Parallel Lattice Programming

Lattice Theory for Parallel Programming

Pierre Talbot
pierre.talbot@uni.lu

22nd October 2025, 29th October 2025

University of Luxembourg

What in this presentation?

We are going to overview two parallel programming models:

1. Pessimistic Parallel Programming (state of the art).
2. Optimistic Parallel Programming (contribution).

Characteristics of our model

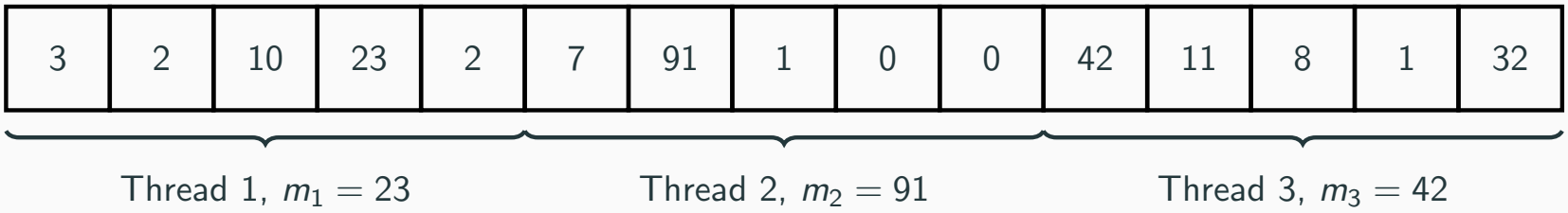
- Lock-free and correct.
- Based on fixpoint over lattices.
- Useful for programming parallel constraint solvers.

Pessimistic Parallel Programming

Running example: parallel maximum

Each thread computes its local max (map), then we compute the max of all local max (reduce).

- Map:

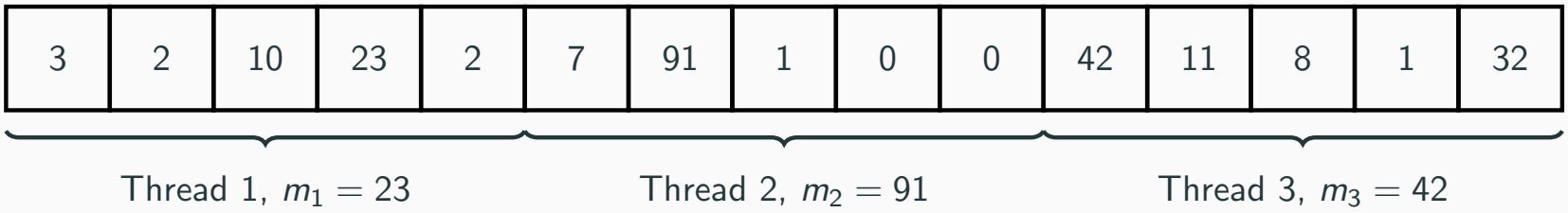


- Reduce: $max([23, 91, 42]) = 91$.

Running example: parallel maximum

Each thread computes its local max (map), then we compute the max of all local max (reduce).

- Map:



- Reduce: $max([23, 91, 42]) = 91$.

Sequential bottleneck: With 100 elements (10 threads), the reduce step takes as much time as the map step.

How to program the reduce step in parallel?

Parallel max

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Parallel max

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Good? No! Data-race.

Parallel max fixed!?

```
/** Suppose as many threads as elements in ‘data‘. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Good? No!

Can produce wrong results.

Parallel max fixed again!?

```
/** Suppose as many threads as elements in ‘data‘. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed again!?

```
/** Suppose as many threads as elements in ‘data‘. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Good? Yes!

But our “parallel” algorithm is now sequential.

Atomics to the rescue (?)

C++26 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.fetch_max(data[tid]);
}
```

Atomics to the rescue (?)

C++26 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.fetch_max(data[tid]);
}
```

Atomic operations are (much) slower than traditional operations.

Reduction in CUDA

Chapter 10 in book "Programming Massively Parallel Processors: A Hands-on Approach".

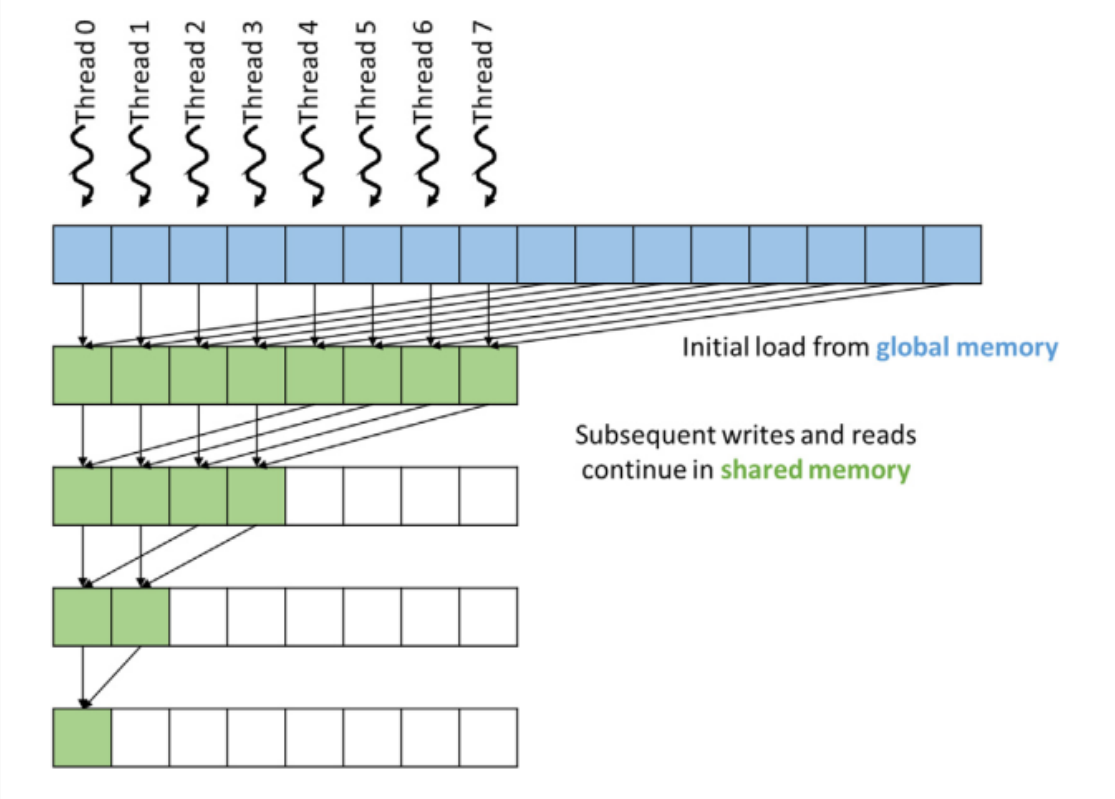
Reduction
And minimizing divergence

10

Chapter Outline

Reduction in CUDA

Chapter 10 in book "Programming Massively Parallel Processors: A Hands-on Approach".



Not easy, and eventually, some threads inactive.

The problem with threads

Multithreading programming is pessimistic.

For a data race that happens once in million instructions, this model:

- Makes parallel programming painful and difficult.
- Slows down computation.
- Prevents us from thinking with a true parallel mindset.

Optimistic Parallel Programming

Let's be optimistic

Instead of being afraid of data races, let's welcome them as part of the programming model itself.

```
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

What happens in case of a data race?

- Suppose two threads with `data = [1, 2]`.
- If a data race occurs, `*m == 1`.
- But if we run `max` again, then we must obtain `*m == 2`.

Let's do extra work only when data races occur (optimistic)

In case of n data races, we run the algorithm $n + 2$ times:

```
int old = *m + 1;
while(old != *m) {
  old = *m;
  max(0, data, m) || ... || max(n-1, data, m);
}
```

This is called the *fixpoint loop*.

Interlude: atomicity of load and store...

At start, suppose `x,y = 0`.

T1	T2
`x ← 512`	`x ← 1`

What are the possible outcomes?

Interlude: atomicity of load and store...

At start, suppose `x,y = 0`.

T1	T2
`x ← 512`	`x ← 1`

What are the possible outcomes? $x = 512$, $x = 1$ and... $x = 513$.

Really? 513?

Assignment is not necessarily atomic. View x as an array of two bytes `x[0]x[1]`:

```
T2: x[0] ← 0
T1: x[0] ← 1   (x = 512)
T2: x[1] ← 0
T1: x[1] ← 1   (x = 513)
```

But in practice, most architectures (x86, x64, ARM, ...) will atomically load and store 32 bits values (if correctly aligned).

Fixing optimistic max in parallel

Therefore, we still need atomic load and store:

```
void max(int tid, const int* data, std::atomic<int>& m) {
  if(data[tid] > m.load()) {
    m.store(data[tid]);
  }
}
```

Note that, we only need atomic load and store, every other operation can be performed non-atomically.

Optimization: Relaxed Memory Consistency

Relaxed memory consistency is enough here: the changes are *eventually* seen by everybody.

```
void max(int tid, const int* data, std::atomic<int>& m) {
  if(data[tid] > m.load(std::memory_order_relaxed)) {
    m.store(data[tid], std::memory_order_relaxed);
  }
}
```

Optimization: Relaxed Memory Consistency

Relaxed memory consistency is enough here: the changes are *eventually* seen by everybody.

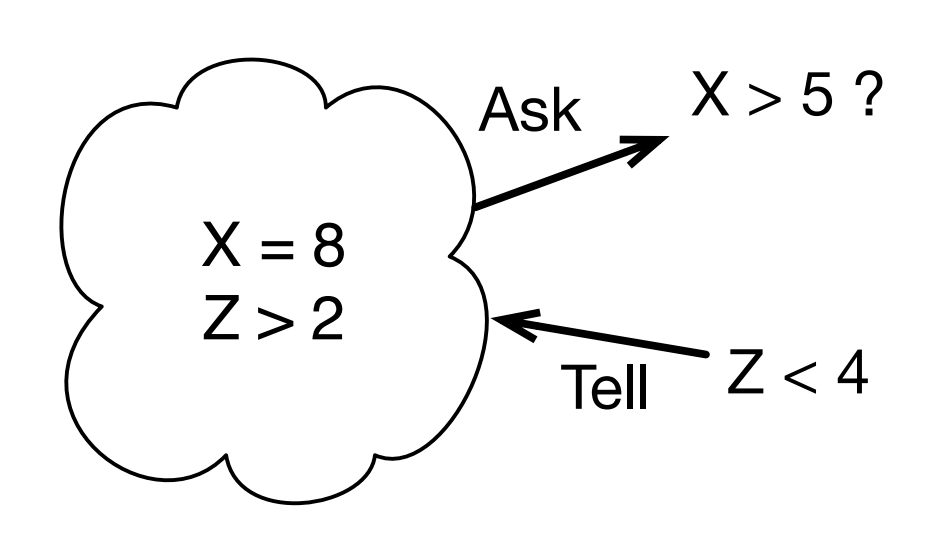
```
void max(int tid, const int* data, std::atomic<int>& m) {
  if(data[tid] > m.load(std::memory_order_relaxed)) {
    m.store(data[tid], std::memory_order_relaxed);
  }
}
```

How to design such algorithms? What properties do we need?

Concurrent Constraint Programming

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[1].

Two main operations: ask and tell.

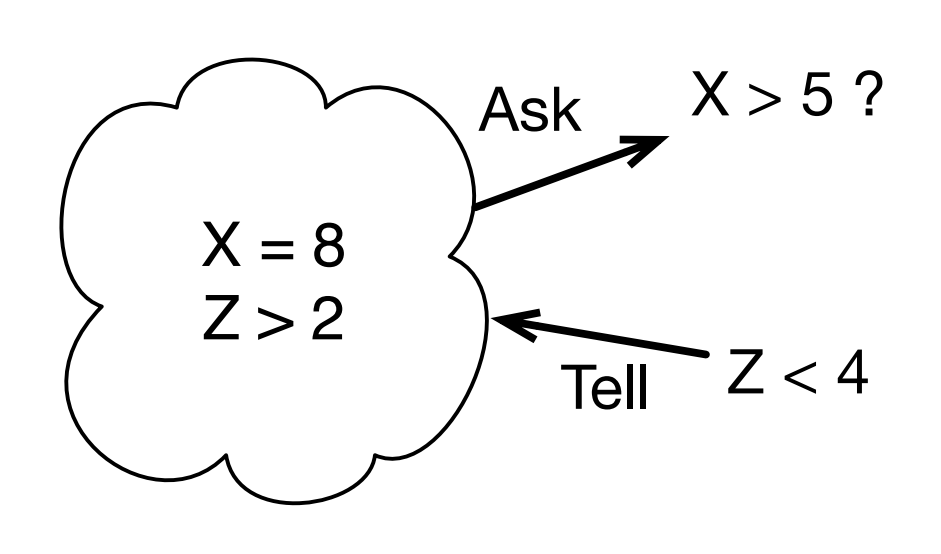


Conceptual idea: allow to compute with partial information; replace the "Von Neumann" memory model by a constraint store.

[1] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[1].

Two main operations: ask and tell.



Conceptual idea: allow to compute with partial information; replace the "Von Neumann" memory model by a constraint store.

Formally, what is the constraint store, tell and ask?

[1] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Syntax of First-Order Logic (FOL)

Let $S = \langle X, F, P \rangle$ be a *first-order signature* where X set of variables, F set of function symbols and P set of predicate symbols.

$$
\begin{array}{lr}
\langle t \rangle ::= x & \textit{variable } x \in X \\
\quad | \quad f(t, \ldots, t) & \textit{function } f \in F \\
 & \\
\langle \varphi \rangle ::= p(t, \ldots, t) & \textit{predicate } p \in P \\
\quad | \quad \neg\varphi & \textit{negation} \\
\quad | \quad \varphi \diamond \varphi & \textit{connector } \diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \\
\quad | \quad \exists x,\ \varphi & \textit{existential quantifier} \\
\quad | \quad \forall x,\ \varphi & \textit{universal quantifier}
\end{array}
$$

A *theory* is a set of formulas without free variables.

Constraint System

A constraint system is a tuple $CS = \langle S, \Delta \rangle$ where S is a first-order structure and Δ is a (consistent) theory.

- The store of constraints is a set of formula $C = \{c_1, \ldots, c_n\}$.
- *tell*(c) adds the constraint c to the store C (simple set union).
- *ask*(c) checks whether c can be *deduced* from C using the theory Δ.

Constraint System

A constraint system is a tuple $CS = \langle S, \Delta \rangle$ where S is a first-order structure and Δ is a (consistent) theory.

- The store of constraints is a set of formula $C = \{c_1, \ldots, c_n\}$.
- *tell*(c) adds the constraint c to the store C (simple set union).
- *ask*(c) checks whether c can be *deduced* from C using the theory Δ.

Example

Let $C = \{x > y, y > z, y = 2\}$ under the standard theory of arithmetic.

- *ask*$(x > 2)$?

Constraint System

A constraint system is a tuple $CS = \langle S, \Delta \rangle$ where S is a first-order structure and Δ is a (consistent) theory.

- The store of constraints is a set of formula $C = \{c_1, \ldots, c_n\}$.
- *tell*(c) adds the constraint c to the store C (simple set union).
- *ask*(c) checks whether c can be *deduced* from C using the theory Δ.

Example

Let $C = \{x > y, y > z, y = 2\}$ under the standard theory of arithmetic.

- *ask*$(x > 2)$? *true*.
- *ask*$(x < 2)$?

Constraint System

A constraint system is a tuple $CS = \langle S, \Delta \rangle$ where S is a first-order structure and Δ is a (consistent) theory.

- The store of constraints is a set of formula $C = \{c_1, \ldots, c_n\}$.
- *tell*(c) adds the constraint c to the store C (simple set union).
- *ask*(c) checks whether c can be *deduced* from C using the theory Δ.

Example

Let $C = \{x > y, y > z, y = 2\}$ under the standard theory of arithmetic.

- *ask*$(x > 2)$? *true*.
- *ask*$(x < 2)$? *false*.
- *ask*$(x = z)$?

Constraint System

A constraint system is a tuple $CS = \langle S, \Delta \rangle$ where S is a first-order structure and Δ is a (consistent) theory.

- The store of constraints is a set of formula $C = \{c_1, \ldots, c_n\}$.
- *tell*(c) adds the constraint c to the store C (simple set union).
- *ask*(c) checks whether c can be *deduced* from C using the theory Δ.

Example

Let $C = \{x > y, y > z, y = 2\}$ under the standard theory of arithmetic.

- *ask*$(x > 2)$? *true*.
- *ask*$(x < 2)$? *false*.
- *ask*$(x = z)$? *false* (we don't know yet! But possible in the future.)

ask(c) is true iff the formula $(\bigwedge_{\varphi \in C} \varphi) \Rightarrow c$ is also true. We write $c \vDash_\Delta d$ (entailment) to say d can be deduced from c.

Syntax of CCP

Let $x, x_1, \ldots, x_n \in X$ be variables, $c, c_1, \ldots$ be constraints, p a predicate name.

$\langle P,\ Q \rangle ::= \sum_{i \in I} ask(c_i)\ ?\ P_i$	*sum statement*
$\mid \quad tell(c)$	*tell statement*
$\mid \quad \exists x,\ P$	*local statement*
$\mid \quad P \parallel Q$	*parallel composition*
$\mid \quad p(x_1, \ldots, x_n)$	*predicate call*
$\langle A,\ B \rangle ::= p(x_1, \ldots, x_n) = P$	*predicate definition*
$\mid \quad A\ B$	*list of predicates*

Syntax of CCP

Let $x, x_1, \ldots, x_n \in X$ be variables, $c, c_1, \ldots$ be constraints, p a predicate name.

$$
\begin{array}{lll}
\langle P,\ Q\rangle ::= \sum_{i\in I} ask(c_i)\ ?\ P_i & & \textit{sum statement} \\
\quad \mid \quad tell(c) & & \textit{tell statement} \\
\quad \mid \quad \exists x,\ P & & \textit{local statement} \\
\quad \mid \quad P \parallel Q & & \textit{parallel composition} \\
\quad \mid \quad p(x_1, \ldots, x_n) & & \textit{predicate call} \\
\langle A,\ B\rangle ::= p(x_1, \ldots, x_n) = P & & \textit{predicate definition} \\
\quad \mid \quad A\ B & & \textit{list of predicates}
\end{array}
$$

Example

$$
\begin{array}{l}
\exists x, y, z, \\
\qquad ask(y = 1)\ ?\ tell(z > 10) \\
\parallel ((ask(x = 0)\ ?\ tell(y = 1))\ +\ (ask(x = 1)\ ?\ tell(y = 2)))
\end{array}
$$

Exercise: define a CCP predicate $max(x, y, z)$ such that $z = \max(x, y)$.

Sketch of Semantics

Definitions

- A *configuration* is a pair $\langle P, C \rangle$ where P is a CCP process to execute, and C is a store of constraint.
- A "step of execution" is given by a relation $\langle P, C \rangle \to \langle P', C' \rangle$.

TELL

$$\langle tell(c), C \rangle \to \langle tell(c), C \cup \{c\} \rangle$$

PAR-LEFT

$$\frac{\langle P, C \rangle \to \langle P', C' \rangle}{\langle P \parallel Q, C \rangle \to \langle P' \parallel Q, C' \rangle}$$

PAR-RIGHT

$$\frac{\langle Q, C \rangle \to \langle Q', C' \rangle}{\langle P \parallel Q, C \rangle \to \langle P \parallel Q', C' \rangle}$$

Main Properties

- *Monotonicity*: $\rightarrow$ is monotone over the store of constraints, in particular it means:
  - If $ask(c)$ is true in a store C then it is true in every store C' such that $C \subseteq C'$.
- *Extensive*: $\rightarrow$ is extensive over the store of constraints (we cannot remove information).
- *Closure operator*: $\rightarrow^*$ is a closure operator over the store.[2]
- *Restartable*: Suppose we perform a partial execution $\langle P, C\rangle \rightarrow \ldots \rightarrow \langle P', C'\rangle$, then we can restart the execution from $\langle P, C'\rangle$ (and obtain the same result).

[2] Supposing the branches of the sum are all disjoints—called *determinate CCP*.

Herbrand Constraint System

The following constraint system $H = \langle S, \Delta \rangle$ is the basis of *logic programming*, also known as *finite tree CS*.

- $S = (X, F, P)$ where F is the infinite set of all function symbols, and $P = \{=\}$ (only equality).
- Δ is given by Clark's equality theory[3]:
  - $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$.
  - $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m) \Rightarrow \texttt{false}$ where $f \neq g \vee n \neq m$.
  - $x = f(\ldots x \ldots) \Rightarrow \texttt{false}$

[3] Clark, K. L. (1977). Negation as failure. In Logic and databases.

Herbrand Constraint System

The following constraint system $H = \langle S, \Delta \rangle$ is the basis of *logic programming*, also known as *finite tree CS*.

- $S = (X, F, P)$ where F is the infinite set of all function symbols, and $P = \{=\}$ (only equality).
- Δ is given by Clark's equality theory[3]:
  - $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$.
  - $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m) \Rightarrow \texttt{false}$ where $f \neq g \vee n \neq m$.
  - $x = f(\ldots x \ldots) \Rightarrow \texttt{false}$

Example

Set r to *true* if x occurs in the list l.

$$\begin{aligned} contains(l, x, r) = \exists h, t, \\ & ask(l = list(h, t) \wedge h = x) \ ?\ tell(r = true) \\ + \ & ask(l = list(h, t) \wedge \neg(h = x)) \ ?\ contains(t, x, r) \\ + \ & ask(l = empty) \ ?\ tell(r = false) \end{aligned}$$

Exercise: Define merge sort (and use the parallel combinator).

[3] Clark, K. L. (1977). Negation as failure. In Logic and databases.

Parallel CCP

Parallel Concurrent Constraint Programming (PCCP)

Observation: CCP lacks a proper connection to parallel architecture, and had limited impact despite a beautiful theory.
We worked on that by simplifying the language (no recursion) and using lattice to define constraint system[4].

[4] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Parallel Concurrent Constraint Programming (PCCP)

Observation: CCP lacks a proper connection to parallel architecture, and had limited impact despite a beautiful theory.
We worked on that by simplifying the language (no recursion) and using lattice to define constraint system[4].

Syntax of PCCP

Let $x, y, y_1, \ldots, y_n \in X$ be variables, L a lattice, f a monotone function, and b a Boolean variable of type $\langle \{true, false\}, \Leftarrow \rangle$:

$\langle P,\ Q \rangle ::= \texttt{if}\ \ b\ \texttt{then}\ \ P$	*ask statement*
$\mid \quad x \leftarrow f(y_1, ..., y_n)$	*tell statement*
$\mid \quad \exists x{:}\mathrm{L},\ P$	*local statement*
$\mid \quad P \parallel Q$	*parallel composition*

[4] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Examples: Minimum and Constraint

Let ZUB the lattice of increasing integers, and ZLB the lattice of decreasing integers.

Minimum

$\exists m : \texttt{ZUB},\ m \leftarrow x_1 \parallel \ldots \parallel m \leftarrow x_n$ (unfolded for-loop)

$x + y \leq c$ constraint

Suppose the variables x and y are defined by four variables xl, xu, yl, yu modelling the intervals $[xl, xu]$ and $[yl, yu]$.

$$[\![x + y \leq c]\!] \triangleq xu \leftarrow c - yl \parallel yu \leftarrow c - xl$$

(see lecture on "abstract satisfaction").

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let $Proc$ be the set of all PCCP processes.

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let $Proc$ be the set of all PCCP processes.

Denotational Semantics

We define a function $\mathcal{D} : Proc \to (Store \to Store)$:

$$\mathcal{D}(x \leftarrow f(y_1, .., y_n)) \triangleq \lambda s.s[x \mapsto s(x) \sqcap f(s(y_1), .., s(y_n))]$$
$$\mathcal{D}(\texttt{if } b \texttt{ then } P) \triangleq \lambda s.(\!| \, s(b) \mathbin{?} \mathcal{D}(P)(s) \mathbin{:} s \, |\!)$$
$$\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$$

Executing the program: **gfp** $\mathcal{D}(P)$.

Sequential Computation = Parallel Computation

We obtain the same result if we execute P in parallel or if we replace all parallel $\|$ by a sequential operator ; (a transformation we write $\mathtt{seq}\ P$) defined as follows:

$$\mathcal{D}(P\ ;\ Q) \triangleq \mathcal{D}(Q) \circ \mathcal{D}(P)$$

Let **fix** f be the set of fixpoints of a function f.

Theorem (Equivalence Between Sequential and Parallel Operators)

fix $\mathcal{D}(\mathtt{seq}\ P) =$ **fix** $\mathcal{D}(P)$

Did We Cheat Using Math?...

OK Ok, we use a "parallel operator", but it is not really executed in parallel on a machine?
Also: $\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$, by unfolding this mathematics definition we have:

$$\begin{aligned} \mathcal{D}(P \parallel Q)(S) &= (\mathcal{D}(P) \sqcap \mathcal{D}(Q))(S) \\ &= \mathcal{D}(P)(S) \sqcap \mathcal{D}(Q)(S) \end{aligned}$$

What's the problem?

Did We Cheat Using Math?...

OK Ok, we use a "parallel operator", but it is not really executed in parallel on a machine?
Also: $\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$, by unfolding this mathematics definition we have:

$$\begin{aligned} \mathcal{D}(P \parallel Q)(S) &= (\mathcal{D}(P) \sqcap \mathcal{D}(Q))(S) \\ &= \mathcal{D}(P)(S) \sqcap \mathcal{D}(Q)(S) \end{aligned}$$

What's the problem?

We have copied the store!!

Not very "Von Neumann Architecture"-friendly.

Operational Semantics

Let's go to a lower-level of semantics, closer to the machine.

Guarded Normal Form (GNF)

We lift all parallel compositions at top-level, and obtain a set of guarded commands of the form $\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)$. The transformation is as follows:

$$gnf(A, x \leftarrow f(y_1, ..., y_n)) \triangleq \{A \Rightarrow x \leftarrow f(y_1, ..., y_n)\}$$
$$gnf(A, \texttt{if } b \texttt{ then } P) \triangleq gnf(A \cup \{b\}, P)$$
$$gnf(A, P \parallel Q) \triangleq gnf(A, P) \cup gnf(A, Q)$$

Theorem

fix $\mathcal{D}(gnf(\{\}, P)) =$ **fix** $\mathcal{D}(P)$

Operational Semantics

- A state is a pair $\langle s, G \rangle$ where s is the store and G a set of guarded commands.
- The operational semantics of PCCP is defined by a transition function $\hookrightarrow$ between states.

$$\text{SELECT}\quad \frac{(\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)) \in G \qquad \bigwedge_{i \leq n} s(b_i)}{\langle s, G \rangle \hookrightarrow \langle s[x \mapsto s(x) \sqcap f(s(y_1), \ldots, s(y_m))], G \rangle}$$

- The execution of a PCCP program is a possibly infinite sequence of states $\langle s_1, G \rangle \hookrightarrow \ldots \hookrightarrow \langle s_n, G \rangle \hookrightarrow \ldots$.

Fairness

Definition (Fairness)

A scheduling strategy is fair if, for each process $P \in G$, it generates transitions such that:

$$\forall i \in \mathbb{Z},\ \exists j \in \mathbb{Z},\ j \geq i \wedge$$
$$\langle s_j, G \rangle \hookrightarrow \langle s_{j+1}, G \rangle \text{ selects the process } P$$

A result of Cousot on chaotic iterations[5] guarantees that the limit of all fair scheduling strategies coincide.

It means the order of execution of the parallel processes doesn't matter!

[5] P. Cousot, *Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice*, Research Report 88, Sep. 1977.

Equivalence Denotational and Operational Semantics

We denote by $\mathcal{O}(P)$ the function mapping a store s to a store s', such that $\langle s, gnf(\{\}, P)\rangle \hookrightarrow \langle s', gnf(\{\}, P)\rangle$ where $\hookrightarrow$ follows a fair scheduling strategy.

Theorem (Fixpoint Equivalence)

$\textbf{fix}\ \mathcal{D}(P) = \textbf{fix}\ \mathcal{O}(P)$

OK OK, but can I really execute my program P in parallel on my machine or math is cheating us again??

Equivalence Denotational and Operational Semantics

We denote by $\mathcal{O}(P)$ the function mapping a store s to a store s', such that $\langle s, gnf(\{\}, P)\rangle \hookrightarrow \langle s', gnf(\{\}, P)\rangle$ where $\hookrightarrow$ follows a fair scheduling strategy.

Theorem (Fixpoint Equivalence)

$\textbf{fix}\ \mathcal{D}(P) = \textbf{fix}\ \mathcal{O}(P)$

OK OK, but can I really execute my program P in parallel on my machine or math is cheating us again??

We are almost there! But to prove it formally, we need to consider the instructions executed by the processor and not some abstract "meet" operations.

Load/Store Semantics

- Load and store instructions: **L** a r and **S** r a where a is the location of a variable in shared memory and r refers to a thread-local memory location such as a register.
- We compile a guarded command $\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)$ to a sequence of loads and stores.

```
while true:
  L b_1 rb_1; ...; L b_n rb_n;
  if rb_1 ∧ ... ∧ rb_n then
      L y_1 ry_1; ...; L y_m ry_m;
      C(f(ry_1, ..., ry_m), rf);
      L x rx;
      C(rx ⊓ rf, ox);
      C(ox < rx, bx);
      if bx then
          S ox x;
```

- The compilation function $\mathcal{C}(E, r)$ compiles the expression E, with its result stored into r.
- As the corresponding expressions are only defined over thread-local variables, they do not pose any concurrent threats.
- We note that $\sqcap$ and $<$ are compiled according to the lattice-type of the variable x.

Load/Store Operational Semantics

- Let r be a variable local to a thread.
- Let l be a sequence of load/store of a guarded command.
- Let Pl be a set of sequences, representing a program P, that can be executed in parallel.
- The transition $\langle s, a; l\rangle \twoheadrightarrow \langle s', l\rangle$ atomically updates the store s with the instruction a.

$$\textsc{load}\quad \langle s, \mathbf{L}\ a\ r; l\rangle \twoheadrightarrow \langle s[r \mapsto a], l\rangle \qquad \textsc{store}\quad \langle s, \mathbf{S}\ r\ a; l\rangle \twoheadrightarrow \langle s[a \mapsto r], l\rangle$$

$$\textsc{select2}\ \frac{l \in Pl \qquad \langle s, l\rangle \twoheadrightarrow \langle s', l'\rangle}{\langle s, Pl\rangle \Rightarrow \langle s', (Pl \setminus \{l\}) \cup \{l'\}\rangle}$$

- The rule $\textsc{select2}$ is similar to $\textsc{select}$ but executes instructions at a finer grain.
- Each step of the transition $\Rightarrow$ models an atomic action on the shared memory.

Very weak requirements on the parallel hardware

The load/store operational semantics make the following assumptions on the memory consistency model and cache coherency protocol:

(ATOM) Load and store instructions must be atomic.

(EC) The caches must eventually become coherent.

(OTA) Values cannot appear *out-of-thin-air*.

Main Results

- If there is a fixpoint reachable in a finite number of steps, we always reach it.
- We compute the same fixpoint than the sequential program.

Theorem (Soundness)

Let GI be the load/store compilation of a PCCP program P. Then for any sequence $\langle s_1, GI\rangle \Rightarrow \ldots \Rightarrow \langle s_i, GI'\rangle$, we have $s_i \geq (\mathbf{gfp}_{s_1}\ \mathcal{O}(P))$.

Theorem (Completeness)

Suppose the fixed point $os = (\mathbf{gfp}_{s_1}\ \mathcal{O}(P))$ is reachable in a finite number of steps. Then, $\exists i \in \mathbb{N}$, $\langle s_1, GI\rangle \Rightarrow \ldots \Rightarrow \langle s_i, GI'\rangle \Rightarrow \ldots$ such that $s_i \leq os$. Further, for all $j \in \mathbb{N}$, $j > i$, $s_i = s_j$.

Conclusion

C++ Abstraction: Lattice Land Project

`lattice-land` is a collection of libraries abstracting our parallel model.

It provides various data types and fixpoint loop:

- `ZLB`, `ZUB`: increasing/decreasing integers.
- `B`: Boolean lattices.
- `VStore`: Array (of lattice elements).
- `IPC`: Arithmetic constraints.
- `GaussSeidelIteration`: Sequential CPU fixed point loop.
- `AsynchronousIteration`: GPU-accelerated fixed point loop.
- ...

```
void max(int tid, const int* data, ZLB& m) {
  m.tell(data[tid]);
}
AsynchronousIteration::fixpoint(max);
```

`https://github.com/lattice-land`

Conclusion

Data races occur rarely, so we should avoid working so much to avoid them.

Properties of the model

A Variant of Concurrent Constraint Programming on GPU (AAAI 2022)[6].

- Correct: Proofs that $P; Q \equiv P||Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Modular: Add more threads without fear of breaking existing code.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[6] `http://hyc.io/papers/aaai2022.pdf`