A SHORT OVERVIEW OF EXECUTING Γ CHEMICAL REACTIONS OVER THE ΣC AND TC DATAFLOW PROGRAMMING MODELS
MANY-CORE PROCESSORS

- Massively parallel chips
  - Hundreds of generic cores
  - Cores are interconnected through a network-on-chip (NoC)
  - Growing complexity of NoC topologies and memory hierarchies

SoC complexity trends
International Technology Working Group, System Driver Chapter 2010 Updates (2011)
PROGRAMMING MODELS FOR MANY-CORES

- Shared address space (Posix, OpenMP)
  - Handles synchronization, data localization and transfer
  - Convenient for the developer
  - Does not scale well

- Message passing (MPI, MCAPI)
  - Widely used in distributed systems
  - Fully managed by the developer
  - Long cycle of development and testing

- Dataflow programming: process networks (Streamit, OpenCl, Sigma-C)
  - Popular in Systems-on-Chip: signal processing, video encoding..
  - Communications are specified within the application code
  - Synchronization and data transfer handled by the runtime

- Can we go further in the abstraction of the system?
Chemical programming (Banâtre et al., INRIA)
- Proposed in the late 1980
- Mathematical formalism for distributed programs
- Declarative programming with replacement rules

Context: large parallel and distributed systems
- Computing clusters (1990-)
- Computing grids (2000-)
- Manycore processors (2010-)?

Replace P by M if C
Replace x,y by x if x > y

Chemical programming implementation challenges

- **Abstraction hides complexity** into the underlying system
- Must provide a complete runtime
- Task management: planning, scheduling, deployment
- Data management: localization, transfer, synchronization

Implementations

- Sequential implementation (Christian, 1991)
- Distributed, synchronous with centralized hypervisor (Banâtre, 1987)
- Distributed, asynchronous with ring topology (Banâtre, 1988)
- Over cloud computing runtime (Németh, Pérez, Priol, 2005, 2011)
Chemical programming implementation challenges

- Abstraction hides complexity into the underlying system
- Must provide a complete runtime
- Task management: planning, scheduling, deployment
- Data management: localization, transfer, synchronization

Implementations

- Sequential implementation (Christian, 1991)
- Distributed, synchronous with centralized hypervisor (Banâtre, 1987)
- Distributed, asynchronous with ring topology (Banâtre, 1988)
- Over cloud computing runtime (Németh, Pérez, Priol, 2005, 2011)

Full dynamic implementations must address complex distributed issues

Strength: Adaptability, real spirit of the chemical paradigm
Weakness: Some online decisions may be inefficient
(chemical reaction order and localization)

Proposition: relying on the dataflow paradigm

- Mapping chemical concepts over dataflow
- Benefit from decision algorithms at the compilation step
Network of connected agents

Amounts of data productions and consumptions are specified on each communication port

Buffer sizing, deadlock detection, place and route, scheduled execution

Industrial products

---


---

**Listing 1:** A simple filter agent definition in Sigma-C

```java
agent Filter (int width) {
  interface {
    in <int> input;
    in <float> random;
    out <float> output;
    spec { input [width]; random; output[width]};
  }
  map {
    agent myRandom = new Random();
    connect (myRandom . output , random);
  }
  void start () exchange (input i [width], random r, output o[width]) {
    int k = 0;
    for (k = 0; k < width; k++)
      o[k] = i[k] + r;
  }
}
```
Mapping chemical programming over dataflow

- **Agents** represent chemical reactions
- **Communication ports** represent molecules

**Chemical metaphor**

**Corresponding dataflow (Application graph)**

**Chemical program**
At compile time, **decide how to connect agents** to build the application graph.

- Connection between two ports follows regular rules (data type check)
- Operators can be duplicated as needed
- Several independent graphs are allowed (parallel execution)
- Ports left unused are connected to collector agents
INTEGRATING WITHIN THE TOOLCHAIN

Application graph

Static Analysis (code parsing) → C → Instantiating (agents and links)

Compilation host → Target host

Buffer sizing Place and route → Scheduling Runtime

Chemical graph building

Collecting results

Run

ΣC

ΣC
ITERATIVE COMPILATION

Initial Chemical Program

Chemical Program from the previous run

Additional Chemical Program

Iterative compilation Loop

Compilation

Choose Realization

Result calculated at run time

Compilation

Compilation

Choose Realization

Result calculated at run time

Compilation

Compilation

Choose Realization

Result calculated at run time
Partitioning the initial molecule set
Pros: parallel speedup
Cons: discards some reactions (different test tubes)
DIFFERENT REALIZATIONS

Instantiated gamma-Sigma-C

- Instantiated Sigma-C agent
- State variable
- Operator

Connection between ports

- Instantiated Sigma-C agent created at compile time
- Result calculated online

Two possible realizations

Calculating possible realizations

Instantiated application
Simple Algorithm to Build a Solution

1. Sequentially applying operators to available communication ports
2. Randomly choose ports to avoid patterns
3. Stops when no operator can be applied
4. At this step, no constraints or goals are taken into account

```java
agent [] molecules = initStateData;
agent [] operators = initOperators;
int stable = operators.size;

/* First step: Applying reactions */
while (stable > 0) {
    agent ope = operators.getNext().copy();
    agent [] mol =
        getSomeCompatibleMolecules(molecules, ope);
    if (mol.size > 0) {
        connectReaction(mol, ope);
        molecules.remove(mol);
        molecules.add(ope);
        stable = operators.size;
    } else {
        free(ope);
        stable--;
    }
}

port [] ports = getUnconnectedPorts(molecules);

/* Second step: Collecting results */
for (int i = 0; i < ports.size; i++) {
    agent result = newResultAgent();
    connect(ports[i], result.input);
    molecules.remove(ports[i].agent);
    molecules.add(result);
}
```
Example, initial state:

- **Operators**
  - 64 molecules
  - 16 operators
  - Up to 4 input or output ports
  - 4 data types

- **Molecules**
Possible realization:
- 5 independent graphs
- 13 collectors
Some initial states can generate never-ending reactions.

The chemical model does not limit cascading patterns (liveness).

Practically, we introduce the energy concept to limit reactions:
- Each molecule has an amount of energy (integer \( \geq 0 \)).
- Reactions can only occur between molecules with (energy \( > 0 \)).
- Output energy is given by the following (arbitrary) formula:

\[
\text{sum}(\text{input\_port}[i].\text{energy} - 1) / \text{nb\_output\_ports}
\]

Liveness is possible through iterative compilation.
Comparison application
Core-2 Duo @ 2.4GHz (2009)
Sequential implementation
From 16 to **32768** molecules
Up to **2500** molecules in a second of calculation
65% time used to find free communication ports (valgrind)
Mapping of the chemical model over the dataflow model

- Pros: benefit from offline compilation and online runtime
- Cons: liveness of the application relies on iterative compilation
- **Tradeoff** between the **expressiveness** of the paradigm and the **ease** of implementation and deployment

Application graph building: **decision process**

- Simple algorithm is fast enough to be included in a decision process
- **Constraints and goals:**
  - Task parallelism (maximize graph width)
  - Data parallelism (maximize graph depth)
  - Reduce task context switching, memory use (minimize agent number)
- **Algorithms:** exhaustive, genetic, metaheuristic
A SHORT OVERVIEW OF EXECUTING CHEMICAL REACTIONS OVER THE \( \Sigma C \) AND TC DATAFLOW PROGRAMMING MODELS