# A Short Overview of Executing Γ Chemical Reactions over the ΣC and τC Dataflow Programming Models

Loïc Cudennec, Thierry Goubier

# A short overview of executing Γ Chemical Reactions over the ΣC and $\tau$C Dataflow Programming Models

Loïc Cudennec and Thierry Goubier

CEA, LIST
91191, Gif-sur-Yvette, France
surname.name@cea.fr

**Abstract**
Many-core processors offer top computational power while keeping the energy consumption reasonable compared to complex processors. Today, they enter both high-performance computing systems, as well as embedded systems. However, these processors require dedicated programming models to efficiently benefit from their massively parallel architectures. The chemical programming paradigm has been introduced in the late eighties as an elegant way of formally describing distributed programs. Data are seen as molecules that can freely react thanks to operators to create new data. This paradigm has also been used within the context of grid computing and now seems to be relevant for many-core processors. Very few implementations of runtimes for chemical programming have been proposed, none of them giving serious elements on how it can be deployed onto a real architecture. In this paper, we propose to implement some parts of the chemical paradigm over the ΣC dataflow programming language, that is dedicated to many-core processors. We show how to represent molecules using agents and communication links, and to iteratively build the dataflow graph following the chemical reactions. A preliminary implementation of the chemical reaction mechanisms is provided using the $\tau$C dataflow compilation toolchain, a language close to ΣC, in order to demonstrate the relevance of the proposition.

*Keywords:* Chemical programming, dataflow programming, many-core processor

## 1 Introduction

Massively parallel architectures become prevalent in computing systems, from large scale infrastructures (computing grids and clouds) to computing chips (GPGPU, many-core accelerators). One key challenge is to write applications that efficiently and transparently benefit from this parallelism. In many-core processors, several hundreds to thousands cores are packed into a single chip, the size of which is close to a regular processor. These cores are interconnected thanks to a dedicated network-on-chip (NoC), making the architecture possibly hierarchical with non-uniform access times to distributed memories (NUMA). Some examples of many-core

processors are proposed by Adapteva [3], Intel [14], Kalray [1], Tilera [26] and other research projects [2, 23]. As for large distributed and parallel systems, writing efficient applications for many-core architectures requires programming skills, long development steps and complex debugging [27]. This is particularly true if based on regular programming models such as shared memory (e.g. OpenMP) or message passing (e.g. MPI).

One audacious and yet elegant approach has been proposed in the late eighties, with the concept of chemical programming [10]. In this model, data are seen as molecules that can react together, following a given set of rules, to *disappear* or *create* new ones. In the early 2000, there has been a new fad for chemical programming with the introduction of computing grids. This new context was obviously conducive to propose new programming models. Computing grids are composed by tens of thousands processors spread in universities, research institutes and private companies, and are subject to quite high dynamicity: entire sites can join and leave the network. In this context, chemical programming allows to write applications as a mathematical abstraction, without having to care about complex software and hardware architectures. Interactions between molecules transparently occur on the platform, using available physical resources. Unfortunately, to our knowledge, there exist no efficient implementation of such a chemical programming language, mainly because of the critical complexity that is hidden in the compilation toolchain and the runtime.

In this paper, we explore the possibilities to implement a chemical programming model over a compilation toochain designed for a dataflow language. The dataflow model organizes processings within communicating agents. Each agent reads data on its input ports, executes its user-code and writes results onto its output ports. The resulting application is described by a communication graph. In order to adapt the semantics of chemical programming into this graph, we represent molecules with the data exchanged on communication ports and the chemical reactions with the user-code of each agent. The paper is organized as follow. Sections 2 and 3 respectively introduce the Γ and ΣC programming models. Section 4 shows how to use the ΣC model to instantiate Γ programs. Section 5 present the implementation in the $\tau$C compiler and gives preliminary results on implementing chemical programs using this compiler.

## 2 Simplified Γ programming model

The Γ programming model [10] takes its concept from the chemical metaphor, in which molecules interact to transform into new ones. From the programming point of view, it consists in declaring a set of data and a set of operators. These operators are applied onto data, without any particular order (it is not specified by the developer). This order is resolved online, and only depends on the current execution context, as chemical reactions can occur within a test tube. Chemical programming is therefore a massively parallel programming model, in which reactions can occur in a concurrent way. Γ is also related to the (Stochastic) Chemical Reaction Network model (CRN) [22] that opens research on programmability and new hardware [15].

The Γ programming model is formally defined using multi-sets and rewrite rules in [9]. A multi-set contains a collection of elements that can potentially show up several times. In Γ, one element represents a molecule, and is also equivalent to a data. This collection represents one state of the solution that is contained in the test tube. Rewrite rules let modify this multi-set using the following scheme: *Replace* $P$ *by* $M$ $if$ $C$ With $P$ and $M$ molecule types and $C$ a condition. Using these semantics, a program that gives the maximum value of a set of integers can be written using the following rewrite rule: *Replace* $x, y$ *by* $x$ $if$ $x \geq y$ Integers are compared two-by-two and are replaced by the greatest. The solution becomes inert when it contains only one integer. Other applications, far more complex, have been proposed in [25],

such as different graph algorithms (cycle detection, minimax, path-finding), a complete e-mail server, a distributed concurrent versioning system, a multi-filter image processing pipeline and an operating system kernel [8]. Chemical programs are massively parallel by nature, different reactions can occur in the *same* time. However, the execution model guarantees the atomicity of each reaction, which removes the need to add synchronizations in the application.

The chemical programming paradigm eases the development of parallel applications by hiding all distributed programming considerations. As a counterpart, the chemical concepts have to be handled by the language, the compilation toolchain and the communication middleware. Several implementations of such an abstract machine have been proposed since the introduction of the model in the late 1980. A first sequential implementation [13] has functionally validated the approach. Another implementation, distributed, synchronous, using a centralized hyperviser is proposed in [5]. An asynchronous version, based on a ring topology is then after proposed in [7]. More recently, a higher-order language, HOCL [6], has been proposed to go further in writing programs in the Γ-calcul. This language considers reaction rules as molecules. A preliminary sequential implementation of HOCL written in JAVA is given in [25].

Analogies between Γ-calcul and a language based on communicating agents has been studied in several works, especially in the context of computing grids. In [24] and [11], the authors define a workflow coordination model based on the chemical metaphor. Processings are applied on data without having been planned in advance. Chemical reactions are decided according to data availability, physical resources availability and a given set of application rules. These works focus on the programming model and the workflow execution model. While opening new research directions towards efficient chemical programming, it does not provide clues on how to implement and deploy chemical workflows over a massively parallel and distributed infrastructure. One critical step is to efficiently coordinate reactions while being able to scale up to modern architectures.

In this paper, we propose to implement the chemical programming metaphor within a complete dataflow compilation toolchain designed for many-core processors. As opposed to the fully dynamic approach, we calculate a scheduling of the chemical reactions *before* going online. This planning can directly lead to a stable state. It can also be partial and generate temporary data that will be used to calculate the next reactions. In this latter case, we propose a loop-back system between compilation and execution steps. In the following chemical metaphor implementation [11], reactions are unpredictable. This approach can lead to irrelevant decisions that are based on a local view of the system, due to the necessity of scaling up with the architecture. In our approach, operational research algorithms can be used to calculate a set of reactions, taking advantage of the global view of the chip. These algorithms can take into consideration different cost functions, to maximize the application performance or to fit into the many-core architecture (number of processing cores, power consumption, limited available memory).

# 3 ΣC programming model

The ΣC dataflow language [21] has been proposed to ease the programmability of massively parallel architectures such as many-cores. Its compilation toolchain [4] allows verification and off-line optimization such as limiting the number of concurrent tasks [17], optimizing the dataflow throughput [12], dimensioning shared communication buffers [28], detecting deadlocks and scheduling tasks [18], calculating the place-and-route of tasks onto processing elements [19], calculating reorganization patterns to access data [16], and so on. One advantage of using this toolchain is to get all these properties for free, and a top-down approach from the language to the hardware. The C programming model is based on networks of connected agents. An agent

is an autonomous entity, with its own address space and thread of control. It has an interface describing a set of ports, their direction and the type of data accepted; and a behavior specification describing the behavior of the agent as a cyclic sequence of transitions with consumption and production of specified amounts of data on the ports listed in the transition, which makes the programming model a CSDF (Cyclo-Static DataFlow). A subgraph is a composition of interconnected agents and it too has an interface and a behavior specification. The contents of the subgraph are entirely hidden and all connections and communications are done with its interface. Recursive composition is possible and encouraged; an application is in fact a single subgraph named root. The directional connection of two ports creates a communication link, through which data is exchanged in a FIFO order with non-blocking write and blocking read operations (the link buffer is considered large enough). An application is a static dataflow graph, which means there is no agent creation or destruction, and no change in the topology during the execution of the application. Entity instantiation, initialization and topology building are performed offline during the compilation process. System agents ensure distribution of data and control, as well as interactions with external devices. Data distribution agents are Split, Join (distribute or merge data in round robin fashion over respectively their output ports / their input ports), Dup (duplicate input data over all output ports) and Sink (consume all data).

# 4 Simplified Γ over ΣC

On several aspects, the Γ-calcul model mechanisms can be compared to a dataflow, where reactions are represented by agents and molecules are represented by communication links (more precisely, the data types that are carried in). Reciprocally, a dataflow application can be compared to set of chemical reactions applied on molecules, in a partial order. The ΣC application graph precisely describes molecules, reactions and some causal dependencies between reactions. This graph corresponds to one possible *realization* of the test tube, from the initial state to the stable state. Running this graph allows to calculate the values attached to each molecules that are part of the stable state.

**Chemical semantics and ΣC.** In this chemical programming analogy, we represent the initial state of the test tube, as well as reactions, using a set of agents. These agents are split into three subsets. The *initial state subset* is composed by agents with one and only one output port. Each of them represent one initial molecule of the test tube. The value of the molecule can be read, at runtime, on the output port. The *operator subset* is composed by agents with at least one input port and one output port. These agents can be duplicated while building the application graph, as many times as needed, using the same user-code and instantiation parameters. This allows to apply the same kind of reaction onto different molecules. Chemical reactions are implemented by connecting typed communication ports together. This is done following *any* algorithm that builds a whole application graph. The purpose of this algorithm is discussed later. Once this graph is built, a third set of agents is defined, the *inert state subset*, composed by agents with one and only one input port. These agents are automatically created and inserted within the graph, connected to each output that has been left free. The value of the resulting molecules can be read at runtime on these output ports. Writing chemical programs in ΣC consists in declaring the initial state subset (instantiating a set of agents with one output port) and the operator subset (a set of agents with at least one input port and one output port). No connections between communication ports are expected within the user source code. However, the model does allow this. In that case, this means that some reactions

are already planned and enforced in the source code.

**Integrating within ΣC toolchain.** The ΣC toolchain is composed by four main compilation steps. The first step is devoted to lexical and syntax analysis that rewrites ΣC source code into regular ANSI C code. The second step instantiates the application by building the communication graph. Using this representation, the third and fourth steps are in charge of buffer sizing, placing and routing, scheduling, runtime generating and binary building. Chemical programming over ΣC only requires to modify the second step of the toolchain. In the classic compilation mode it is not possible to build an application graph with a user code that let communication ports unconnected. In chemical mode, the toochain is allowed to create connections between these empty ports and complete the communication graph. New connections are established following regular rules, such as matching the data types. Once a graph is built, the toolchain instantiates agents into the inert state subset, one instance per output port left unconnected, to collect the results. As a general observation, there is a tough point in the process of building and deciding what application graph should be executed on the target. The algorithm has to calculate a relevant realization, in a reasonable processing time, what can be very tricky when compiling large scale chemical applications. Building realizations can either be done by a sequential or parallel algorithm. However, what is expected at the end of the compilation, is to get an application graph with good parallelism properties, in order to benefit from the many-core architecture. Other considerations include a proper dimensioning of the number of agents, the size of the communication buffers and so on, in order to tightly adapt the binary to the chip.

**Algorithm to implement chemical reactions.** In this paper we do not go further into the search methods for realizations. However, we give a very first algorithm that calculates a proper solution. This algorithm builds a realization by sequentially applying operators. No constraints are taken into account in this version. This is a two-step algorithm: the first step calculates a solution while the second step creates the agents in charge of collecting the results. *The basic idea is to iteratively try each operator on the current set of molecules.* The resulting application graph built by this algorithm depends on the order the elements are picked from the *molecules* and *operators* sets. It is however quite easy to modify the algorithm to try different strategies in parallel. Furthermore, operational research algorithms can help in building the most appropriate graphs. Search constraints include the application sizing: the number of operations and molecules (connections), the width and depth of the graph, which is related to the degree of parallelism and the longest sequential path in the dataflow.

**Dynamicity and iterative compilation.** Once a realization of the test tube has been built by the compilation toochain and executed on the target, a set of results is collected. These results represent molecules that can not react together using the initial operators. However, it is possible to add new molecules and operators in the test tube, provoking new reactions. This dynamicity can be implemented thanks to *iterative compilation*: compilation and execution steps are repeated several times. Iterative compilation is commonly used in the context of optimization, but as an off-line feature, before deploying the application on its final environment. Here, we propose to use iterative compilation as part of the execution of the application. A new chemical program is built after each run of the application. This program is composed by the original operators, the molecules that result from the previous run, and potentially new molecules and operators that are fed to the system. The dynamicity that is introduced with iterative compilation is coarse-grained: it not possible to add new molecules and operators while building a realization or executing the application. Here, the solution has to become inert prior to each modification. Another limitation coming with this granularity is that it

is not possible take into account the values of the intermediate molecules in the application graph to choose the next reaction, because there is a strong separation of the compilation and execution steps. However, we can imagine that the compilation toochain is allowed to build a partial realization of the test tube, that does not necessarily lead to an inert solution. The intermediate molecules would then be part of the decision to build the following application graph. As for many complex systems, this is a trade-off between performances (building a single graph without iterative compilation) and dynamicity (allowing several loops between the compilation toolchain hosted on one platform, and the dataflow application hosted on possibly another platform).

**Chain reactions, graph cycles and energy.** In the Γ calcul, it is possible to write some replacement rules and initial molecules that continuously react without reaching a stable state. For example, let's consider a rule that replaces a molecule with the same one. In that case, the program is not supposed to terminate. This example is simple and it is obvious to detect a chain reaction. However, in more complex applications, this can occur after applying a rather sophisticated number of replacement rules. Furthermore, this also depends on the availability of a particular set of molecules at each reaction step. Chain reactions are legit in a chemical program. However, in the dataflow model, it generates cycles within the application graph. These cycles prevent the compiler from terminating the building of the application graph and from running the program. A first solution consists in detecting graph cycles and to break the building sequence. Another solution - that has been implemented in the compiler - is to introduce a limit on how many reactions can sequentially occur. This limit is implemented through the concept of *energy*. Each reaction consumes an amount of energy that is proportional to the number of molecules it involves. We choosed to attach this amount of energy onto each molecule, and not to manage a global energy in the system, the latter solution would make difficult the parallelization of the graph building algorithm. With this system, one additional condition to apply an operator to molecules is that each input molecule has a positive amount of energy. The operator consumes one unit of energy on each input and each resulting molecule receives a shared part of the sum of the energy left in the operator. This energy management system can also be implemented in a chemical distributed runtime, unlike the graph cycle breaking system that would be quite tricky to detect.

**Towards a full Γ support.** The mapping of the chemical paradigm over the dataflow language does not fully implement the Γ calcul. The powerful expressiveness of the chemical paradigm partly resides in the reaction condition rule (the $C$ term of the replacement rule in section 2). In this condition it is possible to *evaluate* the current value of the molecules before taking the decision of reacting. If we consider the chemical program that returns the maximum of a set of integers, reactions occur *only if* the value of one integer is greater or equal to another one (and not because there exist two integers). In our system, molecules can react based on their inherent property of being a molecule, not on the value it carries. This is directed by the fact that reactions are statically decided at compile time, by connecting ports according to their data type. Therefore, the full support of the Γ calcul implies a *dynamic runtime environment* that has been the purpose of most of the previous implementations, with all the complexity that comes with large-scale parallel and distributed systems. Once again, our implementation is a trade-off between the expressiveness of the paradigm and the ease of implementation and deployment of the applications. Dynamic iterative compilation presented in section 4 can be used to implement reaction conditions that depend on the value of molecules. The compilation toolchain should be able to process separately conditions that need results from a previous execution, from the ones that only involve the type of molecules. This directly determines the

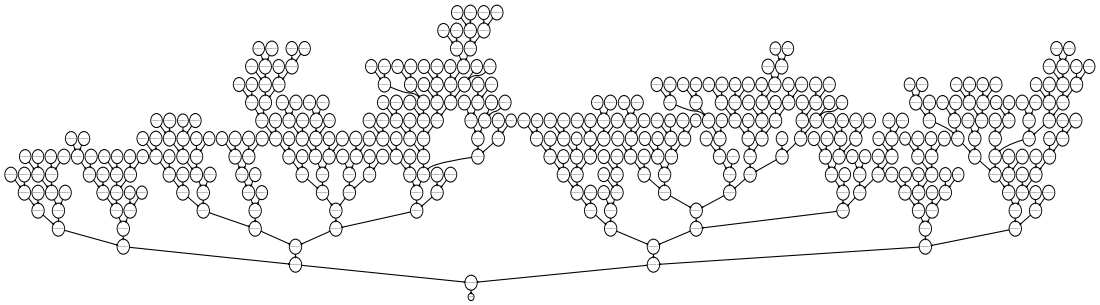

Figure 1: Application graph built compiling the *compare* application with 200 initial molecules.

granularity of the compilation-execution cycles, as discussed previously.

# 5 Experimentations

## 5.1 Building a simple chemical application in $\tau$C

Experimentation have been conducted using the $\tau$C compiler [20], given the latter focus on an agile experimental infrastructure compared to the industrial focus of ΣC. For the experimentation purpose and for the sake of simplicity, we consider a chemical program that returns the greatest value of a set of integers. We use this program to demonstrate the top-down approach, from writing the source code to the execution onto a multi-threaded host. From this program, the compilation toolchain builds the application graph, connecting molecules and operators. One possible solution is given in Figure 1, where 200 initial molecules are put into a chain reaction thanks to 199 instances of the comparison operator. The resulting communication graph - this is the actual *dot* output of the $\tau$C compilation toolchain - includes all initial molecules within a reversed tree. At execution time, the data stream goes from the leaves to the root. Each run of the compiler produces a similar graph, with random properties in terms of width and depth. The fact that all molecules are connected within a single graph comes from the monotonic aspect of the application in which only one operator is declared with only one data type. In case of more complex operators with multiple input and output ports, as well as several data types, the toochain may build a solution consisting of several disjoint graphs because all molecules does not fit into a single graph. These graphs can be deployed and executed in parallel. This just means that there is no causal dependencies between tasks belonging to different graphs, which is great news because it avoids distributed synchronizations.

## 5.2 Benchmarking a naïve building algorithm

One central step in the chemical compilation toolchain is the application graph building program. This program has to deal with a possibly large number of tasks to build a solution, while running operational research algorithms to take into account some given constraints. In this experiment, we evaluate the time needed to build such a graph, keeping in mind this has to be fast enough to be embedded within a compilation toolchain. We use the simple comparison application made of one operator that takes two data and returns only one of the two, based on an arbitrary choice. The graph building algorithm is implemented as an ANSI C sequential program. No particular optimization have been applied to the code and data structures,

making room for further improvements. All Experimentation have been passed on a 2009 Intel Core 2 Duo CPU P8600 at 2.40$GHz$ running Debian 3.2.63-2 x86_64 GNU/Linux. Only one core out of the two was used in the system. Note that this hardware is used for compiling the application and not to measure the execution performance of the resulting dataflow onto a manycore processor.

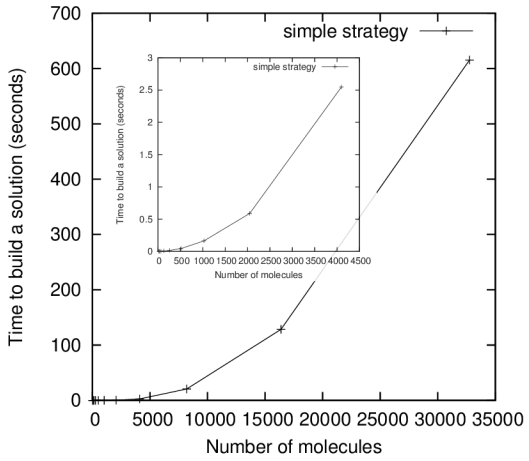


Figure 2: Time to build a solution for the chemical *compare* application, using a naïve algorithm (simple strategy), depending on the number of initial molecules.

We choose to run the Experimentation on this mobile device in order to demonstrate that the compiler extension can be used on a regular laptop station. Figure 2 gives the time to build a solution depending on the number of initial molecules instantiated in the program, from 16 to 32768 molecules. The compilation time follows a parabola from 0.0002 to 615 seconds. According to the valgrind/cachegrind binary instrumentation framework, more than 65% of the time is spent looking for free compatible communication ports, which can be largely improved using optimized data structures and caching systems. However, even with a naïve implementation, the algorithm can process more than 2500 molecules in less than a second. With these performances, we can reasonably consider calculating several solutions and keep the one that fits best to the user or target constraints, as it would be done in a regular optimization loop.

# 6 Conclusion

The chemical paradigm is an elegant and concise approach to the programming of massively parallel and distributed systems. However, its powerful expressiveness and abstraction of the architecture rely on complex mechanisms. These mechanisms have to manage code and data over large distributed systems. In today's regular programming languages, these mechanisms are mainly implemented by the developer, based on her/his skills and knowledge of the platform. In a straight implementation of the chemical model, it has to be managed by the compiler and the runtime, falling back into application deployment, scheduling, communications and other common issues that are related to distributed systems. In this paper, we have proposed to map the chemical paradigm over the dataflow paradigm. Today, there exists several compilers for dataflow language that target industrial chips and perform efficient application compilation, deployment and runtime generation. The mapping of the chemical paradigm over a dataflow language can benefit from the efficiency of the whole compilation toolchain. We have proposed to represent molecules by the output ports of the agents and to define molecules according to the data types. Chemical reactions are represented by agents that take some input molecules, fire up user code and output (or not) new molecules. In this system, a partial order of the chemical reactions is calculated by the compilation toolchain before running the application on the targeted host. This static strategy has a major advantage, as well as a major inconvenience over a fully distributed implementation. One advantage is to run optimization algorithms to

build the application graph that fits best to some constraints. For example, it is possible to search a good trade-off between the width of the application graph to get more parallelism and a proper use of the shared physical resources on the targeted host. The main disadvantage of this solution is that it loses the all-distributed spirit of the system: we have shown that some complex mechanisms based on iterative compilation have to be introduced to re-enforce the dynamicity of the chemical paradigm, as well as the expressiveness of the replacement condition rule. Furthermore, this approach will mostly work with applications that are well balanced between the off-line compilation step and the online calculating step. To demonstrate the mapping of the chemical paradigm over the dataflow model, we have modified the $\tau$C compilation toolchain by allowing unconnected ports, by adding a new graph building module and by defining tasks to collect results. This top-down prototype is used to compile a simple comparison application written as a chemical program. We also show that building the application graph can be calculated in a reasonable time and be therefore integrated within an optimization loop. Ongoing works are now conducted on how to implement the replacement condition rule based on the results collected in a previous run.

# References


[1] The kalray mppa 256 manycore processor. Kalray S.A. http://www.kalray.eu/.

[2] The tera-scale architecture project (tsar). Bull and LIP6. https://www-soc.lip6.fr/trac/tsar.

[3] adapteva Inc. A 1024-core 70 gflop/w floating point manycore microprocessor. In *Proceedings of the 15th Annual Workshop on High Performance Embedded Computing*, HPEC 2011, Lexington, Massachusetts, USA, September 2011.

[4] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Doré, Paul Dubrulle, Benoît Dupont De Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, and Renaud Sirdey. Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *Alchemy 2013 - Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems*, volume 18, pages 1624–1633, Barcelona, Espagne, June 2013.

[5] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133 – 144, 1988.

[6] J.-P. Banâtre, P. Fradet, and Y. Radenac. Higher-order chemical programming style. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 84–95. Springer Berlin Heidelberg, 2005.

[7] Jean-Pierre Banâtre, Anne Coutant, and Daniel Le Métayer. Parallel machines for multiset transformation and their programming style. Rapport de recherche RR-0759, INRIA, 1987.

[8] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In CristianS. Calude, Gheorghe Pun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer Berlin Heidelberg, 2001.

[9] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, January 1993.

[10] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55 – 77, 1990.

[11] Manuel Caeiro Rodriguez, Zsolt Németh, and Thierry Priol. A chemical model for dynamic workflow coordination. In Yiannis Cotronis, George Angelos Papadopoulos, and Marco Danelutto, editors, *The 19th Euromicro International Conference on Parallel, Distributed and Network-Based*

*Computing*, Ayia Napa, Chypre, February 2011. University of Cyprus, Conference Publishing Services.

[12] Sergiu Carpov, Loïc Cudennec, and Renaud Sirdey. Throughput constrained parallelism reduction in cyclo-static dataflow applications. In *International Conference on Computational Science (ICCS 2013)*, volume 18, pages 30–39, Barcelona, Espagne, June 2013.

[13] Creveuil Christian. *Techniques d'analyse et de mise en oeuvre des programmes GAMMA*. PhD thesis, Université de Rennes 1, Rennes, 1991. Thèse de doctorat.

[14] George Chrysos. Intel xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium*, HC 2012, Stanford, California, USA, August 2012.

[15] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic Bioprocesses*, Natural Computing Series, pages 543–584. Springer Berlin Heidelberg, 2009.

[16] Loïc Cudennec, Paul Dubrulle, Franois Galea, Thierry Goubier, and Renaud Sirdey. Generating code and memory buffers to reorganize data on many-core architectures. *Procedia Computer Science*, 29(0):1123 – 1133, 2014. 2014 International Conference on Computational Science.

[17] Loïc Cudennec and Renaud Sirdey. Parallelism reduction based on pattern substitution in dataflow oriented programming languages. In *Proceedings of the 12th International Conference on Computational Science*, ICCS'12, Omaha, Nebraska, USA, June 2012.

[18] Paul Dubrulle, Stéphane Louise, Renaud Sirdey, and Vincent David. A low-overhead dedicated execution support for stream applications on shared-memory cmp. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 143–152, New York, NY, USA, 2012. ACM.

[19] François Galea and Renaud Sirdey. A parallel simulated annealing approach for the mapping of large process networks. In *IPDPS Workshops*, pages 1787–1792. IEEE Computer Society, 2012.

[20] Thierry Goubier, Damien Couroussé, and Selma Azaiez. tau-c: C with process network extensions for embedded manycores. *Procedia Computer Science*, 29(0):1100 – 1112, 2014. 2014 International Conference on Computational Science.

[21] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. Sigma-C: A programming model and language for embedded manycores. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin / Heidelberg, 2011.

[22] Donald A. McQuarrie. Stochastic approach to chemical kinetics. *J. Appl. Prob*, 4:413 – 478, 1967.

[23] Julien Mottin, Mickael Cartron, and Giulio Urlini. The sthorm platform. In Massimo Torquati, Koen Bertels, Sven Karlsson, and Franois Pacull, editors, *Smart Multicore Embedded Systems*, pages 35–43. Springer New York, 2014.

[24] Zsolt Németh, Christian Pérez, and Thierry Priol. Distributed workflow coordination: molecules and reactions. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 241–241, Washington, DC, USA, 2006. IEEE Computer Society.

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

[26] Richard Schooler. Tile processors: Many-core for embedded and cloud computing. In *Fourteenth Annual Workshop on High Performance Embedded Computing*, HPEC 2010, Lexington, Massachusetts, USA, September 2010.

[27] Vtor Schwambach, Sébastien Cleyet-Merle, Alain Issard, and Stéphane Mancini. Application-level performance optimization: A computer vision case study on sthorm. *Procedia Computer Science*, 29(0):1113 – 1122, 2014. 2014 International Conference on Computational Science.

[28] Renaud Sirdey and Pascal Aubry. A linear programming approach to general dataflow process network verification and dimensioning. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *ICE*, volume 38 of *EPTCS*, pages 115–119, 2010.