Claud: Coordination, Locality And Universal Distribution

Jossekin BEILHARZ and Frank FEINBUBE [1] and Felix EBERHARDT and Max PLAUTH and Andreas POLZE

Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany

Abstract. Due to the increasing heterogeneity of parallel and distributed systems, coordination of data (placement) and tasks (scheduling) becomes increasingly complex. Many traditional solutions do not take into account the details of modern system topologies and consequently experience unacceptable performance penalties with modern hierarchical interconnect technologies and memory architectures. Others offload the coordination of tasks and data to the programmer by requiring explicit information about thread and data creation and placement. While allowing full control of the system, explicit coordination severely decreases programming productivity and disallows implementing best practices in a reusable layer.

In this paper we introduce Claud, a locality-preserving latency-aware hierarchical object space. Claud is based on the understanding that productivity-oriented programmers prefer simple programming constructs for data access (like key-value stores) and task coordination (like parallel loops). Instead of providing explicit facilities for coordination, our approach places and moves data and tasks implicitly based on a detailed topology model of the system relying on best performance practices like hierarchical task queues, concurrent data structures, and similarity-based placement.

Keywords. Distributed Object Space, Hierarchical NUMA, Federated Cloud

Introduction

With the introduction of clouds and cloud federations, computer systems have reached a new layer of complexity. Globally distributed, clouds offer easy access to vast resources at low cost enabling all kinds of parallel and distributed applications. Moreover, cloud federations promise adaptive region-aware service execution policies and load balancing, specialization, strong replication and fault-tolerance, vendor-independence, and much more [1]. In order to make good use of these resources, applications running in a cloud environment need to be capable of scaling from a single compute node to several thousands. Cloud-ready scaling can only be achieved by putting a strong focus on parallelism and locality: data and threads need to be placed in such a way that access latencies are minimal. In contrast to classical HPC-clusters, federated clouds can have arbitrary inter- and intra-connection networks of nodes resulting in heavy latency and bandwidth varia-

[1] Frank.Feinbube@hpi.de

tions. Since these characteristics are load-sensitive and thus can change during runtime, a static mapping as described by a programmer has limited feasibility. Competitive applications need to use sophisticated thread and data placement strategies that dynamically adapt the resource usage to their needs and the system topology.

The severe performance impact of threads and data coordination with respect to latencies and locality can be found in all layers of state-of-the-art system topologies. A prominent example are non-uniform memory access (NUMA) systems that are the foundation of modern server systems, especially the ones that are tailored for Big Data Analytics. Built around sophisticated interconnect technologies, NUMA systems can be regarded as a mix of parallel and distributed shared memory systems. Therefore, they share many similarities with cluster and cloud architectures: in modern hierarchical NUMA systems, access latencies vary severely depending on the distance of the NUMA nodes and the current load on the interconnects [2]. The performance impacts have become so predominant that a parallel implementation that regards a NUMA system as a conventional shared memory system runs longer with more resources than a serial implementation. Purely distributed implementations perform well, but do not benefit from the shared access capabilities of NUMA. As of today, it is unclear which programming model is best suited for parallel-distributed hybrids such as hierarchical NUMA systems. The de facto standard is a combination of message passing with MPI [3] for distribution and OpenMP [4] constructs for intra-node parallelization. In federated clouds, developers not only have to consider all the layers of it, but also use separate technologies to express and coordinate tasks and data on each level. We find that distinguishing between distributed and parallel systems is neither realistic nor helpful in modern computer architectures, where even single processors are essentially networks-on-chip.

In this paper we introduce our approach for a framework that allows developers to express tasks and data in a consistent way without the need for a detailed understanding of the underlying system topology. As in Google's MapReduce framework, the fundamental idea is to provide a programming model that is simple enough to allow for productivity while being powerful enough to allow for large-scale parallelism. We are convinced that the metaphor of a Tuple / Object Space combined with the application of performance optimization best practices to preserve locality and hide latencies provide us with a rich basis for our approach. Following the tradition of Ada and Linda, we name our framework after Claud Lovelace.

1. Related Work

Coordination Languages As described in the introduction, the intelligent coordination of tasks and data is crucial for application performance and scalability. Consequently the effort that is required from developers to instruct the coordination frame to make good use of the topology of the target system, is a major productivity factor. When we studied different approaches to create a framework that has a concise interface while providing enough information for efficient placements, we identified Tuple Spaces [5] as one of the most promising designs. The applicability of tuple spaces all the way up to large-scale architectures such as cloud was already emphasized in the original vision [6]. Furthermore, the interface required to work with a tuple space is understandable and can easily be adapted to reflect the expectations of contemporary programmers. Before presenting our application of tuple space concepts in the design of Claud in Section 2, we discuss interesting tuple space implementations as depicted in Figure 1.

	Target Architecture					Placement Strategy					Motivation		
	Shared-Memory	NUMA-aware	Cluster/NOW	Cloud/Grid/WA	Mobile	Central	Local	Round-Robin	Hash	Access Pattern	Fault Tolerance	HPC	Scalability
JavaSpaces			●			●							●
GigaSpaces XAP	●		●				●	●	●		●		●
TupleWare	●		●				●					●	●
DTuples			●						●		●		
MTS-Linda			●						●				●
PageSpace				●		●							●
C2AS				●		●							●
WCL				●		●				●			●
Jada				●		●							●
GridTS				●		●					●		
SwarmLinda			●	●					●				●
LIME					●		●				●		

Figure 1. An overview of the landscape of existing Tuple Space research. The selection is not intended to be complete, but rather to demonstrate the variety of existing implementations. We classify implementations based on our understanding of the system architecture they target, the placement strategy that they employ and the main motivation for their development. (”NOW” stands for ”network of workstations”, ”WA” for ”wide area”, ”HPC” for ”high performance computing”)

1.1. Tuple Space Classification

Target Architecture The computer system architecture that the tuple space was designed for or is predominantly being used with. *Shared-Memory* if the system is optimized for local accesses; *NUMA-aware* if the system is taking into account the non-uniform topology of hierarchical shared memory systems. Systems that would benefit from the topology-awareness of the underlying execution system (such as a NUMA-aware Java Virtual Machine) are not considered NUMA-aware; *Cluster/Network of Workstations* if the system mainly aims at (homogeneous) clusters with a fast network; *Cloud/Grid/Wide Area* if the system aims at multiple computers with a slow network; *Mobile* if the system is designed for mobile devices with sporadic connections.

Placement Strategy The strategy that is applied to place the tuples within the system. *Central* where the tuple space is placed on a single node and all the other nodes are accessing it remotely; *Local* where tuples are placed on the node of the process that created them; *Round-Robin* where tuples are distributed evenly around the system; *Hash* where tuples are placed using a hash based algorithm on either the full tuple or parts of the tuple; *Access Pattern* where the run-time system monitors the tuple access patterns and migrates them accordingly. We found the access pattern placement strategy only in WCL where it is applied not to tuple, but to tuple space placement.

One of the hardest challenges placement strategies have to face is tuple retrieval overhead. While the *Central* and the *Hash* placement strategies allow for straight-forward tuple identification, the other three strategies have to either use multicast or implement a more sophisticated tuple retrieval mechanism as discussed for the respective implementations. There are two variations of tuple hashing: *Cryptographic Hashing*, *Random Hashing*, or *Avalanche Effect Hashing* produce a balanced distribution throughout the network and are therefore beneficial for systems that experience a high rate of random tuple reads. The alternative of *Similarity Preserving Hashing* algorithms are a powerful strategy to ensure that tuples with similar characteristics reside on the same node. If an application is structured in such a way that computations are focussed on closely related tuples, similarity preserving hashing is an efficient means to ensure locality and thus circumvent the access penalties in high-latency system topologies.

Motivation The main driving factor for the tuple space implementation as emphasized in the original paper and the envisioned use cases. *Fault Tolerance* for tuple spaces that explicitly replicate data to tolerate node failures; *HPC* for tuple spaces that are designed for applications in high performance computing; *Scalability* for tuple spaces that are designed to improve scalability regarding their respective target architecture. The investigated systems achieve this by either distributing the tuple space across the nodes or by explicitly distinguishing between multiple tuple spaces and requiring programmers to address them correctly.

1.1.1. Coordination Language Implementations

A prominent representative of the Tuple Space landscape is JavaSpaces [7], where tuple space concepts were applied to objects, coining the term Object Spaces, and the interface was enriched with the concept of transactions. It was integrated with Sun Jini, which is now named Apache River. GigaSpaces XAP [8] is a commercialized version of JavaSpaces that offers a distributed object space with tuple redundancy that supports different placement strategies including hash-based tuple distribution. Tupleware [9] is an implementation aimed at computationally intensive applications running in a cluster. It includes a decentralized search algorithm where each node asks other nodes one by one based on a success factor of previous searches. DTuples [10] uses a distributed hash table for tuple placement and retrieval. Each tuple has to begin with a name which is then used for the hashing, resembling a key-value-store. MTS-Linda [11] was of the earliest attempts using multiple tuple spaces. It uses a tuple-to-node hash for placement.

There have been different attempts to scale the tuple space model to what we would today call a cloud architecture. PageSpace [12] is using a tuple space to coordinate distributed applications on the web. Rowstron et al. extended this notion first with C2AS [13], adding multiple tuple spaces, later with WCL [14] where a ”control system” that monitors tuple space usage and migrates tuple spaces to the right location was added. Analogous to Linda, C2AS and WCL can be embedded in any host language. Jada (Java+Linda) [15] similarly implements multiple distributed but disjoint tuple spaces for Java. GridTS [16] uses a replicated tuple space for fault-tolerant scheduling.

In addition to these cloud-scale implementations still relying on the programmer to specify which tuple space (and thus which node) she wants to access, there are two interesting implementations providing one distributed, or transparently shared tuple space. SwarmLinda [17] is an attempt to transfer the ideas from the field of swarm intelligence to distributed tuple spaces. Natural multi-agent systems – such as ant colonies – show intelligent behavior, while coordinating only with local interactions in their neighborhood. This transfer results in an adaptive system that can react to changes in topology and is highly scalable while retaining some locality of similar tuples. LIME: Linda in a mobile environment [18] implements “transiently shared tuple spaces” that span multiple physical ones. The disconnection of nodes is viewed as a normal operation which results in the tuples on that node being removed from the transiently shared tuple space. The placement strategy defaults to local but can also be specified.

CnC [19] is a coordination model strongly influenced by Linda, which further allows to declaratively express data and control dependences between computational steps.

We miss two things in the systems described above, which we will describe in greater detail in section 1.3. Firstly, we want to investigate the implementation of tuple spaces at the two ends of the hierarchy: NUMA-awareness and federated clouds. Secondly, we want to be guided by the minimal set of information from a programmer needed to achieve good performance, sticking closely to modern programming models.

1.2. Hierarchical NUMA systems

Additional cores in modern business servers are either introduced by increasing the amount of cores per processor or by adding additional processors. In both cases all cores need to have access to other cores, processors and memory. These interconnects constitute the von-Neumann bottleneck, and have become one of the most crucial recent performance design challenges. Modern processor architectures, such as Intel's Haswell processors, facilitate an on-chip ring interconnection network with two opposing rings to connect cores, memory controller and processor interconnect. [20] One level higher, the reference architecture for processor interconnects provided by the processor vendors is usually a point-to-point interconnect between all processor sockets that is designed to support systems up to a certain size. (Intel for example supports up to eight processor sockets.) These systems are called glue-less systems, because the processors and the interconnection technology are provided by the same vendor. The alternative are glued systems, where third party interconnection technologies are used to build systems that support more sockets than the reference architecture. [2,21,22] In addition to the increased processor count, glued systems usually also facilitate special caching and pre-fetching solutions to compensate for the latencies and improve the overall system performance. Besides all-to-all interconnects between the processor sockets, glued architectures can realize various other popular topologies such as hypercubes and cross-bar switches.

Hierarchical NUMA systems combining multiple layers of interconnect technologies are programmed using a combination of message passing (usually with MPI [3]) and shared memory task parallelism (usually with OpenMP [4]). The application of both programming models allows programmers to account for the distributed as well as the parallel nature of hierarchical NUMA systems. Shared memory task parallelism is used on the intra-processor level where performance bottlenecks are often introduced by task and data access synchronization. Due to the significant latencies on the inter-processor level, considering the system a fully distributed one and using the message passing programming model for task and data coordination excels. The message passing model requires developers to structure their algorithm in a way that allows for the computation of independent tasks on local data and the explicit exchange of data updates via messages. The application of local data access and data duplication in form of messages reduces the load on the interconnect, while the explicit distribution ensures that the characteristics of the interconnect can be respected by the message passing framework implementation.

While allowing to achieve close to optimal application performance, the current approach restricts productivity due to the fact that programmers need to develop a detailed understanding of two programming models and their complex interplay with the systems hardware. Learning from both approaches, we designed Claud to encapsulate the best practices for parallel and distributed models into a layer that allows programmers to reuse them, while simplifying the programming model to improve productivity without sacrificing much of the performance.

1.3. Research Gap

When we set out to evaluate the design space for programming models that would allow us to coordinate task and data from the core of parallel systems up to distributed cloud federations, we assessed possible approaches based on the following question: What is the minimal set of constructs, that we need the programmer to use to express the algorithm in a way that allows for correct and efficient execution? We found tuple spaces to be a promising answer to that question. No only do they provide a very concise interface, their suitability has also been proven for both, the parallel and the distributed domain.

To apply tuple space concepts to our objective of a coherent performance framework for the whole system topology we identified some research gaps to be filled (Figure 1):

While existing tuple space implementations have a strong focus on scalability and fault tolerance, we want to evaluate how best practices for performance can be incorporated into a framework to make them reusable. Since different problems demand different optimization strategies, we want to start with a limited subset and extend our framework iteratively to support techniques for additional problem classes. The problem class of graph based algorithms possesses inherent locality characteristics which is why it is particularly well suited for a mapping onto the hierarchical system topology of modern computer systems. In this paper we describe how Claud can support developers with the coordination of tasks and data of graph problems.

Existing tuple space implementations mastered parallel shared memory systems, clusters and cloud systems. While they are probably also very well suited for both, hierarchical NUMA systems (Section 1.2) and federated clouds [23], we find that it is important to evaluate this with a number of real-world examples. We intend to study the potential and possible opportunities for improvement in the SSICLOPS [23] project.

Finally, we want to identify a minimal set of programming constructs that is required to allow a coordination framework like Claud to perform efficient data placement and task scheduling. We hope to identify programming constructs that allow programmers to express parallelism and locality without understanding the target topology in detail, while enabling us to achieve portable performance with adaptive mappings.

2. Approach

The objective of Claud is to allow task and data coordination throughout the whole system topology (from core to cloud federation) with a single coherent programming model. The overhead that the programming model imposes should to be low enough to allow productive development while allowing Claud to enable acceptable performance.

2.1. Assumptions and Design Decisions

We assume that programmers do not know the hardware topology upfront, and that the characteristics of the topology change during runtime. These assumptions are based on the fact that there is a huge variety of possible topologies and the fact that the characteristics of the topology are load dependent. Since the programmer does not know the topology, she cannot specify the data placement and task distribution statically beforehand. Since the topology changes, Claud needs to adapt to the current system characteristics by using auto-balancing and fault-tolerance techniques to provide portable performance.

We assume that programmers are more productive, if they do not need to explicitly specify coordination and dependence. Consequently, coordination should be inferred from the system topology and the logical representation of the algorithm and its data structures to make coordination as implicit as possible.

We assume that besides homogeneous tasks of similar compute intensity, there are also task sets containing a mix of compute intensive and light-weight tasks. Furthermore, we assume that programmers do not want to specify compute intensity of tasks. In contrast to the common approach of either distributing all tasks or none, we want to integrate heuristics that account for varying task profiles. The heuristics will be based on the computation-to-communication ratio of a task, resulting in appropriate task coordination. One extreme coordination decision would be to execute a set of tasks serially on one processor if the computer is very cheap and the distribution would be relative expensive.

From our experience we assume that programmers have a better intuition for the *read* and *write* programming constructs than the notion of *in* and *out* as proposed by Linda. Therefore, we provide primitives such as barriers and NUMA-aware reader-writer locks for data access synchronization, allowing programmers to see Claud as a distributed shared memory framework mimicking a familiar execution environment.

We assume that latency and bandwidth limitations are the predominant bottlenecks, and that as a consequence locality is beneficial to performance. Preserving locality means that a task working on data should executed at the subgraph of the topology (preferably on the same node) where the respective memory is located.

Since the layers of the topology have differing characteristics, we do not assume that an optimal distribution can be derived simply based on the programmer's input. Instead we want to incorporate multiple distribution modules in Claud as described in Section 2.2. These modules allow us to add constructs to Claud's interface, presenting a concise and familiar interface to programmers while providing Claud with all the information that is required for effective coordination. When evaluating Claud with real-life applications, we will iteratively integrate additional distribution modules.

2.2. Architecture and Implementation

Figure 2 shows the interfaces that Claud provides to algorithm programmers as well as the associated techniques used for the coordination of task and data on the present system. As a basis for the coordination of data and tasks, Claud comprises an extensible set of Distribution Modules. Each module may imply an extension of the interface for the programmer, as well as, a more sophisticated mapping strategy to the topology. Distribution Modules have to be designed so that they can be used in unison with the other modules or so that a module is explicitly overwriting the policies of another module. As an example, the Concurrent Data Structures Module (number III in the picture) may overwrite the policies of the Hierarchical Task Queues Module (number I), but can still be improved by the Run-Time-Analysis Module (number II).

Distributed Module I Object/Data access is presented using the usual shared memory metaphor: memory can be read and written. Write accesses automatically allocate or update data in the Object Space, read accesses retrieve data from the Object Space. The mental model of the Object Space that the programmer can use is similar to a Key-Value Store. As our objective is to support all topology levels, we need to provide a software-managed cache for distributed levels, that integrates with the hardware caches on the parallel levels. Since reading will basically result in a local copy of the data, coherency needs to be guaranteed by invalidating all copies if data is written. The coherency requirements of the algorithm can be enforced by either by implicit synchronization barriers or by explicit synchronization with synchronization primitives like Reader/Writer Locks. Internally we have a hierarchical structure that keeps track of the data locations

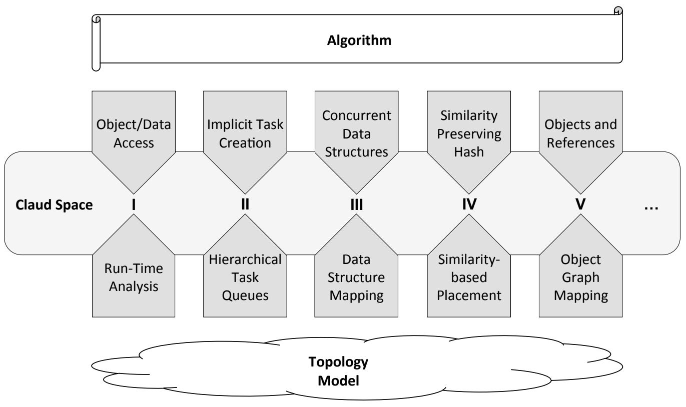

Figure 2. Architecture Overview: Claud acts as a mediator between the logical domain of the programmer and the topology model derived from the current system configuration. Currently Claud comprises five Distribution Models, each providing a distinct set of programming constructs and extended mechanisms for coordination.

and is closely modeled after the system topology allowing us to assess the distance to the data. We employ Run-Time Analysis methods to assess data access patterns, which allows us to prefetch data and migrate tuple responsibility based on auto-tuned heuristics.

Distributed Module II Claud offers several ways to create tasks implicitly such as parallel loops and recursive task creation. [4] If no other module provides a more intelligent algorithm for the task distribution, we utilize hierarchical task queues and work-stealing. [4] Each core in the topology will have its own local queue and can also access an additional queue that it shares with its neighbors. Hierarchical task queues with work stealing provide a pretty good distribution scheme for average algorithms, but can easily be outperformed by modules exploiting additional information about the data and the algorithm like III and IV.

Distribution Module III Another module provides Concurrent Data Structures. If programmers use our arrays, lists, trees, etc. they provide us with insight about the way their data is supposed to be structured and accessed. From this information, Claud can infer the inherent notion of locality and distribute data and tasks accordingly. As described before this module can benefit from other modules like the run-time analysis. Furthermore, if the tasks have varying complexities, work stealing can help to balance the work.

Distribution Module IV Similarity (or Bonding-) preserving multi-dimensional hashing allows the programmer to provide a similarity measure for the data in form of a multi-dimensional vector. In a two dimensional index space (think matrix-matrix multiplication) this could be a vector describing the horizontal and vertical coordinate of the cell. Based on these vectors, Claud can determine the similarity of tuples and put similar tuples in the same place or closely together.

Distribution Module V Another way to gather information about the data structures and presumed access patterns is by looking at the object graph. If this module is used, we map the graph that is accessible from the current context (e.g. loop body) onto the topology at the beginning of each code block and transfer data and tasks if necessary. As with all the other modules, this can improve the performance significantly or produce an additional overhead. Consequently, using heuristics and auto-tuning to find the right balance between the modules is essential.

2.3. Restrictions

In large scale scenarios such as federated clouds, some traditional operating system functionalities become increasingly challenging to provide. Hence, cloud providers usually implement such features as a part of their Infrastructure as a Service offer. The same restrictions apply to Claud: an application being executed in a distributed fashion using Claud is expected not to work with its own operating system handles. This means: no access to file handles, socket, I/O, operating system synchronization primitives, etc. To compensate for this, Claud provides its own synchronization primitives such as barriers and NUMA-aware reader-writer locks. The set of features is currently restricted, but will be extended to meet the requirements of further use cases.

3. Conclusion

We have shown that the modular design of Claud is a suitable approach to coordinate complex graph problems on hierachical system topologies. We have demonstrated several coordination techniques that are integrated into our hierarchical object space, which maximize locality and thus minimize latency penalties. We found that additional programming constructs to realize such a coordination are not only concise, but can also be tailored to fit the expectations of the programmer.

In the future, we want to implement all presented ideas and thoroughly evaluate Claud on hierarchical NUMA systems and in federated clouds. To demonstrate the applicability of the approach to real-world scenarios, we plan to evaluate it in the context of the SSICLOPS [23] and the sHiFT [24] project. The *Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS)* [23] project tackles the challenge of managing federated private cloud infrastructures. We are particularly interested in is the aspect of workload scheduling: In cloud systems, data can be scattered across different datacenters. Since even modern wide area datacenter interconnections such as rented dark fibers or the public internet come with severely constrained connectivity compared to intra-datacenter connectivity, ignoring the lack of locality results in degraded performance and is not an economic option. Therefore, either data needs to move close to the processing resources or vice versa. At a much lower level on the intra-system scale, the same issues apply to modern NUMA architectures, where remote memory access caused by improper workload placement limits performance. Hence, our goal for Claud is to enable developers to benefit from versatile workload placement strategies on various scales ranging from groups of CPU cores to entire datacenter federations. The sHiFT [24] project aims at creating a high performance framework for business process analysis. Business processes can be represented as graphs annotated with natural language artifacts. Various algorithms work with these graphs to extract business information: process matching, reference model and process mining, identification of isomorphic subgraphs, and natural language processing. Most of these algorithms can be parallelized, making the mapping of the process graphs to the system topology and the efficient coordination of computations the core performance challenges of the project. We see this project as an opportunity to identify potential for further improvements in Claud's capabilities.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] J. Costa-Requena, M. Kimmerlin, F. Eberhardt, M. Plauth, A. Polze, S. Klauck, and M. Uflacker, "Use-case scenarios for the evaluation of federated private clouds," Scalable and Secure Infrastructures for Cloud Operations, Tech. Rep., 2015, to be published.

[2] Silicon Graphics International Corp., "Technical Advances in the SGI© UV™Architecture," Tech. Rep., June 2012.

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1," June 2015.

[4] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0," July 2013.

[5] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[6] ——, *Mirror worlds: Or the day software puts the universe in a shoebox... How it will happen and what it will mean*. Oxford University Press, 1992.

[7] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[8] Gigaspaces. (2015, Jun.) Gigaspaces xap. [Online]. Available: http://www.gigaspaces.com/xap

[9] A. Atkinson, "Tupleware: A distributed tuple space for cluster computing," *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, pp. 121–126, 2008.

[10] Y. Jiang, G. Xue, Z. Jia, and J. You, "DTuples: A distributed hash table based tuple space service for distributed coordination," *Proceedings - Fifth International Conference on Grid and Cooperative Computing, GCC 2006*, pp. 101–106, 2006.

[11] B. Nielsen and T. Sørensen, "Distributed Programming with Multiple Tuple Space Linda," 1994.

[12] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali, "PageSpace: An architecture to coordinate distributed applications on the Web," *Computer Networks and ISDN Systems*, vol. 28, 1996.

[13] A. Rowstron, S. Li, and R. Stefanova, "C2AS: a system supporting distributed Web applications composed of collaborating agents," *Journal of Engineering and Applied Science*, pp. 127–132, 1997.

[14] A. Rowstron, "WCL: A co-ordination language for geographically distributed agents," *World Wide Web*, vol. 1, no. 3, pp. 167–179–179, 1998.

[15] P. Ciancarini and D. Rossi, "Jada: Coordination and Communication for Java Agents," *Mobile Object Systems Towards the Programmable Internet*, vol. 1222, pp. 213–228, 1997.

[16] F. Favarim, J. Fraga, L. C. Lung, M. Correia, and J. a. F. Santos, "GridTS: Tuple spaces to support fault tolerant scheduling on computational grids," pp. 1–24, 2006.

[17] R. Tolksdorf and R. Tolksdorf, "A New Approach to Scalable Linda-systems Based on Swarms," *Computing*, no. March, pp. 375–379, 2003.

[18] G. Picco, a.L. Murphy, and G.-C. Roman, "LIME: Linda meets mobility," *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pp. 368–377, 1999.

[19] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, 2010.

[20] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik *et al.*, "Haswell: The fourth-generation Intel core processor," *IEEE Micro*, no. 2, 2014.

[21] Hewlett-Packard Development Company, L.P, "HP Integrity Superdome 2 - The ultimate mission-critical platform," Tech. Rep., July 2013.

[22] T. P. Morgan, "Balancing Scale And Simplicity In Shared Memory Systems," *http://www.theplatform.net*, March 2015.

[23] Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS) Project. [Online]. Available: https://ssiclops.eu/

[24] DFKI GmbH, Hasso-Plattner-Institut and Software AG. sHiFT Project. In German, application to a bidding, unpublished. [Online]. Available: http://www.bmbf.de/foerderungen/26683.php