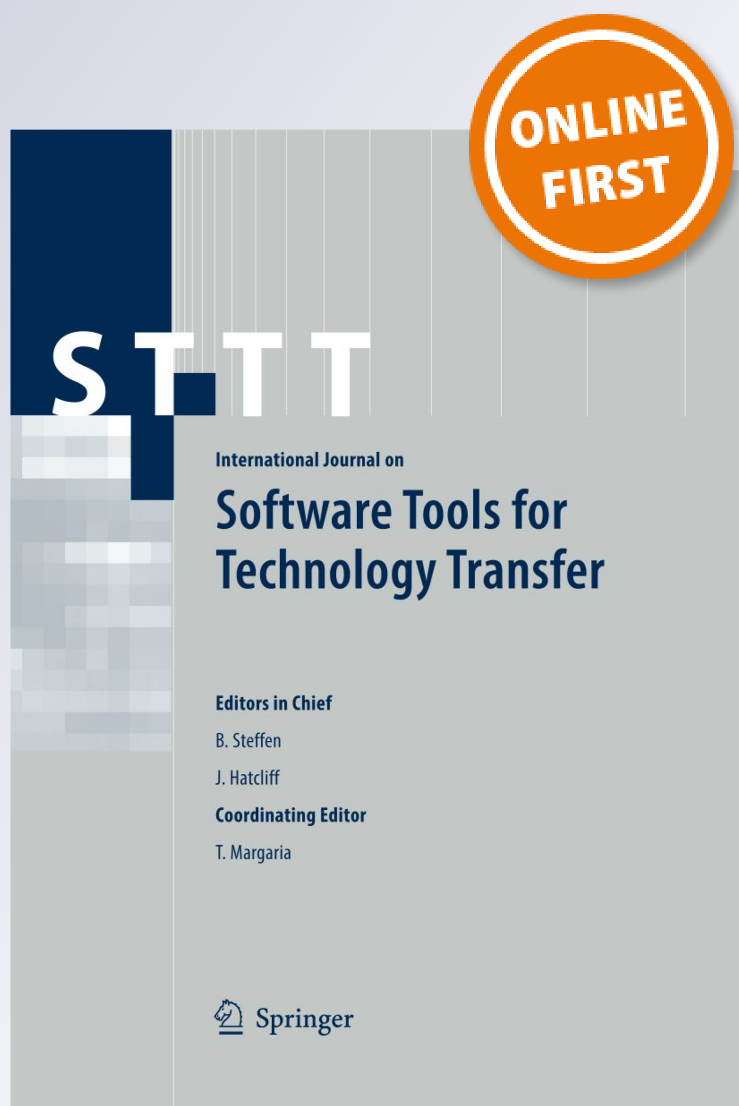
Formal specification and implementation of an automated pattern-based parallel-code generation framework

Gervasio Pérez & Sergio Yovine

International Journal on Software Tools for Technology Transfer

ISSN 1433-2779

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0465-2



Springer

Your article is protected by copyright and all rights are held exclusively by Springer-Verlag GmbH Germany. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0465-2

CrossMark

REGULAR PAPER

Formal specification and implementation of an automated pattern-based parallel-code generation framework

Gervasio Pérez[1] · **Sergio Yovine**[1]

© Springer-Verlag GmbH Germany 2017

Abstract Programming correct parallel software in a cost-effective way is a challenging task requiring a high degree of expertise. As an attempt to overcoming the pitfalls undermining parallel programming, this paper proposes a pattern-based, formally grounded tool that eases writing parallel code by automatically generating platform-dependent programs from high-level, platform-independent specifications. The tool builds on three pillars: (1) a platform-agnostic parallel programming pattern, called `PCR`, (2) a formal translation of `PCRs` into a parallel execution model, namely Concurrent Collections (`CnC`), and (3) a program rewriting engine that generates code for a concrete runtime implementing `CnC`. The experimental evaluation carried out gives evidence that code produced from `PCRs` can deliver performance metrics which are comparable with handwritten code but with assured correctness. The technical contribution of this paper is threefold. First, it discusses a parallel programming pattern, called `PCR`, consisting of *producers*, *consumers*, and *reducers* which operate concurrently on data sets. To favor correctness, the semantics of `PCRs` is mathematically defined in terms of the formalism `FXML`. `PCRs` are shown to be composable and to seamlessly subsume other well-known parallel programming patterns, thus providing a framework for heterogeneous designs. Second, it formally shows how the `PCR` pattern can be correctly implemented in terms of a more concrete parallel execution model. Third, it proposes a platform-agnostic `C++` template library to express `PCRs`. It presents a prototype source-to-source compilation tool, based on `C++` template rewriting, which automatically generates parallel implementations relying on the Intel `CnC` `C++` library.

Keywords Formal methods · Software design patterns · Parallel programming · Automated code generation

1 Introduction

Issues related to the physics of processor design have made hardware industry to shift from improving the speed of a single processor to increasing the number processing cores [4]. This paradigm change puts software engineering in front of the challenging task of providing appropriate tools for effectively building software that correctly and efficiently exploits parallel processing power. Indeed, besides the well-known, inherent pitfalls of concurrent programming, such as deadlocks and data races, which are the cause of numerous bugs [29], developing software for multicore hardware demands taking care of different parallel patterns and execution models [30], and integrating legacy code which cannot always be easily or simply rewritten from scratch [8]. This complexity makes engineering correct and efficient parallel software to require a high degree of expertise.

The aforementioned situation creates a need of techniques and tools that ease building parallel software in a cost-effective way. This paper looks forward contributing in that direction by following a theory-based practical approach. More precisely, our work relies on two principles. First, parallel software should be designed in a platform-independent way, so as the same piece of software could end up running either in a many-core server, a cluster of inexpensive

✉ Gervasio Pérez
gdperez@dc.uba.ar
http://lafhis.dc.uba.ar/~gperez

Sergio Yovine
syovine@dc.uba.ar
http://lafhis.dc.uba.ar/~syovine

[1] ICC-CONICET and Universidad de Buenos Aires, Ciudad Autónoma de Buenos Aires, Argentina

Published online: 01 August 2017

Author's personal copy

nodes, or a processor grid. Second, tuning for a concrete execution model should be done by formally translating such a platform-independent design into a specific solution. Platform-specific and environment characteristics are to be factored in at relevant phases of a formally grounded code generation process. Relying on sound theoretical bases guarantees correctness.

A key aspect is the use of *abstract* program descriptions. According to [6], parallel programming methodologies can be categorized by their level of abstraction as follows. Thread management and synchronization primitives together with IPC mechanisms (e.g., MPI [22]) are considered to be *low*-level abstractions. Language extensions (e.g., OpenMP [13] and Cilk [7]) and frameworks (e.g., TBB [31], TPL [28] and `CnC` [9]) provide a *middle* level of abstraction by relieving part but not all of the coordination and synchronization efforts from the programmer. Similarly, we could argue that emerging parallel programming languages such as X10 [32] and Chapel [10] belong to this category.[1] *Pattern-based* parallel-software design [12,20,30], also known as *structured* parallelism or algorithmic *skeletons*, provides *high*-level parallel programming abstractions. They consist of common constructs that hide from the programmer all low-level coordination and synchronization mechanisms which are necessary to perform the actual parallel execution. This is done by mapping, at compile time and/or at runtime, the high-level abstractions into middle/low-level libraries and/or language constructs. It has been shown that resorting to structured parallelism allows harnessing the computing power of heterogeneous architectures [19].

Numerous works advocate using *pattern-based* parallel programming [20]. Among the most recent and successful ones, we should cite [1,3,14,16,18,27,40]. Despite their contribution to the field, it is worth observing that they have some drawbacks. First, they provide little or no formal foundations. Notable exceptions are [3,17,27], which give abstract semantics to the patterns, but do not establish any formal relationship with the concrete underlying execution model. This decoupling inhibits proving whether runs of the actual program indeed correspond to behaviors defined by the high-level abstraction. Second, they provide no easy means of combining different patterns (or instances of the same pattern) in a compositional way, a problem which has been identified and partially addressed in [40].

The main motivation of this paper is to overcome these issues. To do so, this work starts by defining a parallel programming pattern, called `PCR`, which describes computations performed concurrently by communicating *Producers*, *Consumers*, and *Reducers*, each one being either a basic function (business logic), or a nested `PCR`. It combines in a single and composable pattern several concepts like collectives [22], eureka computations [25], unbounded iteration and recursion, and stream programming [33]. The semantics of `PCRs` is formalized using `FXML` [5,39], a formal specification language for expressing parallelism. `FXML` does not rely on any concrete execution model of concurrency, enabling multiple implementations of a program. `PCRs` are shown to behave as functions which ensures seamless composition. With relevant case studies, we illustrate how `PCRs` can ease parallel programming in practice. To enable writing actual programs, we designed and implemented a platform-agnostic `C++` template library supporting `PCRs`. As a second step, we propose a sound and complete formal translation of `PCRs` into an executable parallel model, namely Concurrent Collections (`CnC`) [9]. To complete the contribution, we developed a code generation tool which encompasses a template rewriting engine for translating `PCRs` into `CnC`-based implementations.

Outline The paper is structured in two parts. The first part (Sects. 2–4) is devoted to describing syntax, semantics and applications of `PCRs`. Section 2 presents the `PCR` pattern. It starts with a high-level description together with a motivating example. Then, the semantics is formalized with `FXML`. Section 3 discusses extensions to the basic `PCR` pattern which allow composing `PCRs` beyond the `PCR` computational model. Section 4 explores case studies of increasing complexity to illustrate how `PCRs` express commonly used parallel programming patterns [30]. The second part (Sects. 5 and 6) explains the implementation of `PCRs`, together with its associated tool. Section 5 briefly introduces `CnC` and proposes a translation of `PCRs` into `CnC`. Section 6 describes a `C++` template library which provides a programming framework for `PCRs`. It also sketches a concrete `CnC`-based implementation and compares it with `CnC` through a set of benchmarks. Finally, Sects. 7 and 8 discuss related and future work.

2 The produce–consume–reduce pattern

2.1 Informal presentation

The `PCR` pattern aims at expressing computations consisting of a *producer* consuming input data items and generating, for each one of them, a data set to be consumed by several *consumers* working in parallel. Their outputs are finally aggregated back into a single result by a *reducer*. `PCRs` emphasize the independence between different computations in order to expose all parallelization opportunities.

Figure 1 depicts the general form of a `PCR`. Arrows represent *data connections*. Full ones model the possibly multiple *input sources* and the single *output channel* to the external environment. Dashed arrows denote *internal* data channels.

[1] An in-depth discussion of parallel programming languages is out of the scope of this paper.

Author's personal copy

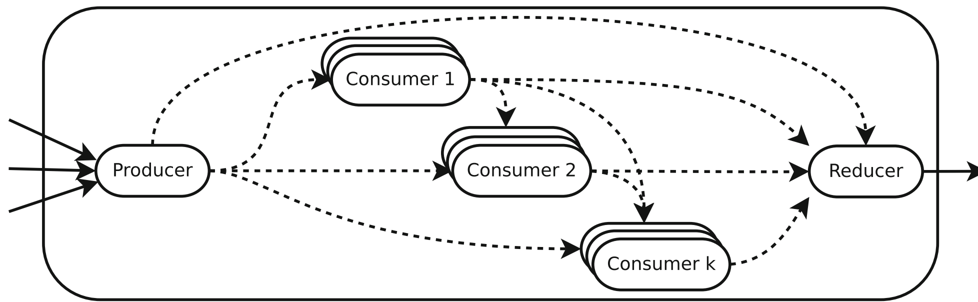


Fig. 1 The *PCR* pattern

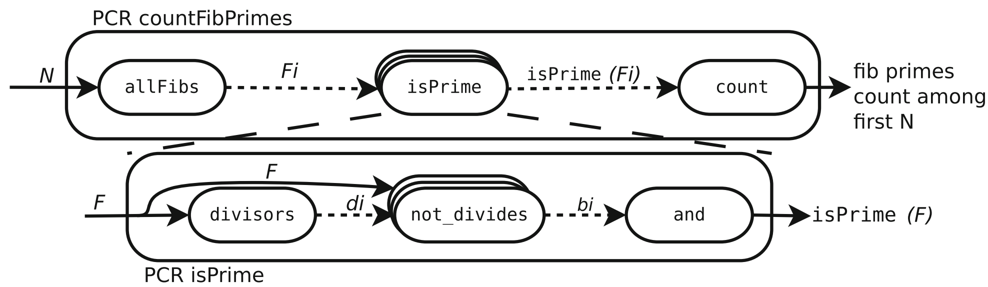


Fig. 2 *PCR* for counting Fibonacci primes

Notice that all *PCR* external inputs are available to any inner component. Cycles between components are not allowed.[2] *Data flow* inside a *PCR* is as follows. For each input data item, the *producer* component generates a set of output values; each one being immediately available for reading. *Consumers* read values from the outer scope and from the private data channels to perform their computations. At the end, a *reducer* combines values from one or more data sources coming from the *producer* and one or more *consumers*, generating a single output item for every input item processed by the producer. Reads in data channels are nondestructive, i.e., the same value can be read multiple times by any consumer and by the reducer. No input is ignored, i.e., every item is handled by some component—all dashed arrows carry the same number of data items to be read. *Producer*, *consumers*, and *reducer* work in parallel subject to data dependencies: all input items must be available for a consumer/reducer instance in order to perform its calculation. Each producer, consumer and reducer can potentially spawn as many parallel execution instances as necessary for any specific workload. Both the nature of an execution instance (local and/or remote thread or process) and the scheduling policy are defined by each *PCR* underlying implementation.

We illustrate the concept by specifying a program that counts primes among the first N Fibonacci numbers. Figure 2 (top) shows the *PCR* `countFibPrimes`. The producer `allFibs` generates the sequence $F_1, F_2, \ldots, F_N$ of Fibonacci numbers. Each instance $i \in [1 \ldots N]$ of the `isPrime` consumer checks, in parallel, the primality of F_i, resulting in the unordered output of indexed boolean values `isPrime`(F_i). The reducer `count` counts the number of those which are *true*. Figure 2 (bottom) shows the *PCR* of consumer `isPrime` which checks in parallel all possible d_i divisors. The `and` reducer computes the conjunction of all the b_i outputs by the parallel instances of consumer `not_divides`. This is an example of a consumer reading the producer output and the *PCR* input as well. The ability of *nesting PCRs* allows reusing components and controlling the desired grain of parallelism in a simple way. The *PCR* `countFibPrimes` admits parallel execution at several levels. First, many instances of `isPrime` could be executed simultaneously as allowed by the available processing engines and the F_i production rate. Second, since the `count` reduce operation is associative and commutative, it could also be parallelized. It is worth noticing that, even if at *PCR* scope the producer and reducer components are single instances, *PCR* nesting allows for concurrent execution of multiple instances of the same producer/reducer pair. In this example, there are as many logical instances of the `divisors` producer and of the `and` reducer as the number of F_i to be processed by consumer `isPrime` in the outer scope.

PCRs and collectives Collectives ([22], Chapter 5 of [30]) are parallel patterns based on different communication strategies between nodes: *gather*, collecting data from several senders; *scatter*, partitioning data among several recipients; *broadcast*, sending the same data to multiple recipients; *reduce*, combining multiple elements); and *scan*, produc-

[2] Cyclic composition through recursion is discussed in Sect. 3.

Author's personal copy

ing all partial reductions of a collection. `PCRs` combine collective operations into a single, composable, high-level pattern. The *producer* in a `PCR` is a scatter/broadcast component, sending each produced item to different instances of the same consumer (*scatter*), and every produced item to all different consumers reading its output (*broadcast*). The *reducer* is a *gather* component, combining all the consumer outputs. Composability follows since producers, consumers and reducers are themselves `PCRs`. *Scan* is obtained by composing `PCRs`. Moreover, the consumer/reducer combination is a map/reduce fusion.

2.2 Formal definition of *PCRs*

To give both a specification language and a formal semantics to the `PCR` concept, we choose the `FXML` language and propose syntactic extensions to it in order to ease writing `PCR` instances. As an introduction, we informally provide `FXML` syntax and semantics. The reader is referred to [5,39] for an in-depth and formal definition.

2.2.1 FXML

An `FXML` specification describes parallel computations by defining the expected behavior of any valid implementation of it as a *set of partial orders*. The *body* of an `FXML` specification is composed of blocks called *pnodes*. The basic *pnode*-types are variable (**var**) and function (**fun**) *declarations*, *assignments*, and *basic code*. Basic *pnodes* are executed *atomically*. *Pnodes* can be combined with *sequential execution* constructs: **seq**, **while**, **for**, **if-then-else**, and with *parallel execution* constructs: **par** (parallel code blocks) and **forall** (parallel `for` loop). *Pnodes* can be labeled. *Pnodes* inside loops (**for**, **while**, **forall**) are automatically and dynamically indexed.

Parallelism can be restricted by specifying data dependencies. The statement **dep** `Q(i)` $\rightarrow$ `P(i)` specifies a *data dependency* between occurrences of assignments labeled `Q` and `P`, meaning that the i-th occurrence of `P` must use the value of the variable, say x, written by the i-th occurrence of `Q`. We call this an (i, i) dependency. `FXML` supports dependencies of the form $(i, g(i))$, where g is an affine function. Besides, `FXML` provides some predefined types of dependencies: *weak*, i.e., the read value could be any written one, *strong*, i.e., every written value must be read at least once, and *bijective*, i.e., every written value must be read exactly once. Data and control dependencies determine a partial order of the execution of statements. The semantics of a *pnode* is a (possibly infinite) *set* of (possibly infinite) partial orders, called *executions*, consistent with the *conjunction* of constraints imposed by dependencies.

`FXML` semantics describes the full history of assignments. This is achieved by keeping track of all values carried out by a variable through dynamic and automatic indexing of each assignment. This property is leveraged into a syntactic mechanism by enabling `FXML` *pnodes* to refer to *specific indexes* of a variable. Given a computable function g, the operation x**[**g**]** on variable x refers to the value $x^{i+g(i)}$ assigned to x by the assignment indexed $i + g(i)$, where i is the dynamic index given by the semantics to the innermost *pnode* where the expression x**[**g**]** appears. This allows stream programming operations *look-ahead* and *look-behind*, to be used on `FXML` variables. For example, x**[**-1**]** (resp., x**[**1**]**) references the value of x at the *previous* (resp., *next*) index. Whenever needed, we will use the syntax x**[**0**]** to make it clear we are specifically referring to the value x^i where i is the index of the *current* context, as opposed to the complete history of values. The behavior of `FXML` variables is further explained in Sect. 6.1 along with the implementation of *look-ahead* and *look-behind*.

Example Figure 3 (left) shows an `FXML` specification of the Fibonacci primes counting problem. Program indentation is only for pretty printing. Figure 3 (right) depicts the schematic diagram of its semantics. Notice that only one partial order, actually the less restrictive one, is shown. Arrows model data and sequential control dependencies. Occurrences of *pnodes* are indexed. For instance, P^i represents the occurrence of the i-th assignment to variable p, or equivalently, the value p^i. Indeed, indexes are vectors whose dimension increases along with loop nesting. The dependency `B(i,j)` $\rightarrow$ `C(i)` entails the i-th evaluation of (basic function) `and` depends on *all* values of variable b with index (i, j), that is $b^{i,j}$, where $j \in J_i$, and $J_i = [0 \ldots (\mathtt{sqrt}(p^i)\text{-}1)/2]$. Assuming `and` computes the conjunction of all these values, the value c^i is $\bigwedge_{j \in J_i} b^{i,j}$. Similarly, the dependency `C(i)` $\rightarrow$ `R` entails the evaluation of (basic function) `count` depends on *all* values c^i, where $i \in [0 \ldots N)$. Assuming `count` computes the number of all these values which are true, the value of this occurrence of r is $\sum_{i \in [0 \ldots N)}$ **if** c^i **then** 1 **else** 0.

2.2.2 PCR syntax

The syntax of `PCRs` is defined in Table 1. *var* is a variable *name*, and *param* is a *formal parameter*. We refer as *basic functions* to user provided functions implemented in the host language. For the sake of simplicity, we restrict the grammar to always include a producer and reducer. In Sect. 4, we will relax this requirement in special cases where we omit the producer/reducer pair.

Example Figure 4 shows the example from Fig. 2, reusing some elements from the pure `FXML` example from Fig. 3. We omit the **par** keyword inside **forall** if there is only

Author's personal copy


```
1 fun fib(f, i) = if i < 2 then 1 else f[-1] + f[-2]
2 fun divisors(j) = if j==0 then 2 else 3+2*(j-1)
3 fun sqrt, not_divides, count, and
4 dep P(i) -> Q(i)
5 dep D(i,j) -> B(i,j)
6 dep B(i,j) -> C(i)
7 dep C(i) -> R
8 var p, c, r
9 par
10  forall (i = 0; i < N+1; i++)
11    P: p = fib(p, i) // allFibs producer
12  forall p
13    par
14      Q: forall (j=0; j <= (sqrt(p)-1)/2; j++)
15        D: d = divisors(j) // divisors producer
16      forall d
17        B: b = not_divides(d, p)
18      C: c = and(b)
19  R: r = count(c)
```

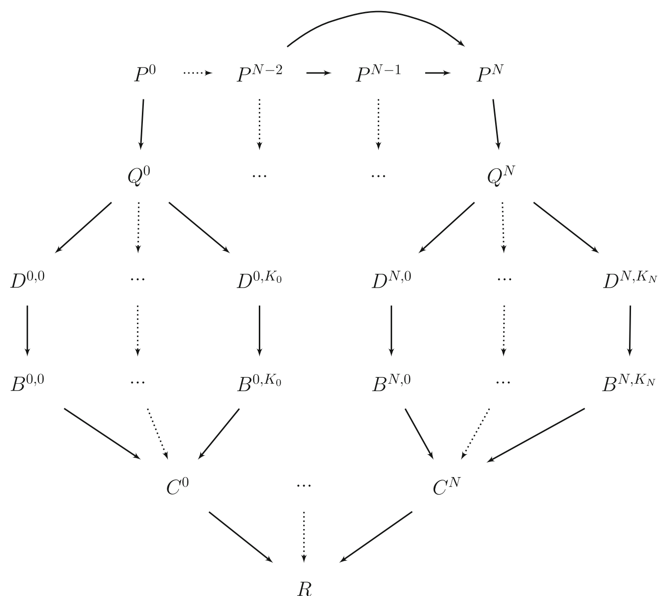


Fig. 3 (*Left*) Fibonacci primes counter in *FXML*. (*Right*) Diagram of its semantics

Table 1 *PCR* grammar

⟨*PCR*⟩	::=	⟨*PCR-name*⟩ (⟨*param-list*⟩) ⟨*body*⟩
⟨*body*⟩	::=	**par** ⟨*producer*⟩
		{**forall** *p* {**par** ⟨*cons-list*⟩$_1$ } }
		⟨*reducer*⟩
⟨*producer*⟩	::=	*p* = **produce** ⟨*f-name*⟩ ⟨*var-list*⟩
⟨*cons-list*⟩$_i$	::=	⟨*consumer*⟩$_i$ ⟨*cons-list*⟩$_{i+1}$ | ϵ
⟨*consumer*⟩$_j$	::=	c_j = **consume** ⟨*f-name*⟩ ⟨*var-list*⟩ |
		c_j = **iterate** ⟨*cnd*⟩ ⟨*f-name*⟩ ⟨*var-list*⟩
⟨*reducer*⟩	::=	*r* = **reduce**⟨*cnd*⟩⟨*f-name*⟩ ⟨*init*⟩ ⟨*v-list*⟩
⟨*param-list*⟩	::=	⟨*param*⟩ , ⟨*param-list*⟩ | ⟨*param*⟩
⟨*v-list*⟩	::=	⟨*var*⟩ ⟨*v-list*⟩ | ⟨*var*⟩
⟨*f-name*⟩	::=	⟨*PCR-name*⟩ | ⟨*basic-fun-name*⟩
⟨*init*⟩	::=	⟨*basic-fun-name*⟩ ⟨*param-list*⟩

```
1 fun sum(i,b) = i + (if b then 1 else 0)
2 fun bnd_fib(x) = x+1
3 fun bnd_divisors = lambda p: (sqrt(p)-1)/2
4
5 PCR countFibPrimes(N):
6   par
7     p = produce fib N  // allFibs producer
8     forall p
9       c = consume isPrime p
10    r = reduce sum 0 c

1 PCR isPrime(F):
2   par
3     p = produce divisors F
4     forall p
5       c = consume not_divides p F
6     r = reduce && true c
```

Fig. 4 Fibonacci primes counter written in *PCR* syntax

one children. Notice that basic functions `and` and `count` have been replaced by lower-level functions `&&` and `sum` as arguments of **reduce**. This enables taking care of the potential parallelism at this level (see Sect. 2.2.3). The role of $\mathbf{bnd}_f$ will become clear later.

2.2.3 Semantics

We start by providing the corresponding *FXML* specification of each building block for the case where f is a *basic function* (Table 2). Nesting is considered afterward.

Specification of **produce** A producer generates the set of indexed values to be processed by consumer and reducer elements. Formally, it is a **forall**-*pnode*, say P, that iterates a basic function f. Let I be the index dynamically assigned by the underlying *FXML*-semantics to a particular execution of P, denoted P^I. In P^I, $\mathbf{bnd}_f$ determines the number of parallel instances of f as a function of input variables $x_1..x_n$. For each instance $i \in [0, \ldots, \mathbf{bnd}_f(x_1..x_n))$, the producer writes its output variable p, setting its $(I \circ i)$-th value $p^{I \circ i}$, that is, the value of index i produced by the I-th instance of producer P. This indexing allows for the concurrent execution of any two instances $I \neq J$ of the producer, each one generating its own set of p values, namely $p^{I \circ i}$ and $p^{J \circ j}$. To compute $p^{I \circ i}$, f can use any value of the input variables $x_1..x_n$, any *previous* value $p^{I \circ i'}$ of p, $i' < i$, and i. That is, a producer can *look-ahead/behind* at will on input variables

Author's personal copy

Table 2 *PCR* building blocks and their *FXML* specification

PCR	FXML
p = **produce** $f\ x$	1 `forall(i=0;i<bnd`$_f(x)$`;++i)` 2 $p = f(x, p, i)$
c_j = **consume** $f_j\ x\ p\ c_1..c_k$	1 $c_j = f_j(x, p, c_1..c_k)$
c_j = **iterate** $cnd\ f_j\ z$	1 **seq** 2 $y = z$ 3 **repeat** $y = f_j(y)$ 4 **until** $cnd(y)$ 5 $c_j = y^{last}$
r = **reduce** $cnd \oplus v_0\ z_1..z_q$	1 **seq** 2 $v = v_0$ 3 **for** p 4 $v = v$`[-1]` $\oplus \langle z_1..z_q\rangle$ 5 **if** $cnd(v)$ **then break** 6 $r = v^{last}$

and *look-behind* on its own output. We omit these dependencies in the table.

Specification of **consume** A basic *PCR*-consumer reads a set of input parameters and applies a basic function on them in order to compute a single output. Formally, a consumer is an *assignment-pnode*, say C_j, whose left-hand side is its output variable c_j, and its right-hand side is function f_j, possibly depending on *PCR* input variables $x_1..x_n$, producer's output p, and other consumers' output variables $c_1..c_k$, $k < j$. We restrict the j-th consumer to only read outputs of previous consumers to avoid data-dependency loops inside a *PCR*. However, it is allowed to *look-ahead/behind* on any of them. Associated with the surrounding **forall** p, there is an (i, i) dependency $p^{I \circ i} \rightarrow c_j^{I \circ i}$, which is omitted in the Table. Thus, for each value $p^{I \circ i}$ written by the producer, there is an instance $C_j^{I \circ i}$ writing value $c_j^{I \circ i}$.

Specification of **iterate** This construct enables looping a *PCR* until a given condition is met. In each iteration, it *looks-behind* to use previous values. Like **produce**, **iterate** is restricted to *look-behind* operations, as *look-ahead* would generate a deadlock. Setting *cnd* to y**[0]** $==$ y**[-1]** allows computing a fixpoint. y^{last} is the *last* value of y.

Specification of **reduce** The reducer uses a *commutative and associative operation* $\oplus$ and an initial neutral value v_0 to combine consumers' outputs into a single result, until *cnd* holds. This condition allows specifying *eureka computations* ([25]) making possible early termination. This is a common pattern in search and optimization problems in

```
1 // parallel partition
2 forall k ∈ K
3   r_k = v_0
4   // combine values
5   for i ∈ k
6     r_k = r_k[-1] ⊕ S
7     if cnd(r_k) then break
```

```
1 // combine partitions
2 r = v_0
3 for k ∈ K
4   r = r[-1] ⊕ r_k^last
5   if cnd(r) then break
```

Fig. 5 Generic *FXML* specification of **reduce**

which a solution space is searched in parallel until the first (or best) solution is found. Setting *cnd* to `false` corresponds to reducing all values. Hereinafter, we assume that this is the default condition for the reducer and omit it in this case. A *reducer* is modeled as a **for**-*pnode* that reads the variables $z_1..z_q$. The dependencies are $z_j^{I \circ i} \rightarrow v^{I \circ i}$, where $v^{I \circ i}$ is the i-th value of the assignment $v = \ldots$ inside the **for**-loop of the I-th instance of the reducer, and $z_j^{I \circ i}$ is the i-th value assigned to z_j in the *PCR*. The value v_0^I represents the result of evaluating the initializer function. The reducer computes $v^I = v_0^I \oplus Z^{I \circ 0} \cdots \oplus Z^{I \circ b^I}$ where $Z^{I \circ i} = \langle z_1^{I \circ i}..z_q^{I \circ i}\rangle$ and assigns v to r, therefore obtaining its I-th value r^I. Here, b^I is the minimum between the number of iterations of producer P^I (there are as many iterations as values of p written by the I-th instance of the producer) and the index of the first iteration in which $cnd(v)$ becomes true.

The sequential reducer is a special case of the generic specification shown in Fig. 5, where K is a partition of a given set J indexing the set of values to reduce. Indexes $k \in K$ are processed in parallel and sequentially reduced. The sequential reducer is a single-partition implementation of this general model.

Nesting of PCRs *PCRs* do not support recursion. Therefore, using *PCR* B in the definition of *PCR* A is semantically equivalent to inlining the definition of B inside A, renaming all local variables in B as fresh variables, and renaming the formal parameters of B as the variables referenced in the usage of A. In Sect. 3, we discuss recursive calling of *PCRs*.

Example Figure 6 shows the result of inlining `isPrime` inside `countFibPrimes` (inner **par** block).

Remark It is worth making two observations. First, nesting entails the inner *PCR* inputs are references to outputs from the outside scope. Variables of the outer scope have an index with lower dimension than the reading *pnode*. Therefore, the value is obtained truncating the index of the reading *pnode* to the dimension of the read variable. Second, besides the (i, i) dependencies enforced so far, *look-ahead/behind* operations introduce their own data constraints. Since these are

Author's personal copy


```
1 par
2   F = produce fib N
3   forall F
4     par
5       K = produce divisors F
6       forall K
7         D = consume not_divides K F
8       P = reduce && true D
9   R = reduce sum 0 P
```

Fig. 6 Flattened code for `countFibPrimes` (Fig. 4)

data dependencies, they are automatically accounted for by the underlying *FXML* semantics [39]. The way indexes are handled in these situations is revisited in detail in Sect. 5 where the proposed implementation of *PCRs* is explained. See Sect. 6.1 for a discussion about this.

Property 1 The result of evaluating a *PCR* in *FXML* semantics is a function on the input parameters assuming (a) the basic producer, consumer and reducer functions are total, and (b) no data cycles are introduced by *look-ahead* in basic functions.

3 PCR extensions

Property 1 shows that any *PCR* behaves like a total function. This allows calling a *PCR* from any basic function in a *blocking* way, where the caller *holds* until the call returns. Of course, even if the caller is blocked, the parallelism *inside* the callee is preserved. In this section, we discuss various extensions to the basic *PCR* model defined so far that exploit this capability in different manners. Their use is later illustrated in Sect. 4.

3.1 Divide and conquer

Calling a *PCR* as a function enables *recursive parallelism*. A prominent example is *divide and conquer*, an algorithmic technique consisting in partitioning a complex instance of a problem into several smaller subproblems, solving each one independently, and combining their solutions in order to calculate the final result. Each subproblem can be solved directly if it is simple enough. Otherwise divide and conquer can be recursively applied. This is done by defining the following functions:

- `is_base`: checks whether a problem is a base case;
- `base`: computes the solution for a base case;
- `divide`: partitions a problem into subproblems;
- `conquer`: describes how to combine solutions.

Figure 7 shows a *PCR*-based parallel solution. The producer partitions the original problem into subproblems

```
1 fun divide, is_base, base, conquer, terminate
2 fun subproblem(x) =
3   if is_base(x) then base(x) else divide_and_conquer(x)
4 fun iter_divide(x,i) = divide(x)[i]
5
6 PCR divide_and_conquer(x):
7   par
8     p = produce iter_divide x
9     forall p
10      c = consume subproblem p
11    r = reduce terminate conquer null x c
```

Fig. 7 *PCR* definition for *divide and conquer*

using the `iter_divide` function. Consumers process each subproblem, either using `base` or recursively calling *PCR* `divide_and_conquer`, depending on the result of `is_base`. The reducer uses `conquer` to combine all the subproblems' solutions. *null* is the empty subproblem. Function `terminate` is used to define an eureka stopping condition.

The divide and conquer pattern is illustrated with the N-Queens problem in Sect. 4.3.

3.2 Feedback loops

Some computations involve producing, for each input item, one or more results, each of which being either output or feedback into the same component. Programming languages for data-parallel streaming applications, like StreamIt [35], provide explicit constructs for specifying feedback loops. In the context of task-based parallelism, this behavior corresponds to the *workpile* pattern, where an instance of a task can generate more instances and add them to a pile of tasks to be done [30]. To cope with such behaviors, we extend *PCRs* with:

$$o = \texttt{feedbackloop}\ f\ v$$

where o is the output variable, f is a function and v is a value. Table 3 sketches the *big-step* operational semantics of **`feedbackloop`**. The notation $\mathcal{A} \vdash p \Downarrow \mathcal{A}'$ means that executing program p in a set of indexed assignments

Table 3 Semantics of **`feedbackloop`**

FDB
$$\frac{\langle \mathcal{E}, \{v\}\rangle \vdash o = \texttt{feedbackloop}\ f\ v \Downarrow \langle \mathcal{E}', \emptyset\rangle}{\mathcal{E} \vdash o = \texttt{feedbackloop}\ f\ v \Downarrow \mathcal{E}'}$$

DOWORK
$$\frac{x^i \in \mathcal{X} \qquad f^i(x^i) = O^i \cup X^i}{\langle \mathcal{E}, \mathcal{X}\rangle \vdash o = \texttt{feedbackloop}\ f\ v \Downarrow \langle \mathcal{E} \uplus O^i, \mathcal{X} \setminus x^i \uplus X^i\rangle}$$

Author's personal copy


```
1 PCR preProcess(x): ...
2 PCR doStatistics(x): ...
3 fun makeBucket(size) = lambda x :
4   if (mod (idx x) size == 0)
5   then doStatistics(bucket(x[0]..x[-size+1])
6
7 // Connect preProcess to makeBucket (size=3)
8 connect(preProcess, makeBucket(3))
```

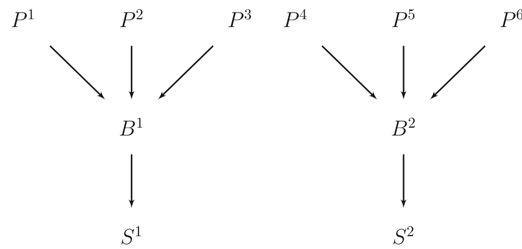


Fig. 8 (*Left*) Connection of `preProcess` and `doStatistics` by means of delegate `makeBucket`. (*Right*) Diagram of its semantics

$\mathcal{A}$ yields the set of indexed assignments $\mathcal{A}'$. We use $\mathcal{E}$ to denote the set of indexed assignments of the "external" variables, while $\mathcal{X}$ denotes the indexed assignments of the variable x, which is "internal" to the **feedbackloop**. We use the notation $\langle \mathcal{E}, \mathcal{X} \rangle$ to make explicit the separation between external and internal variables. Rule [FDB] states that the initial value x^0 of x is v. That is, $\mathcal{X}$ starts being $\{v\}$. Whenever all values of x have been consumed, i.e., the set $\mathcal{X}$ is empty, the **feedbackloop** terminates, yielding the set of assignments $\mathcal{E}'$, which extends $\mathcal{E}$ with the indexed values of variable o. Rule [DOWORK] gives the semantics of the actual work done by **feedbackloop**. It states that for each value $x^i \in \mathcal{X}$, there is an instance f^i such that $f^i(x^i)$ performs a set of assignments O^i and X^i on variables o and x, respectively. The assignments to o are visible "outside" the **feedbackloop**, so they are added to $\mathcal{E}$. The assignments X^i on x "spawn" further executions of f, so they are added to $\mathcal{X}$, while x^i is removed since it has already been consumed. The $\uplus$ operation ensures the attribution of appropriate indexes to the added assignments.

A major difference with the **forall** construct of *FXML* is that x is assigned inside the **forall** body; therefore, the total number of instances of x is not known in advance and the overall structure of the computation is not regular. Besides, **feedbackloop** is different from **iterate** in two aspects: (1) each instance may generate an output, and (2) instances can be executed as soon as their dependencies hold.

Indeed, the **feedbackloop** construct entails a proper extension of the basic *PCR* model as it somehow combines producer and consumer capabilities: it *consumes* each instance of x^i of x and *produces* a set of indexed of outputs $o^{j_0} \ldots o^{j_{m_i}}$ and of new instances $x^{k_{o}0} \ldots x^{k_{o}n_i}$ of x to be consumed. Nevertheless, it can be composed with a *reducer* to get a *PCR* as follows:

```
1 PCR P(v):
2  par
3    o = feedbackloop f v
4    r = reduce cnd ⊕ r0 o
```

The use of this pattern is illustrated with the N-Queens case study in Sect. 4.

3.3 *PCR* networks

PCRs enforce an (i, i) dependency between input and output. In some scenarios, it is useful to relax that dependency in order to forward more (or less) elements from a source *PCR* to one or more target *PCRs*. Some examples of connections between two *PCRs* A and B are: *grouping* outputs of A in variable-sized *buckets* to be processed by B; *partitioning* each output of A in a variable number of items read by B; and *monitoring* (some) outputs of A with B without changing A's behavior. In all these use cases, enforcing the (i, i) dependency is either incompatible or costly in practical terms.

To solve this, we propose a mechanism of *connecting* two *PCRs* as follows. The **connect**(A, d) operation *bridges* the output of *PCR* A to a *delegate* function d: for each output value v of A, $d(v)$ will be called. Execution of d could ignore v based on some condition or *forward* $f_k(v)$, for some function f_k, to *PCRs* $P_1, \ldots, P_q$, by calling $P_k(f_k(v))$.

Figure 8 shows a **connect** example along with its execution semantics. Delegate function `makeBucket` reads inputs from *PCR* `preProcess` and calls the *PCR* `doStatistics` with a bucket of size *size* whenever the index of the current input is a multiple of *size*. The construction of the buckets is done using *look-behind*. At call i of `makeBucket`, variable x in line 3 gets instantiated with the i-th output of `preProcess`. The diagram depicts one possible execution. Labels P, B and S represent instances of `preProcess`, `makeBucket`, and S, respectively.

Adding an intermediate and opaque delegate allows for breaking the (i, i) dependency. Therefore, the resulting *network* of *PCRs* may no longer be a *PCR*. Indeed, **connect** extends the model into a two-level hierarchy: a level of *PCRs* with (i, i) dependencies, and a second level of connections with dependencies between indexed sets of potentially different sizes, derived from the behavior of the participating delegates. A complete formalization of this two-level model is out of the scope of this paper.

Author's personal copy


```
1 PCR lowPassFilter(S):
2   par
3     c1 = consume LPF S
4     c2 = consume DEMOD c1
5     c3 = consume EQU c2
```

```
1 fun DEMOD(v)
2   // consume a window of
3   // width 2 from v
4   return GAIN*atan(v+v[1])
```

Fig. 9 *PCR* "Low-pass Filter" (*left*) and DEMOD (*right*)

4 Case studies

We illustrate how *PCRs* and its extensions allow expressing commonly used parallel programming patterns [30].

4.1 Low-pass filter

This example is a pipeline with three computing stages: *low-pass filter* (LPF), *demodulator* (DEMOD), and *equalizer* (EQU) [35]. An interesting aspect is that pipeline stages rely on *look-ahead*. LPF consumes NT consecutive input elements in order to produce a single output element, while DEMOD uses the two previous values produced by LPF. EQU needs the NT previous outputs from DEMOD in order to generate one value. The *PCR* shown in Figure 9 has no producer and reducer. This is equivalent to having a producer/reducer pair with the identity operation.

A typical implementation would interconnect stages with buffers. Instead, the *PCR*-based description abstracts out any buffering scheme between stages using *look-ahead* inside the consumers to describe this behavior (e.g., DEMOD). Besides, this example involves two parallel programming patterns: (a) it showcases an instance of a *stateless parallel pipeline* composed of **consume** components, and (b) each **consume** can be regarded as a one dimensional *stencil computation*, as

```
1 fun lines, words, joinCounts, count, elem
2
3 PCR count-words-by-lines(T, W):
4   par
5     l = produce lines T
6     forall l
7       c = consume count W l
8     r = reduce joinCounts {} c
9
10 PCR count-words-by-words(T, W):
11   par
12     w = produce elem W
13     forall w
14       c = consume count-words-by-lines T [w]
15     r = reduce joinCounts {} c
```

Fig. 10 *Count-words*

```
1 fun is_base(c) = c.complete() or not c.canAddQueens()
2 fun base(c) = if c.complete() then [c] else []
3 fun is_big(c) = c.currentRow < MAXDEPTH
4 fun conquer(s1, s2) = s1 ++ s2
5 fun divide(c) =
6   cs = []
7   for i in 1..c.boardSize
8     if c.canAddQueen(i) then cs += [c.addQueen(i)]
9   return cs
```

Fig. 11 N-Queens using *PCR*-based divide and conquer

```
1 fun initial, found
2
3 fun nqueens(c,cs) =
4   for i in 1..c.boardSize
5     if c.canAddQueen(i) then
6       z  = c.addQueen(i)
7       if z.complete() then cs = [z] else c = z
8
9 PCR NQueens(N):
10   par
11     c = produce initial N
12     cs = consume NQueens_fb c
13
14 PCR NQueens_fb(c):
15   par
16     cs = feedbackloop nqueens c
17     r = reduce found ++ [] cs
```

Fig. 12 N-Queens with **feedbackloop**

```
1 fun extend(c) = if c.complete() then [c] else divide(c)
2
3 PCR nqueens_step(cs):
4   par
5     c = produce elem cs
6     forall c
7       cs = consume extend c
8     r = reduce found ++ [] cs
9
10 PCR nqueens_iterative(cs):
11   c = iterate found nqueens_step cs
```

Fig. 13 N-Queens with **iterate**

look-ahead is used to access several neighbor values of the input.

4.2 Count words

Given a text T and a set W of words, *count-words* computes the number of appearances of each $w \in W$ in T. This is a typical MapReduce example [14].

Author's personal copy

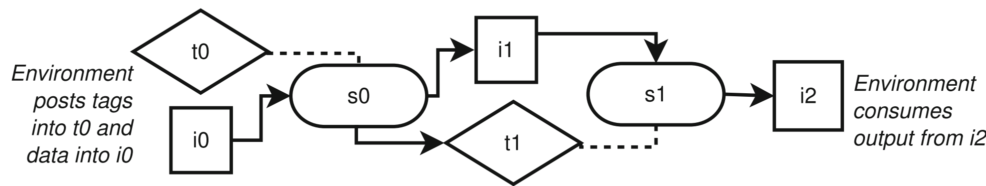


Fig. 14 A diagram of a *CnC* graph. Diamonds, ovals and boxes, respectively, represent tag, step and item collections. Dotted lines indicate *prescribes* (control) relationships between tag and step collections and arrows denote *produce/consume/control* relationships between step, item and tag collections

Figure 10 shows two *PCRs* for counting words. *PCR* `count-words-by-lines` splits T in lines and counts the appearances of words in W for each line in parallel, using basic function `count`. *PCR* `count-words-by-words` adds an extra level of parallelization, by calling, for each word $w \in W$, *PCR* `count-words-by-lines`.

4.3 N-Queens problem

This problem consists in placing N queens in an $N \times N$ chessboard, with no two queens sharing the same row, column or diagonal. Figure 11 shows a *PCR* for finding all solutions using *divide and conquer*: `c` is a *configuration* of the chessboard with the placed queens, and `cs` is a list of configurations. Function `complete` checks whether a configuration is a solution, and `canAddQueens` checks whether it is possible to place a queen, etc. Predicate `is_big` controls if a subproblem is large enough to warrant solving it in parallel: *MAXDEPTH* is the maximum recursion depth done in parallel before starting to work sequentially.

A variation consists in finding only one solution. This is done by appropriately defining `terminate` in the `divide_and_conquer` *PCR* of Fig. 7 to stop the recursion as soon as a valid solution is found. Figure 12 shows an alternative specification using **`feedbackloop`** composed with an *eureka* reducer. Function `initial(N)` constructs a start configuration of size `N`, and `found(cs)` verifies whether `cs` contains a complete configuration in order to stop the computation. Figure 13 shows a *PCR* using **`iterate`**.

5 Implementing PCRs

In this section, we present Concurrent Collections (*CnC*) [9], a concrete target model for implementing *PCRs*.

5.1 The concurrent collections model

A *CnC* program consists of a high-level description of a computation graph, legacy code to be executed, and the environment. The basic atoms are computation *steps* hosting legacy code to be run; control instances or *tags*, where each tag value represents the signal of one unit of work to be done by each dependent step; and data *items*, which are read and written by the steps during their computation work.

The *CnC* graph is constructed by interconnecting collections. Each tag collection *prescribes* a number of computation steps. Tag values can be **`put`** into tag collections by the *environment* or by steps; in the latter case, each step *controls* the tag collection. Posting a new tag value will spawn one instance of each controlled step collection with the tag value as a parameter. Each item collection is a concurrent map storing items indexed by tag values, providing **`get`** and **`put`** operations. Items are get/put by the *environment* or by steps; in the latter case, the steps *consume-from*/*produce-to* the item collection. To ensure determinism, the semantics prohibit overwriting tags or items in collections (*Dynamic Single Assignment*).

Figure 14 shows an instance of a *CnC* graph and illustrates the basic building blocks.

CnC (operational) semantics is given considering a simplified core language called *Featherweight*-*CnC* [9], formalizing the concept of launching steps when tags are posted. Item collections are expressed as a flat memory array. Tag collections are modeled as step-spawning rewriting rules in the operational semantics.

We briefly describe the relevant *Featherweight*-*CnC* syntax elements. *CnC* computation *steps* are written as functions with a single tag input parameter. Execution of a step S is triggered by the **`prescribe`** S operation. Memory is represented by the **`data`** object with **`get`**(i) for reading the value stored in memory location i and **`put`**(i, v) for writing value v into memory location i. The *CnC* semantics requirement of *Dynamic Single Assignment* means there can be at most one **`data.put`**(i, v) execution for each possible value of tag i.

5.2 Translation of *PCRs* into *CnC*

We define a translation of a *PCR* into *Featherweight*-*CnC*. Afterward, we show that the semantics of the *CnC* code is functionally equivalent to the *FXML* semantics.

CnC memory model As *CnC* computation steps are stateless, the only memory used by a *CnC* program is its item collections' storage. In its operational semantics definition, *CnC* simplifies the formal memory model by mapping all item col-

Author's personal copy

lections into a single flat *data* structure. In this translation, we assume this memory flattening mapping as given but keeping track of the memory mapped to each item collection. To achieve this, we denote, for the item collection associated with *FXML* variable x, $\mathbf{data}_x$ as the memory reserved to it; therefore, $\mathbf{data}_x.\mathbf{get}(t)$ denotes the value stored in **data** for tag value t in the item collection associated to x. Likewise, $\mathbf{data}_x.\mathbf{put}(t, v)$ denotes a write operation into the item collection associated to x of value v for tag value t.

FXML variables and indexes In *CnC*, item collections represent the assignment history for *FXML* variables. The index of each assignment is the tag value used as key in the collection. As *FXML* indexes may be multidimensional because of iteration space nesting, the tag type is a vector of integer values. When reading an *FXML* variable indexed with a lower dimension, the implementation truncates the reader tag dimension to the read variable dimension. In this way, the **get** operation done in the source item collection is always well defined.

Translation of look-ahead/behind To translate these operations, we need to rewrite functions in **produce**, **consume** and **reduce** primitives. Given $f(s_1, \ldots, s_k) = E$ with input *FXML* variables s_i written as an expression E, and an index I, we define the translation operator $\overline{f}^I$ as

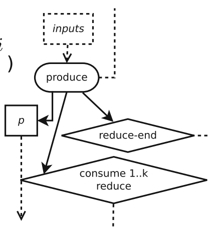
$$\overline{f(s_1, \ldots, s_k)}^I \equiv E[s_i\,\mathtt{[}d\mathtt{]} := \mathbf{data}_{s_i}.\mathbf{get}(I + d)]_{i\in 1\ldots k}$$

where $E[x := y]$ denotes syntactic substitution of x by y in expression E.

Note Without loss of generality, we assume that every variable s_i appears in E in the form $s_i\,\mathtt{[}d\mathtt{]}$ for some integer d, and that d is a value and not an expression.

Producer Given a producer function $f(x_1..x_n, p, i)$ and a bound expression $\mathbf{bnd}_f(x_1..x_n)$, the *CnC* translation of p = **produce** $f\ x_1..x_n$ is defined as

```
1 produce_f(I) {
2   for (i=0; i < bnd_f(x1..xn)^I ; ++i)
3   {
4     data_p.put(I∘i, f(x1..xn,p,i)^(I∘i))
5     PCR_prescribe consume_1...k(I∘i)
6     prescribe reduce(I∘i)
7   }
8   prescribe reduce-end(I∘i) }
```

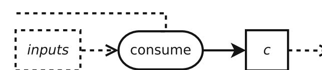


Note that the producer assignment increments the index dimension. This producer translation prescribes a special reducer step `reduce-end` with a tag value encoding the total number of items to process.

For prescribing the consumers, we define the macro **PCR_prescribe** C which translates into **prescribe** C if C is a basic function, and into **prescribe** $\mathtt{produce}_C$ if C is a nested *PCR*, where $\mathtt{produce}_C$ is the corresponding produce step for the *CnC* translation of *PCR* C.

Consumer Given a basic function $f(x_1 \ldots x_n, p, c_1 \ldots c_k)$, the *CnC* translation of c_j = **consume** $f\ x_1..x_n,\ p,\ c_1 \ldots c_k$ is

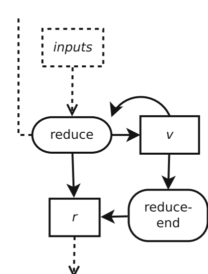
```
1 basic_consume_f(I) { data_cj.put(I, f(x1..xn,p,c1..ck)^I ) }
```



If the consumer is a *PCR* P, the translation does not generate a *CnC* step: It recursively expands the definition into the corresponding *CnC* translations of producer, consumer and reducer of P. The nested producer step does the **get** operations for the input parameters, and the reducer step does the **put** operation on its output item collection which corresponds to variable c_j.

Reducer Given $f(v, z_1 \ldots z_q)$ and v_0, the *CnC* implementation of r = **reduce** *cnd* $f\ v_0\ z_1 \ldots z_q$ is defined as

```
1 reduce_f(I∘i) {
2   if (i==0) u = v0
3   else u = data_v.get(I∘(i−1))
4   u = f(u,z1..zq)^(I∘i)
5   if cnd(u) data_r.put(I, u)
6   else data_v.put(I∘i, u)
7 }
8 reduce-end_f(I∘i) {
9   data_r.put(I, data_v.get(I∘(i−1))) }
```



The reducer folds the nested iteration space (with tags $I \circ 0 \ldots I \circ (k-1)$) into that of the outside scope (with tag I). Each `reduce` step executes one operation reading the output of the previous step stored in collection v, or using the initial value v_0 (for the first reducer). `u` is a local variable of the step in the host language. After checking *cnd*, the executing step either posts the result to the final output r or to v. `reduce-end` is eventually prescribed by the producer and forwards the last value from v to the output r. Exactly one of the **put** operations of lines 5 and 9 is executed. If the *cnd* operation is omitted, the constant function `false` is assumed (i.e., no input will produce an eureka event).

Iterate The implementation of c_j = **iterate** *cnd* $f\ z$ is a direct translation of the pseudocode in Table 2.

Author's personal copy

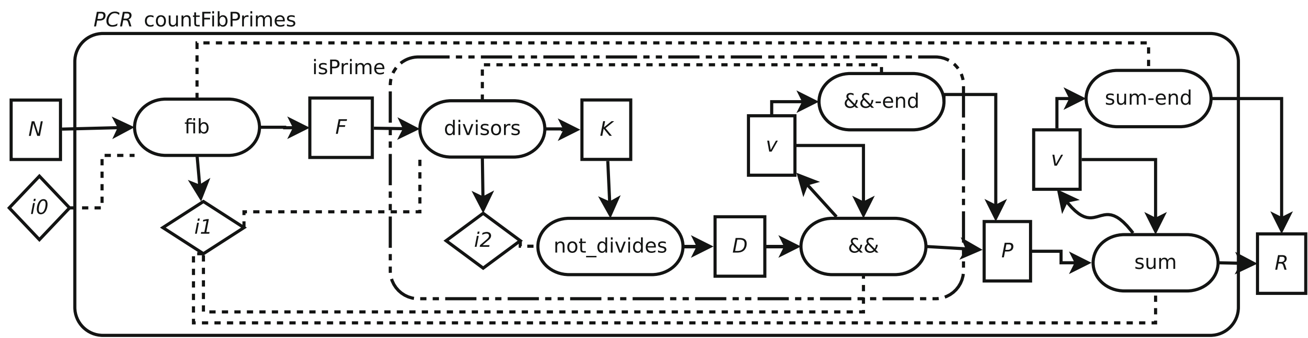
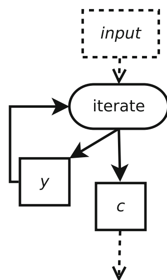


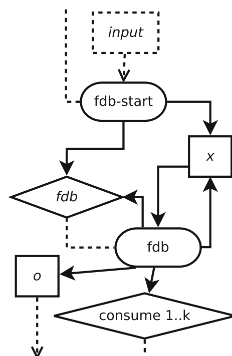
Fig. 15 CnC implementation of nesting of `isPrime` as consumer in `countFibPrimes`

```
1 iterate_f(I) {
2   i=0
3   data_y.put(I ∘ i, overline(f(z))^I)
4   repeat
5     data_y.put(I ∘ (i+1), overline(f(y))^(I∘i));
6     i=i+1;
7   until overline(cnd(y))^(I∘i)
8   data_cj.put(I, data_y.get(I ∘ i)) }
```



Feedback Loop Given a function f, the *CnC* translation of $o =$ **feedbackloop** $f\ v$ is

```
1 fdb-start_f(I) {
2   i = 0
3   data_x.put(I ∘ i, data_v.get(I))
4   prescribe fdb_f(I ∘ i)
5 }
6
7 fdb_f(I ∘ i) {
8   O, X = overline(f(x))^(I∘i)
9   for u ∈ X
10    j = nextindex(data_x, I)
11    data_x.put(I ∘ j, u)
12    prescribe fdb_f(I ∘ j)
13  for u ∈ O
14    j = nextindex(data_o, I)
15    data_o.put(I ∘ j, u)
16    PCR_prescribe consume_1...k(I ∘ j)
17 }
```



The *CnC* code in steps $\texttt{fdb}_f$ and $\texttt{fdb-start}_f$ uses an atomic function **nextindex** to obtain a fresh index to use in the **put** operations. Performing k calls to **nextindex** will return consecutive indexes from 0 to $k-1$ for each combination of input parameters. Again, u is a local variable of the step in the host language.

Remark In this translation, f is presented as returning sets of assignments and outputs O, X as in rule [DoWork] from the **feedbackloop** semantics in Sect. 3.2.

Evaluation In order to implement the evaluation of a *PCR* $P(x_1 \dots x_n)$ on a set of input values $v_1 \dots v_n$, we provide the following *CnC* code generated from the definition of P, where index I is provided by the environment, and $reducer(P)$ is the output variable of P's reducer.

```
1 evaluate_P(I, v_1...v_n) {
2   data_x1.put(I, v_1)
3   ...
4   data_xn.put(I, v_n)
5   PCR_prescribe P(I)
6   return data_reducer(P).get(I)
7 }
```

Complete CnC translation example Figure 15 shows the *CnC* graph of *PCR* `countFib Primes` (Fig. 6). The complete translation into *CnC* is shown in Fig. 16.

Property 2 The implementation of *PCRs* by *CnC* computation graphs is sound.

6 A C++ template library for PCRs

Here we present a platform-independent API for *PCRs* based on *C++* templates and its implementation on Intel's *C++ CnC* runtime through template rewriting.

6.1 Implementation agnostic *PCR C++* API

In *C++*, *PCRs* are specified as compositions of *template type definitions*. A *PCR* is specified as a `pcr` template instance parameterized with: a list of input types T_i, producer P, consumers C_j, and reducer R:

```
1 typedef pcr<T_1,... ,T_n, P, C_1,
2   ..., C_k, R> PCR_NAME;
```

Author's personal copy


```
1 // PCR: countFibPrimes(N)
2 // F = produce fib N,  bnd_fib(N) = N
3 produce_fib(I) {
4   for(i=0; i < data_N.get(I); ++i) {
5     data_F.put(I ∘ i, fib(F,i)^(I∘i))
6     PCR_prescribe consume_isPrime(I ∘ i)
7     prescribe reduce_sum(I ∘ i) }
8   prescribe reduce-end_sum(I ∘ i)
9 }
10
11 // P = consume isPrime F
12 // isPrime is a basic function
13 consume_isPrime(I) {
14   data_P.put(I, isPrime(F)^I)
15 }
16
17 // R = reduce sum 0 P
18 // v is a local item collection
19 // u is a local variable
20 reduce_sum(I ∘ i) {
21   if (i==0) u = 0
22   else u = data_v.get(I ∘ (i − 1))
23   data_v.put(I ∘ i, sum(u, P)^(I∘i))
24 }
25
26 reduce-end_sum(I ∘ i) {
27   data_R.put(I, data_v(I ∘ (i − 1))
28 }
29
30 env(I, N) {
31   data_N.put(I, N)
32   PCR_prescribe fibPrimes(I)
33   Y = data_R.get(I)
34 }
```

```
1 // PCR: isPrime(F)
2 // K = produce divisors F, bnd_divisors(F) = (sqrt(F)-1)/2
3 produce_divisors(I) {
4   for(i=0; i < (sqrt(F)-1)/2^I; ++i) {
5     data_K.put(I ∘ i, divisors(i))
6     PCR_prescribe consume_not_divides(I ∘ i)
7     prescribe reduce_&&(I ∘ i)
8   }
9   prescribe reduce-end_&&(I ∘ i)
10 }
11
12 // Basic consumer
13 // P = consume not_divides K F
14 consume_not_divides(I) {
15   data_D.put(I, not_divides(V, F)^I)
16 }
17
18 // R = reduce && true D
19 // v is a local item collection
20 // u is a local variable
21 reduce_&&(I ∘ i) {
22   if (i==0) u = true
23   else u = data_v.get(I ∘ (i − 1))
24   data_v.put(I, u && D^(I∘i))
25 }
26
27 reduce-end_&&(I ∘ i) {
28   data_P.put(I, data_v(I ∘ (i − 1))
29 }
30
31 // Example of fib function rewrite
32 // fib(F,i) = if i<2 then 1 else F[-1] + F[-2]
33 fib(F,i)^I = if (i<2) 1
34              else data_F.get(I − 1) + data_F.get(I − 2)
```

Fig. 16 Translated *CnC* code for the `countFibPrimes` example

Table 4 summarizes the syntax for specifying T_i, P, C_j, and R. In all cases, `(parameters...)` is a list $i_1, \ldots, i_k$ of positive integer constants specifying the source parameters as relative positions backwards in the *PCR* body list; i.e., i_k means "*the output of the* i_k*-th preceding PCR body element*". `PROD`, `CONS` and `RED` parameters are basic code binding specifications to be explained in what follows. `CONS` can be alternatively an existing **`pcr`** template specification, allowing nesting of *PCRs* in consumers.

Figure 17 shows the *C++* specification of `countFibPrimes` *PCR* from Sect. 2.2.2. Note the positional syntax for input parameter specification in the *PCR* body.

Table 4 Summary of *C++* templates used for *PCR* specifications

Element	*C++* Specification
T_i	**`pcr_in`**`< type(X_i) >`
P	**`pcr_produce`**`< PROD, parameters...>`
C_j	**`pcr_consume`**`< CONS, parameters...>`
R	**`pcr_reduce`**`< RED, parameters...>`

```
1 typedef pcr<                          // notes on parameters
2     pcr_in< int >,                   // input N of type num_t
3     pcr_produce< fibs,      1 >,     // 1 is N
4     pcr_consume< isPrime,  1 >,      // 1 is fibs output
5     pcr_reduce< count,      1 >      // 1 is isPrime output
6 > countFibPrimes;                    // output of count
```

Fig. 17 `countFibPrimes` written as a *C++* template

Host Language Code Integration The `countFibPrimes` example is incomplete as there is no definition of the components `fibs`, `isPrime` and `count` which corre-

Author's personal copy


```
1 num_t fib(pcr_var<num_t>& F, int i) {
2   if (i<2) return 1; else return F[0] + F[-1];
3 }
4
5 // id = bound_fib
6 num_t id(pcr_var<num_t>& N) { return num_t(N); }
7
8 typedef producer<       //  producer:
9   num_t,               //  output type,
10  decltype(fib), &fib, //  producer op type and pointer,
11  decltype(id), &id,   //  bound op type and pointer,
12  num_t                //  input parameters types
13 > fibs;
```

Fig. 18 Host language code and binding for `fib` producer

Table 5 Parameter wrapping operators for basic functions

Operation	Description
`operator T()`	Read implicit index value
`int idx ()`	Return current implicit index
`T operator[] (int)`	Look-ahead/behind

spond to template parameters PROD, CONS and RED from Table 4. Figure 18 completes the example giving the complete code for the `fib` operation.

Function parameters In `countFibPrimes`, basic functions have their input parameters decorated with the **`pcr_var`** template interface. This wrapper type abstracts *FXML* variables in host language code, and allows for applying *look-ahead* and *look-behind* stream operations described in Sect. 2.2.1. Table 5 summarizes the operations implemented by this interface.

Remark Decorating function parameters corresponds to the translation operator $\overline{E}^I$ presented in Sect. 5.2.

Evaluation of PCRs as functions To enable interaction with the calling environment, the **`pcr`** template provides the function call operator `operator(...)` which passes the given inputs to the called *PCR* to feed it with the stream of values to process.

Connecting PCRs The following template defines a *type* implementing the behavior of the combination of a given *PCR* with a delegate function. It provides a *C++* interface for the concepts described in Sect. 3.3.

```
1 template <typename P, typename
2  Delegate> struct connect;
```

6.2 Platform-dependent target code generation

Code generation is done through template metaprogramming to unfold *PCR* definitions into *CnC C++*. The *CnC* graph is represented by an instance of the `CnC::context` class. Every step, item and tag collection is bound to the containing context instance. The compiler generates a `CnC::context` subclass representing the nested *PCR* in a flat form. The template expansion rules convert a definition of the form **`pcr`**`<X1,...,Xn,P,C1,...,Ck,R>` into a class public inheritance chain of the form:

```
1 cnc_class(R): cnc_class(C_k) : ... :
2 cnc_class(C_1): cnc_class(P):
3 cnc_class(X_n): ... : cnc_class(X_1)
4 : CnC::context
```

Here, `cnc_class` is the synthesized implementation class for the corresponding *PCR* component. Table 6 sketches the generated subcontexts for each type of component.

CnC implementation of **`pcr_var`** Each *FXML* variable is implemented as an item collection mapping every *FXML* assignment to the assigned value. Figure 19 shows an abridged implementation of the read operations given in Tab. 5. Type `tag_t` represents a static integer vector whose dimension depends on the *PCR* nesting level of a **`pcr_var`** instance. Operation `last` returns the index value for the innermost nesting level. Type `var_t` abstracts out the type of the value stored by the **`pcr_var`**.

CnC implementation of PCR evaluation For this purpose, the translation synthesizes a *CnC*-based implementation of `operator(...)` of template **`pcr`**, as mentioned in 6.1.

6.3 Performance tuning

The *CnC* code synthesis procedure presented so far ensures correctness, but it is not focused on performance. In this sec-

Table 6 Overview of *CnC* implementations of *PCR* building blocks

PCR component	Generated *CnC* context
`pcr_in``<T>`	Context with reference to the actual item/tag collections storing values of type T
`pcr_produce``<P,...>`	Item collections for output and number of items to reduce; one tag collection to prescribe consumers; producer steps are prescribed by the input (controlled by outer scope)
`pcr_consume``<C,...>`	One item collection for output; steps prescribed by the producer
`pcr_reduce``<R,...>`	One item collection for output; steps prescribed by tag collection prescribing the producer (from outer scope)

Author's personal copy


```
 1 template <typename tag_t, typename var_t>
 2 class cnc_pcr_var : pcr_var {
 3   item_collection_t& items;
 4   tag_t current_tag;
 5   operator var_t() { return *this[0]; }
 6   int idx() { return current_tag.last(); }
 7   pcr_var& operator[] (int idx) {
 8     var_t out;
 9     items.get(current_tag+idx, out);
10     return out;
11   }
12 }
```

Fig. 19 *C++ CnC* abridged implementation of the **pcr_var** type

tion, we discuss *CnC* tuning options and their application in the *PCR*-to-*CnC* translation.

6.3.1 General considerations

Several concerns affect performance:

Locality An efficient implementation should allow taking advantage of time and space locality. This is achieved by controlling processor affinity of threads and allocation in computing nodes for distribution.

Early cancellation Eureka computations require terminating ongoing tasks as soon as a result is found.

Dependencies The translation ensures that data dependencies are enforced at runtime as it preserves program correctness. Nevertheless, explicitly knowing dependencies ahead of time may help improving the scheduling of computations.

Lifetime If a data item has potentially multiple concurrent readers, it is not trivial to determine when to dispose of it. Knowing the number of times an item will be read allows for more efficient memory management.

Distribution Distribution performance is affected by the frequency and size of data transmission between nodes, and communication latencies. Minimal communication between nodes is desired.

PCR level performance hints To allow tuning implementation performance, the *PCR C++* API provides optional template parameters for the **pcr_produce**, **pcr_consume** and **pcr_reduce** constructs, affecting runtime behavior related to the categories discussed above. In Table 7, we summarize the set of high-level tuning parameters available to the *PCR* user through the **pcr_tune** type. A tradeoff between flexibility and portability exists, having favored the latter in our approach. The set of parameters described in the table can be used by the translation and runtime of the target execution framework to tune performance. All the parameters are functions on each *FXML* index in order to allow a different tuning decision for each processed input value.

6.3.2 Performance tuning in the CnC translation

The *C++ CnC* framework supports specifying *tuners* which are optimization hints applied on a *CnC* graph step, tag, and item collections. *Tag tuners* allow for partitioning of tag ranges; which enables processing of a group of tag values by the same step instance, improving locality. Tag tuners also enable memoization of tag values. *Item tuners* let the user specify the number of expected read operations on each stored item (**get_count**). These also allow specifying in which process (for distributed scenarios) each item is to be produced/consumed on. *Step tuners* let the programmer: (a) declare data dependencies by specifying which tag values a step will consume from which item collections in a specific step execution and (b) specify the step relative priority and give hints of thread/process affinity to the scheduler. A **cancel_tuner** provides the function **cancel_all()** to signal the early termination of all running instances of a step.

Considering the performance hints described in 6.3.1, we analyze their application to the *CnC* implementation:

Table 7 *PCR* optional performance tuning hints given for each component using the `pcr_tune` type as container

PCR Parameter	Meaning
`typedef tuple<int,int> dep_t`	Index and expected number of read operations done on that index
`list<dep_t> dependencies(int p, int i)`	List of `dep_t` of parameter *p* on which the *i*-th execution depends
`int location(int i)`	Implementation-dependent execution place of the component for the given index value
`int affinity(int i)`	Implementation-dependent thread affinity for the given index value
`int priority(int i)`	Implementation-dependent priority for the given index value

Author's personal copy

- `dependencies(int p, int i)` is used in the consumer step tuner to declare data dependencies and in the item tuner of each consumed input to declare the **`get_count`** property for memory usage optimization.
- `location(int i)` is used in the item tuner in order to specify the **`produced_on`** and **`consumed_on`** distributed properties for its output item collection.
- `affinity(int i)` maps directly into the `affinity` function of the step tuner to specify thread affinity.
- whenever the termination condition *cnd* of the reducer holds (line 5 of the implementation of `reduce` described in Sec. 5.2), **`cancel_all()`** is called to cancel every producer and consumer step instance on which the reducer depends on.

6.4 Performance evaluation

In this section, we make a preliminary analysis of the current *CnC* implementation of our *PCR* parallel code library. Given that at this stage the *CnC* implementation of *PCRs* is not heavily optimized, the goal is to provide an initial overview of the running time behavior to assess the feasibility of the approach. For benchmarking analysis, we followed the recommendations given in [24]. Since our motivation is to ease parallel programming, the goal of this performance evaluation is not to analyze the scalability of applications with respect to the available number of processors, but to validate the practical interest of using automatically generated *CnC* code from *PCR* templates in place of handwritten implementations.

Benchmarking platform Test case runs were performed in a server with 4 processors (AMD Opteron(TM) Processor 6276 @ 2.30GHz) of 16 computing cores each (64 total cores) with 128GB of system memory running the 64-bit version of RedHat Linux Enterprise 6.7.

Remark The implementations compared in this section do not necessarily scale linearly with the number of physical processors; linear scaling depends on the nature of the parallel computation, the input data sets, and the hardware architecture. We stress that the focus of this section is to compare different implementations solving the same problem and to compare *PCR*-generated *CnC* code against a similar pure *CnC* implementation.

6.4.1 N-Queens performance study

In this section, we compare the three implementations of N-Queens presented in 4.3. We use the *eureka* versions, finishing the computation as soon as the first result is found. We set the time limit to one minute.

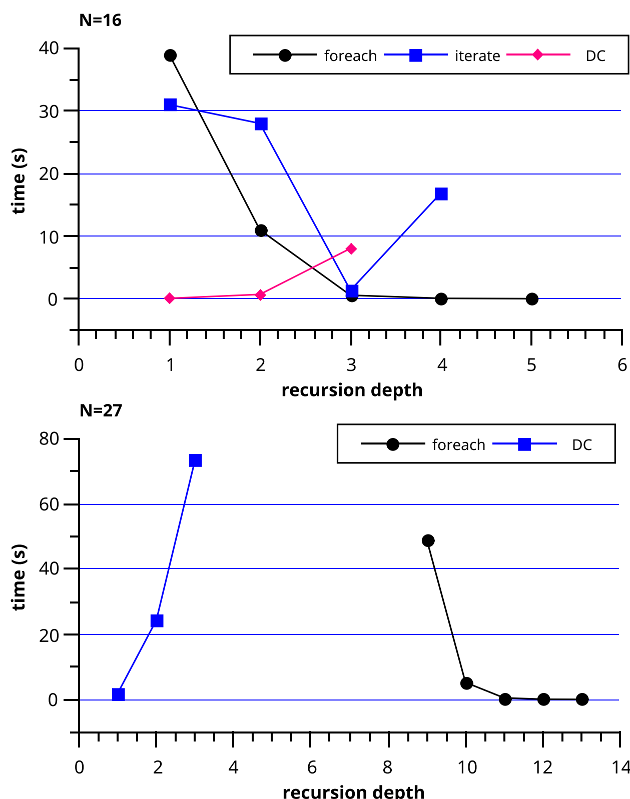


Fig. 20 Runtime comparison between N-Queens *PCR* implementations with varying recursion depth for different N

First, we analyze running times for various recursion depths and problem sizes (Fig. 20).

The **`iterate`** implementation with $N = 16$ shows its best performance with recursion depth 3 and reaches the time limit for recursion depths higher than 4. For $N = 27$, this implementation does not finish within the time limit for any recursion depth. The divide and conquer implementation shows much better running times than **`iterate`**. We observe that increasing the recursion depth worsens running times, eventually hitting the time limit with recursion depths higher than 3. This implementation does not scale with increasing problem sizes, as shown in the $N = 27$ graph. The **`feedbackloop`** implementation performs badly for small recursion depths, reaching the time limit. We observe that increasing recursion depth improves running times, eventually achieving better times than the other implementations. Figure 21 summarizes the best running time achieved in each implementation for different problem sizes, showing that only **`feedbackloop`** scales beyond $N = 29$.

As a second performance measure, Fig. 22 box-plots the running times of the best *PCR* implementation we identified so far (based on **`feedbackloop`**) compared to the hand-coded *CnC* version which is part of the *CnC* suite. This implementation follows the *workpile* pattern. Both implementations use equal recursion depth. All runs were repeated

Author's personal copy

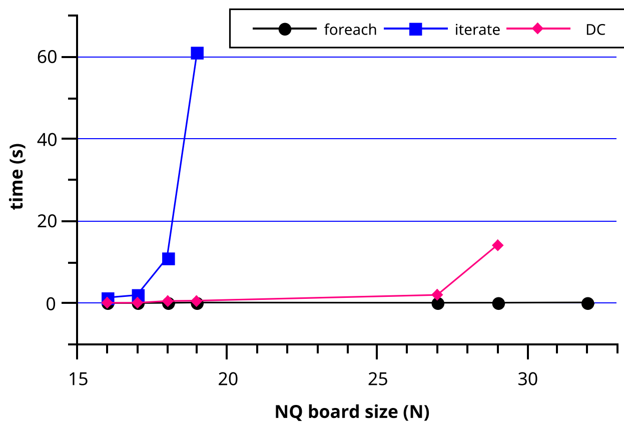


Fig. 21 Summary of the best running time in each implementation for $N = 16 \ldots 32$

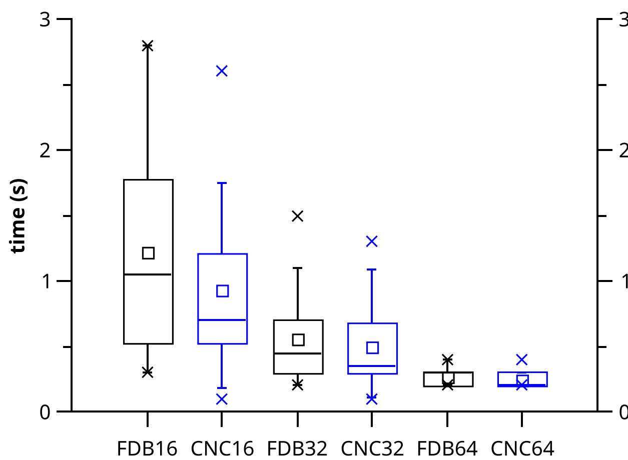


Fig. 22 Time comparison between the **`feedbackloop`** N-Queens implementation and a hand-coded CnC implementation

20 times and were performed using 16, 32 and 64 cores. With 16 cores, both implementations show high running time variability, **`feedbackloop`** having a slightly worse maximum time. With 32 cores, variability reduces, but a small advantage is still apparent in the *CnC* implementation. Finally, with 64 cores little variability is observed; with both implementations having almost equal running times.

6.4.2 Count-words performance study

In this section, we perform a performance comparison of the two *PCR* solutions proposed for the *count-words* problem in Sect. 4.2. The actual implementations partition the file in *chunks* of several lines. Runs were repeated 10 times, and the median of the measurements was recorded. An input text file of 111M lines with a total size of 8GB was used.

Figure 23 shows running times of *PCR*-based implementations for counting 9 words with a chunk size of 10K lines. It also compares *PCR* versions against a pure *CnC* implementation of *count-words* taken from the *CnC*

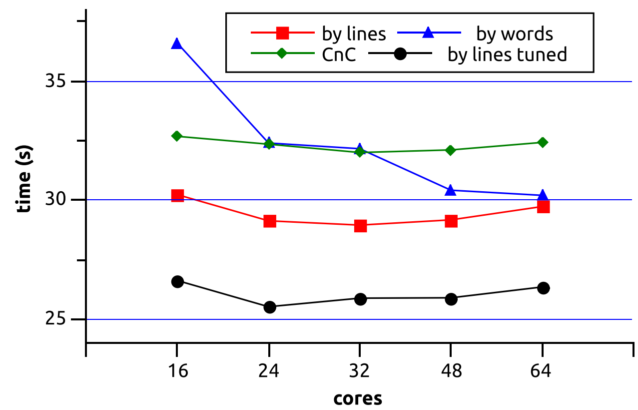


Fig. 23 Comparison of count-words implementations for increasing numbers of available computing cores

samples which uses a parallel reduce graph. Globally, `count-words-by-lines` exhibits better performance than `count-words-by-words`, but the gap tends to diminish for larger number of cores. For more than 32 cores, *PCR*-based versions show slightly better running times than the *CnC* one. We also analyzed the effect of providing tuning hints at *PCR*-level for `count-words-by-lines`. From its specification on Fig. 10, it follows that variables `l` and `c` are written as many times as lines are produced, but read *exactly once* by the consumer and by the reducer, respectively. A simple tuning action is to specify this fact, with `dependencies`, enabling the runtime to free memory as soon as each value is read. Figure 23 shows that the tuned version delivers the best performance.

Clearly, the number of words to count and the chunk size used for partitioning the input file affect performance. Figure 24 shows running times of both non-tuned *PCR*-based implementations on 64 cores, for counting 100 words with increasing chunk sizes. The running time of `count-words-by-lines` increases steadily with the chunk size. This is consistent with the fact that it searches each word in every chunk sequentially, and so bigger chunks reduce the exploitable parallelism. On the other hand, `count-words-by-words`, which counts each word in parallel, takes advantage of a larger number of words to count while too many small chunks affect its performance negatively. The figure shows that its running time goes down up to a chunk size of 100K, without further improvement. For chunk sizes of 10M, `count-words-by-lines` shows an abrupt decrease in performance compared to `count-words-by-words`. This is consistent with the fact that for this size there are less chunks than the number of available processors. In this scenario, `count-words-by-words` benefits of the extra dimension of parallelism.

Finally, to evaluate the memory gained by tuning, we measured the maximum Resident Set Size (RSS) of the runs

Author's personal copy

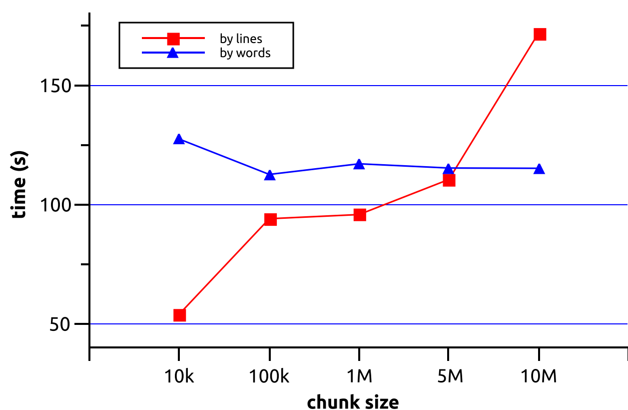


Fig. 24 Comparison of two count-words implementations with varying chunk sizes

using 64, 48, 32, 16, 8 and 4 cores. With 4 cores, the tuned implementation consumes in average around 45% less memory. For 8 cores and up, the median RSS peak drops from about 8GB to 10MB, yielding an impressive reduction ranging between 98 and 99%.

Final remarks The performance measurements obtained for the case studies illustrate that our approach can ease comparing different parallelization strategies written as `PCRs`. Also, comparisons made against pure `CnC` implementations show that even without tuning the generated code can behave comparably in terms of performance and can achieve the scalability of a hand-coded `CnC` implementation.

7 Related work

`PCRs` are related to both algorithmic skeleton frameworks [20] and stream processing models [33]. Following the conclusions summarized in [34], we analyze the capabilities of current streaming and skeleton frameworks with respect to: (a) exposing task, data, and pipeline parallelism; (b) exposing the presence of sliding window parallelism; (c) preventing the usage of stateful filters; (d) naturally describing complex computation topologies; and (e) keeping the parallel problem description platform-agnostic.

Hereinafter, we refer to the atomic components of a framework as *Single/Multiple Input* (SI/MI) or *Single/Multiple Output* (SO/MO) with respect to the number of input/output channels a component can have.

`PCRs` cover the mentioned requirements by (a) execution semantics allowing for concurrent instances of the same consumer and chaining of consumers, together with `FXML` dependencies describing data parallelism; (b) providing look-ahead and look-behind operations on `PCR` variables in order to enable expression of sliding window and stencil computations; (c) use of pure functions in basic code; (d) supporting nested `PCRs` with named parameters for the producer, consumer and reducer inputs (MISO), allowing for complex connections of `PCRs` while keeping a functional behavior; and (e) separating `PCR` specifications from the execution platform.

Algorithmic skeleton frameworks We limit the discussion to those which are more closely related to `PCRs`.

Quaff [18] declares a concurrent computation coordination topology by the composition of basic skeletons. It uses legacy code functions restricted to a single input parameter (SISO). Another limitation is that the `farm` construct requires fixing at compile time the number of processors to be used at runtime. In [17], a CSP [23] semantics of *Quaff* is provided without showing the implementation correctness with respect to this semantics.

Muesli [11] supports *task parallelism* by constructing a topology of connected processes and *data parallelism* by using distributed data structures. The topology is constructed by composing object instances (*dynamic polymorphism*) of basic skeletons (Pipe, Filter, Farm) and algorithmic ones (DivideAndConquer, BranchAndbound). All communication in data-parallel structures is explicit so the problem description has embedded communication logic. No formal model is provided.

MapReduce [14] defines a simple computation model together with a reference implementation handling work distribution. An extension [15] of the *MapReduce* implementation allows iterative computations, similar to `PCRs` **`iterate`** construct. `PCRs` extend *MapReduce* with the concepts of composition by nesting, iteration, producer components, and "native" look-ahead/look-behind.

The *Orleans Skeleton Library* [27] follows the Bulk Synchronous Parallel (BSP) [36] computation model with SISO basic atoms. Its semantics is formalized by term rewriting, but no formal relationship with the actual execution model (MPI) is provided.

SkePU [16] is a multicore/GPU-oriented skeleton library. It supports MISO computations but limits the number of input parameters to 3. No formal semantics of the skeleton execution is given.

STAPL [41] focuses on compositional reuse of skeletons and provides a domain-specific syntax for describing complex component interconnections. It supports iteration, but it lacks formal semantics.

Summarizing, to the best of our knowledge, most algorithmic skeleton approaches are restricted to SISO components, lack formal semantics, neither support iteration nor eureka computations, and do not provide specific support for look-ahead and look-behind.

Stream programming models *StreamIT* [35] is a programming language based on *pipelines* of *filters*. Each filter

Author's personal copy

has one input and one output streams. Communication is achieved by push, pop and peek (look-ahead) operations on streams. Complex topologies can be assembled using FeedbackLoop (iteration) and SplitJoin connectors which first separate and later combine items from/to one stream to/from many. The main differences with `PCRs` are: (a) destructive pop operations on streams preventing filters from using the full history of stream values (look-behind); (b) lack of direct support of filters with multiple independent input streams; and (c) SplitJoin is restricted to duplicate and round-robin, while `PCR` **`connect`** supports any *user-defined* policy.

FastFlow [2] is a layered stream programming framework with a bottom layer of SISO components connected by anonymous channels used to provide MISO and MIMO components in the middle layer; a top layer of skeletons completes the framework. It does not provide look-ahead/behind capabilities.

RISC-pb^{2}l [1] is a set of parallel building blocks implemented over FastFlow allowing the construction of complex parallel computation patterns. RISC does not model directly complex data connections because channels are anonymous and inherits FastFlow's lack of support for look-ahead/look-behind in its input channels.

S-Net [21] is a streaming computation model consisting of stateless SISO *boxes* interconnected by streams of records forming an acyclic computation graph. *Record subtyping* is provided to enable composition and adaptation of boxes to different environments. The S-Net model does not provide look-ahead/behind capabilities on its input streams and can not easily model complex data connections because channels are anonymous.

In summary, none of the cited frameworks fully supports the concepts of look-ahead/behind. Besides, they rely on unnamed connections making it difficult to specify complex interactions between components.

Other models *Multi-BSP* [37] is an extension of BSP [36]. This multi-level tree model aims at describing the concurrency capabilities of an architecture made of a combination of hardware and software elements. Each level defines runtime parameters such as number of processors, synchronization/-communication costs, and cache sizes. Its goal is to write concurrent algorithms aware of architectural parameters in order to achieve an optimal implementation for any architecture for any value of the parameters in some specifiable sense. An important difference with `PCRs` is that *Multi-BSP* imposes several requirements on architectures to support it, while `PCRs` are completely architecture-oblivious.

Dryad [26] models coarse-grained computations as directed acyclic graphs with sequential pieces of code in the nodes. The description language is flat although a graph composition operator is provided. The main focus is on implementation performance and scheduling adaptation to available resources.

Clearly, both models are interesting targets for automated code generation from `PCR` specifications.

8 Conclusions

From a theoretical point of view, we defined a composable parallel pattern which combines concepts like collectives, eurekas, iteration, recursion, and stream programming. We formalized its abstract semantics and proposed a concrete one through a formal translation of `PCRs` into `CnC`. We illustrated with several case studies from different application domains that `PCRs` can ease writing parallel programs.

Tool-wise, we developed a framework which consists of (i) a library for writing platform-independent `PCRs`, and (ii) a code generation engine based on template metaprogramming for translating them into `CnC`-based implementations. It is worth mentioning that `PCR C++` templates are designed to enhance portability in the sense that different target runtimes could be used for code generation.

The framework provides theory-driven automated generation of parallel code from platform-independent, high-level, structured descriptions. The experimental results provided evidence that the synthesized code can achieve performances which are comparable to those of low-level, unstructured, platform-dependent programs. This is a sign that the approach could be feasible in practice. From a software engineering perspective, it would result in less coding effort, more reliability and faster prototyping of several implementations.

One envisaged extension of the framework consists in enabling *asynchronous* composition of `PCRs`, letting caller and callee proceed in parallel by not blocking the former until it needs the latter's output or it returns (e.g., Futures [38], Cilk [7]). Other future research directions include experimenting `PCRs` on other platforms, such as distributed clusters, many-core architectures, and GPUs; running experiments on groups of programmers to evaluate average code size, coding time, etc; and developing concrete semantics composing heterogeneous parallel computation models.

Acknowledgements Partially funded by LIA INFINIS (CNRS, Université Paris Diderot, CONICET, Universidad de Buenos Aires), PEDECIBA and SNI. Thanks to CSC-CONICET for granting use of cluster TUPAC.

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming frame-

Author's personal copy

work implementation. Int. J. Parallel Program. **42**(6), 1012–1031 (2014)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing* (2014)
3. Anand, C.K., Kahl, W.: Synthesizing and verifying multicore parallelism in categories of nested code graphs. In: Process Algebra for Parallel and Distributed Processing, vol. 2, pp. 3–45. Chapman & Hall (2009)
4. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. CACM **52**(10), 56–67 (2009)
5. Assayad, I., Bertin, V., Defaut, F.-X., Gerner, P., Quévreux, O., Yovine, S.: Jahuel: a formal framework for software synthesis. In Formal Methods and Software Engineering, pp. 204–218. Springer (2005)
6. Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., Loidl, H.-W.: A survey of high-level parallel programming models (2013)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
8. Buchty, R., Karl, V., Weiss, W., Weiss, J.-P.: A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. Concurr. Comput. Pract. Exp. **24**(7), 663–675 (2012)
9. Budimlić, Z., et al.: Concurrent collections. Sci. Program. **18**(3), 203–217 (2010)
10. Chamberlain, B.L.: Chapel. MIT Press, Cambridge (2015)
11. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster skeleton library Muesli—a comprehensive overview. Technical report, Münster (2009)
12. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman, London (1989)
13. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
14. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. CACM **51**(1), 107–113 (2008)
15. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: 19th ACM International Symposium on High Performance Distributed Computing, pp. 810–818, New York, NY, USA. ACM (2010)
16. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In Proceedings of 4th International Workshop on High-level parallel programming and applications, pp. 5–14. ACM (2010)
17. Falcou, J., Sérot, J.: Formal semantics applied to the implementation of a skeleton-based parallel programming library. Parallel Comput. Archit. Algorithms Appl. **38**, 243–252 (2008)
18. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.-T.: Quaff: efficient c++ design for parallel skeletons. Parallel Comput. **32**(7), 604–615 (2006)
19. González-Vélez, H., Cole, M.: Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. Concurr. Comput. Pract. Exp. **22**(15), 2073–2094 (2010)
20. Horacio, G.-V., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exp. **40**(12), 1135–1160 (2010)
21. Grelck, C., Scholz, S.-B., Shafarenko, A.: Asynchronous stream processing with s-net. Int. J. Parallel Program. **38**(1), 38–67 (2010)
22. Hempel, R.: The MPI standard for message passing. In: International Conference on High-Performance Computing and Networking, pp. 247–252. Springer (1994)
23. Hoare, C.A.R.: Communicating sequential processes. In: The Origin of Concurrent Programming, pp. 413–443. Springer (1978)
24. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, pp. 73:1–73:12, New York, NY, USA, ACM (2015)
25. Imam, S., Sarkar, V.: The Eureka programming model for speculative task parallelism. In: LIPIcs-Leibniz Internatinal Proceedings in Informatics, vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
26. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: ACM SIGOPS Operating Systems Review, vol. 41, No. 3, pp. 59–72. ACM (2007)
27. Javed, N., Loulergue, F.: OSL: optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. Springer, Berlin (2009)
28. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. ACM Sigplan Notices **44**(10), 227–242 (2009)
29. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM Sigplan Notices, vol. 43, No. 3, pp. 329–339. ACM (2008)
30. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, Amsterdam (2012)
31. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media Inc, Sebastopol (2007)
32. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: Proceedings of 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 271. ACM (2007)
33. Stephens, R.: A survey of stream processing. Acta Inform. **34**, 491–541 (1997)
34. Thies, W., Amarasinghe, S.P.: An empirical characterization of stream programs and its implications for language and compiler design. In: Knoop, J., Salapura, V., Gschwind, M. (eds.) 19th International Conference on Parallel Architecture and Compilation Techniques, pp. 365–376. ACM (2010)
35. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: Compiler Construction, pp. 179–196. Springer (2002)
36. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103–111 (1990)
37. Valiant, L.G.: A bridging model for multi-core computing. In: Algorithms-ESA 2008, pp. 13–28. Springer (2008)
38. Walker, E.F., Floyd, R., Neves, P.: Asynchronous remote operation execution in distributed systems. In: Proceedings of 10th International Conference on Distributed Computing Systems, pp. 253–259 (1990)
39. Yovine, S., Assayad, I., Defaut, F.-X., Zanconi, M., Basu, A.: A formal approach to derivation of concurrent implementations in software product lines. In: Alexander, M., Gardner, W. (eds.) Algebra for Parallel and Distributed Processing, Chapter 11, pp. 359–401. Chapman and Hall, CRC Press, Boca Raton (2008)
40. Zandifar, M., Jabbar, M.A., Majidi, A., Keyes, D., Amato, N.M., Rauchwerger, L.: Composing algorithmic skeletons to express high-performance scientific applications. In: Proceedings of 29th ACM on International Conference on Supercomputing, pp. 415–424. ACM (2015)
41. Zandifar, M., Thomas, N., Amato, N.M., Rauchwerger, L.: The STAPL Skeleton Framework. Springer, Cham (2015)