Choreographies in Practice*

Luís Cruz-Filipe, Fabrizio Montesi

University of Southern Denmark, Department of Mathematics and Computer Science
{lcf,fmontesi}@imada.sdu.dk

Abstract. Choreographic Programming is a development methodology for concurrent software that guarantees correctness by construction. The key to this paradigm is to disallow mismatched I/O operations in programs, called choreographies, and then mechanically synthesise distributed implementations in terms of standard process models via a mechanism known as EndPoint Projection (EPP).
Despite the promise of choreographic programming, there is still a lack of practical evaluations that illustrate the applicability of choreographies to concrete computational problems with standard concurrent solutions. In this work, we explore the potential of choreographies by using Procedural Choreographies (PC), a model that we recently proposed, to write distributed algorithms for sorting (Quicksort), solving linear equations (Gaussian elimination), and computing Fast Fourier Transform. We discuss the lessons learned from this experiment, giving possible directions for the usage and future improvements of choreography languages.

Keywords: Choreographies, Correctness by Construction, Distributed Algorithms

1 Introduction

Choreographic Programming is an emerging paradigm for the programming of concurrent software based on message passing [18]. The key aspect of this paradigm is that programs are choreographies, i.e., global descriptions of communications based on an "Alice and Bob" notation inherited from security protocol notation [21]. Since the syntax of such notation disallows the writing of mismatched I/O actions, choreographies always describe deadlock-free systems by construction. Given a choreography, a distributed implementation can be projected automatically (synthesis) in terms of a process model. This transformation is typically called EndPoint Projection (EPP) [2,3]. If EPP is defined correctly, then the generated code behaves exactly as specified by the originating choreography, yielding a correctness-by-construction result: since a choreography cannot describe deadlocks, the generated process implementations must also be deadlock-free. Previous works have presented formal models for capturing different aspects of choreographic programming, e.g., web services [2,11], multiparty

* Supported by *CRC (Choreographies for Reliable and efficient Communication software)*, grant no. DFF–4005-00304 from the Danish Council for Independent Research.

sessions and asynchrony [3], runtime adaptation [23], modular development [20], protocol compliance [3,4], and computational expressivity [7]. More in general, looking also at other applications rather than just programming system implementations, choreographies have been investigated in the realms of type theory [14], automata theory [10,16], formal logics [5], and service contracts [1,24].

Despite the rising interest in choreographies found in the communities of programming languages and concurrent computing, there is still a lack of evidence about what kind of nontrivial programs can actually be written with choreographic programming. This is due to the young age of the paradigm [19]. Indeed, most works on languages for choreographic programming still focus on showcasing representative toy examples (e.g., [2,3,6,11,18,20]), rather than giving a comprehensive practical evaluation of how known algorithms can be implemented using choreographies. Here, we aim at contributing to filling this gap by investigating how choreographies can be used to tackle some common computational problems in the setting of concurrent programming.

Our investigation is based on the language of Procedural Choreographies (PC), and its corresponding process calculus of Procedural Processes (PP), which we presented in [8]. PC, introduced in § 2, is a choreography language developed with the aim of capturing divide-and-conquer concurrent algorithms, by extending previous choreography models with primitives for parameterised procedures. Like several other choreography languages (e.g., [3,20]), PC supports implicit parallelism by means of a flexible semantics that allows non-interfering communications to take place in any order.

In this work, we provide an empirical evaluation of the expressivity of PC, by showing how it can be used to program some representative and standard concurrent algorithms: Quicksort (§ 3), Gaussian elimination (§ 4), and Fast Fourier Transform (§ 5). As a consequence of using choreographies, all these implementations are guaranteed to be deadlock-free. We also illustrate how implicit parallelism has the surprising effect of automatically giving concurrent behaviour to traditional sequential implementations of these algorithms. Our exploration brings us to the limits of the expressivity of PC, which arise when trying to tackle distributed graph algorithms (§ 6), due to the lack of primitives for accessing the structure of process networks, e.g., broadcasting a message to neighbouring processes.

2 Background

In this section, we recap the definition of Procedural Choreographies (PC), our choreography language, and its main properties. We refer the reader to the original presentation of PC, in [8], for a more comprehensive technical discussion.

2.1 Procedural Choreographies

The syntax of PC is introduced in Figure 1. A procedural choreography is a pair $\langle \mathcal{D}, C \rangle$, where C is a choreography and $\mathcal{D}$ is a set of procedure definitions. Process

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \eta ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}.f \mid \mathsf{p} \texttt{ -> } \mathsf{q}[l] \mid \mathsf{p} \texttt{ start } \mathsf{q} \mid \mathsf{p} \texttt{ : } \mathsf{q} \texttt{ <-> } \mathsf{r}$$
$$\mathcal{D} ::= X(\tilde{\mathsf{q}}) = C, \mathcal{D} \mid \emptyset \qquad I ::= \texttt{if } \mathsf{p}.e \texttt{ then } C_1 \texttt{ else } C_2 \mid X\langle\tilde{\mathsf{p}}\rangle \mid \mathbf{0}$$

Fig. 1. Procedural Choreographies, Syntax.

names, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$, identify processes that execute concurrently. Each process p is equipped with a memory cell that stores a single value of a fixed type. In the remainder, we will omit such types in the language and our examples since they can always be inferred using the technique given in [8]. Statements in a choreography can either be communication actions (η) or compound instructions (I), and both can have continuations. Term $\mathbf{0}$ is the terminated choreography, which we sometimes omit. The term $\mathbf{0}; A$ is needed at runtime to capture the termination of procedure calls with continuations.

Processes communicate via direct references (names) to each other. Communications are synchronous, as they are simple and suffice for our purposes here, but can be made asynchronous by adopting the asynchronous extension proposed in [3]. In a value communication $\mathsf{p}.e$ -> $\mathsf{q}.f$, process p sends the result of evaluating expression e to q; the expression e can contain the placeholder c, which is replaced at runtime with the data stored at process p. When q receives the value from p, it applies to it the (total) function f, of the form $\lambda x.e'$, replacing the formal parameter x with the value sent by p. The result of the computation will be stored in q. The expression e', the body of f, can also contain the placeholder c, allowing it to read the contents of q's memory. We assume that expressions and functions are written in a pure functional language, which we leave unspecified.

The selection term p -> $\mathsf{q}[l]$ is standard, as in session types [13]: p communicates to q its choice of label l. Labels l range over an infinite enumerable set.

In term p start q, process p spawns the new process q. Process name q is bound in the continuation C of p start $\mathsf{q}; C$. Also, after executing p start q, process p is assumed to be the only process who knows the name of process q. (Or, in other words, process p is the only process connected to q.) In order to propagate knowledge of q to other processes, PC includes the action p : q <-> r, read "p introduces q and r", where p, q and r are distinct.

In a conditional term if $\mathsf{p}.e$ then C_1 else C_2, process p evaluates expression e to choose between the possible continuations C_1 and C_2.

The set $\mathcal{D}$ defines global procedures that can be invoked in choreographies. Term $X(\tilde{\mathsf{q}}) = C$ defines a procedure X with body C, which can be used anywhere in $\langle \mathcal{D}, C \rangle$ – in particular, inside the definitions of X and other procedures. The names $\tilde{\mathsf{q}}$ are bound to C, and they are assumed to be exactly the free process names in C. The set $\mathcal{D}$ contains at most one definition for each procedure name. Term $X\langle\tilde{\mathsf{p}}\rangle$ then invokes procedure X, instantiating its parameters with the processes $\tilde{\mathsf{p}}$.

The semantics of PC, which we do not detail, is a reduction semantics that relies on two extra elements: a total state function that assigns to each process the value it stores, representing the local memory of processes; and, a connection graph that keeps track of which processes know (are connected to) which other processes [8]. In particular, two processes can only communicate if there is an edge between them in the connection graph. Therefore, it is possible for choreographies to deadlock (be unable to reduce) because of errors in the programming of communications: if two processes are supposed to communicate but they are not connected according to the connection graph, the choreography gets stuck. This issue is addressed by a simple typing discipline that we do not discuss here (see [8]). When a choreography is well-typed in PC, it is guaranteed to be deadlock-free.

Theorem 1 (Deadlock-freedom [8]). *Let $\langle \mathcal{D}, C \rangle$ be a well-typed procedural choreography. Then, $\langle \mathcal{D}, C \rangle$ is deadlock-free.*

2.2 Procedural Processes

Choreographies in PC are compiled into a distributed implementation represented in terms of a process calculus: the calculus of Procedural Processes (PP).

The syntax of PP is reported in Figure 2. A term $\mathsf{p} \triangleright_v B$ is a process, where p is its name, v is the value stored in its memory cell, and B is its behaviour. Networks, ranged over by N, M, are parallel compositions of processes, where $\mathbf{0}$ is the inactive network. Finally, $\langle \mathcal{B}, N \rangle$ is a procedural network, where $\mathcal{B}$ defines the procedures that the processes in N may invoke. Values, expressions and functions are as in PC.

We comment on behaviours. A send term $\mathsf{q}!e; B$ sends the evaluation of expression e to process q, and then proceeds as B. Term $\mathsf{p}?f; B$ is the dual receiving action: it receives a value from process p, combines it with the value in memory cell of the process executing the behaviour as specified by f, and then proceeds as B. Term $\mathsf{q}!!\mathsf{r}$ sends process name r to q and process name q to r, making q and r "aware" of each other. The dual action is $\mathsf{p}?\mathsf{r}$, which receives a process name from p that replaces the bound variable r in the continuation. Term $\mathsf{q} \oplus l; B$ sends the selection of a label l to process q. Selections are received by the branching term $\mathsf{p}\&\{l_i : B_i\}_{i \in I}$, which can receive a selection for any of the labels l_i and proceed according to B_i. Branching terms must offer at least one branch. Term $\mathsf{start}\, \mathsf{q} \triangleright B_2; B_1$ starts a new process (with a fresh name) executing B_2, proceeding

$$
\begin{array}{rcl}
B & ::= & \mathsf{q}!e; B \mid \mathsf{p}?f; B \mid \mathsf{q}!!\mathsf{r}; B \mid \mathsf{p}?\mathsf{r}; B \mid \mathsf{q} \oplus l; B \mid \mathsf{p}\&\{l_i : B_i\}_{i \in I}; B \mid \mathbf{0} \\
 & & \mid \mathsf{start}\, \mathsf{q} \triangleright B_2; B_1 \mid \mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2; B \mid X\langle \tilde{\mathsf{p}} \rangle; B \mid \mathbf{0}; B \\
N, M & ::= & \mathsf{p} \triangleright_v B \quad \mid \quad N \mid M \quad \mid \quad \mathbf{0} \\
\mathcal{B} & ::= & X(\tilde{\mathsf{q}}) = B, \mathcal{B} \mid \emptyset
\end{array}
$$

Fig. 2. Procedural Processes, Syntax.

$$[\![\mathsf{p}.e \texttt{ -> } \mathsf{q}.f; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q}!e; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?f; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{else} \end{cases} \qquad [\![\mathsf{p} \texttt{ -> } \mathsf{q}[l]; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q} \oplus l; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}\&\{l : [\![C]\!]_\mathsf{r}\}; \mathbf{0} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{else} \end{cases}$$

$$[\![\mathsf{p} : \mathsf{q} \texttt{ <-> } \mathsf{r}; C]\!]_\mathsf{s} = \begin{cases} \mathsf{q}!!\mathsf{r}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{p} \\ \mathsf{p}?\mathsf{r}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{q} \\ \mathsf{p}?\mathsf{q}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{r} \\ [\![C]\!]_\mathsf{s} & \text{else} \end{cases} \qquad [\![X\langle \tilde{\mathsf{p}} \rangle; C]\!]_\mathsf{r} = \begin{cases} X_i\langle \tilde{\mathsf{p}} \rangle; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p}_i \\ [\![C]\!]_\mathsf{r} & \text{else} \end{cases}$$

$$[\![\mathbf{0}]\!]_\mathsf{r} = \mathbf{0} \qquad [\![\mathbf{0}; C]\!]_\mathsf{r} = [\![C]\!]_\mathsf{r}$$

$$[\![\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_\mathsf{r} = \begin{cases} \mathsf{if}\ e\ \mathsf{then}\ [\![C_1]\!]_\mathsf{r}\ \mathsf{else}\ [\![C_2]\!]_\mathsf{r}; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ ([\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r}); [\![C]\!]_\mathsf{r} & \text{else} \end{cases}$$

$$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}; C]\!]_\mathsf{r} = \begin{cases} \mathsf{start}\ \mathsf{q} \triangleright [\![C]\!]_\mathsf{q}; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_\mathsf{r} & \text{else} \end{cases}$$

Fig. 3. Procedural Choreographies, Behaviour Projection.

in parallel as B_1. The other terms are standard (conditionals, procedure calls, and termination), while procedural definitions are stored globally as in PC.

Some terms bind names: $\mathsf{start}\ \mathsf{q} \triangleright B_2; B_1$ binds q in B_1, and $\mathsf{p}?\mathsf{r}; B$ binds r in B. The semantics of PP is again a reduction semantics, capturing the intuitive description of the operators given above.

2.3 EndPoint Projection (EPP)

We now exhibit the compilation of procedural choreographies in PC to processes in PP.

Behaviour Projection. We start by defining how to project the behaviour of a single process p, a partial function denoted $[\![C]\!]_\mathsf{p}$. The rules defining behaviour projection are given in Figure 3. Each choreography term is projected to the local action of the process that we are projecting. For example, for a communication term $\mathsf{p}.e$ -> $\mathsf{q}.f$, we project a send action if we are projecting the sender process p, a receive action if we are projecting the receiver process q, or we just proceed with the continuation otherwise.

The rule for projecting a conditional uses the standard (and partial) merging operator $\sqcup$: $B \sqcup B'$ is isomorphic to B and B' up to branching, where the branches of B or B' with distinct labels are also included [2]. Merging allows the process that decides a conditional to inform the other processes of its choice later on, using selections [15]. Unlike in previous work, our conditionals have continuations, which have to be moved inside the different branches if they do not coincide for all cases (see the example in § 3).

Building on behaviour projection, we define how to project the set $\mathcal{D}$ of procedure definitions. Formally, the projection $[\![\mathcal{D}]\!]$ is the component-wise extension

of the projection of a single procedure, defined below.

$$[\![X(\tilde{\mathsf{q}}) = C]\!] \;=\; X_1(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_1}, \ldots, X_n(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_n} \qquad \text{where } \tilde{\mathsf{q}} = \mathsf{q}_1, \ldots, \mathsf{q}_n\,.$$

Definition 1 (EPP from PC to PP). *Given a procedural choreography $\langle \mathcal{D}, C\rangle$ and a map of initial process values σ, the endpoint projection $[\![\mathcal{D}, C, \sigma]\!]$ is defined as the parallel composition of the processes in C with all global definitions derived from $\mathcal{D}$:*

$$[\![\mathcal{D}, C, \sigma]\!] \;=\; \langle [\![\mathcal{D}]\!], [\![C, \sigma]\!]\rangle \;=\; \left\langle [\![\mathcal{D}]\!], \textstyle\prod_{\mathsf{p}\in \mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}} \right\rangle$$

where $[\![C, \sigma]\!]$, the EPP of C wrt state σ, is independent of $\mathcal{D}$.

Above, σ is a total function mapping process names to their current values. For our purposes here, we will only consider a default mapping that assigns an initial undefined value to each process, and omit further discussions on σ since it does not influence our presentation in any way (see [8] for details).

Properties. EPP guarantees the typical operational correspondence between PC and PP: the projection of a choreography implements exactly the behaviour of the originating choreography. This implies, in particular, that the projections of typable PC terms never deadlock.

3 Quicksort

In this section, we illustrate PC's capability of supporting the programming of divide-and-conquer algorithms, by providing a detailed implementation of (concurrent) Quicksort.

We begin by defining procedure `split`, which splits the (non-empty) list stored at `p` among three processes: $\mathtt{q}_<$, $\mathtt{q}_=$ and $\mathtt{q}_>$. Before giving the code for `split`, we describe the (standard) auxiliary functions and procedures that we are going to use. We assume that all processes store objects of type `List(T)`, where `T` is some type. We also assume that these lists are implemented such that the following operations are supported and take constant time: accessing the first element (`fst`); accessing the second element (`snd`); checking that the length of a list is at most 1 (`short`); appending an element (`add`); and, appending another list (`append`). This can be readily achieved, for example, by an implementation of linked lists with pointers to the first, second and last node (and `short` simply checks where the pointer to the second node is null). We use the predicates `fst<snd` and `fst>snd` to test whether the first element of the list at a process is, respectively, smaller or greater than the second element. Finally, the procedure `pop2` (which we omit) removes the second element from the list at its argument process.

We use the abbreviation `p -> q1,...,qn[l]` to signify that `p` sends the label `l` to the processes `q1`,...,`qn`, i.e., as an abbreviation for the sequence of selections `p -> q1[l]; ...; p -> qn[l]`. We can now show the code for `split`, reported in the following.

```
split(p,q<,q=,q>) =
  if p.short then p -> q<,q=,q>[stop]; p.fst -> q=.add
  else if p.fst<snd then p -> q<[get]; p.snd -> q<.add; p -> q=,q>[skip]
       else if p.fst>snd then p -> q>[get]; p.snd -> q>.add; p -> q<,q=[skip]
            else p -> q=[get]; p.snd -> q=.add; p -> q<,q>[skip]
       ;
       pop2<p>; split<p,q<,q=,q>>
```

Procedure `split` starts by testing whether the list at process `p` is `short`. If so, its element is stored in process $q_=$ and the procedure terminates. Otherwise, we test whether the second element in the list is smaller, greater, or equal to the first element in the list, and add it to the respective process $q_<$, $q_=$, or $q_>$; then, we pop the second element of the list at `p` and recursively invoke `split`. When `split` terminates, we know that all elements in $q_<$ and $q_>$ are respectively smaller and greater than those in $q_=$.

Sending the label `skip` to the processes that will not receive messages in an iteration is required for projectability. (So that they know whether they will receive a value or not.) Using `split` we can implement a robust version of Quicksort (in the sense that it works with lists containing duplicates), the procedure `QS` below. We use `p start q1, ..., qn` as a shortcut for the sequence `p start q1; ...; p start qn`. Observe that `split` is only called when `p` stores a non-empty list.

```
QS(p) = if p.short then 0
        else p.start q<,q=,q>;
             split<p,q<,q=,q>>; QS<q<>; QS<q>>;
             q<.c -> p.id; q=.c -> p.append; q>.c -> p.append
```

Procedure `QS` implements Quicksort using its standard recursive structure. However, the recursive calls run completely in parallel here. Indeed, the created processes $q_<$, $q_=$ and $q_>$ do not even have references to each other, so they cannot exchange messages. Therefore, the network of processes becomes tree-like, as exemplified in Figure 4.

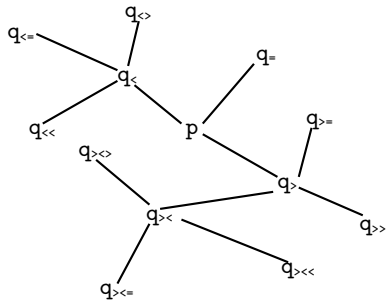

Fig. 4. Example of a network connection graph after some recursive calls of `QS`.

Applying EPP, we get the following process procedures, where we simplified the projections of procedure definitions to include only the arguments that are actually used inside the procedures (see [8]).

```
split_p(p,q< ,q= ,q> ) =
  if short then q<⊕stop; q=⊕stop; q>⊕stop; q= !fst
  else if fst<snd then q<⊕get; q< !snd; q=⊕skip; q>⊕skip
       else if fst>snd then q>⊕get; q> !snd; q<⊕skip; q=⊕skip
            else q=⊕get; q= !snd; q<⊕skip; q>⊕skip
       ;
       pop2<p>; split_p<p,q< ,q= ,q> >

split_q<(p,q) = p&{stop: 0, get: p?add;split_q<(p,q), skip: split_q<(p,q)}

split_q=(p,q) = p&{stop: p?add, get: p?add;split_q=(p,q), skip: split_q=(p,q)}

split_q>(p,q) = p&{stop: 0, get: p?add;split_q>(p,q), skip: split_q>(p,q)}

QS_p(p) = if small then 0
          else (start q< ▷ split_q<<p,q< >; QS_p<q< >; p!c);
                (start q= ▷ split_q=<p,q= >; p!c);
                (start q> ▷ split_q><p,q> >; QS_p<q> >; p!c);
                q<?id; q=?append; q>?append
```

Remark 1. Our implementation of `split` is suitable in a context where communication is cheap, e.g., as in object-oriented programming and/or a multi-threaded application. In architectures where communications are costly, it could be better to use a `select` function at `p` to compute the lists of elements smaller than, equal to, or larger than the pivot and send each of these in a single message to $\mathtt{q}_<$, $\mathtt{q}_=$ or $\mathtt{q}_>$, respectively.

4 Gauss Elimination

We now show how we can program the distributed resolution of systems of linear equations by Gaussian elimination. Let $A\boldsymbol{x} = \boldsymbol{b}$ be a system of linear equations in matrix form; our procedure `gauss` will transform this into an equivalent system $U\boldsymbol{x} = \boldsymbol{y}$, where U is an upper triangular matrix (so this system can be solved by direct substitution). We use parameter processes $\mathtt{a}_{ij}$, with $1 \leq i \leq n$ and $1 \leq j \leq n{+}1$. Each $\mathtt{a}_{ij}$ such that $1 \leq i, j \leq n$ stores one value from the coefficient matrix and $\mathtt{a}_{i,n+1}$ stores the independent term in one equation. (Including the independent terms in the coefficient matrix substantially simplifies the notation, as Gaussian elimination treats the independent vector exactly as the columns in the coefficient matrix.) After execution, each $\mathtt{a}_{ij}$ stores the corresponding term in the new system. For simplicity, we assume that the matrix A is non-singular and numerically stable.

Implementing this algorithm in PC cannot be done directly, as our procedure `gauss` needs to take a variable number of parameters (the $\mathtt{a}_{ij}$). However, it is straightforward to extend PC so that procedures can also take process *lists* as parameters, instead of only processes. We succinctly describe how to do this.

Syntax of PC. We extend parametric procedures with arguments that are lists of process names. In procedure calls, we can use standard list functions (e.g., head, tail) to manipulate these lists. These functions must be pure and take a list as their only argument. The processes in these lists are assumed to have all the same type, and process lists can only be used in procedure calls.

In our examples, we will use uppercase letters to identify process lists and lowercase letters for normal process identifiers.

Semantics of PC. We assume that a procedure that is called with an empty list as one of its arguments is equivalent to the terminated process **0**.

Connections. We assume that the connections between processes are uniform wrt argument lists, i.e., if `p` is a process and `A` is a list of processes that are arguments to some procedure `X`, then `X` requires/guarantees that `p` be connected to none or all of the processes in `Q`. The type system presented in [8] can be trivially extended to check for this requirement.

Syntax of PP. We extend it in the same way as PC.

Semantics of PP. We assume that a procedure unfolds to **0** if the process unfolding it does not occur inside its arguments.

EndPoint Projection. We project procedures as before, with one PP procedure for each argument of each PC procedure – regardless of whether the argument is a single process or a list of processes. The merge operator $\sqcup$ also needs to be slightly adjusted; we explain how this is done in § 5, as this change is not required for the example in this section.

This extension preserves the properties of PC, PP, and the EPP from the former to the latter.

We use `hd` and `tl` to compute the head and tail of a list of processes, respectively; `fst` and `rest`, which take a list of processes representing a matrix and return, respectively, the first row of the matrix, or the matrix without its first row; and `minor`, which removes both the first row and the first column from a matrix. (Formally, some of these functions would need the size of the rows as arguments, but we omit these for simplicity.) Each process uses standard arithmetic operations to combine its value with the one it receives.

The code of procedure `gauss` follows.

```
gauss(A) = solve(fst(A));
           eliminate(fst(A),rest(A));
           gauss(minor(A))

solve(A) = divide_all(hd(A),tl(A)); set_to_one(hd(A))

divide_all(a,A) = divide(a,hd(A)); divide_all(a,tl(A))

divide(a,b) = a.c -> b.div

eliminate(A,B) = elim_row(A,fst(B)); eliminate(A,rest(B))

elim_row(A,B) = elim_all(tl(A),hd(B),tl(B)); set_to_zero(hd(B))

elim_all(A,m,B) = elim_one(hd(A),m,hd(B)); elim_all(tl(A),m,tl(B))

elim_one(a,m,b) = b start x; b: x <-> a; b: x <-> m;
                  a.c -> x.id; m.c -> x.mult; x.c -> b.minus

set_to_zero(a) = a start p; p.0 -> a.id
set_to_one(a) = a start p; p.1 -> a.id
```

Procedure `solve` divides the first equation by the pivot, obtaining the new first equation in the reduced system. Then, `eliminate` uses this row to perform an elimination step, setting the first column of the coefficient matrix to zeroes.

The auxiliary procedure `elim_row` performs this step at the row level, using `elim_all` to iterate through a single row and `elim_one` to perform the actual computations. The first row and the first column of the matrix are then removed in the recursive call, as they will not change further.

This implementation follows the standard sequential algorithm for Gaussian elimination, as described in, e.g., Algorithm 8.4 in [12]. However, the implicit parallelism in the semantics of choreographies allows it to run concurrently. We explain this behaviour by focusing on a concrete example. Assume that A is a 3×3 matrix, so there are 12 processes in total. For legibility, we will write `b1` for the independent term `a14` etc.; `A=`$\langle$`a11,a12,a13,b1,a21,a22,a23,b2,a31,a32,a33,b3`$\rangle$ for the matrix; `A1=`$\langle$`a11,a12,a13,b1`$\rangle$ for the first row (likewise for `A2` and `A3`); and, `A'2=`$\langle$`a22,a23,b2`$\rangle$ and likewise for `A'3`.

Calling `gauss(A)` unfolds to

```
solve(A1); eliminate(A1,⟨A2,A3⟩);
solve(A'2); eliminate(A'2,A'3);
solve(⟨a33,b3⟩)
```

or, unfolding `eliminate`,

```
solve(A1); elim_row(A1,A2); elim_row(A1,A3);
solve(A'2); elim_row(A'2,A'3);
solve(⟨a33,b3⟩)
```

Unfolding `solve(A1)` is straightforward, leading to

```
a11.c -> a12.div; a11.c -> a13.div; a11.c -> b1.div;
a11 start x1; x1.1 -> a11.id
```

and likewise for the remaining calls. In turn, `elim_row(A1,A2)` becomes

```
elim_all(⟨a12,a13,b1⟩,a21,⟨a22,a23,b2⟩); set_to_zero(a21)
```

which can be expanded to

```
elim_one(a12,a21,a22); elim_one(a13,a21,a23); elim_one(b1,a21,b2);
set_to_zero(a21)
```

and we note that each of these procedure calls involves only communication between the processes explicitly given as arguments.

Since all these procedures involve `a21`, the semantics of choreographies requires them to be executed in this order. Likewise, the call to `elim_row(A1,A3)` must be executed afterwards (since it also involves processes `a11` through `a13`), and unfolds to a sequential composition of procedure calls with `a31` as argument.

The interesting observation is that none of the processes intervening in `elim_row(A1,A3)` occur in the expansion of `solve(A'2)`. In other words,

```
elim_row(A1,A3); solve(A'2)
```

expands to

```
elim_one(a12,a31,a32); elim_one(a13,a31,a33); elim_one(b1,a31,b3); set_to_zero(a31);
a21.c -> a22.div; a21.c -> a23.div; a21.c -> b2.div; a21 start x2; x2.1 -> a21.id
```

and the semantics of PC therefore allows the communications in the second line to be interleaved with those in the first line in any possible way; in the terminology of [7], the calls to `elim_row(A1,A3)` and `solve(A'2)` run in parallel.

This corresponds to the implementation of Gaussian elimination with pipelined communication and computation described in § 8.3 of [12]. Indeed, as soon as any row has been reduced by all rows above it, it can apply `solve` to itself and try to begin reducing the rows below it. It is a bit surprising that we get such parallel behaviour by straightforwardly implementing an imperative algorithm; the explanation is that the EndPoint Projection encapsulates the part of determining which communications can take place in parallel, thus removing this burden from the programmer. In the next section, we will include a simple example of the EPP of a procedure with parameter lists.

5 Fast Fourier Transform

We now present a more complex example: computing the discrete Fourier transform of a vector. We refer the reader to § 13.1 of [12] for details.

Definition 2. *Let* $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$ *be a vector of* n *complex numbers. The* discrete Fourier transform *of* $\boldsymbol{x}$ *is* $\boldsymbol{y} = \langle y_0, \ldots, y_{n-1} \rangle$*, where* $y_j = \sum_{k=0}^{n-1} x_k \omega^{kj}$ *with* $\omega = e^{2\pi i/n}$*.*

Given $\boldsymbol{x}$, its discrete Fourier transform can be computed efficiently by the *Fast Fourier Transform* (FFT) as follows (Algorithm 13.1 in [12]). We assume n to be a power of 2; in the first call, ω has the value defined earlier.

```
procedure R_FFT(X,Y,n,ω)
if n = 1 then y_0 = x_0
        else R_FFT(⟨x_0, x_2, ..., x_{n-2}⟩,⟨q_0, q_1, ..., q_{n/2}⟩,n/2,ω^2)
             R_FFT(⟨x_1, x_3, ..., x_{n-1}⟩,⟨t_0, t_1, ..., t_{n/2}⟩,n/2,ω^2)
             for j = 0 to n − 1 do
                 y_j = q_{(j%n/2)} + ω^j t_{(j%n/2)}
```

To implement this procedure in PC, we need a way to communicate labels in selections to a group of processes. We do this by means of two auxiliary procedures `gsel_then(p,Q)` and `gsel_else(p,Q)`, where `p` is a process broadcasting a selection of label `then` or `else`, respectively, to all the processes in `Q`. In order for EPP to work correctly, we also need to extend the merge operator $\sqcup$ slightly so as to recognize these procedures. We give the definition of `gsel_then`; this can trivially be adapted to any other label.

```
gsel_then(p,Q) = gsel1_then(p,hd(Q)); gsel_then(p,tl(Q))
gsel1_then(p,q) = p -> q[then]
```

The EPP of `gsel_then` looks as follows.

```
gsel_then_p(p,Q) = gsel1_then_p(p,hd(Q)); gsel_then_p(p,tl(Q))
gsel_then_Q(p,Q) = gsel1_then_q(p,hd(Q)); gsel_then_Q(p,tl(Q))

gsel_then_p(p,q) = q⊕l
gsel_then_q(p,q) = p&{then: 0}
```

The key aspect is that, during execution, the call to `gsel_then_Q(p,Q)` will reduce to *either* `gsel1_then_q(p,hd(Q))` *or* `gsel_then_Q(p,tl(Q))`, as each process can only be on one of `hd(Q)` or `tl(Q)`! So, while `gsel_then_p(p,Q)` essentially unfolds to a sequence of selections from `p` to each of the processes in `Q`, each local copy of `gsel_then_Q(p,Q)` at a process `q`∈`Q` unfolds exactly to the reception of one selection from `p`.

We will use (without specifying them) the following auxiliary procedures.

- `intro(n,m,P)`, where `n` introduces `m` to every process in `P` (defined similarly to `gsel_then` above)
- `power(n,m,nm)`, where at the end `nm` stores the result of exponentiating the value in `m` to the power of the value stored in `n` (see [7] for a possible implementation in a sublanguage of PC).

Before we present our implementation of FFT, we point out the one major difference wrt the algorithm R_FFT reported above: we are not able to create a variable number of fresh processes and pass them as arguments to other procedures (corresponding to the auxiliary vectors $\boldsymbol{q}$ and $\boldsymbol{t}$). Therefore, we have to use the result vector $\boldsymbol{y}$ to store the result of the recursive calls, and then create two auxiliary processes inside each iteration of the final for loop.

```
fft(X,Y,n,w) = if n.is_one
               then gsel_then(n,join(X,Y)); n -> w[then]; base(hd(X),hd(Y))
               else gsel_else(n,join(X,Y)); n -> w[else];
                    n start n'; n.half -> n'; intro(n,n',Y);
                    w start w'; w.square -> w'; intro(w,w',Y);
                    n: n' <-> w; w: n' <-> w';
                    fft(even(X),half1(Y),n',w');
                    fft(odd(X),half2(Y),n',w');
                    n' start wn; n': w <-> wn; power(n',w,wn);
                    w start wj; w.1 -> wj; intro(w,wj,Y);
                    combine(half1(Y),half2(Y),wn,w,wi)

base(x,y) = x.c -> y

combine(Y1,Y2,wn,w,wj) = combine1(hd(Y1),hd(Y2),wn,wj);
                         w.c -> wj.mult;
                         combine(tl(Y1),tl(Y2),wn,w,wj)

combine1(y1,y2,wn,wj) = y1 start q; y1.c -> q; y1: q <-> y2;
                        y2 start t; y2.c -> t; y2: t <-> y1; y2: t <-> wj;
                        q.c -> y1; wj.c -> t.mult; t.c -> y1.add;
                        q.c -> y2; wn.c -> t.mult; t.c -> y2.add
```

The level of parallelism in this implementation is suboptimal, as the two recursive calls to `fft` both use `n'` and `w'`; by duplicating these processes, however, these calls are able to run in parallel exactly as in the previous example. (We chose the current formulation for simplicity.) Process `n'` is actually the main orchestrator of the whole execution.

6 Graphs

Another prototypical application of distributed algorithms is graph problems. In this section, we focus on a simple example (broadcasting a token to all nodes of a graph) and discuss the limitations of implementing these algorithms in PC.

The idea of broadcasting a token in a graph is very simple: each node receiving the token for the first time should communicate it to all its neighbours. The catch is that, in PC, there are no primitives for accessing the connection graph structure from within the language. Nevertheless, we can implement our simple example of token broadcasting if we assume that the graph structure is statically encoded in the set of available functions over parameters of procedures. To be precise, assume that we have a function `neighb(p,V)`, returning the neighbours of `p` in the set of vertices `V`. (The actual graph is encapsulated in this function.) We also use `++` and `\` for appending two lists and computing the set difference of two lists. We can then write a procedure `broadcast(P,V)`, propagating a token from every element of `P` to every element of `V`, as follows.

```
broadcast(P,V) = bcast(hd(P),neighb(hd(P),V));
                 broadcast(tl(P)++neighb(hd(P),V),V\neighb(hd(P),V))

bcast(p,V) = bcast_one(p,hd(V)); bcast(p,tl(V))

bcast_one(p,v) = p.c -> v.id
```

Calling `broadcast(`$\langle$`p`$\rangle$`,G)`, where `G` is the full set of vertices of the graph and `p` is one vertex, will broadcast `p`'s contents to all the vertices in the connected component of `G` containing `p`. Furthermore, implicit parallelism again ensures that each node will start broadcasting as soon as it receives the token, independently of the remaining ones.

This approach is however not very satisfactory as a graph algorithm, since it requires encoding the whole graph in the definition of `broadcast`; furthermore, it does not generalise easily to more sophisticated graph algorithms. Adding primitives for accessing the network structure at runtime is not simple, as it would influence the definitions of EPP and the type system of PC [8] (which we omitted in this presentation). We leave this as an interesting direction for future work, which we plan to pursue in order to be able to implement more sophisticated graph algorithms, e.g., for computing a minimum spanning tree.

7 Related Work

To the best of our knowledge, this is the first experience report on using choreographic programming for writing real-world, complex computational algorithms.

The work nearest to ours is the evaluation of the Chor language [18], an implementation of the choreographic programming model in [3]. Chor supports multiparty sessions (similar to channels in the π-calculus [17]) and their mobility, which recalls introductions in PC. Chor is evaluated by encoding some representative examples from Service-Oriented Computing, such as distributed authentication and streaming, but none of the presented examples cover interesting algorithms as in here.

Previous works based on Multiparty Session Types (MPST) [14] have explored the use of choreographies as protocol specifications for the coordination of message exchanges in some real-world scenarios [9,22,25]. Differently from our approach, these works fall back to a standard process calculus model for defining

implementations. Instead, our programs are choreographies. As a consequence, programming the composition of separate algorithms in PC is done on the level of choreographies, whereas in MPST composition requires using the low-level process calculus. Also, our choreography model is arguably much simpler and more approachable by newcomers, since much of the expressive power of PC comes from allowing parameterised procedures, a standard feature of most programming languages. The key twist in PC is that parameters are process names.

8 Conclusions

We have reported our experience with the writing of some representative concurrent algorithms in the paradigm of choreographic programming.

What have we learned from this experience?

First, that choreographies make it easy to produce a simple concurrent implementation of a sequential algorithm. This is obtained by choosing process identifers to maximise the effect of implicit parallelism. Then, EPP takes care of generating the concrete separate programs and the required I/O actions to implement the described behaviour. This is a striking difference from how concurrent algorithms usually differ from their sequential counterparts. Although we do not necessarily get the most efficient possible distributed algorithm, this automatic concurrency is a pleasant property to observe.

The second interesting realisation is that it is relatively easy to implement nontrivial algorithms in choreographies. We exemplified this point with our implementations of Gaussian elimination and Fast Fourier Transform. This is an important deviation from the typical use of toy examples, of limited practical significance, that characterises previous works in this programming paradigm.

In conclusion, we showed that the current state of choreographic programming can already be used for addressing complex real-world computational problems. We also identified a future direction for extending the paradigm towards settings that require accessing the structure of process networks, such as some algorithms on graphs.

References

1. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In M. Lumpe and W. Vanderperren, editors, *Proc. of SC*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.
2. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
3. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *Proc. of POPL*, pages 263–274. ACM, 2013.
4. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In P. Baldan and D. Gorla, editors, *Proc. of CONCUR*, volume 8704 of *LNCS*, pages 47–62. Springer, 2014.

5. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. In L. Aceto and D. de Frutos-Escrig, editors, *Proc. of CONCUR*, volume 42 of *LIPIcs*, pages 412–426. Schloss Dagstuhl, 2015.
6. Chor. Programming Language. `http://www.chor-lang.org/`.
7. L. Cruz-Filipe and F. Montesi. Choreographies, computationally. *CoRR*, abs/1510.03271, 2015. Submitted for publication.
8. L. Cruz-Filipe and F. Montesi. Choreographies, divided and conquered. *CoRR*, abs/1602.03729, 2016. Submitted for publication.
9. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In T. Ball and M. Sagiv, editors, *Proc. of POPL*, pages 435–446. ACM, 2011.
10. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *Proc. of ESOP*, LNCS, pages 194–213. Springer-Verlag, 2012.
11. M. Gabbrielli, S. Giallorenzo, and F. Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
12. A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson, 2003. 2nd edition.
13. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin, editor, *Proc. of ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *Proc. of POPL*, pages 273–284. ACM, 2008.
15. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proc. of SEFM*, pages 323–332. IEEE, 2008.
16. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In S. K. Rajamani and D. Walker, editors, *Proc. of POPL*, pages 221–232. ACM, 2015.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, Sept. 1992.
18. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. `http://fabriziomontesi.com/files/m13_phdthesis.pdf`.
19. F. Montesi. Kickstarting choreographic programming. *CoRR*, abs/1502.02519, 2015.
20. F. Montesi and N. Yoshida. Compositional choreographies. In P. R. D'Argenio and H. C. Melgratti, editors, *Proc. of CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
21. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.
22. N. Ng and N. Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015.
23. M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies – safe runtime updates of distributed applications. In T. Holvoet and M. Viroli, editors, *Proc. of COORDINATION*, volume 9037 of *LNCS*, pages 67–82. Springer, 2015.
24. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *Proc. of WWW*, pages 973–982. ACM, 2007.
25. N. Yoshida, P. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In C. L. Ong, editor, *Proc. of FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.