, 15 workers because for one worker generally tuple space is not used). The curve depends weakly on the size of the grid for all systems and much more on the number of workers. Indeed, from Figure 15, we can appreciate that TUPLEWARE performs a smaller number of nodes visits and that when the number of workers increases the difference is even more evident[§].

[§]Figure 15, the curves for KLAIM and GIGASPACES are overlapping and purple wins over blue.

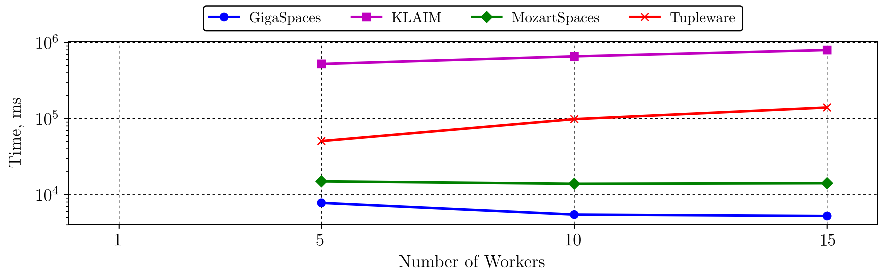

FIGURE 17 Ocean model. Total time (the grid size is 1200)

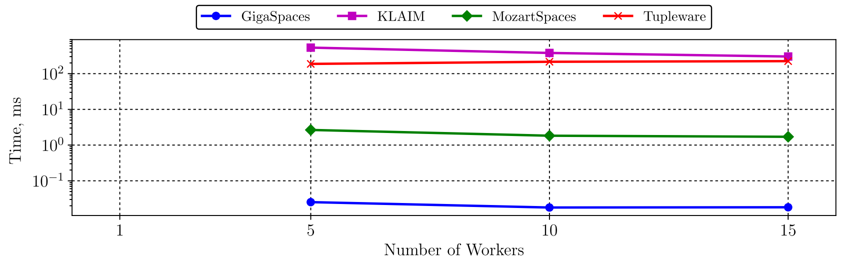

FIGURE 18 Ocean model. Remote reading time (the grid size is 1200)

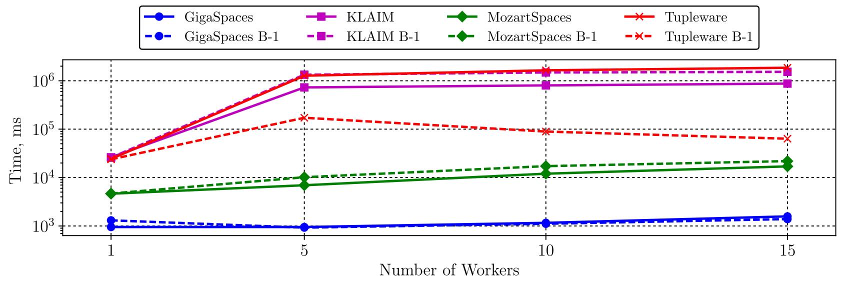

FIGURE 19 Matrix multiplication. Total time (the matrix order is 100)

The difference in the number of visited nodes does not affect significantly the total time of execution for different values of the grid size (Figures 16-17) mostly because the case study requires many read operations from remote tuple spaces (Figure 18).

As shown in Figure 18, the time of remote operation varies for different tuple space systems. For this case study, we can neglect the time of the pattern matching and consider that this time is equal to the time of communication. For KLAIM and TUPLEWARE, these times were similar and significantly greater than those of GIGASPACES and MOZARTSPACES. KLAIM and TUPLEWARE communications rely on TCP, and to handle any remote tuple space, one needs to use exact addresses and ports. GIGASPACES, that has a centralized implementation, most likely does not use TCP for data exchange but relies on a more efficient memory-based approach. The communication time of MOZARTSPACES is in the middle (in the plot with logarithmic scale) but close to GIGASPACES by its value: for GIGASPACES, this time varies in the range of 0.0188 to 0.0597 ms, for MOZARTSPACES in the range of 2.0341 to 3.0108 ms, and for TUPLEWARE and KLAIM, it exceeds 190 ms. Therefore, as it was mentioned before, GIGASPACES and MOZARTSPACES implements a read operation differently from TUPLEWARE and KLAIM, and it is more effective when working on a single host.

Figure 16 provides evidence of the effectiveness of TUPLEWARE when its total execution time is compared with the KLAIM one. Indeed, KLAIM visits more nodes and spends more time for each read operation, and the difference increases when the grid size grows and more data have to be transmitted as it is shown in Figure 17.

Matrix multiplication. This case study mostly consists of searching tuples in remote tuple spaces, and this implies that the number of remote read operations is by far bigger than the other operations. Therefore, GIGASPACES and MOZARTSPACES outperform other tuple space systems total execution time (Figure 19).

As discussed in Section 3.1, we consider two variants of this case study: one in which matrix B is uniformly distributed among the workers (as the matrix A), and one in which the whole matrix is assigned to one worker. In the following plots, solid lines correspond to the experiments with uniform distribution and dashed lines correspond to ones with the second type of distribution (names ending with B-1 are used to refer this kind of distribution).

Figure 20 depicts the average number of the nodes that it is necessary to visit in order to find a tuple for each worker. When considering experiments with more than one worker, all tuple space systems except TUPLEWARE demonstrate similar behavior: the total time almost coincides for

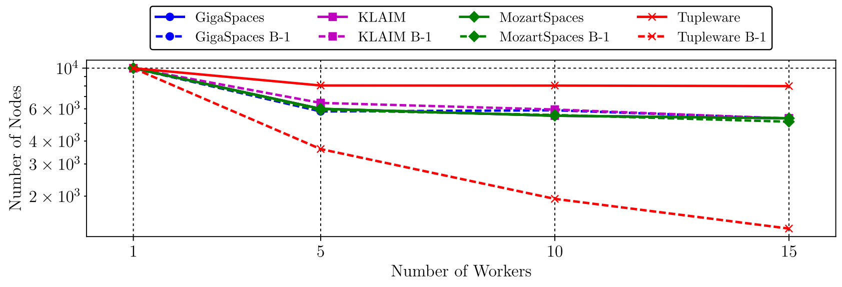

FIGURE 20 Matrix multiplication. Number of visited nodes (the matrix order is 100)

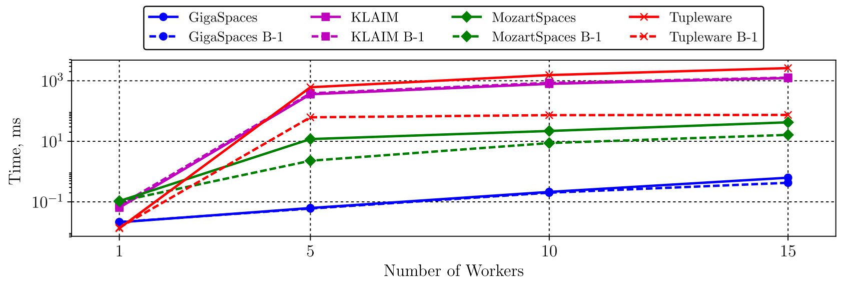

FIGURE 21 Matrix multiplication. Search time (the matrix order is 100)

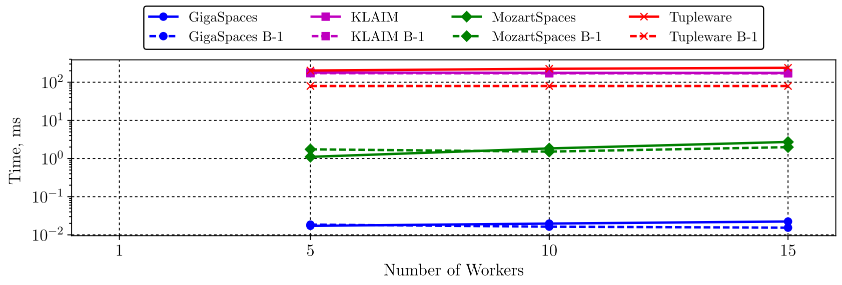

FIGURE 22 Matrix multiplication. Remote reading time (the matrix order is 100)

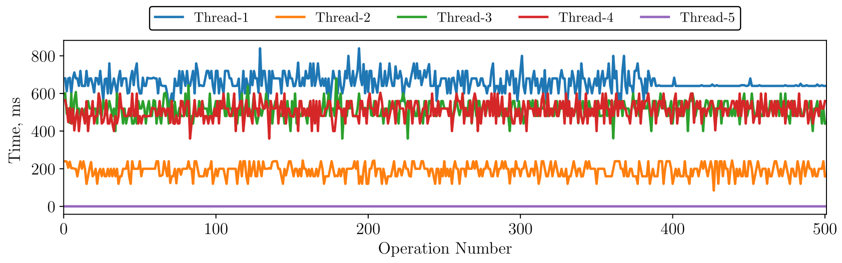

FIGURE 23 Matrix multiplication. Search time per thread (KLAIM, the matrix order is 50, 5 workers)

both types of the distribution. However, for the uniform distribution, TUPLEWARE exhibits always greater values, and for the second type of distribution, the values are significantly lower. The second case reaffirms the results of the previous case study because in this case, all workers know where to search the rows of the matrix B almost from the very beginning that leads to the reduction of the amount of communication, affects directly the search time (Figure 21), and, in addition, implicitly leads to the lower remote reading time (Figure 22, the remote reading time is not displayed for one worker because only the local tuple space of the worker is used). In contrast, for the uniform distribution, TUPLEWARE performs worse because of the same mechanism which helps it in the previous case: when it needs to iterate over all the rows one by one, it always starts the checking from the tuple spaces which were already checked at the previous time and which do not store required rows. Therefore, every time it checks roughly all tuple spaces.

As shown in Figure 19, the runs with the uniform distribution outperform (i.e., they show a lower search time) the others, except the ones where TUPLEWARE is used. This is more evident in the case of KLAIM, where the difference in execution time is up to two times. To explain this behavior, we looked at the time logs for one of the experiments (matrix order 50, workers 5) which consist of several files for each worker (e.g., Java Thread) and

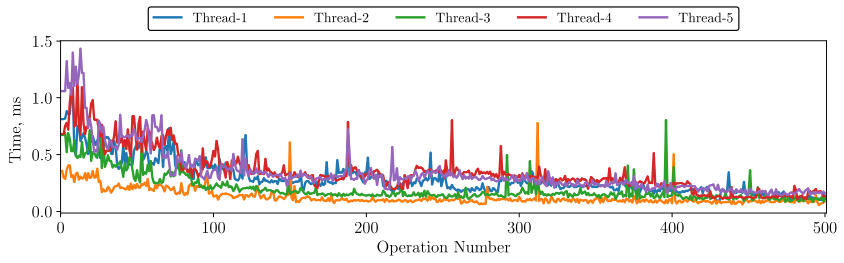

FIGURE 24 Matrix multiplication. Search time per thread (GIGASPACES, the matrix order is 50, 5 workers)

paid attention to the search time that mostly affects the execution time. The search time of each search operation performed during the execution of the case study for KLAIM and GIGASPACES is shown in Figures 23 and 24, respectively (these two figures are presented not in a logarithmic scale). Every colored line represents one of five threads of workers and shows how the search time changes when the program executes. As we can see, although the search time of GIGASPACES is much less than the time of KLAIM, there is a specific regularity for both tuple space systems: the average value of search time for each thread significantly differs from each other. At the same time, the search operation is the most frequent one, and it mostly defines the time of the execution. Therefore, a thread with the greatest average value of the search time determines the time of execution. The situation is similar with GIGASPACES and MOZARTSPACES, but less visible since the search time of KLAIM is much greater (Figure 21).

The results of this case study are generally consistent with the previous ones: remote operations of GIGASPACES and MOZARTSPACES are much faster and better fits to the application with frequent inter-process communication; TUPLEWARE continues to have an advantage in the application with a specific pattern of communication. At the same time, we revealed that in some cases, this feature of TUPLEWARE had the side-effect that negatively affected its performance.

Block-wise matrix multiplication Here we introduce a variant of the matrix multiplication algorithm that supports a higher degree of parallelization and relies on a different communication strategy. In the "row-wise" multiplication, each worker gets a part of the initial matrices, and each searches for data in tuple spaces of the others. In the block-wise approach, instead, the master keeps all data, and workers access its space to get them. The block-wise algorithm of multiplication operates with matrices of smaller size (called blocks) obtained by evenly splitting the initial matrices.

Suppose that matrix A is split into blocks of q rows and n columns and matrix B is split into blocks of n rows and r columns. Then each cell $c_{i,j}$ of the resulting matrix C can be iteratively obtained using formula $c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}$ where $i \in \{1, .., q\}, j \in \{1, .., r\}$ and $k \in \{1, .., n\}$.

In Listing 5, we report the actual code for *Block-wise matrix multiplication*. The master first initializes matrix C (line 2) and adds to its local tuple space the blocks for matrices A, B, C (lines 3-7), then it assigns tasks (lines 9-10). Finally, it collects the results sent by the workers (lines 12-14) and notifies termination to them (lines 17-18). Each worker performs the assigned tasks (lines 26-30) and terminates its work (line 24-25). After getting a task (line 23), a worker gets the two assigned matrix blocks (lines 26-27) and adds their product to the intermediate result (lines 28-30). Finally, it puts the result into the space of the master (line 31) and continues its work.

We compared the performances of the two matrix multiplication algorithms using GIGASPACES, because this implementation exhibited the best performances in the tests for the row-wise version. In our experiments, we divided the matrices into square blocks (e.g., $q = r = n$) to consider just one parameter that we call *block size* (we used *block size* $\in \{5, 10\}$). As shown in Figure 25, depending on the size of blocks and the number of workers, the block-wise multiplication exhibits different times. When blocks of bigger size are used, we have that row-wise multiplication is constantly outperformed.

As shown in Figure 26, the remote reading time is much higher for the block-wise implementation; in this case, reading requests are directed to master whereas in the row-wise case they are directed to workers.

Modifying KLAIM. The previous experiments provided us with the evidence that KLAIM suffers from a high cost of communications when one process accesses the tuple space of another. At the same time, for instance, MOZARTSPACES does not have such a problem, and therefore, we have chosen to improve the part of the network communication in KLAIM without touching others.

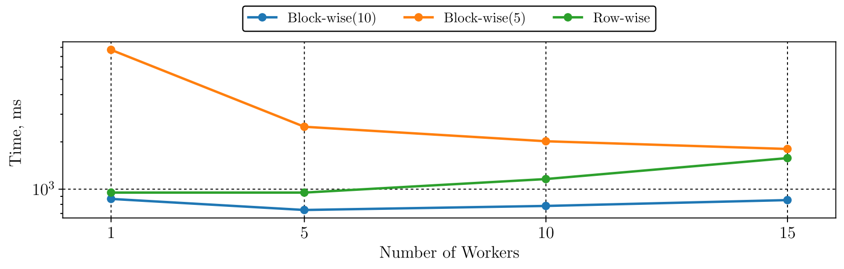

FIGURE 25 Block-wise and row-wise matrix multiplications. Total time (GIGASPACES, the matrix order is 100)

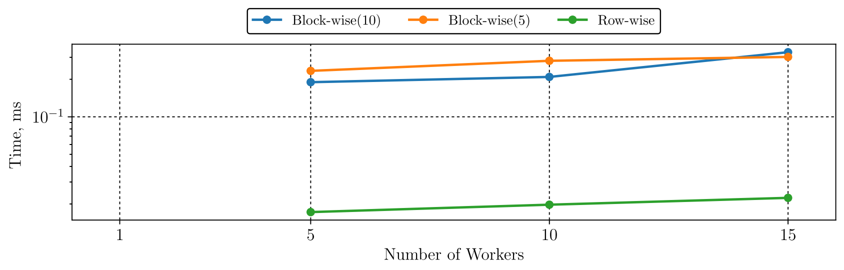

FIGURE 26 Block-wise and row-wise multiplications. Remote reading time (GIGASPACES, matrix order is 100)

Listing 5: Block-wise matrix multiplication. Listing of master and worker processes

```
Master():
  initMatrix(matrixC)
  for i in range(blockNumber):
    for j in range(blockNumber):
      masterTS.out("matrixA", i, j, matrixA.block(i, j))
      masterTS.out("matrixB", i, j, matrixB.block(i, j))
      masterTS.out("matrixC", i, j, 0, matrixC.block(i, j))

      for k in range(blockNumber):
        masterTS.out("task", i, j, k)

  for k in range(blockNumber*blockNumber):
    masterTS.in("matrixC", ?i, ?j, blockNumber, ?blockC)
    C.block(i, j) = blockC

  // poison tuples for all workers
  for i in range(workerNumber):
    masterTS.out("task", -1, -1, -1)


Worker():
  while true:
    masterTS.in("task", ?i, ?j, ?k)
    if i == -1:
      break
    masterTS.rd("matrixA", i, k, ?blockA)
    masterTS.rd("matrixB", k, j, ?blockB)
    tempBlock = blockA*blockB
    masterTS.in("matrixC", i, j, ?count, ?blockC)
    blockC = blockC + tempBlock
    masterTS.out("matrixC", i, j, count+1, blockC)
```

We then have substituted the part of KLAIM responsible for sending and receiving tuples. It was based on Java IO, the package containing classes for the data transmission over the network. For the renewed part, we have opted to use Java NIO, non-blocking IO,[38] which is a modern version of IO and in some cases allows for an efficient use of resources. Java NIO is beneficial when used to program applications dealing with many incoming connections. Moreover, for synchronization purposes, we used a more recent package (java.util.concurrent) instead of synchronization methods of the previous generation.

To evaluate the effectiveness of the modified KLAIM, we tested it with the *Matrix multiplication* case study since it depends on remote operations more than other case study and benefits of the modification are clearer. We just show the results for the case in which matrix B is uniformly distributed among the workers since the other case shows similar results. As shown in Figure 27, the remote writing time decreased significantly. The remote reading time of the modified KLAIM is close to the one of MOZARTSPACES (Figure 28) and demonstrates similar behavior. In Figure 28, the remote reading time for the runs with one worker is not shown since in this case, just the local tuple space of the worker is used. The remote reading operations mostly determine the total time and that is why graphics of modified KLAIM and MOZARTSPACES in Figure 29 are similar.

Performing experiments on *Sorting*, we noticed that the remote reading time for KLAIM was much higher than that of the others (Figure 13). Figure 30 shows that the remote reading time is much less for the modified KLAIM than for the original one and it is similar to that of GIGASPACES and MOZARTSPACES. These results show that the problem with KLAIM was indeed due to the actual implementation of communication.

Experiments with several host machines. The results of the previous experiments which were conducted using only one host machine provide us evidence that GIGASPACES has a more efficient implementation of communication and that is very beneficial when many operations on remote tuple spaces are used. Since we do not have access to its source code, we conjecture that GIGASPACES uses an efficient inter-process communicating mechanism and do not resort to socket communications as the other implementations.

To check whether GIGASPACES remains efficient when running over a network, we have used Docker (www.docker.com). This technology allowed us to create a number of lightweight containers, similar to virtual machines, that are isolated environments equipped with an operating system

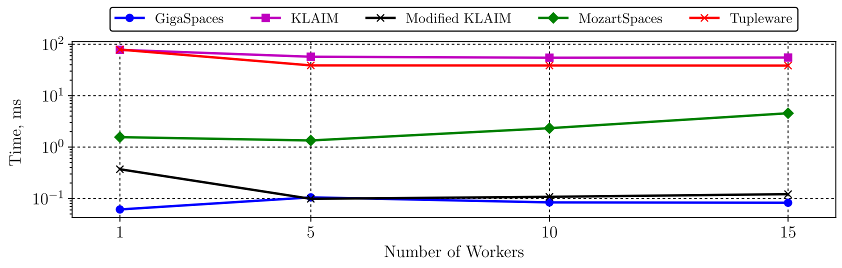

FIGURE 27 Matrix multiplication. Remote writing time (with modified KLAIM, the matrix order is 100)

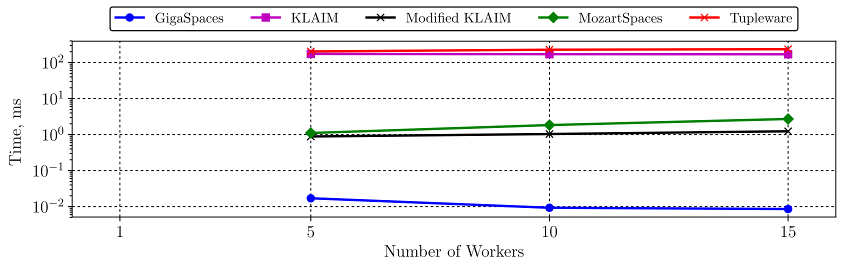

FIGURE 28 Matrix multiplication. Remote reading time (with modified KLAIM, the matrix order is 100)

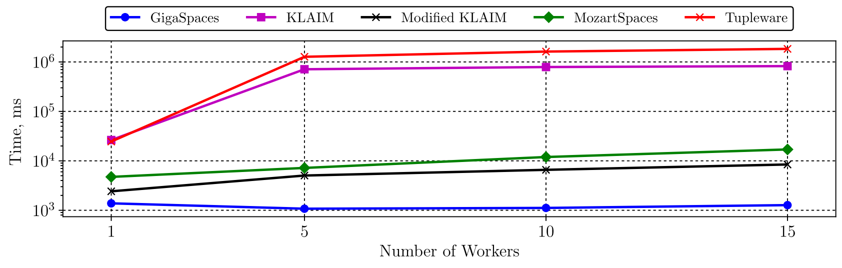

FIGURE 29 Matrix multiplication. Total time (with modified KLAIM, the matrix order is 100)

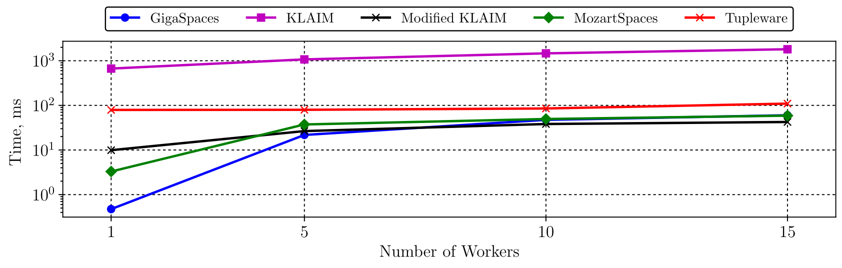

FIGURE 30 Sorting. Remote reading time (with modified KLAIM, 10 million elements)

where a minimal set of software components is installed that allows to use less resources. Each container has a network interface to communicate with others. Masters and workers were launched in their own containers, and each set of experiments was conducted 10 times. We conducted experiments over two case studies: *Sorting* and *Matrix multiplication* with uniform distribution and we focused on remote reading time, the most frequently used operation in all case studies.

To further explore the impact of GIGASPACES's inter-machine communication, we have also conducted experiments on a real network: 16 identical virtual machines (Oracle VM VirtualBox[¶]) evenly placed in 3 real servers (see in Remark 2). Each virtual machine is equipped with 3 GB RAM and Linux Lubuntu 16.04.

When evaluating performances of *Sorting*, we have that the remote reading time of the networked version is smaller than the one of the single host version (Table 2). This might be related to the amount of processed data (much larger than those of other case studies) and to the way Java Virtual Machine (JVM) works. In the networked version, each tuple space is managed by an independent JVM, whereas in a single host version only, one JVM is used.

[¶]Oracle VM VirtualBox is a free and open-source hypervisor for x86 computers from Oracle Corporation (https://www.virtualbox.org/)

TABLE 2 Sorting (host machines, GIGASPACES, 10 million elements, 15 workers)

	Remote Reading Time	Total Time
Single host version	60.0817	25701
Docker version	12.4407	28773
Networked version	12.2382	11112

TABLE 3 Matrix multiplication (host machines, GIGASPACES, the matrix order is 100, 15 workers)

	Remote Reading Time	Total Time
Single host version	0.0223	1577
Docker version	2.867	19641
Networked version	1.3340	13918

When evaluating *Matrix multiplication*, we have that the remote reading time exceeds significantly the one for the single host version (Table 3). In our view, this is an evidence of the fact that GIGASPACES does not use network protocols when all processes run on a single machine.

The numbers of Tables 2 and 3 show that the tests based on dockers and real networks provide similar results for remote reading times. The total execution time is lower for the network-based tests only because more powerful machines were used.

4 | CONCLUSIONS

Distributed computing is getting increasingly pervasive, with demands from various applications domains and highly diverse underlying architectures from the multitude of tiny things to the very large cloud-based systems. Tuple spaces certainly offer valuable tools and methodologies to help develop scalable distributed applications/systems.[39,40] This paper has first surveyed and evaluated a number of tuple space systems, then it has analyzed more closely four different systems. We considered GIGASPACES, because it is one of the few currently used commercial products, KLAIM, because it guarantees code mobility and flexible manipulation of tuple spaces, MOZARTSPACES as the most recent implementation that satisfies the main criteria we do consider essential for tuple-based programming, and TUPLEWARE, because it is the one that turned out to be the best in our initial evaluation. We have then compared the four system by evaluating their performances over four case studies: one testing performance of local tuple space, a communication-intensive one, a computational-intensive one, and one demanding a specific communication pattern.

In our experiments, we vary the workload by changing the size of the input data; in this way, we can track the different performances of the different tuple space systems. In our opinion, the results of our experiments are sufficient to provide a full account of all tuple space systems and increasing the workload would not introduce any significant difference.

Our work follows the lines of Wells et al,[41] but we have chosen more recent implementations and conducted more extensive experiments. On purpose, we ignored implementations of systems that have been directly inspired by those considered in the paper.

After analyzing the outcome of the experiments, it became clear what are the critical aspects of a tuple space system that deserve specific attention to obtain efficient implementations. The critical choices are concerned with inter-process and inter-machine communication and local tuple space management.

The first aspect is related to data transmission and is influenced by the choice of algorithms that reduce communication. For instance, the commercial system GIGASPACES differs from the other systems that we considered for the technique used for data exchange, exploiting memory-based inter-process communication, that guarantees a considerably smaller access time to data. Therefore, the use of this mechanism on a single machine does increase efficiency. However, when working with networked machines, it is not possible to use the same mechanism, and we need to resort to other approaches (e.g., the TUPLEWARE one) to reduce inter-machine communication and to have more effective communications. To compare GIGASPACES with the other tuple space systems under similar conditions and thus to check whether it remains efficient also in the case of distributed computing, we have carried out experiments using a network where workers and masters processes are executed in separate environments (Docker containers, virtual machines). The results of these experiments show that, although the remote operations are much slower, the overall performance of GIGASPACES remains high.

The second aspect, concerned with the implementation of local tuple spaces, is heavily influenced by the data structure chosen to represent tuples, the corresponding data matching algorithms, and by the lock mechanisms used to prevent conflicts when accessing the tuple space. In our experiments, the performance of different operations on tuple spaces varies considerably; for example, KLAIM provides fast writing and slow reading, whereas TUPLEWARE exhibits high writing time and fast reading time. The performances of a tuple space system would depend also on the chosen system architectures which determine the kind of interaction between their components. Indeed, it is evident that all the issues should be tackled together because they are closely interdependent. Another interesting experiment would be to use one of the classical database systems that offer fast I/O operations to model tuple spaces and their operations, in order to assess their performances over our case studies. The same experiment can be carried out by considering modern no-SQL databases, such as Redis (https://redis.io/) or MongoDB (https://www.mongodb.com/).

It is worth noting that all the considered implementations exhibit very poor performances especially when the level of parallelism is increased. For all case studies, increasing the number of workers leads to a (feeble) gain in execution time only when considering password search and matrix multiplication with GIGASPACES and MOZARTSPACES. In general, the potential gain provided by additional workers is hampered by the cost of inter-process communications. To this end, one could think of using MPI[7] libraries when using different machines, or to look for more efficient and compact ways of data transmission over networks.

Once we determined, via our experiments, the critical aspects and the good design choices of the different Linda implementations, we have reconsidered the implementation of KLAIM. We have focused on the part which evidently damages its overall performance, i.e., the one concerned with data transmission over the network. The outcome of the experiments on the new version are promising. The time of remote writing and reading is comparable to that of MOZARTSPACES and is significantly lower than that required by the previous implementation of KLAIM. Moreover, the modified version allows faster execution of the tasks where many inter-process communications are considered. It is worth noting that the plots and results concerning GIGASPACES, TUPLEWARE, and KLAIM are pretty similar to those presented in a previous version of our paper,[15] even though the ones presented in this paper were conducted on a much more powerful machine.

We plan to use the results of this work as the basis for designing an efficient tuple space system which offers programmers the possibility of selecting (e.g., via a dashboard) the desired features of the tuple space according to the specific application. In this way, one could envisage a distributed middleware with different tuple spaces implementations each of them targeted to specific classes of systems and devised to guarantee the most efficient execution. The set of configuration options will be a key factor of the work. One of such options, that we consider important for improving performances, is data replication. In this respect, we plan to exploit the results of RepliKlaim[42] which enriched KLAIM with primitives for replica-aware coordination. Indeed, we will use the current implementation of KLAIM as a starting point for the re-engineering of tuple space middleware.

ACKNOWLEDGMENTS

We would like to thank Lorenzo Bettini and Michele Loreti for fruitful discussions on tuple spaces implementations, and Alberto Lluch Lafuente for suggestions and feedback on a preliminary version of this document. Furthermore, we are grateful to the anonymous reviewers for their remarks and very useful suggestions. One of them even provided with the alternative code for matrix multiplication that we used in our experiments.

ORCID

Claudio Antares Mezzina http://orcid.org/0000-0003-1556-2623

REFERENCES

1. Nitzberg B, Lo VM. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer*. 1991;24(8):52-60. https://doi.org/10.1109/2.84877
2. Gelernter D, Bernstein AJ. Distributed communication via global buffer. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1982; 1982; Ottawa, Canada. 10-18.
3. Birrell A, Nelson BJ. Implementing remote procedure calls. *ACM Trans Comput Syst*. 1984;2(1):39-59. https://doi.org/10.1145/2080.357392.
4. Pitt E, McNiff K. *java.rmi: The Remote Method Invocation Guide*. USA: Addison-Wesley Longman Publishing Co., Inc.; 2001.
5. Andrews GR. *Concurrent Programming: Principles and Practice*. USA: Benjamin-Cummings Publishing Co., Inc.; 1991.
6. Atkinson RR, Hewitt C. Parallelism and synchronization in actor systems. In: Graham RM, Harrison MA, Sethi R, eds. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. Los Angeles, California: ACM; 1977:267-280.
7. Barker B. Message Passing Interface (MPI). *Workshop: High Performance Computing on Stampede*. Ithaca, NY: Cornell University; 2015.
8. Gelernter D. Generative communication in Linda. *ACM Trans Program Lang Syst*. 1985;7(1):80-112.
9. Eugster PT, Felber P, Guerraoui R, Kermarrec A. The many faces of publish/subscribe. *ACM Comput Surv*. 2003;35(2):114-131. http://doi.acm.org/10.1145/857076.857078.
10. Carriero N, Gelernter D. *How to Write Parallel Programs—A First Course*. Cambridge, Massachusetts: MIT Press; 1990.
11. Ceriotti M, Murphy AL, Picco GP. Data sharing vs. message passing: synergy or incompatibility? an implementation-driven case study, *Proceedings of the 2008 ACM symposium on Applied computing*; 2008; Fortaleza, Ceara, Brazil. 100-107.
12. Patterson LI, Turner RS, Hyatt RM. Construction of a fault-tolerant distributed tuple-space, *Sac'93*; 1993:279-285. http://doi.acm.org/10.1145/162754.162899.
13. Bessani AN, Correia M, da Silva Fraga J, Lung LC. An efficient byzantine-resilient tuple space. *IEEE Trans Comput*. 2009;58(8):1080-1094. https://doi.org/10.1109/TC.2009.71.
14. Barbi R, Buravlev V, Mezzina CA, Schiavoni V. Block placement strategies for fault-resilient distributed tuple spaces: an experimental study. In: Chen LY, Reiser JP, eds. *Distributed applications and interoperable systems - DAIS 2017*, Lecture Notes in Computer Science, vol. 10320. Neuchâtel, Switzerland: Springer; 2017. To appear.
15. Buravlev V, De N R, Mezzina CA. Tuple spaces implementations and their efficiency. In: Lluch-Lafuente A, Proença J, eds. *Coordination Models and Languages - COORDINATION 2016*, Lecture Notes in Computer Science, vol. 9686. Heraklion, Crete, Greece: Springer; 2016:51-66.

16. Murphy AL, Picco GP, Roman G-C. Lime: a coordination model and middleware supporting mobility of hosts and agents. *ACM Trans Softw Eng Methodol*. 2006;15(3):279-328. https://doi.org/10.1145/1151695.1151698.
17. Julien C, Roman G-C. Active coordination in ad hoc networks. In: Nicola RD, Ferrari GL, Meredith G, eds. *Coordination Models and Languages, COORDINATION 2004*, Lecture Notes in Computer Science, vol. 2949. Pisa, Italy: Springer; 2004:199-215.
18. Omicini A, Zambonelli F. TuCSoN: a coordination model for mobile information agents. In: Schwartz DG, Divitini M, Brasethvik T, eds. *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*. Pisa, Italy: IDI – NTNU, Trondheim (Norway); 19988:177-187.
19. Microsystems S. JS—JavaSpaces Service Specification. https://river.apache.org/doc/specs/html/js-spec.html Accessed September 15, 2016.
20. Lehman T, McLaughry S, Wyckoff P. T Spaces: the next wave. *In: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences, HICSS 1999*, Vol. 8. Maui, Hawaii, USA: Springer Berlin Heidelberg; 1999.
21. GigaSpaces. Concepts—XAP 9.0 Documentation—GigaSpaces Documentation Wiki. http://wiki.gigaspaces.com/wiki/display/XAP9/Concepts. Accessed September 15, 2016.
22. Atkinson A. *Tupleware: A Distributed Tuple Space for the Development and Execution of Array-Based Applications in a Cluster Computing Environment*, University of Tasmania School of Computing and Information Systems thesis. Tasmania: University of Tasmania; 2010.
23. Capizzi S. *A Tuple Space Implementation for Large-Scale Infrastructures*, Department of Computer Science Univ. Bologna thesis. Bologna, Italy: University of Bologna; 2008.
24. Van Der Goot R. *High Performance Linda Using a Class Library*, PhD thesis. Rotterdam, The Netherlands: Erasmus Universiteit Rotterdam; 2001.
25. Jiang Y, Jia Z, Xue G, You J. DTuples: a distributed hash table based tuple space service for distributed coordination. In: Grid and Cooperative Computing, 2006. GCC 2006. Fifth International Conference; 2006; Changsha, Hunan, China. 101-106.
26. Leal MA, Rodriguez N, Ierusalimschy R. LuaTS—a reactive event-driven tuple space. *J Universal Comput Sci*. 2003;9(8):730-744.
27. De Nicola R, Ferrari GL, Pugliese R. KLAIM: a kernel language for agents interaction and mobility. *IEEE Trans Software Eng*. 1998;24(5):315-330. https://doi.org/10.1109/32.685256.
28. Craß S, eva K, Salzer G. Algebraic foundation of a data model for an extensible space-based collaboration protocol. *In: InternationalDatabase Engineering and Applications Symposium (IDEAS 2009)*; 2009; Cetraro, Calabria, Italy. 301-306.
29. Bettini L, De Nicola R, Loreti M. Implementing mobile and distributed applications in X-Klaim. *Scalable Comput: Pract Experience*. 2006;7(4).
30. Mordinyi R, eva K, Schatten A. Space-based architectures as abstraction layer for distributed business applications. *In: The Fourth International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2010*. Krakow, Poland: IEEE Computer Society; 2010:47-53.
31. Craß S, Dönz T, Joskowicz G, eva K, Marek A. Securing a space-based service architecture with coordination-driven access control. *JoWUA*. 2013;4(1):76-97. http://isyou.info/jowua/papers/jowua-v4n1-4.pdf.
32. Bessani AN, Alchieri EAP, Correia M, da Silva Fraga J. DepSpace: a byzantine fault-tolerant coordination service; 2008:163-176. https://doi.org/10.1145/1352592.1352610.
33. Distler T, Bahn C, Bessani AN, Fischer F, Junqueira F. Extensible distributed coordination. In: Proceedings of the Tenth European Conference on Computer Systems, Eurosys 2015, Bordeaux, France, April 21-24, 2015; 2015:10:1-10:16. https://doi.org/10.1145/2741948.2741954.
34. Mattson T, Sanders B, Massingill B. *Patterns for Parallel Programming*, The software patterns series: Addison-Wesley; 2005.
35. Quinn MJ. *Parallel Programming in C with MPI and OpenMP*: McGraw-Hill Higher Education; 2004.
36. TIOBE index web site. http://www.tiobe.com/tiobe-index/ Accessed September 15, 2016.
37. Cass S. The 2017 Top Programming Languages. *IEEE Spectr*. 2017. https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
38. Oracle. Package java.nio. https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html Accessed September 15, 2016.
39. Bjornson RD. Linda on distributed memory multiprocessors. *PhD thesis*; 1992.
40. Carriero N. *Implementation of Tuple Space Machines*, Research report: Yale University; 1987.
41. Wells G, Chalmers A, Clayton PG. Linda implementations in java for concurrent systems. *Concurrency - Practice and Experience*. 2004;16(10):1005-1022.
42. Andric M, De Nicola R, Lluch-Lafuente A. Replica-based high-performance tuple space computing. In: Holvoet T, Viroli M, eds. *Coordination Models and Languages - COORDINATION 2015*, Lecture Notes in Computer Science, vol. 9037: Springer; 2015:3-18.

How to cite this article: Buravlev V, De Nicola R, Mezzina CA. Evaluating the efficiency of Linda implementations. *Concurrency Computat Pract Exper*. 2018;30:e4381. https://doi.org/10.1002/cpe.4381