A Variant of Concurrent Constraint Programming on GPU

Pierre Talbot, Frédéric Pinel, Pascal Bouvry
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
Esch-sur-Alzette L-4243, Luxembourg
surname.name@uni.lu

Abstract

The number of cores on graphical computing units (GPUs) is reaching thousands nowadays, whereas the clock speed of processors stagnates. Unfortunately, constraint programming solvers do not take advantage yet of GPU parallelism. One reason is that constraint solvers were primarily designed within the mental frame of sequential computation. To solve this issue, we take a step back and contribute to a simple, intrinsically parallel, lock-free and formally correct programming language based on *concurrent constraint programming*. We then re-examine parallel constraint solving on GPUs within this formalism, and develop TURBO, a simple constraint solver entirely programmed on GPUs. TURBO validates the correctness of our approach and compares positively to a parallel CPU-based solver.

Introduction

The number of cores of graphical computing units (GPUs) steadily increases over the years, reaching thousands nowadays, whereas central processing unit (CPU) clock speed and core numbers stagnate (Thompson and Spanuth 2021). However, it is notoriously difficult and error prone to write parallel programs on shared memory systems (Lee 2006). Threads introduce nondeterminism that must be pruned using synchronization mechanisms such as locks and barriers. However, too much synchronization quickly reduce the efficiency of a parallel program, sometimes making it slower than its sequential counterpart.

This phenomenon is observed in the field of *constraint programming*, where constraint solvers do not benefit yet from the parallelism offered by GPUs. One of the reasons is that most of the key optimisations of constraint solvers, including propagation loop, search strategies, global constraints and various forms of constraints learning, were primarily designed in the mental frame of sequential computation.

To grasp the challenge, we consider a standard solving algorithm named *propagate and search* (see, *e.g.*, Tack (2009)). Each constraint is implemented by a function, called a *propagator*, which removes inconsistent values from the domain of its variables. The propagation is the process of executing the propagators until a fixed point is reached and no more values can be removed. The *search step* then splits the problem into two complementary subproblems. Propagation and search alternates recursively until either a solution is found or no solution could be found.

Search can be parallelized without much hurdles by executing each subproblem on a different thread. This approach has been explored in-depth on CPUs and in distributed settings (Perron 1999; Schulte 2000; Malapert, Régin, and Rezgui 2016). However, it is not very efficient to run a subproblem per thread on a GPU; in part due to the limited cache memory available per thread. On the other hand, propagation is almost never parallelized in modern constraint solvers because propagators are executed in a highly sequential manner (Schulte and Stuckey 2008). Moreover, as propagators share variables, it seems that synchronization will unavoidably hinder efficiency and complicate the solver architecture (Gent et al. 2018).

In this paper, we contribute to a simple, intrinsically parallel, lock-free and formally correct programming language based on *concurrent constraint programming* (CCP) (Saraswat 1993), that we call parallel CCP (PCCP). CCP has been very successful to describe concurrent computations at the theoretical level, but has had limited impact on practical implementations. PCCP aims to bridge this gap between concurrency theory and parallel programming.

The key characteristics of PCCP is to allow threads to progress and communicate on a shared memory without synchronization, and still to always reach a unique deterministic result. The main ingredient is the notion of fixed point over *lattices* (Davey and Priestley 2002), a practical and mathematical structure, which guarantees the correct combination of parallel processes. To show the correctness of our language, we successively refine the semantics of PCCP from denotational, operational and then to a low-level semantics based on load and store operations. In particular, we show that whether the fixed point of a PCCP program is computed sequentially or in parallel, the result of the execution remains the same.

Through the lens of PCCP, we then re-examine parallel constraint solving on GPUs. We propose TURBO, the first propagate and search constraint solver fully implemented on GPUs. To accommodate the architecture of GPUs, we have made several algorithmic simplifications which are normally considered crucial for efficiency in sequential solvers. For instance, our propagation loop is eventless and is reminiscent of the AC-1 algorithm (Mackworth 1977). Our experiments show that TURBO obtains slightly better performance when compared to GECODE (Schulte, Tack, and Lagerkvist 2020), a parallel CPU-based solver. TURBO is only a proof of concept and several key components of modern constraint solvers are missing, in particular global constraints and constraint learning. Nevertheless, constraint decomposition of global constraints is an efficient approach that can be immediately used in TURBO (Narodytska 2011; Bessiere et al. 2009; Schutt et al. 2009). Still, it is clear that many improvements will be required to make TURBO a competitive solver.

Parallel Concurrent Constraint Programming

Parallel concurrent constraint programming (PCCP) is a variant of determinate CCP (Saraswat, Rinard, and Panangaden 1991) adapted to parallel programming. The main changes are an explicit usage of lattice-typed variables instead of a constraint system, and a reduced set of instructions (essentially without recursive procedures).

A *lattice* is an ordered structure $\langle L, \leq\rangle$ where L is a set and $\leq$: $L \times L$ a partial order relation. The *join operation* $\sqcup : L \times L \to L$ is obtained by $x \leq y \Leftrightarrow x \sqcup y = y$. When they exist, we write $\bot$ the smallest element and $\top$ the greatest element in L. The dual lattice of L, written L^∂, is the lattice with the reversed partial order ($\geq$ instead of $\leq$) on the same set of elements. A function $f : L \to L$ is *extensive* on L if $x \leq f(x)$, *monotone* if $x \leq y \Rightarrow f(x) \leq f(y)$ for all $x, y \in L$ and *idempotent* if $f(x) = f(f(x))$. Closure operators are extensive, monotone and idempotent functions. The pointwise lifting is a lattice construction ordering functions $f, g \in L \to K$ where $f \leq g$ iff $\forall x \in L,\ f(x) \leq g(x)$. The pointwise join is defined as $f \sqcup g \triangleq \lambda x.f(x) \sqcup g(x)$. A fixed point of f is an element $x \in L$ such that $f(x) = x$. We write $\mathbf{fp}_x(f)$ the fixed point $f(f(...f(x)))$, and $\mathbf{fix}\ f$ the function $\lambda x.\mathbf{fp}_x(f)$ mapping any element x to its fixed point. Details on lattice theory can be found in (Davey and Priestley 2002).

We now overview several useful lattices and the Cartesian product of two lattices. Firstly, we have the lattice of *increasing integers* $ZInc = \langle Z \cup \{-\infty, \infty\}, \leq\rangle$, such that $Z \subset \mathbb{Z}$. The order is given by the natural arithmetic order, the smallest element is $-\infty$ and the greatest one is ∞. The join operation is defined as the maximum of the two elements: $a \sqcup b \triangleq max(a, b)$. Similarly, we can define $ZDec$ for the decreasing integers (note that $ZDec = ZInc^\partial$), $FInc$ and $FDec$ for the increasing and decreasing floating-point numbers[1], and $BDec$ and $BInc$ for the lattice of Boolean where $\{true, false\}$ where $true \leq false$ in $BDec$. We note that these lattices form chains. In order to obtain new lattices from these primitive ones, the Cartesian product of two lattices $L \times K$ can be used. In this case, the order and join are defined pointwise, *i.e.*, $(a, b) \leq (a', b') \Leftrightarrow a \leq a'$ and $b \leq b'$, and $(a, b) \sqcup (a', b') \triangleq (a \sqcup a', b \sqcup b')$. The Cartesian product can be generalized to a n-ary product $L_1 \times \ldots \times L_n$. It is equipped with a monotone projection function $\pi_i((a_1, \ldots, a_n)) \triangleq a_i$ and a monotone embedding function $embed_i((a_1, \ldots, a_n), b_i) \triangleq (a_1, \ldots, a_i \sqcup b_i, \ldots, a_n)$. This way, we can obtain the lattice of integer intervals $IZ = ZInc \times ZDec$ where an element (ℓ, u) represents the set of values $\{v \in \mathbb{Z} \mid \ell \leq v \leq u\}$. The lattice of intervals forms a partial order that is not a chain. We write the monotone projection functions π_1 and π_2 more explicitly by $\lceil(\ell, u)\rceil \triangleq u$ and $\lfloor(\ell, u)\rfloor \triangleq \ell$.

[1]We can follow the total order given by IEEE-754-2019 (iee 2019).

Lattices are types in PCCP, and elements of these lattices are the data manipulated by PCCP programs. The syntax of PCCP is defined as follows, where $x, y, y_1, \ldots, y_n \in Vars$ are variables, L a lattice, f a monotone function, and b a Boolean variable of type $BInc$:

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

The ask statement tests the value of b, and if it is equal to $true$, executes the process P. The tell statement joins the value of the variable x and the result of a monotone function f. We declare a new variable using the local statement $\exists x{:}\texttt{L},\ P$ which initializes a variable x of type L to $\bot$, such that x is only visible to P. The parallel composition executes two processes P and Q in parallel. There is no explicit sequential composition, but sequentiality is still present due to ask statements—some processes might not be able to execute until some information is provided by other processes. A useful syntactic sugar is to allow an expression e in an ask statement `if` e `then` P, which is equivalent to $\exists b{:}BInc,\ b \leftarrow e \parallel \texttt{if}\ b\ \texttt{then}\ P$ where b is not free in P.

PCCP instructions are purposefully low-level because they should be directly implementable in a programming language such as C++ or Java. For clarity, we introduce a light modelling layer on top of PCCP, which generates PCCP programs at compile-time. Firstly, we use a generator expanding parallel processes at compile-time, *e.g.*, $\forall i \in \{1, 2\},\ x_i \leftarrow \bot$ expands to $x_1 \leftarrow \bot \parallel x_2 \leftarrow \bot$. We sometimes use generators inside expressions such as in $\sum_{i \in S} x_i$. All indices in names are resolved at compile-time, hence x_1 is really just the name of a variable. Our usage of generators is similar to those in traditional modelling languages such as MiniZinc (Nethercote et al. 2007), and their meanings should be clear from the context. Secondly, we compile constraints into corresponding PCCP programs using the function $[\![.]\!] : \Phi \to Proc$ where Φ is the set of constraints and $Proc$ the set of PCCP processes. The obtained process $[\![\varphi]\!]$ is called a *propagator* for the constraint φ. Finally, we rely on the monotone function $entailed : \Phi \to BInc$ which maps a formula φ to $true$ whenever φ is entailed from the store, and to $false$ otherwise.

We illustrate PCCP on the resource-constrained project scheduling problem (RCPSP). RCPSP is an optimization problem where one must find a minimal schedule such that resources usage do not exceed some capacities and some precedence constraints are satisfied. RCPSP is defined by a tuple $\langle T, P, R\rangle$ where T is the set of tasks, P is the set of precedences among tasks, written $i \ll j$ to indicate that task i must terminate before j starts, and R is the set of resources.

Each task $i \in T$ has a duration $d_i \in \mathbb{N}$ and, for each resource $k \in R$, a resource usage $r_{k,i} \in \mathbb{N}$. Each resource $k \in R$ has a capacity $c_k \in \mathbb{N}$ quantifying how much of this resource is available in each instant. The decision variables are the starting date s_i for each task i, and n^2 Boolean variables $b_{i,j}$ such that $b_{i,j}$ is true iff the tasks i and j overlap. This is a standard model of RCPSP which decomposes the cumulative global constraint (Schutt et al. 2009). The PCCP model for RCPSP is given as follows (where h is the horizon, *e.g.*, the sum of the durations):

$$\begin{array}{l}
\exists s_1 : IZ, \ldots, \exists s_n : IZ, \\
\exists b_{1,1} : IZ, \ldots, \exists b_{n,n} : IZ, \\
\quad \forall i \in [1..n],\ s_i \leftarrow (0, h) \\
\ \parallel \forall i \in [1..n],\ \forall j \in [1..n],\ b_{i,j} \leftarrow (0,1) \\
\ \parallel \forall (i \ll j) \in P,\ [\![s_i + d_i \leq s_j]\!] \\
\ \parallel \forall j \in [1..n],\ \forall i \in [1..n], \\
\qquad [\![b_{i,j} \Leftrightarrow (s_i \leq s_j \land s_j < s_i + d_i)]\!] \\
\ \parallel \forall k \in R,\ \forall j \in [1..n],\ [\![\sum_{i \in [1..n]} r_{k,i} * b_{i,j} \leq c_k]\!]
\end{array}$$

The compilation of constraints into PCCP processes is reminiscent of the compilation of constraints into indexicals, a simple CCP constraint system (Van Hentenryck, Saraswat, and Deville 1998; Carlsson, Ottosson, and Carlson 1997). The precedence and resource constraints are compiled as follows:

$$[\![x + y \leq c]\!] \triangleq x \leftarrow (\bot, c - \lfloor y \rfloor) \parallel y \leftarrow (\bot, c - \lfloor x \rfloor)$$

$$\begin{array}{l}
[\![\sum_{i \in [1..n]} r_{k,i} * b_{i,j} \leq c_k]\!] \triangleq \exists lsum{:}ZInc, \\
\qquad lsum \leftarrow r_{k,1} * \lfloor b_{1,j} \rfloor + \ldots + r_{k,n} * \lfloor b_{n,j} \rfloor \\
\qquad \parallel \texttt{if } r_{k,1} + lsum > c_k \texttt{ then } b_{1,j} \leftarrow (0,0) \\
\qquad \parallel \ldots \\
\qquad \parallel \texttt{if } r_{k,n} + lsum > c_k \texttt{ then } b_{n,j} \leftarrow (0,0)
\end{array}$$

The entailment operations are given by:

$$entailed(x + y \leq c) \triangleq \lceil x \rceil + \lceil y \rceil \leq c$$

$$\begin{array}{l}
entailed(\sum_{i \in [1..n]} r_{k,i} * b_{i,j} \leq c_k) \triangleq \\
\qquad r_{k,1} * \lceil b_{1,j} \rceil + \ldots + r_{k,n} * \lceil b_{n,j} \rceil \leq c_k
\end{array}$$

We note that the partial order $\leq$ is viewed as a function returning $true$ whenever $a \leq b$ holds. Logical connectors can also be defined in a similar manner, for instance we have:

$$[\![\varphi \land \psi]\!] \triangleq [\![\varphi]\!] \parallel [\![\psi]\!]$$

$$\begin{array}{l}
[\![\varphi \Leftrightarrow \psi]\!] \triangleq \\
\qquad \texttt{if } entailed(\varphi) \texttt{ then } [\![\psi]\!] \\
\quad \parallel \ \texttt{if } entailed(\psi) \texttt{ then } [\![\varphi]\!] \\
\quad \parallel \ \texttt{if } entailed(\neg\varphi) \texttt{ then } [\![\neg\psi]\!] \\
\quad \parallel \ \texttt{if } entailed(\neg\psi) \texttt{ then } [\![\neg\varphi]\!]
\end{array}$$

We have defined various PCCP processes without proving the functions used were indeed monotone, as required by the PCCP model. Automatically proving the monotonicity of arbitrary functions is a challenging task. Rules are given by Carlson, Carlsson, and Diaz (1994) for the restricted language of indexicals, and Monette, Flener, and Pearson (2012) discuss the topic on an extended indexicals language. In PCCP, the next lemma provides a useful lattice-theoretic view of partial orders as monotone functions:

Lemma 1. *Let L be a lattice and L^∂ its dual lattice, and $a \in L$ and $b \in L^\partial$. The function $f : L \times L^\partial \to BInc$ defined as $f(a,b) = true$ iff $a \geq_L b$ and false otherwise, is monotone.*

Proof. $a' \geq_L a$ implies $f(a,b) \geq f(a',a)$ since $a' \geq_L a \geq_L b$. Also, $b' \geq_{L^\partial} b$ (equivalent to $b' \leq_L b$) implies $f(a,b) \geq f(a,b')$ since $b' \leq_L b \leq_L a$. □

A particular case of this lemma is very useful to prove monotonicity of the functions in the previous examples. When $c \in L$ is a constant and a is a variable taking a value in L, a corollary is that $c \geq a$ (equivalently $a \leq c$) is monotone. For instance, $\lceil x \rceil + \lceil y \rceil \leq c$ is monotone because the projection functions and the integer addition are monotone (and the functional composition of monotone functions is monotone as well).

Further discussion on monotone functions is out of scope of this paper, and we will suppose that all functions in PCCP processes are monotone. Our focus is now on proving that PCCP programs can be correctly executed on a parallel hardware, and provide the same result than if executed on a sequential machine.

Semantics and Correctness

We introduce the denotational semantics, useful to prove properties on PCCP programs, the operational semantics, useful for implementation purposes, and the formal equivalence between the two. A PCCP process is an extensive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables. Because PCCP does not have recursive definitions, the number of local variables is finite and known at compile-time. Therefore, all local statements $\exists x{:}L,\ P$ can be erased at compile-time by replacing each variable's name with an index in $Store$ representing the lattice element of this variable. For instance, given a store $s \in Store$, we write $s(x) \triangleq \pi_x(s)$ the value of the variable x. The three remaining statements are the ask, tell and parallel operations.

Our semantic treatment of PCCP is based on the one of CCP (Saraswat, Rinard, and Panangaden 1991). However, instead of a constraint system, we manipulate a Cartesian product of lattices, allowing for a closer mapping between the theory and a practical implementation. We first give the denotational semantics of PCCP by defining a function $\mathcal{D} : Proc \to (Store \to Store)$:

$$\begin{array}{l}
\mathcal{D}(x \leftarrow f(y_1, .., y_n)) \triangleq \lambda s. embed_x(s, f(s(y_1), .., s(y_n))) \\
\mathcal{D}(\texttt{if } b \texttt{ then } P) \triangleq \lambda s. if\, s(b)\ then\ \mathcal{D}(P)(s)\ else\ s \\
\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcup \mathcal{D}(Q)
\end{array}$$

The tell operation joins the result of the function with the component indexed x of the store. The ask operation verifies if the variable b is $true$ in the store, in which case it maps to the denotation of P, and otherwise behaves like the identity function. Finally the parallel composition is the join of the denotations of the two processes. This mathematical definition seems to forbid P and Q to share the store s as it is "copied". When we introduce the operational semantics, this will not be the case anymore, but still, we will show that both semantics are equivalent.

Applying the function $\mathcal{D}(P)$ to the store might not yield a fixed point, and therefore it must be applied several times. We characterize this function in the next theorem, which is largely due to Saraswat, Rinard, and Panangaden (1991).

Theorem 2. *Let $L_1, \ldots, L_n$ be complete lattices, and $Store = L_1 \times \ldots \times L_n$. Then, for any process P, the least fixed point of $\mathcal{D}(P)$ exists and is unique. Moreover,* $\mathbf{fix}\ \mathcal{D}(P)$ *is a closure operator.*

Proof. $\mathcal{D}(P)$ is extensive and monotone as *embed* and $\sqcup$ are extensive and monotone operations. Moreover, *Store* is a complete lattice as the Cartesian product of complete lattices yields a complete lattice. By Tarski fixed point theorem, any monotone function on a complete lattice has a least fixed point. This shows $\mathcal{D}(P)$ is completely described by its fixed points, thus $\mathbf{fix}\ \mathcal{D}(P)$ is idempotent and is a closure operator since $\mathcal{D}(P)$ is already extensive and monotone. $\square$

The claim made earlier that we can write PCCP programs with the same hypothesis than in sequential programs can be made more precise now. Given a process P, if we replace all parallel compositions by sequential compositions (a transformation we write $\texttt{seq}\ P$), then the denotation of both programs remains the same. We define sequential composition as:

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

Proposition 3. $\mathbf{fix}\ \mathcal{D}(\texttt{seq}\ P) = \mathbf{fix}\ \mathcal{D}(P)$

Proof. Firstly, we notice that Theorem 2 still holds for sequential programs as the functional composition preserves extensiveness and monotonicity. Hence both $\mathcal{D}(\texttt{seq}\ P)$ and $\mathcal{D}(P)$ have a least fixed point, and we must show they coincide. It boils down to proving the fixed points of $f \circ g$ equals the fixed points of $f \sqcup g$. Because $f \leq f \circ g$ and $g \leq f \circ g$, necessarily we have $f \sqcup g \leq f \circ g$, and by monotonicity of f and g, $\mathbf{fix}\ f \sqcup g \leq \mathbf{fix}\ f \circ g$. The other direction is shown by $(f \sqcup g) \circ (f \sqcup g) \geq f \circ g$ which holds because $f \sqcup g \geq g$ and $f \circ (f \sqcup g) \geq f \circ g$. Moreover, by monotonicity of f and g, we have $(f \sqcup g) \circ (f \sqcup g) \geq f \circ g \Rightarrow$ $\mathbf{fix}\ (f \sqcup g) \circ (f \sqcup g) \geq \mathbf{fix}\ f \circ g \Leftrightarrow \mathbf{fix}\ f \sqcup g \geq \mathbf{fix}\ f \circ g$. Therefore, $\mathbf{fix}\ f \sqcup g = \mathbf{fix}\ f \circ g$. $\square$

A practical implication of this proposition is that we are not forced to fully parallelize a PCCP program, and can choose, when appropriate, to use a sequential composition. This choice depends on the number of threads available on a particular hardware, and how costly it is to start a thread. Moreover, the proof highlights that executing once all processes sequentially always leads to a stronger store than executing once all processes in parallel. In practice, however, a sequential iteration may take longer to complete than a parallel iteration since it can only be executed on a single core.

The operational semantics of PCCP is more compactly described on a *guarded normal form* (GNF), which is obtained by lifting all parallel compositions at top-level. We obtain a set of guarded commands of the form $\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)$ that can be executed in parallel. This program transformation is obtained by the following function:

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

This function essentially duplicates the ask conditions to lift nested parallel processes. We note that a guarded command $\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_n)$ is just a compact syntax for the PCCP program $\texttt{if}\ b_1\ \texttt{then}\ \ldots\ \texttt{if}\ b_n\ \texttt{then}\ x \leftarrow f(y_1, ..., y_n)$. Next, we show *gnf* preserves fixed points.

Proposition 4. $\mathbf{fix}\ \mathcal{D}(gnf(\{\}, P)) = \mathbf{fix}\ \mathcal{D}(P)$

Proof. The ask operation distributes over parallel composition, *i.e.*, $\mathbf{fix}\ \mathcal{D}(\texttt{if}\ b\ \texttt{then}\ (P \parallel Q))$ equals $\mathbf{fix}\ \mathcal{D}(\texttt{if}\ b\ \texttt{then}\ P \parallel \texttt{if}\ b\ \texttt{then}\ Q)$. The full proof is obtained by structural induction over the processes in *gnf*. $\square$

We describe the operational semantics of PCCP using a transition function $\hookrightarrow$ between states. A state is a couple $\langle s, G\rangle$ where s is the store and G a set of guarded commands. The operational semantics can be specified using a single rule on PCCP programs in GNF:

$$\textsc{select}\quad \frac{(\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)) \in G \qquad \bigwedge_{i \leq n} s(b_i)}{\langle s, G\rangle \hookrightarrow \langle embed_x(s, f(s(y_1), \ldots, s(y_m))), G\rangle}$$

The execution of a PCCP program is a possibly infinite sequence of states $\langle s_1, G\rangle \hookrightarrow \ldots \hookrightarrow \langle s_n, G\rangle \hookrightarrow \ldots$. In each transition, we select one process to be executed. The transition function is monotone and extensive because $embed_x$ is monotone and extensive, and f and the condition $\bigwedge_{i \leq n} s(b_i)$ are monotone.

In the next section, we motivate why this seemingly sequential operational semantics is actually a correct representation of a low-level parallel execution. Before that, we must discuss the nondeterminism of the rule SELECT which allows us, at each step, to execute any process in G. Nondeterminism is important to avoid forcing a particular scheduling strategy of the processes in the semantics. The only requirement is fairness, otherwise the same process could be selected over and over again.

Definition 5 (Fairness)**.** A scheduling strategy is fair if, for each process $P \in G$, it generates transitions such that:

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

A result of Cousot and Cousot (1977b) on chaotic iterations, applied to constraint programming by Apt (1999), guarantees that the limit of all fair scheduling strategies coincide. When the fixed point of $\hookrightarrow$ is reachable in a finite number of steps, which is the case if the lattices underlying the store are finite[2], the choice of a fair scheduling strategy

[2]Or satisfy the ascending chain condition, *i.e.*, for all chains $x_1 \leq \ldots \leq x_n \leq \ldots$ there exists $i \in \mathbb{Z}$ such that $x_i = x_{i+1}$.

has no impact on the result computed. This is particularly important in practice as from one parallel hardware to another, or even from one execution to another, the scheduling strategy might change.

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 6. $\mathbf{fix}\ \mathcal{D}(P) = \mathbf{fix}\ \mathcal{O}(P)$

Proof. By Proposition 4, we consider the equivalent process $Q = gnf(\{\}, P)$, and by Proposition 3, the denotation of the equivalent process $\texttt{seq}\ Q$. The function $\mathcal{D}(\texttt{seq}\ Q)$ computes the functional composition of each guarded command in sequential order. This corresponds to a particular fair scheduling strategy. By Apt (1999), the fixed points of any fair execution of monotone functions coincide, and therefore it coincides with the operational execution $\mathbf{fix}\ \mathcal{O}(P)$. Therefore, $\mathbf{fix}\ \mathcal{D}(P) = \mathbf{fix}\ \mathcal{O}(P)$. □

Load/Store Semantics

We show our semantic correct w.r.t. a *weak memory consistency model* (see, *e.g.* (Nagarajan et al. 2020)). Consistency defines a correct behavior in terms of loads and stores in shared memory. Load and store instructions are written $\mathbf{L}\ a\ r$ and $\mathbf{S}\ r\ a$ where a is the location of a variable in shared memory and r refers to a thread-local memory location such as a register. For both instructions, the direction of the memory movement is from the first to the second argument. To investigate the low-level details of execution, we compile a guarded command $\{b_1, \dots, b_n\} \Rightarrow x \leftarrow f(y_1, ..., y_m)$ to a sequence of loads and stores:

```
1  while true then
2      L $b_1$ $rb_1$; ...; L $b_n$ $rb_n$;
3      if $rb_1 \wedge \ldots \wedge rb_n$ then
4          L $y_1$ $ry_1$; ...; L $y_m$ $ry_m$;
5          $\mathcal{C}(f(ry_1, \ldots, ry_m), rf)$;
6          L $x$ $rx$;
7          $\mathcal{C}(rx \sqcup rf, ox)$;
8          $\mathcal{C}(ox > rx, bx)$;
9          if $bx$ then
10             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 $\sqcup$ and $>$ are compiled according to the lattice-type of the variable x. Although the compilation of a guarded command is a sequential program, only a few of the program order dependencies are important.

Lemma 7. *The only important program order (po) dependencies are* $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 9 \xrightarrow{po} 10$, $1 \xrightarrow{po} 6 \xrightarrow{po} 7$, *and* $1 \xrightarrow{po} 4 \xrightarrow{po} 5 \xrightarrow{po} 7 \xrightarrow{po} 8 \xrightarrow{po} 9$.

This lemma can be used by compilers to improve efficiency by reordering some of the instructions.

We now define the load/store operational semantics. The difference with the operational semantics of the previous section is that we have an assignment operator which does not perform a join in the store. The assignment is written $s[a \mapsto b]$, meaning the value of a is replaced by $s(b)$ in the store s. We suppose the variable r is local to a thread.

LOAD
$$\langle s, \mathbf{L}\ a\ r; P\rangle \twoheadrightarrow \langle s[r \mapsto a], P\rangle$$

STORE
$$\langle s, \mathbf{S}\ r\ a; P\rangle \twoheadrightarrow \langle s[a \mapsto r], P\rangle$$

SELECT2
$$\frac{I \in GC \qquad \langle s, I\rangle \twoheadrightarrow \langle s', I'\rangle}{\langle s, GI\rangle \Rightarrow \langle s', (GI \setminus \{I\}) \cup \{I'\}\rangle}$$

We skip the operational semantics of **if** and **while** which is standard, and focus on the execution of a sequence of load and store instructions. The rule SELECT2 is similar to SELECT but executes instructions at a finer grain. Each step of the transition $\Rightarrow$ models an atomic action on the shared memory. The load/store operational semantics, and the proofs of correctness below, make the following assumptions on the memory consistency model and cache coherency protocol:

(PO) The program order of Lemma 7 must be enforced.

(ATOM) Load and store instructions must be atomic.

(EC) The caches must eventually become coherent.

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

(**PO**) is merely a consequence of the Von Neumann model of computation. (**ATOM**) is an important assumption to avoid torn reads and writes, *e.g.*, a thread writes the two first bytes of a 32 bits integer and another thread writes the two last bytes. Such a data race is undefined behavior in many languages, *e.g.*, C++ (ISO 2020) (§6.9.2.1) and CUDA (Lustig, Sahasrabuddhe, and Giroux 2019), which is why we must prevent it. (**EC**) is relevant to cache coherency and supposes the threads eventually view the same value of any variable after a finite number of loads. In other terms, a thread cannot work on its own view of the world forever, and communication must happen at one point in time. (**OTA**) is a common assumption to avoid a thread to speculatively write a value which might be wrongly read by another thread (Boehm and Demsky 2014). In addition, and although hardware vendors seldom specify the scheduling strategy employed, we make the additional fairness assumption (defined similarly to Def. 5):

(FAIR) The thread scheduling strategy must be fair.

We now prove soundness and completeness results. Soundness implies the load/store semantics of a PCCP program does not generate more information than the operational semantics, and completeness that it does not generate less information.

Theorem 8 (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 $s_i \leq (\mathbf{fix}\ \mathcal{O}(P))(s_1)$.*

Proof. By (**ATOM**), we must have $s_k(a) = s_k(r)$ after the execution of any instruction $\mathbf{S}\ a\ r$ or $\mathbf{L}\ r\ a$. Moreover, the register r can only evolve monotonically as it is local to a thread and is only modified by monotone operations. Therefore, we have at each step k, $s_k(r) = s_k(a) \leq (\textbf{fix}\ \mathcal{O}(P))(s_i)(a)$ for all variables a. By induction on $\Rightarrow$, it is necessary that $s_i \leq (\textbf{fix}\ \mathcal{O}(P))(s_1)$. □

Because of the assignment operator, the transition $\Rightarrow$ is neither extensive nor monotone. However, we can still show that $\Rightarrow$ converges to the fixed point of $\mathcal{O}(P)$.

Theorem 9 (Completeness)**.** *Suppose the fixed point $os = (\textbf{fix}\ \mathcal{O}(P))(s_1)$ 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 $s_i \geq os$. Further, for all $j \in \mathbb{N}$, $j > i$, $s_i = s_j$.*

Proof. We first show that $\Rightarrow$ eventually makes *progress*. Because of the fairness condition (**FAIR**), we must reach a state $\langle s_k, GI\rangle$ such that all instructions have been executed at least once. If no store operation was executed, then $s_k = os$ because all tell operations are at a fixed point. Otherwise, at least one store operation on a variable x has been executed, and due to the condition on lines 8–9 ($ox > rx$) and (**ATOM**), we necessarily have $s_k(x) > s_1(x)$. This guarantees the progress of the transition $\Rightarrow$ after k steps.

Now, if $s_k < os$, it means that $\mathcal{O}(P)$ is not at a fixed point on s_k, and therefore there is a transition $\langle s_k, G\rangle \hookrightarrow \langle s_{k+1}, G\rangle$ such that $s_k < s_{k+1}$. Suppose the transition executes the process $\{b_1, \ldots, b_n\} \Rightarrow x \leftarrow f(y_1, \ldots, y_m)$. By (**EC**), we must reach a state in which the latest values of $b_1, \ldots, b_n$ are visible to the thread, and the condition on line 3 is true. Similarly for the values $y_1, \ldots, y_m$ in which case, by (**PO**), $f(ry_1, \ldots, ry_m)$ (line 5) and $rx \sqcup rf$ (line 7) must lead to the same result than the one given in the operational semantics. At that point, $ox > rx$ must hold by hypothesis and a store operation must be issued. Therefore, by induction on the finite sequence of operational transitions, we must have $s_i \geq os$.

The fact that $s_i = s_j$ for all $j > i$ is a consequence of Theorem 8. □

Theorems 8 and 9 show that the result of the low-level parallel execution in terms of loads and stores coincides with the fixed point of the operational semantics, when it is reachable in a finite number of steps (which is the only behavior we are interested by). The *asynchronous iterations* of Cousot (1977) generalizes this result, although in a more abstract mathematical setting, to the case of infinite sequences.

We address more practical aspects of the computation such as fixed point and failure detections in the next section.

Turbo: GPU Constraint Solver

TURBO is a constraint solver implementing *entirely on GPU* a standard propagate and search algorithm (Tack 2009). Some of the design choices of TURBO are direct consequences of the TURING and newer architectures of NVIDIA GPUs which we review now. To illustrate our explanations, we take the example of the laptop GPU NVIDIA QUADRO RTX 5000 MAX-Q, which we later use to perform our experiments. It has 48 *streaming multiprocessors* (SMs), each with 64KB of L1 cache and 64 cores. There is a total of $48 * 64 = 3072$ cores. When programming in CUDA, we group parallel processes in *blocks* that are executed on a single SM, and have their own intra-block synchronization primitives. In particular, we will use the barrier instruction `__syncthread()` which forces all threads of a block to wait for each other before continuing. Threads in a block can communicate through up to 48KB of *shared memory*, a very fast on-chip memory local to a block. Finally, we have 16GB of *global memory* shared among all SMs, which can be used for inter-blocks communication.

In order to increase the solving efficiency, the design of TURBO attempts to match the architecture of GPUs. Firstly, we completely avoid communication latency with the CPU as the full solving algorithm is on the GPU. Secondly, we dynamically generate subproblems following a variant of embarrassingly parallel search (EPS) (Malapert, Régin, and Rezgui 2016). Each subproblem is then solved by a block on a single SM, which limits data movements between caches and global memory. To maximize the utilization of the cache, avoid costly memory allocation and synchronization among threads, we have minimal data structures. Propagation is *eventless* (Mackworth 1977), hence we do not need to maintain a queue of propagators to be executed (Schulte and Stuckey 2008) or take care of synchronization and workload balancing issues (Rolf and Kuchcinski 2009). State restoration during backtracking is done by full recomputation (Schulte 1999). Each block maintains two stores of variables: one for the root node of the subproblem and one for the computation. On backtracking, the root node of the subproblem is copied in the current store, and the branching decisions of the new path to explore are committed into the store. Therefore, we avoid maintaining a trailing data structure which would induce complicated thread synchronization. Adaptative recomputation (Schulte 1999) incurs copies of the stores which are not cache friendly, hence we chose to avoid extra copies completely. This design is simple, yet efficient as shown below.

Lustig, Sahasrabuddhe, and Giroux (2019) formalizes the memory consistency model of PTX, the instruction set architecture of NVIDIA GPUs. The assumptions made by the load/store semantics are correct w.r.t. the PTX memory consistency model. Indeed, Lustig, Sahasrabuddhe, and Giroux (2019) introduce 6 axioms which match or strengthen our assumptions. In particular, the *SC-per-location axiom* implies (**PO**) as it states "morally strong [...] communication order cannot contradict program order". The *no-thin-air axiom* matches the (**OTA**) assumption, as both target the same issue. Eventual cache coherence (**EC**) is enforced at the level of blocks because threads in a block share the same cache, therefore observe any store operation directly. The remaining assumption (**ATOM**) is slightly more tricky. When considering the CUDA programming model, store and load are not necessarily atomic. Fortunately, NVIDIA recently implemented the C++11 atomic library for CUDA[3], and therefore we simply need to wrap shared integer and Boolean variables in a templated class `atomic<T>`. Another way to fulfil the (**ATOM**) assumption is to store the shared variables in the *shared memory*. Load and store accesses to

[3]https://nvidia.github.io/libcudacxx/

solver	feas.	opt.	nodes-per-sec.	time (sec.)
Patterson	110	110	-	-
Turbo	110	108	473k	1362s
GeCode	109	103	49k	3812s
j30	480	480	-	-
Turbo	379	309	423k	5349s
GeCode	376	307	63k	5361s

Table 1: Comparing Turbo to GeCode on scheduling problems.

shared memory are sequentialized[4], and therefore satisfy the **(ATOM)** assumption. However, this method is less generic than atomics and is proper to the hardware architecture of recent GPUs. Nevertheless, in our experiments, we rely on the shared memory as we measured it to be the fastest solution.

Fixed point loop We now present the propagation loop with fixed point and failure detections, improving on the `while true` loop of the guarded commands compilation. We define the propagation loop as a function `void propagation(int tid, int stride, VStore& s, Vector<Propagator*>& props)` where `tid` is a unique thread identifier, `stride` is the number of threads in the block, `s` is a store of interval variables and `props` is the array of propagators (which are PCCP processes):

```
__shared__ bool has_changed[3] = {true};
for(int i = 1; !s.is_top() &&
    has_changed[(i-1)%3]; ++i) {
  for (int t = tid; t < props.size(); t
      += stride){
    if(props[t]->propagate(s))
      has_changed[i%3] = true; }
  has_changed[(i+1)%3] = false;
  __syncthreads();
}
```

The method `s.is_top()` is `true` whenever an interval is empty in the store, hence detecting if a failure occurred. Fixed point detection is done using three Boolean variables `has_changed[3]`. In each iteration, we manipulate the past, present and future values of `has_changed`. First, we continue to iterate if a variable changed in the previous iteration (`has_changed[(i-1)%3]`). Second, we set the current value to true if a propagator successfully changes a domain (`has_changed[i%3]`). Note that for each guarded command, the Boolean value bx indicates if a change occurred. Thirdly, we must initialize to false the value of the next iteration. At the end of each iteration, all threads reach a barrier, thus the current and next value of `has_changed` is correctly set.

This propagation loop algorithm is reminiscent of the AC-1 algorithm (Mackworth 1977) which iterates until arc consistency is achieved on all constraints. The similarity only concerns the propagation loop as we do not control the level of consistency achieved by the propagators here.

[4]https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory

Evaluation We evaluate TURBO[5] on the RCPSP problem introduced above. As it is common knowledge in CUDA, it is best to over-saturate the SMs with blocks, and over-saturate the blocks with threads. We have identified that multiplying by 4 the SMs—to obtain 192 blocks—and cores on SMs—to obtain 256 threads—yield the best efficiency on this particular GPU model. We compare TURBO with the well-known constraint solver GECODE 6.2.0 (Schulte, Tack, and Lagerkvist 2020) in parallel mode on a processor i7-10750@2.60GHz with 6 cores and 12 threads. The constraint model and search strategy are the same for both solvers. We experiment on 2 data sets: Patterson (Patterson 1984) (timeout of 5 minutes) which has instances with various numbers of tasks and resources, and j30 from PSPSLIB (Kolisch and Sprecher 1997) (timeout of 30 seconds) with 30 tasks and 4 resources. The results are presented in Table 1. TURBO positively compares to GeCode, being slightly better on both data sets. An important aspect is that TURBO processes up to an order of magnitude more nodes per seconds than GeCode. Interestingly, GeCode failed on three instances due to an assertion relevant to the parallel code. This shows the difficulty to code correctly multithreaded program, hence one advantage of the formally correct and simpler design of TURBO. Our goal with these experiments was only to show a GPU-based solver can be competitive, but there are methods and solvers more efficient on RCPSP as shown by Schutt et al. (2013).

Related Work and Discussion

We are not aware of similar work connecting concurrent constraint programming and the low-level aspects of parallel programming. Therefore, we primarily compare parallel constraint solvers to TURBO.

The literature on constraint solving on GPUs is scarce. Actually, as far as we know, NVIDIOSO (Campeotto et al. 2014a) is the only propagate and search constraint solver running on GPU using CUDA. A difference with TURBO is that NVIDIOSO relies on the CPU for the backtracking and propagation of some constraints, whereas one guiding principle behind TURBO is to only rely on the GPU. Moreover, in contrast to our work, the search component is not parallelized. PHACT is another parallel constraint solver running on heterogeneous architectures (CPU, GPU and others) using OPENCL (Roque et al. 2018). However, it is not clear from the paper if and how propagation is parallelized.

A larger number of works is available when considering other solving algorithms or CPU parallelism (Hamadi and Sais 2018); we only overview a few selected ones. Local search algorithms have been successfully implemented on GPUs, showcasing speed-up of one order of magnitude in comparison to sequential versions (Arbelaez and Codognet 2014; Campeotto et al. 2014b). However, local search is not an exact method, which means there is no guarantee to find the best solution. Fioretto et al. (2018) successfully apply a *dynamic programming* solving algorithm to GPUs. Their results are very encouraging and lead to speed-up of

[5]**Replicate**: the version of TURBO used in this paper is available at https://github.com/ptal/turbo/tree/aaai2022.

two orders of magnitude. However, as the authors warn us, this dynamic programming approach can require exponential time and space, and is only suited for specific problems. In the case of CPU parallelism, the traditional approach is to parallelize the search component, see for instance Perron (1999) and Schulte (2000). TURBO relies on EPS which is a recent and robust search parallelization method (Malapert, Régin, and Rezgui 2016). The class of *distributed constraint satisfaction problems* is mostly relevant in a distributed computing context where agents communicate by message-passing (Yokoo and Hirayama 2000), whereas we situate ourselves in a shared-state memory model. We refer to the survey of Gent et al. (2018) for additional references.

It is worth noting that parallelizing propagation and search dates back to the eighties, with parallel logic programming languages. Conjunctive and disjunctive logic predicates can be executed in parallel, similarly to the propagate and search components. Gupta et al. (2001) survey the parallelization of logic programming languages.

Abstract interpretation (Cousot and Cousot 1977a), and its recent application to constraint reasoning (D'Silva, Haller, and Kroening 2014; Cousot, Cousot, and Mauborgne 2013; Pelleau et al. 2013), is deeply connected to PCCP, as both are based on lattice theory. We have purposely avoided to introduce PCCP in the framework of abstract interpretation to keep the paper brief. This framework would allow us to prove more properties, in particular when considering lattices that cannot be exactly represented by a machine; for instance the lattice of intervals of real numbers is approximated by the lattice of intervals of floating-point numbers. Moreover, in the context of abstract interpretation, Kim, Venet, and Thakur (2020) introduce a method to compute deterministically a fixed point in parallel. In our terms, they build a dependency graph among propagators and statically generate a scheduling of the propagators. It improves our naive propagation loop where propagators are all executed, even those that are already at fixed points. An adaptation of this work to constraint reasoning and GPUs is an interesting lead to improve the efficiency of the propagation loop.

Conclusion

We have laid the mathematical and practical foundation of PCCP, a new parallel programming language, useful for correctly computing fixed points in parallel. We applied this paradigm to constraint solving with TURBO a parallel GPU-based constraint solver. We experimentally validated TURBO on a scheduling problem, where the propagation and search are parallelized, and showed it competes positively with a CPU-based constraint solver. This work is only a first step towards a full-fledged GPU-based constraint solver. In particular, constraint learning techniques such as lazy clause generation (Ohrimenko, Stuckey, and Codish 2009) pose additional challenges for their executions on GPUs, as already investigated in SAT solvers (Hamadi and Sais 2018). Moreover, while some global constraints such as `table` natively support parallelization (Campeotto et al. 2014a), the applicability of PCCP for programming other global constraints remains to be shown. We intend to explore a parallelization of the `range` and `roots` constraints which can construct many other global constraints (Bessiere et al. 2009).

Acknowledgments

We are grateful to the reviewers for the useful comments, and the reference to the AC-1 algorithm.

References

2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. https://doi.org/10.1109/IEEESTD.2019.8766229.

2020. Programming languages—C++. *ISO/IEC 14882:2020*.

Apt, K. R. 1999. The essence of constraint propagation. *Theoretical computer science*, 221(1-2): 179–210. https://doi.org/10.1016/S0304-3975(99)00032-8.

Arbelaez, A.; and Codognet, P. 2014. A GPU Implementation of Parallel Constraint-Based Local Search. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 648–655. Torino, Italy: IEEE. https://doi.org/10.1109/PDP.2014.28.

Bessiere, C.; Hebrard, E.; Hnich, B.; Kiziltan, Z.; and Walsh, T. 2009. Range and Roots: Two common patterns for specifying and propagating counting and occurrence constraints. *Artificial Intelligence*, 173(11): 1054–1078. https://doi.org/10.1016/j.artint.2009.03.001.

Boehm, H.-J.; and Demsky, B. 2014. Outlawing ghosts: avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, 1–6. Edinburgh United Kingdom: ACM. https://doi.org/10.1145/2618128.2618134.

Campeotto, F.; Dal Palu, A.; Dovier, A.; Fioretto, F.; and Pontelli, E. 2014a. Exploring the use of GPUs in constraint solving. In *International Symposium on Practical Aspects of Declarative Languages*, 152–167. Springer. https://doi.org/10.1007/978-3-319-04132-2.

Campeotto, F.; Dovier, A.; Fioretto, F.; and Pontelli, E. 2014b. A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems. In *European Conference on Artificial Intelligence*, 189–194. https://doi.org/10.3233/978-1-61499-419-0-189.

Carlson, B.; Carlsson, M.; and Diaz, D. 1994. Entailment of finite domain constraints. In *International Conference on Logic Programming*. https://doi.org/10.7551/mitpress/4316.003.0038.

Carlsson, M.; Ottosson, G.; and Carlson, B. 1997. An open-ended finite domain constraint solver. *Programming Languages: Implementations, Logics, and Programs*. https://doi.org/10.1007/bfb0033845.

Cousot, P. 1977. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Research Report 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France.

Cousot, P.; and Cousot, R. 1977a. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings* *of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 238–252. ACM. https://doi.org/10.1145/512950.512973.

Cousot, P.; and Cousot, R. 1977b. Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, 1–12. New York, NY, USA: ACM. https://doi.org/10.1145/872734.806926.

Cousot, P.; Cousot, R.; and Mauborgne, L. 2013. Theories, Solvers and Static Analysis by Abstract Interpretation. *Journal of the ACM*, 59(6). https://doi.org/10.1145/2395116.2395120.

Davey, B. A.; and Priestley, H. A. 2002. *Introduction to lattices and order*. Cambridge University Press. https://doi.org/10.1017/cbo9780511809088.

D'Silva, V.; Haller, L.; and Kroening, D. 2014. Abstract satisfaction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, 139–150. San Diego, California, USA: ACM Press. https://doi.org/10.1145/2535838.2535868.

Fioretto, F.; Pontelli, E.; Yeoh, W.; and Dechter, R. 2018. Accelerating exact and approximate inference for (distributed) discrete optimization with GPUs. *Constraints*, 23(1): 1–43. https://doi.org/10.1007/s10601-017-9274-1.

Gent, I. P.; Miguel, I.; Nightingale, P.; McCreesh, C.; Prosser, P.; Moore, N. C.; and Unsworth, C. 2018. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6): 725–758. https://doi.org/10.1017/S1471068418000340.

Gupta, G.; Pontelli, E.; Ali, K. A.; Carlsson, M.; and Hermenegildo, M. V. 2001. Parallel execution of Prolog programs: a survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4): 472–602. https://doi.org/10.1145/504083.504085.

Hamadi, Y.; and Sais, L., eds. 2018. *Handbook of Parallel Constraint Reasoning*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-63516-3.

Kim, S. K.; Venet, A. J.; and Thakur, A. V. 2020. Deterministic parallel fixpoint computation. *Proceedings of the ACM on Programming Languages*, 4(POPL): 1–33. https://doi.org/10.1145/3371082.

Kolisch, R.; and Sprecher, A. 1997. PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research*, 96(1): 205–216. https://doi.org/10.1016/S0377-2217(96)00170-1.

Lee, E. A. 2006. The Problem with Threads. *Computer*, 39(5): 33–42. https://doi.org/10.1109/MC.2006.180.

Lustig, D.; Sahasrabuddhe, S.; and Giroux, O. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 257–270. Providence RI USA: ACM. https://doi.org/10.1145/3297858.3304043.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial intelligence*, 8(1): 99–118. https://doi.org/10.1016/0004-3702(77)90007-8.

Malapert, A.; Régin, J.-C.; and Rezgui, M. 2016. Embarrassingly Parallel Search in Constraint Programming. *Journal of Artificial Intelligence Research*, 57: 421–464. https://doi.org/10.1613/jair.5247.

Monette, J.-N.; Flener, P.; and Pearson, J. 2012. Towards Solver-Independent Propagators. In *Principles and Practice of Constraint Programming*, 544–560. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33558-7_40.

Nagarajan, V.; Sorin, D. J.; Hill, M. D.; and Wood, D. A. 2020. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture*, 15(1): 1–294. https://doi.org/10.2200/S00962ED2V01Y201910CAC049.

Narodytska, N. 2011. *Reformulation of global constraints*. Ph.D. thesis, University of New South Wales, Sydney, Australia.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming—CP 2007*, 529–543. Springer. https://doi.org/10.1007/978-3-540-74970-7_38.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via Lazy Clause Generation. *Constraints*, 14(3): 357–391. https://doi.org/10.1007/s10601-008-9064-x.

Patterson, J. H. 1984. A Comparison of Exact Approaches for Solving the Multiple Constrained Resource, Project Scheduling Problem. *Management Science*, 30(7): 854–867. https://doi.org/10.1287/mnsc.30.7.854.

Pelleau, M.; Miné, A.; Truchet, C.; and Benhamou, F. 2013. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation*, 434–454. Springer. https://doi.org/10.1007/978-3-642-35873-9_26.

Perron, L. 1999. Search procedures and parallelism in constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, 346–360. Springer. https://doi.org/10.1007/978-3-540-48085-3_25.

Rolf, C. C.; and Kuchcinski, K. 2009. Parallel Consistency in Constraint Programming. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009*, 638–644.

Roque, P.; Pedro, V.; Diaz, D.; and Abreu, S. 2018. Improving Constraint Solving on Parallel Hybrid Systems. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, 726–732. IEEE. https://doi.org/10.1109/ICTAI.2018.00114.

Saraswat, V. A. 1993. *Concurrent constraint programming*. ACM Doctoral dissertation awards. MIT Press. https://doi.org/10.7551/mitpress/2086.001.0001.

Saraswat, V. A.; Rinard, M.; and Panangaden, P. 1991. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, 333–352. New York, NY, USA: ACM. https://doi.org/10.1145/99583.99627.

Schulte, C. 1999. Comparing trailing and copying for constraint programming. In *In Proceedings of the International Conference on Logic Programming*, 275–289. The MIT Press. https://doi.org/10.7551/mitpress/4304.003.0027.

Schulte, C. 2000. Parallel Search Made Simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, 41–57.

Schulte, C.; and Stuckey, P. J. 2008. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1): 2:1–2:43. https://doi.org/10.1145/1452044.1452046.

Schulte, C.; Tack, G.; and Lagerkvist, M. 2020. *Modeling and Programming with Gecode*.

Schutt, A.; Feydy, T.; Stuckey, P. J.; and Wallace, M. G. 2009. Why cumulative decomposition is not as bad as it sounds. In *International Conference on Principles and Practice of Constraint Programming*, 746–761. Springer. https://doi.org/10.1007/978-3-642-04244-7_58.

Schutt, A.; Feydy, T.; Stuckey, P. J.; and Wallace, M. G. 2013. Solving RCPSP/max by lazy clause generation. *Journal of Scheduling*, 16(3): 273–289. https://doi.org/10.1007/s10951-012-0285-x.

Tack, G. 2009. *Constraint Propagation – Models, Techniques, Implementation*. Ph.D. thesis, Saarland University.

Thompson, N. C.; and Spanuth, S. 2021. The Decline of Computers as a General Purpose Technology. *Communications of the ACM*, 64(3): 64–72. https://doi.org/10.1145/3430936.

Van Hentenryck, P.; Saraswat, V.; and Deville, Y. 1998. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3): 139–164. http://doi.org/10.1016/S0743-1066(98)10006-7.

Yokoo, M.; and Hirayama, K. 2000. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2): 185–207. https://doi.org/10.1023/A:1010078712316.