International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 1, pages 37–54, January 2013

Experimental Evaluation of a Hierarchical Chemical Computing Platform

Marko Obrovac and Cédric Tedeschi
IRISA / Université de Rennes I / INRIA
Rennes, France

Received: July 3, 2012
Revised: October 24, 2012
Accepted: December 6, 2012
Communicated by Akihiro Fujiwara

Abstract

Chemical computing was initially proposed as a paradigm capturing the essence of parallel programs. Within such a model, a program is envisioned as a solution of information-carrying molecules, that, at run time, collide non-deterministically to produce new information. Such a paradigm allowed the programmers to focus on the logic of the problem to be solved in parallel, without having to worry about the implementation's considerations. Throughout the years, the model has been enriched with various features related to program structure, control and practicability. More importantly, the model has recently been raised to the higher order, increasing again its expressiveness. With the rise of service-oriented computing, such models have recently regained a lot of interest. They have been shown to provide adequate abstractions to enhance service-oriented architectures with autonomic properties such as self-adaptation, self-healing, or self-organisation.

However, the deployment of chemical programs over large-scale, distributed platforms is still a widely open problem, hindering the model to be leveraged in practice. This paper studies the possibility of building a distributed execution environment for chemical programs, to which end different approaches are discussed. Firstly, the paper envisions such a platform based on the distributed shared memory model to implement this *solution.* Such an approach leads to several issues that prevented us to finalise this approach and led to the exploration of a message-passing-based solution. Thus, secondly, and more importantly, a generic peer-to-peer-based runtime model is proposed.

To complete this study, a software prototype of the second approach was developed and experimented over the Grid'5000 test-bed. Experimental performance results are detailed, allowing for a discussion of the feasibility and performance of such a runtime, paving the way for future works, and lifting a barrier towards the enactment of the chemical programming model.

Keywords: Chemical Programming Model, Distributed Systems, Distributed Shared Memory, Peer-to-peer, Distributed Runtime, Experimental Study

1 Introduction

In 1993, Banâtre and Le Métayer took the opposite position to the at-the-time commonly admitted idea that introducing parallelism in programs was a difficult task. They argued that parallelism was "a powerful structuring facility that could profitably be exploited in program construction" [7]. This claim was precisely at the origin of the development of chemistry-inspired programming models [4]. In such a model, a computation is seen as a succession of chemical reactions in a solution of information-carrying molecules moving and colliding according to Brownian motion. On collisions, and according to some predefined rules (constituting the chemical program), molecules are consumed and new molecules carrying new information are created. Reactions take place locally, autonomously, and non-deterministically, implicitly modelling parallelism and distribution. In other words, such a model is cleared of any artificial sequentiality brought by the process of implementation, and allows programmers to focus on the logic of the computation without having to worry about implementation constraints imposed by parallel architectures.

More formally, as initially proposed by the GAMMA formalism [7], the solution is represented by a multiset of data on which a set of rewriting rules — constituting the chemical program — is applied. Reaction rules are of the form **replace** P **by** M **if** V. They involve a set of molecules N satisfying the pattern P and the reaction condition V. The reaction consumes N, and produces a new set of molecules M. The rules are applied on molecules until no more reaction is possible — a state referred to as *inertia*. Note that, nothing is said by the programming model, about two rules that can be triggered in parallel; the execution is non-deterministic. However, the result of the computation remains deterministic. The execution model only ensures the independence of reactions by the atomic capture of the reactants needed in them.

1.1 Chemical Programming in Context

Throughout the years, the expressiveness of the chemical model has been exploited in different contexts, for instance to verify shared-memory coherence protocols [25], to process large images [24], or to specify software architectures [16, 26]. The model has recently been raised to the higher order, giving birth to HOCL, a highly expressive chemical language [5]. In HOCL, everything is a molecule, including reaction rules, that can themselves be consumed or produced upon a reaction. In other words, the program itself can be modified at runtime. The higher order offers a clean way to express self-adaptation and autonomous behaviours, as such programs are able to self-modify at run time (rules can appear and disappear dynamically). This feature makes the model highly adequate in today's emerging platforms, in which we need to coordinate collections of loosely-coupled services dynamically combined on top of unreliable platforms: services need to self-organise, self-heal, and self-adapt [32]. In the same vein, different works have been conducted to develop intuitive abstractions using this model for self-coordination in such platforms [6, 11, 27].

Let us notice here that this regain of interest for chemical models is part of a bigger phenomenon which sees the renaissance of declarative programming models [22] in the quest for adequate programming abstractions able to tackle the scale and complexity of today's computing platforms. Declarative programming encompasses any programming approach which separates the logic of a computation from its control, allowing the programmer to concentrate on the former while it is assumed the runtime will take care of the latter.

Functional programming shares some similarities with chemical programming. To illustrate them, let us focus on three fundamental features of chemical programming, namely: (i) the immutability of variables (molecules), (ii) the implicit parallelism of instructions (reactions), and (iii) dynamic code rewriting (higher-order). The immutability of objects has been explored in order to tackle the safety issues of concurrent programming. Indeed, if an object's state cannot be changed, it cannot be corrupted or observed in an inconsistent state in the case of its concurrent access. This feature can be found in many languages, and in particular in functional languages such as Scala. The implicit parallelism feature can also be found in several functional languages which implement the concept of *actors*, which can be found for instance in Scala or Haskell. Finally, the possibility to manipulate functions (or rules) as basic types can also be encountered in many common languages such as Java, Python, or Scala and Clojure, on the functional side.

Closer to our work, some recent works have advocated the use of rule-based programming for the specification of distributed systems. For instance, in [13], it has been shown how rule-based languages can be used to specify communication protocols and peer-to-peer applications. In [3], the same idea is applied to web-based data management. On the computing side, rule-based programming was also used as a building block for workflow management systems [33, 18].

1.2 An Example

Let us illustrate this feature with a simple HOCL program that extracts the maximum even number from a set of integers.

```
let $selectEvens =$ replace $x, \omega$ by $\omega$ if $x\%2 \; != 0$
in
let $getMax =$ replace $x, y$ by $x$ if $x \geq y$
in
⟨
    ⟨ $selectEvens, 2, 3, 5, 6, 8, 9$ ⟩,
    replace-one ⟨ $selectEvens = s, \omega$ ⟩ by $getMax, \ \omega$
⟩
```

The $getMax$ rule compares the values of two integers and leaves the greater of the two in the solution, while *selectEvens* removes odd numbers from the solution, by repeated reactions with an integer x, ω denoting the whole solution in which *selectEvens* floats but deprived of x. The solution is composed of (i) a sub-solution containing the input integers along with the *selectEvens* rule, and (ii) a higher-order rule that will *open* the sub-solution, extract the remaining (even) numbers and introduce the $getMax$ rule. Solving the problem requires the sequentiality of the reactions of the two rules. In HOCL, a sub-solution can react with other elements only once it has reached inertia. The higher-order rule will react with the sub-solution only after containing solely even numbers. The result is then as follows :

```
⟨
    ⟨ $selectEvens, 2, 6, 8$ ⟩,
    replace-one ⟨ $selectEvens = s, \omega$⟩ by $getMax, \ \omega$
⟩
```

Then, the higher-order rule reacts, extracting the remaining numbers and introducing the $getMax$ rule. The resulting solution is $\langle 2, 6, 8, getMax \rangle$. This triggers the second phase of the program where only the maximum value (here, 8) is kept. Note that putting both rules directly in the solution of integers could entail a wrong behaviour as the pipeline between the two rules would be broken. For instance, if $getMax$ reacts first with molecules 8 and 9, then 8 is deleted.

1.3 Motivation

Despite the high demand for its properties, the chemical model suffers from a significant lack of means of execution over large platforms and yet executing chemical programs is crucial on the road to its largely-suggested adoption in today's context. Without a *distributed chemical machine* able to execute the coordination, previously mentioned works remain mostly conceptual. To introduce the philosophy of chemical programming, Banâtre and Le Métayer distinguished *logical* parallelism from *physical* parallelism [7] :

> "Physical parallelism is related to the implementation; it corresponds to the distribution of tasks on several processors. By logical parallelism, we mean the possibility of describing a program as a composition of several independent tasks. Of course, a particular implementation can turn logical parallelism into physical parallelism, but these notions have very different natures: the former is a program-structuring tool, whereas the latter is an implementation technique."

It appears that this distinction also holds for distribution : in the chemical runtime model, reactions are — implicitly — local and autonomous, *i.e.*, they take place independently from each other. Time has come to offer an implementation to the chemical model, in other words, to turn its logical parallelism and distribution into physical ones, for large-scale platforms.

Issues of a Distributed Runtime. The execution of a chemical program carries three main problems: (i) finding reactants, *i.e.*, retrieving combinations of molecules that satisfy some reaction rule's pattern and condition, (ii) performing reactions, *i.e.*, deleting the molecules found in step (i), producing the result of the reaction, and inserting it in the solution; and (iii) detecting inertia — the state in which no more reactions are possible. An execution machine performing these steps exists for a single processor [29]. However, when dealing with their parallel and distributed implementation, these three steps raise new problems, especially the third one : how to efficiently detect that molecules can no longer react when they are distributed at large scale ? As we detail in Section 2, the few isolated attempts at tackling the problem had a limited impact, for they focused on some particular topologies or cases. Peer-to-peer (P2P) systems, that were until now unexplored in this context, pave the way to new solutions with properties such as *scalability*, but also *transparency* and *platform independence*.

Contribution. This work is a feasibility and performance study of a large scale runtime for the chemical model. The paper describes the different tracks explored to build such a runtime. Firstly, an intuitive approach to extend the existing centralised runtime, by leveraging the *distributed shared memory* model, is discussed. Based on the discussion of the limitations inherent to such an approach, a second framework is built atop a P2P communication layer and relies on distributed algorithms for the execution of programs. The proposed algorithms are shown to present an optimal distribution of inertia detection : there is no computational overhead when compared to a centralised version. To fulfil the practical requirements of such a study, a software prototype has been developed, based on the FreePastry P2P overlay network [2]. Its experimental campaign conducted on the Grid'5000 platform [8] is detailed, showing the viability of the concept, and constituting a first achievement towards large scale chemical computing in practice.

Organisation of the Paper. The next section explores existing works around the parallel execution of chemical programs. Section 3 discusses the first attempt at modelling a distributed chemical environment relying on a distributed shared memory system. Section 4 presents the P2P framework and the algorithms while Section 5 discusses our prototype and the results of the experiments conducted. Section 6 concludes.

2 Physical Parallelism of the Chemical Programming Model

We here briefly review the previous efforts in building runtimes for the chemical programming model. The pioneering work of Banâtre *et al.* [4] provides two implementation methods, the basic idea of which is a parallel machine. Each processor holds a molecule and compares it with the molecules of all the other processors. Two algorithms are proposed: (a) a synchronous one, where a centralised controller triggers each comparison step, and (b) an asynchronous one, in which the molecules travel along a vector of processors, either until they react, or until they have returned to their starting point. This last algorithm was implemented on top of an iPSC hypercube with 16 processors. In the work by Linpeng *et al.* [15], a program is executed on MasPar MP1, a massively parallel machine, using the fold-over operation. The molecules are placed on a strip and folded over after each vertical comparison. At each step, the elements in the upper segment of the strip are compared in parallel to those in the lower segment. Recently, Lin *et al.* have shown that GAMMA can be used to define programs executable on a cluster exploiting GPU computing power [21]. Although these works present significant speed-ups, they only target specific platforms, and cannot be exploited in large-scale, heterogeneous platforms. Moreover, the speed-ups they depict are achieved by testing programs containing exclusively *reducer* rules, *i.e.*, rules that produce less molecules than they consume, which uniformly reduces the complexity of a problem since the number of molecules in the solution decreases after each reaction.

More importantly, these works regard GAMMA as a specification language : a program's behaviour is only described using GAMMA and then implemented in another language in order to be executed. This paper, however, presents a distributed execution machine which would turn

GAMMA and its descendants from specification languages into implementation languages suitable for distributed execution.

3 A DSM-based Chemical Platform

One important part when coming to implement an HOCL runtime is the *scheduler*. The scheduler is the entity which decides which rule should be applied and when. It is responsible for the actual implementation of the non-deterministic execution model. One possible strategy is *round-robin*: every rule is triggered once, each on its turn, until inertia. The centralized runtime of HOCL presented in [29] deals with this relying on several lists: one containing all molecules of the solution, and one containing the rules. Thus, the scheduler is greatly dependent on the management of these lists. We started our study by trying to keep these original ideas while trying to distribute the process. About the solution, this leads to two contradictory objectives:

1. keep the multiset in an easy-to-access, shared location; and

2. distribute the multiset for the sake of load balancing and bottleneck avoidance

These two objectives seem natural: the first enables the system to quickly search for reactants and access them. The second one is a performance requirement: a shared memory has intrinsic drawbacks regarding performance, as a single memory controller is accessed concurrently by a set of processors. These seemingly contradictory goals have already been pursued together in Distributed Shared Memory (DSM) systems [28], in which programs see a set of distributed memory slots as a single virtual entity to be accessed uniformly.

In the remainder of the section, we are exploring the possibility of using the DSM paradigm to build a distributed runtime for the chemical programming model. The paradigm allows one to conceive a distributed platform while *thinking sequentially*, since the DSM implicitly handles concurrency. In this sense, the DSM paradigm shares some similarities with the chemical programming model, which itself relieves the programmer of handling mutual exclusion, making DSM a natural model to start the study with.

3.1 DSM-inspired Architecture Overview

The DSM model provides a logical abstraction of the shared-memory model atop a message-passing distributed system. It combines the best features of the centralised and distributed worlds: each node is equipped with its own local memory, but all these slots are virtually assembled to provide a unique global memory accessed uniformly by programmers. Even though originally oriented towards homogeneous clusters of computers, the model has been adapted for various types of systems and is considered nowadays for heterogeneous and large-scale platforms [12, 19, 34]. Thus, using such a model enables us to abstract out the physical network and lets us tackle the problem at hand — executing a chemical application using multiple processors.

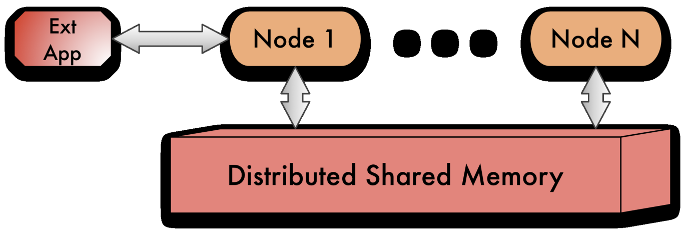

Figure 1: Conceptual view of the DSM-based platform.

The conceptual, logical view of a distributed runtime based on the DSM model is shown in Figure 1. The external application represents any entity which requests the execution of a chemical program. It contacts a node in the DSM-enabled network and transfers it the program to execute. The contacted node will, once the execution finishes, transfers the resulting inert solution to the application.

Every node involved in the execution is equipped with a chemical engine able to apply a reaction rule on a combination of molecules. Therefore, each node participates in the computation by selecting a combination of molecules, checking it against the rule's reaction condition, and, finally, performing the reaction if the condition is positively evaluated. This process is repeated until inertia, which concludes the computation. However, a logical global space where to access the molecules is not enough to build a chemical runtime. A *scheduler* is needed. We must ensure that each possible combination of molecules is checked once (and only once) against the rules. As we detail in the following, this can be achieved through a list of possible reactant combinations, shared between all nodes involved in the computation.

3.2 Course of Execution

The execution of a program involves four main elements described in the following.

Initialisation. After the transfer of the program to be executed has been completed, the contacted node creates two objects in the DSM : a list containing the rules to be executed and the list of molecules present in the program. As soon as other nodes detect the presence of the list of rules in the system, they each copy it locally and start the execution phase. For the sake of simplicity, let us assume that only one rule consuming two molecules is present in the program. However, the algorithm described is also valid for multiple rules, each with a varying number of arguments.

Combination List Filling. The nodes start off by filling the *combination list* — a list containing combinations of molecules which yet have to be tested against the rule's reaction condition. We now briefly describe how the nodes can do it in a cooperative way : each node sequentially picks a molecule from the molecule list and creates a sub-list of combinations involving the picked molecule and puts it in the combination list. As an illustration, suppose a program contains five molecules, $m_1, \ldots, m_5$, and is executed on a three-node system. Then, node 1 will put in the combination list the set of combinations containing the molecule m_1 $(\langle m_1, m_2\rangle; \ldots; \langle m_1, m_5\rangle)$, node 2 will put the combinations with the molecule m_2 but will avoid the combination with m_1 to avoid double insertions $(\langle m_2, m_3\rangle; \langle m_2, m_4\rangle; \langle m_2, m_5\rangle)$. In the same fashion, node 3 will add $(\langle m_3, m_4\rangle; \langle m_3, m_5\rangle)$ to the list.

Condition Checking and Inertia Detection. Once the combination list is in place, each node accesses it, taking the next untested combination of molecules and checking it against the rule's reaction condition. If the molecules cannot react, their combination is removed from the list. Otherwise, the node locks the molecule list and tries to take the molecules from it. If not all of the molecules are available, the combination is deleted from the list and the node simply moves on to the next one. This process is repeated until the combination list has been emptied. When a node notices the list is empty, it repeats the combination list filling process. Inertia has been reached once there are no more combinations to check, *i.e.* when :

1. there are no more elements in the combination list; and

2. the molecule list has been exhausted — there are no more molecules for which combinations can be generated.

The execution is considered to be completed when the state of inertia has been reached. At that point, the node initially contacted by the external application transfers it the inert solution.

Reaction Execution. If, after locking the molecule list, the node is able to obtain all of the molecules it needs to perform a reaction, it removes them from the molecule list and consumes them in the reaction. Then, it removes all of the combinations containing either of the consumed molecules from the combination list. The molecule list is then locked once again. The node fills the combination list with combinations containing the newly created molecules coupled with all of the molecules present in the molecule list, after which the new molecules themselves are added to the molecule list.

3.3 Issues of the DSM-based Platform

While a DSM-based approach seems a natural way to implement a distributed chemical platform, it suffers from several issues.

Lists Management. Primarily, the platform relies on the repeated usage of locks over shared objects, not only the molecules, but also the lists used in the computation. The more nodes there are in the system, the more time each of them is likely to spend in a lock's wait queue, thus increasing their idle times and reducing their work times. This is due to the fact that concurrently manipulating a list while keeping its state consistent is inherently a difficult task. Then, the two lists, or *containers*, are needed to coordinate the nodes and keep the system in a consistent state ; locking the list of molecules prevents two nodes from using the same molecule in concurrent reactions, while the list of combinations serves as a *reminder* to nodes as to which combinations yet have to be checked, in this way preventing nodes to check already checked combinations. The use of lists stems from the DSM model's centralised access to memory — each node has to know *a priori* the memory location (or the object id) it will manipulate. Moreover, each list resides in the memory of one particular node, which entails network traffic penalties during locking, as it has to be transferred from one node to the other.

DSMs in Practice. Finally, on the practical side, implementing such a system would not be an easy task due to the fact that, in spite of the volume of research that has been conducted in the DSM area, there exist only a few DSM implementations, which are tied to specific platforms [9, 17, 20].

In conclusion, while at first glance using a shared-memory approach seems a natural track to be pursued, it suffers from severe drawbacks. Consequently, we shifted our focus towards distributed memory models. The next section describes a distributed runtime for executing chemical programs with the minimal requirement of message-passing facilities.

4 Hierarchical Chemical Computing Platform

The proposed platform is illustrated in Figure 2. The platform can be seen as a service: the applications submit their chemical programs to be executed and await their respective resulting inert solutions. Note that multiple applications can be processed concurrently by the service. Internally, the platform is composed of three layers detailed in the remainder of the section: (i) the overlay network, (ii) the execution engine and (iii) the inertia detection mechanism. The overlay network connects the participating nodes and allows the molecules to be spread around the system. The execution engine takes them over and locally performs reactions until the inertia detection mechanism perceives the local solution has reached a stable state, where no more molecules can react.

We first explain the role of the underlying overlay network. Next, the functioning of the chemical engine is clarified by detailing how molecules are distributed and the computation initiated. Finally, we focus on the distributed mechanisms required to find reactants and detect inertia. In particular, we show, by providing a first *brute-force* algorithm, that sub-optimality in terms of number of reaction tests is easily encountered if the algorithm is not designed properly. An algorithm exhibiting optimality in this regard is then proposed. This optimality will be experimentally illustrated in Section 5.

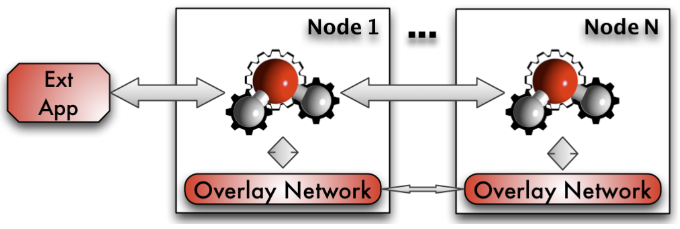

Figure 2: The hierarchical platform.

4.1 Physical Layer Abstraction

As we mentioned earlier, the scale and the heterogeneity of targeted platforms should be abstracted out, before going further in the design of a distributed chemical machine. This requirement is addressed by relying on overlay networking, which build logical virtually homogeneous networks on top of heterogeneous platforms. One good choice to build an overlay are distributed hash tables (DHTs) [1, 30] in the sense that they partially solve the scalability issue as nodes are guaranteed to communicate efficiently regardless of their number. A DHT's routing complexity typically grows logarithmically with the number of nodes in the platform. Another advantage of DHTs is load-balancing. In particular, the external application sending its program to the chemical runtime platform can contact any of the DHT nodes acting as an entry point to the platform. Then, natural load balancing is obtained as each application can choose a different contact node (through an out-of-band mechanism).

Any DHT could fill this role. In the following, we use the Pastry DHT [30]. In Pastry, nodes are given a unique identifier chosen uniformly at random in a circular identifier space, organising nodes as a ring. Each node maintains a routing table of shortcuts, allowing systematic routing from any node to any other node in a logarithmic number of hops in the logical network.

4.2 Execution Flow

The node contacted by the external application is referred to as the *source* in the remainder. The reception of data triggers the creation of an execution tree rooted at the source for the execution requested by the application. The execution will also finish on the source, which will finally deliver the inert solution, *i.e.*, the result, to the requesting application. We now detail the construction of the execution tree.

Once the source node receives the data of the chemical program, it scatters the data molecules across the Pastry ring according to their hash values; the cryptographic hash function of the underlying DHT guarantees uniform dispersion with high probability (w.h.p.). Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops [30], where n denotes the number of nodes in the platform. In the course of the routing process, the path of each molecule is traced by intermediary nodes, called *forwarders*, from the source node to a molecule's destination node, referred to as a *worker*. By passing on molecules, a forwarder maintains a *local state* (in addition to the Pastry's routing table) containing the set of nodes to which it forwarded molecules. This set of nodes constitutes its child nodes in the execution tree. Note that forwarders, together with the source node, will be workers as well, w.h.p. Finally, the source node spreads down the tree one final message, mc containing the rules to execute.

Upon the receipt of mc on a node p from a node s, p completes its state with s, referring to it as its *parent* (in the multicast tree being built). In case p has already received mc from another node, it just drops it, but sends another specific message back to s, which, upon receipt, deletes p from s' local state. This ensures that, combined, the local states form a tree. This tree, rooted at the source node, will be used later to make partial inert solutions move backwards to the source node. The created tree presents some similarities with the Scribe publish/subscribe system [10]. Note that the complexity of the local state is logarithmic to the number of nodes in the system, as nodes referenced in the local state of a node (except its parent) are necessarily inside Pastry's routing table, itself logarithmic in size.

After receiving the multicast, nodes start locally the computation. Every possible combination of molecules residing on a node is checked on this node against the rules, and, if possible, reactions take place. The combinations' cardinality is determined by the number of molecules local to the node and the number of a rule's arguments. When the part of the solution received by a node is inert, it must associate itself with other nodes to continue the computation.

Each of them sends its inert local solution to its parent. Parents then add them to their own and continue the computation. Only when a parent has received all of its children's solutions and when its local solution is inert, the process continues with the parent transferring its local solution to its parent, and so forth until all of the inert local solutions reach the source node, which delivers the global solution after executing it until inertia.

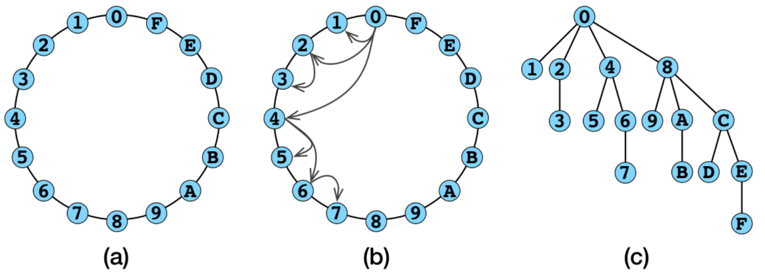

Figure 3: Execution example: (a) the original ring; (b) molecule dissemination and tree creation; (c) the execution tree.

Execution Example. Consider the sixteen-node Pastry ring shown on Figure 3a. Following Pastry's routing scheme, let node 0's routing table contain nodes 1, 2, 4 and 8, node 1's routing table nodes 2, 3, 5 and 9, and so forth. Furthermore, let node 0 be the source node for a chemical program composed of 16 molecules with IDs 0 through F. Node 0 holds on to molecule 0, while sending molecules 1 and 2 to their respective nodes since they are located in its routing table. In doing so, node 0 puts them into its children list in its local state. Molecule 3 is sent to node 2, which forwards it to node 3 and puts it in its own children list. Next, node 0 sends molecules 4 through 7 to node 4. It keeps molecule 4 and forwards the others to nodes 5 and 6, putting them in its children list. The process of disseminating molecules and creating parent-child relations for the first eight molecules is depicted in Figure 3b. Once all of the molecules have reached their destinations, the complete execution tree is in place (Figure 3c). It is then used to spread the program's rules and signal the beginning of the execution. In the final phase, local inert solutions travel in the opposite direction — from node F to node E, then to node C, *etc.* After receiving the inert local solutions from nodes 1, 2, 4 and 8, node 0 starts the last execution cycle and delivers the final result to the requesting application.

Fault Tolerance. While failures can affect our scheme (failures or disconnections of nodes can lead to (i) routing problems, and (ii) loss of molecules), it is not our primary concern here. However, we give here a few hints for its reliability. The sub-tree formerly rooted at a crashed node becomes unable to forward its results up the tree. Inspired by the work in [10], a simple detection and reconnection protocol can be defined: when initiating the last broadcast message, the source can include its ID. Then, when a node is unable to reach its parent, it can dynamically find a new path to the root by launching a reconnection request in the DHT on the source ID, and thus rebuild a connected tree. For dealing with the loss of molecules one can rely on state machine replication [23, 31].

4.3 Condition Checking and Inertia Detection

We now discuss the mechanisms enabling (i) reaction condition checking and (ii) inertia detection. The former can be thought of as being part of the actual execution process — in order to perform a reaction, the runtime has to make sure the reaction condition holds. The latter, which represents a critical problem, is a termination detection mechanism the task of which is to detect the fact that no new reaction can be performed, signalling the execution's completion.

4.3.1 A Difficult Problem

Because the execution effectively stands idle while this step is performed, it needs to be performed as efficiently as possible. For the sake of discussion and accuracy, we present two algorithms distributing the task of trying every possible combination of molecules. The first one, based on a repeated brute-force mechanism, is intuitive but sub-optimal, highlighting the fact that, if one is careless, many unnecessary tests can be done, increasing the complexity of an already hard task. The second one we have devised, referred to as *BucketSolver*, is shown to be optimal in terms of number of combination tests.

Since all of the molecule combinations have to be checked in order to ensure inertia has been reached, detecting it represents an NP-complete problem. We, therefore, refer to an algorithm as being optimal if it checks all of the combinations, but it checks each of them once and only once. Note that one cannot avoid examining all of the combinations in the general case. Then, any algorithm performing every check once and only once will be considered as optimal. A suboptimal algorithm, *invece*, may test a subset of combinations more than once, wasting CPU.

Brute-force Algorithm. The approach which seems the most intuitive is that, upon the receipt of molecules from one of its children, a node starts a computation cycle, during which every possible molecule combination of the local solution is tested, and possible reactions performed, in this way locally leading to inertia. Then, if a parent has got g children, there are exactly $g + 1$ such cycles: the first one occurs after the initial dissemination of molecules, while the other g cycles take place after the result of each child has been received. Note that, even though two or more children's results might be received simultaneously, this will not reduce the number of computation cycles in terms of number of tests. Thus, the total number of tests performed by this algorithm depends not only on the number of molecules and nodes, but on the tree's structure as well. In other words, the number of tests performed can variate considerably from one execution to the next.

4.3.2 BucketSolver Algorithm

As the reader may have noticed, the sub-optimality of the brute-force algorithm comes from its reaction condition checking routine: once a node receives or generates new molecules, it puts them in its unique local solution without keeping track of already checked combinations, leading to future unnecessary tests.

When a node transfers its local solution to its parent, the parent is sure that all of the combinations in the node's local solution have already been tried. The parent does not need to know exactly which combinations have been checked, as long as it knows the set of molecules they derive from. Thus, in this second algorithm, we create *buckets* into which we put sets of molecules the combinations of which have already been tried.

When a node originally receives molecules from the source, it puts them each in its own bucket. A computation cycle comprises checking only inter-bucket combinations — those whose elements belong to different buckets. For the sake of clarity, let us consider two buckets. Formally, when checking a combination of r arguments, j, $0 < j < r$, elements are picked from bucket a, while $r - j$ elements are picked from bucket b. If the combination is evaluated positively, the elements are removed from their respective buckets and once the reaction has been carried out, each resulting molecule is placed in a new, separate bucket. Once two initial buckets' intersection combinations have been checked, they are *fused* — molecules from one bucket are put into the other and the now empty bucket is deleted. The solution, be it local or global, is declared to be inert once there is only one bucket left in the system.

Consider the example illustrated in Figure 4. Imagine a node checked two molecules, a_1 and a_2, which now reside in bucket a. The node then receives a result from its child and creates the bucket

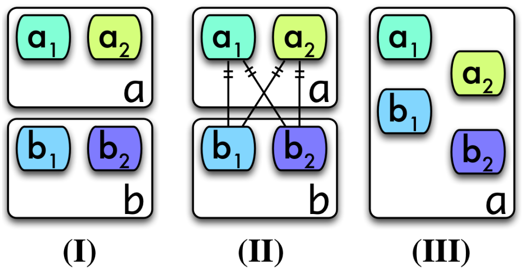

Figure 4: The process of checking buckets.

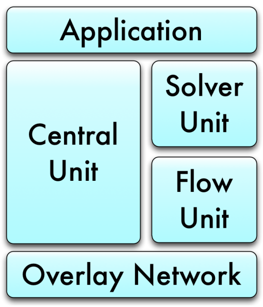

Figure 5: The prototype's logical concept.

b (Figure 4(I)). Then, it checks all of the combinations except those of elements residing in the same bucket, *i.e.* (a_1, a_2) and (b_1, b_2) (Figure 4(II)). Finally, presuming no reaction took place, the two buckets are fused into one (Figure 4(III)).

BucketSolver provides *inertia detection* (all of the combinations will be examined) while being *optimal* (every combination will be checked only once). Experimental results confirm these assertions.

5 Experimental Evaluation

To better capture the viability of the platform designed, and thus go further in the feasibility study, a software prototype of the architecture and algorithms described in Section 4 was developed[1]. It was experimented with on two chemical programs presenting different complexity properties, as outlined in Section 5.1. The results are discussed in Section 5.2.

The abstract concept of the prototype is depicted in Figure 5. It exploits FreePastry [2], an implementation of the Pastry DHT, as its overlay network. Its facilities are used by the two units directly above it — the central and the flow unit. They represent the implementation of the architecture laid out in Section 4.2. The central unit is in charge of communicating with external applications and sending and receiving molecules, while the flow unit builds and manages a node's local state. The central role in the execution is played by the solver unit, which is the implementation of the inertia-detection algorithms. The two versions presented were implemented. The implementation of the brute-force algorithm reflects precisely its theoretical description, while that of BucketSolver carries a slight optimisation — a multi-threaded version of the algorithm is implemented with a minimum amount of synchronisation; this optimisation is possible since the well-defined bucket boundaries imply that two disjoint groups of buckets, each containing more than one bucket, can be executed independently one from the other.

5.1 Test Programs

The evaluation of the proposed architecture and algorithms was conducted using two programs. We consider two families of programs having a different complexity regarding inertia detection. While the expression of programs chosen is quite simple (due to the expressiveness and uncluttered style of the chemical model), their runtime's complexity is similar to many other chemical programs with far more complex logic. Put simply, what matters is the number of test and reactions to be performed over time. The first class of applications has an amount of data that decreases over time, resulting in a decreasing complexity of the combination checking process. The second exhibits a complexity that does not variate during execution.

[1]The sources are available in the *branches/devel-distrib* directory of the *svn* repository located at *http://gforge.inria.fr/scm/?group_id=2125*.

Programs with Decreasing Complexity. A large collection of real-world applications solve problems the complexity of which gradually decreases as computation progresses. In our experiments we chose to represent this class of applications with the *getmax* program :

$$\textbf{let } \mathit{getmax} = \textbf{ replace } x :: \mathit{int}, y :: \mathit{int} \textbf{ by } x \textbf{ if } x \geq y$$

The rule requires two input arguments — two integers —, consumes them, and creates a new molecule which holds the higher value of the two. This is a so-called *reducer* rule (according the terminology given in [14]) since each reaction it activates decreases the total number of molecules, and thus, the program's complexity. This is a typical scenario for virtually all data-processing applications where the amount of data to be processed diminishes over time. In order to simulate data processing, we introduced a pause of one second after each reaction. In the experiments we performed for this program, a solution containing 50,000 molecules was used.

Non-decreasing Complexity Programs. The second is a chemical program containing one multiset comprised of 5000 molecules, each composed of two integer numbers — an index and a value associated with it —, and a single rule, *sort*, operating on them:

$$\begin{aligned} \textbf{let } \mathit{sort} = \ & \textbf{replace } \{x.i, x.v\}, \{y.i, y.v\} \textbf{ by } \{x.i, y.v\}, \{y.i, x.v\} \\ & \textbf{if } (x.i > y.i \ \&\& \ x.v < y.v) \,||\, (x.i < y.i \ \&\& \ x.v > y.v) \end{aligned}$$

The rule consumes two molecules if they are not already sorted in ascending order. Two new molecules are then created, holding the same indices as the original ones, but with swapped values. Although remarkably simple, this program exhibits an important property — it keeps the number of molecules in the solution, as well as the complexity of the program, constant over time, which means that, at the end of the computation, the multiset to be processed by the source node is still large in size, exposing a potential scalability limit of the approach.

5.2 Results

Experiments were carried out on the prototype with the programs presented above. They target the examination of the following aspects: (i) the scalability of the architecture, (ii) the performance of distributed inertia detection (algorithms from Section 4.3), (iii) the network traffic generated during a program's execution, and (iv) the distribution of work amongst nodes involved in the computation.

The experiments were conducted on the French nation-wide Grid'5000 [2] [8] computing platform, where the nodes, varying in number from 100 to 1000, were scattered randomly across nine geographically distant sites. The results represent values averaged over 6 runs. The configuration of the logical (DHT) network was changed upon each run.

Experiment 1 (Execution Time). Firstly, we examine the viability of the framework defined. Figures 6 and 7 show the speed-up in execution time achieved by the runtime when compared to the execution time of a single runtime instance. This experiment confirms that chemical programs can be directly executed over such a distributed platform, and allows to capture better the architecture's effectiveness, discussed below.

When considering the execution time of programs with a decreasing complexity, depicted in Figure 6, we observe that using either of the two algorithms results in considerable speed-ups, which linearly grow when increasing the number of nodes. However, there are significant differences in performance between the two algorithms; while with the brute-force approach a maximum speed-up of 150 is achieved, BucketSolver is able to reduce the execution time by a factor of up to 900. Although both algorithms perform the same number of reactions, when using BucketSolver, the

[2] `http:www.grid5000.fr`

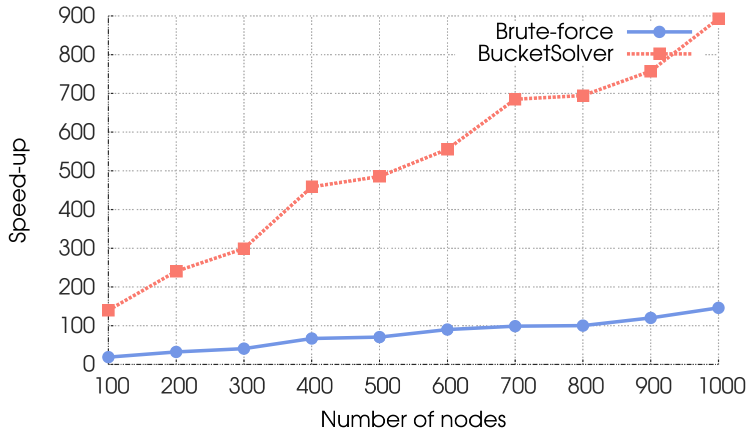

Figure 6: Speed-up in execution time for the *getmax* program.

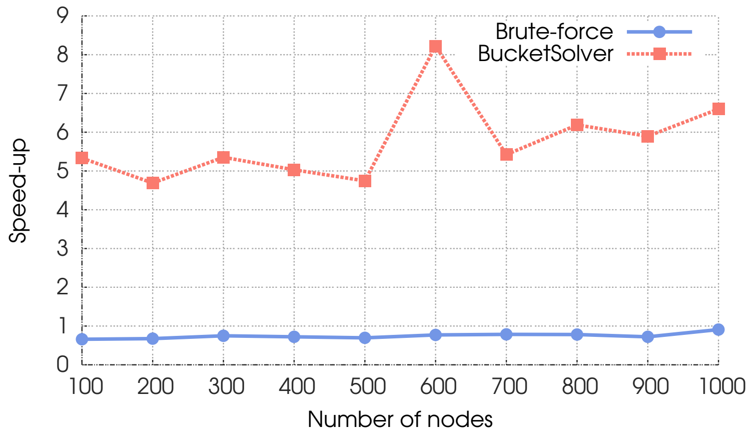

Figure 7: Speed-up in execution time for the *sort* program.

runtime benefits from the implementation's optimisation of multi-threading, and thus performs multiple reactions at a time.

On the other hand, Figure 7 shows that in the case of the *sort* test program distribution of the execution does not induce a significant speed-up. Concretely, the brute-force algorithm is not even able to match a single instance's execution time regardless of the number of nodes, while BucketSolver achieves speed-ups in the range $[5, 7]$. Note that, due to the non-determinism of the execution, the result obtained for 600 nodes, where a speed-up of 8 has been obtained, is in fact an artefact. Still, it is visible that, overall, a global, coherent speed-up is achieved. In contrast to *getmax* where parents in the tree have got less work to do than their child nodes after receiving the results from the, during the execution of *sort* parents receive the same amount of molecules they initially forwarded to their children. Thus, even though the total number of molecules in the system is constant, the complexity of the program increases over time for each of the participating nodes. As a consequence, parents have got to wait longer on their children's responses, ultimately prolonging the entire execution. Finally, BucketSolver performs better due to its optimality and ability to exploit multi-threading. However, its performance is limited by the fact that the number of nodes involved in the computation is negligible compared to the total number of checks and reactions done during execution.

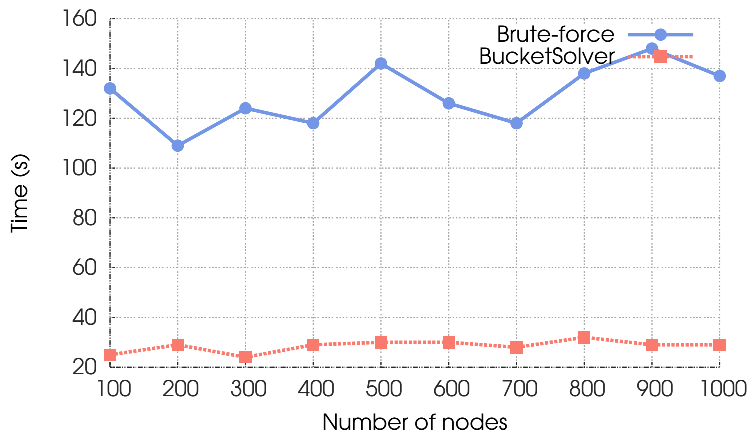

Figure 8: Execution time on an inert solution.

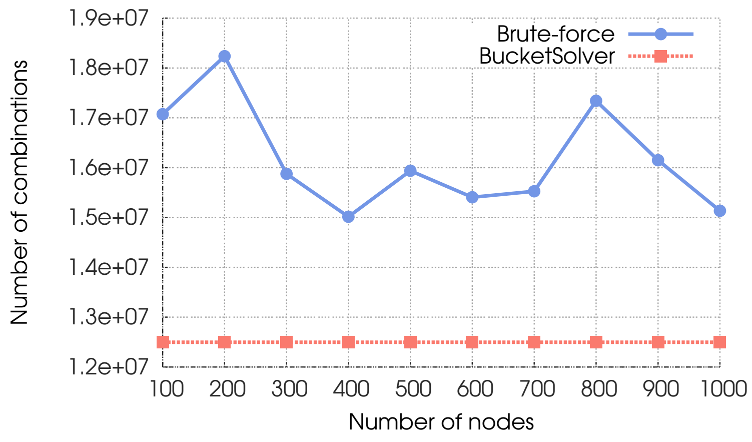

Figure 9: Number of checks done on an inert solution.

Experiment 2 (Inertia Detection). Next, we investigate the overhead of distributed inertia detection, depicted on Figures 8 and 9. The tests were conducted using the *sort* program on an inert solution. The total execution time, depicted in Figure 8, shows the brute-force algorithm's inability to efficiently detect inertia in a distributed fashion. Moreover, the fluctuation in execution time reveals its dependence on the structure of the tree built during execution. On the other hand, BucketSolver decreases the inertia detection time four to five times. However, its performance is not improved with the increase in the number of nodes, suggesting that nodes spend a considerable part of their time waiting on results from other nodes. The total number of combinations checked, shown on Figure 9, confirms BucketSolver's optimality: the total amount of combinations tested matches that of a single instance regardless of the number of nodes involved in inertia detection. As predicted, this optimality does not hold for the brute-force algorithm, where the number of combinations checked fluctuates wildly.

Experiment 3 (Communication Overhead). In this experiment we analyse the scalability related to communication. Depicted in Figure 10 is the total number of bytes sent during execution, normalised per molecule. As expected, the overhead of adding new nodes is smaller for *getmax* than for *sort*, since its complexity (and number of molecules) decreases over time. Although steeper, the curve for *sort* shows that distributing the execution has a rather limited impact on network traffic. Moreover, the trend of this overhead is inversely proportional to the growth of the number of nodes:

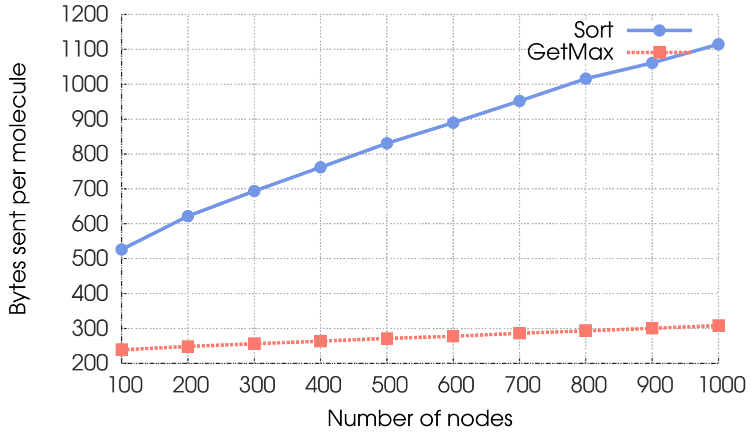

Figure 10: Communication costs per molecule for both test programs.

5 bytes/molecule/node for 100 nodes and 1.1 bytes/molecule/node for 1000 participating nodes. Thus, we can conclude that increasing the number of nodes does not introduce network bottleneck problems.

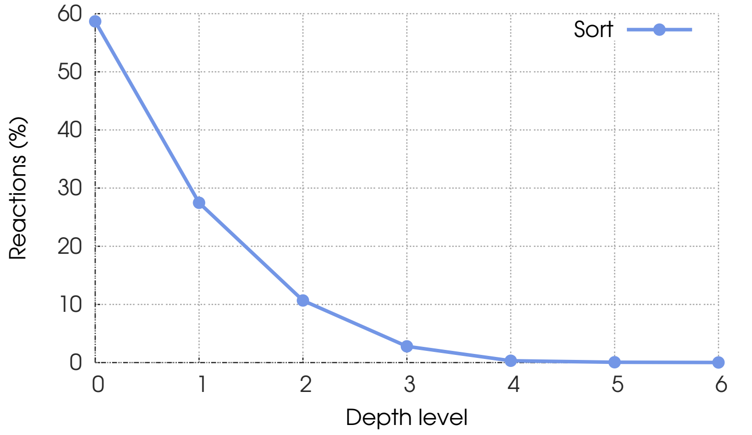

Figure 11: Distribution of reactions done per depth for the *sort* program, $n = 600$

Experiment 4 (Workload Distribution). Lastly, we explore the distribution of work amongst nodes according to their depth level. Figure 11 shows the percentage of reactions done at each depth of the tree when running the *sort* program on 600 nodes. The number of reactions done decreases at each depth. This scenario happens due to the hierarchical organisation of the platform : as parent nodes integrate the inert local solutions of their children, they are bound to do more reactions than their child nodes ; the higher in the tree hierarchy a node is, the bigger the local solutions it receives from its children. This culminates in the source node doing around 60% of all of the reactions. Figure 12 depicts the percentage of reactions done by the source node when varying the size of the program to execute. Even though the curve exhibits a certain stability, one should note that the actual number of reactions done by the source node is in fact increasing with the size of the problem. This experiment establishes a computation bottleneck at the source node — regardless of the amount of work to be done, the source node will always do the greatest number of combination checks, which entails a possibly high number of reactions when compared to other nodes.

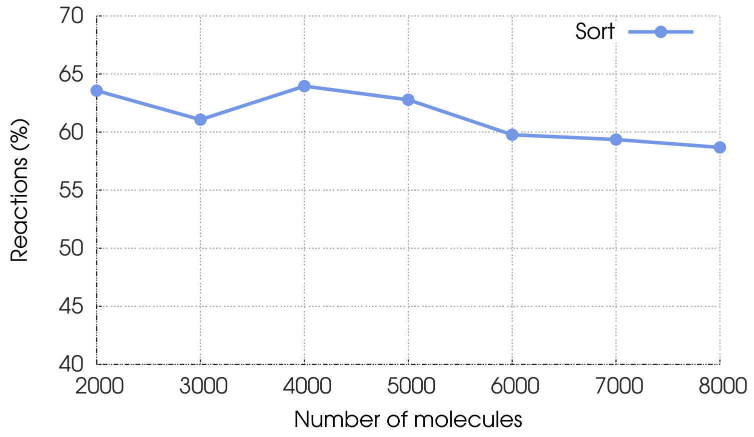

Figure 12: Percentage of reactions done by the source node for the *sort* program, $n = 400$

6 Conclusion

Chemistry-inspired models have been recently highlighted as a promising alternative paradigm to program autonomic service infrastructures. More precisely, the chemical programming model offers one the possibility to easily model such systems due to its non-deterministic and implicitly parallel paradigm. At the same time, the programmer does not have to worry about issues inherent to distributed systems, such as synchronisation or communication between nodes. However, its distributed runtime has until now been largely ignored.

This paper presents a conceptual and experimental study of a generic framework to solve this issue in a transparent way for programmers. We discuss two approaches for achieving this goal: a runtime based on the distributed shared memory model, and a hierarchic one based on peer-to-peer communication.

Our first attempt at a distributed chemical runtime environment exploits the distributed shared memory model. The shared memory space is used to store molecules and rules, as well as a list of combinations of molecules which have to be checked by the nodes against the rules. The management of this list leads to several issues, all of the nodes having to access it continuously in order to progress in the computation.

For this reason we studied the feasibility of a hierarchical architecture based on message passing in a peer-to-peer network. The design of a tree-structured framework based on distributed hash tables is discussed, and algorithms needed to build a chemical runtime are given, in particular dealing with the crucial inertia detection problem for which an optimal distributed mechanism is provided. The software prototype built and the experimental campaign conducted establish the viability of the concepts presented. As a next step, we plan to compare the performance of our model to that of similar parallel architectures used in distributed computing, most notably MapReduce, Linda and PGAS. Beyond these aspects, this work is intended for any programmer willing to deploy chemical specifications on top of a distributed platform.

In spite of its viability, performance and low network overhead, the platform suffers from a computation bottleneck on the tree's root node as a consequence of the runtime's hierarchical structuring. This issue in particular has to be tackled among the fair amount of work remaining to be done on the path towards efficient distributed chemical computing. To address it, we are currently considering the conception of a fully-decentralised chemical platform which would eliminate the currently present computation bottleneck. With regards to the inertia detection problem, integrating intelligent mechanisms to locate and capture specific molecules involved in a reaction would eliminate the need for checking all of the possible combinations, consequently significantly decreasing the cost of inertia detection.

References

[1] Chord. http://pdos.csail.mit.edu/chord/, June 2012.

[2] Freepastry. http://www.freepastry.org, June 2012.

[3] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Émilien Antoine. A rule-based language for web data management. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 293–304, New York, NY, USA, 2011. ACM.

[4] J.-P. Banâtre, A. Coutant, and D. Le Metayer. A Parallel Machine for Multiset Transformation and its Programming Style. *Future Gener. Comput. Syst.*, 4, 1988.

[5] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Generalised Multisets for Chemical Programming. *Mathematical Structures in Computer Science*, 16, 2006.

[6] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Towards Chemical Coordination for Grids. In *SAC*, 2006.

[7] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Comm. ACM*, 36(1), 1993.

[8] Raphal Bolze, Franck Cappello, Eddy Caron, Michel Dayd, Frdric Desprez, Emmanuel Jeannot, Yvon Jgou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Ira Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Winter 2006.

[9] J.B. Carter, J.K. Bennett, and Winy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, page 164. ACM, 1991.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. *IEEE JSAC*, 20, 2002.

[11] Hector Fernandez, Thierry Priol, and Cédric Tedeschi. Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In *IEEE ICWS*, 2010.

[12] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Not.*, 45(3):347–358, March 2010.

[13] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *12th International Symposium on Practical Aspects of Declarative Languages*, pages 88–103, 2010.

[14] Chris Hankin, Daniel Le Métayer, and David Sands. A parallel programming style and its algebra of programs. In *PARLE*, pages 367–378, 1993.

[15] Linpeng Huang, Weiqin Tong, Wing Kam, and Yongqiang Sun. Implementation of GAMMA on a Massively Parallel Computer. *JCST*, 12, 1997.

[16] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Software Eng.*, 21(4), 1995.

[17] Y. Jégou. Dynamic Memory Management on Mome DSM. *Cluster Computing*, 2006.

[18] Zakir Laliwala, Rahul Khosla, Pritha Majumdar, and Sanjay Chaudhary. Semantic and rules based Event-Driven dynamic web services composition for automation of business processes. In *Services Computing Workshops, 2006. SCW '06. IEEE*, pages 175–182, 2006.

[19] Peter Langendoerfer, Krzysztof Piotrowski, Manuel Diaz, and Bartolome Rubio. Distributed shared memory as an approach for integrating wsns and cloud computing. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, pages 1 –6, may 2012.

[20] Adrien Lèbre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing kernel development complexity in distributed environments. In Emilio Luque, Toms Margalef, and Domingo Bentez, editors, *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 576–586. Springer Berlin / Heidelberg, 2008.

[21] Hong Lin, Jeremy Kemp, and Padraic Gilbert. Computing Gamma Calculus on Computer Cluster. *IJTD*, 1(4), 2010.

[22] John W. Lloyd. Practical advtanages of declarative programming. In *Joint Conference on Declarative Programming (GULP-PRODE'94)*, pages 18–30, 1994.

[23] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in distributed hash tables. In *IPTPS*, 2002.

[24] Hugh McEvoy. *Gamma, chromatic typing and vegetation.* Imperial College Press, 1996.

[25] David Mentré, Daniel Le Métayer, and Thierry Priol. Formalization and verification of coherence protocols with the gamma framework. In *PDSE*, 2000.

[26] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7), 1998.

[27] Zsolt Németh, Christian Pérez, and Thierry Priol. Distributed workflow coordination: molecules and reactions. In *IPDPS*, 2006.

[28] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):63 –71, summer 1996.

[29] Yann Radenac. *Programmation "chimique" d'ordre supérieur.* PhD thesis, Université de Rennes 1, 2007.

[30] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, 2001.

[31] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22, 1990.

[32] Mirko Viroli and Matteo Casadei. Chemical-inspired self-composition of competing services. In *SAC*, 2010.

[33] Yi Wang, Minglu Li, Jian Cao, Feilong Tang, Lin Chen, and Lei Cao. An ECA-Rule-Based workflow management approach for web services composition. *In 4th Int. Conference on Grid and Cooperative Computing (GCC)'05*, 3795:143–148, 2005.

[34] Kai-Leung Adam Wong. *Jasmine: A shared-object multi-locking distributed shared memory system for heterogeneous computers.* PhD thesis, University of Queensland Australia, 2004.