Bringing Aggregate Programming towards the Cloud

Giorgio Audrito[0000−0002−2319−0375], Ferruccio Damiani[0000−0001−8109−1706], and Gianluca Torta[0000−0002−4276−7213]

Dipartimento di Informatica, Università di Torino, Italy
{name.surname@unito.it}

Abstract. Aggregate Programming (AP) is a paradigm for developing applications that execute on a fully distributed network of communicating, resource-constrained, spatially-situated nodes (e.g., drones, wireless sensors, etc.). In this paper, we address running an AP application on a high-performance, centralized computer such as the ones available in a cloud environment. As a proof of concept, we present preliminary results on the computation of graph statistics for centralised data sets, by extending FCPP, a C++ library implementing AP. This: *(i)* opens the way to the application of the AP paradigm to problems on large centralised graph-based data structures, enabling massive parallelisation across multiple machines, dynamically joining and leaving the computation; and *(ii)* represents a first step towards developing collective adaptive systems where computations dynamically move across the IoT/edge/fog/cloud continuum, based on mutable conditions such as availability of resources and network infrastructures.

Keywords: Distributed computing · Collective adaptive systems · Cloud computing · Graph algorithms.

1 Introduction

In recent years, Aggregate Programming (AP) [14] has attracted significant attention as an innovative approach for the development of fully distributed systems [32]. The typical applications for which AP is particularly suited involve resource-constrained, spatially-situated nodes that coordinate through point-to-point, proximity-based communications. For example, AP has been adopted in simulations of domains such as swarm-based exploration [21], crowd safety management and monitoring [7, 8, 11], data collection from sensor networks [4], dynamic multi-agent plan repair [5, 6].

The main implementations of AP can be understood as being combinations of two components: the first component provides full support for the constructs of the foundational language of AP, namely the Field Calculus (FC) [12]; the second component connects the FC program with the environment where the distributed system is deployed and executed. In particular, FC is currently supported by the following open-source implementations: the *FCPP* [3, 10] library,

as a C++ internal Domain-Specific Language (DSL); *ScaFi (Scala Fields)* [20], as a Scala internal DSL; and *Protelis* [29], as a Java external DSL. The main execution environment provided by existing implementations consists of simulations that run on a single computer, simulating sets of nodes situated in a 2D or 3D space, their dynamics, and the point-to-point communications between neighbouring nodes. The *FCPP* library has an internal simulator, while *ScaFi* and *Protelis* exploit the Alchemist simulator [28]. Recently, the *FCPP* library has been adapted to deployment on physical Micro Controller Unit (MCU)-based boards in a simplified Industrial Internet of Things (IIoT) scenario [31], targeting Contiki on DecaWave boards. A further porting to Miosix [2] on WandStem boards is also currently in progress.

In this paper we start exploring a new direction for the application of AP, namely, the implementation and execution of distributed algorithms on high-performance, centralized computers such as the ones typically available in a cloud environment. Instead of a single CPU-based system, that can run multiple threads sharing the same memory, it can be interesting to consider clusters of such systems, communicating through high-speed links for sharing data. While we will discuss this possibility, the main purpose of the present work will be handling the execution on a single CPU, as a first preliminary step.

In particular, as a proof of concept, we will describe an extension of the *FCPP* library that allows it to ingest a centralized, large-scale graph structure, and compute some network statistics of its input graph. Preliminary results of experiments with the extended library will be presented. The ability of executing algorithms on large static graphs is per-se an important application [13], that could be further boosted by distribution over several CPU-based systems. Note that this contribution is different from the one described in [9], where AP was adopted to compute centrality statistics of *dynamic* networks, whose structure was induced by (simulated) spatial-based connectivity, using Alchemist and Protelis. In this paper, instead, we compute similar statistics for a *general static* graph (provided as a single file on disk) which does not arise from any spatial arrangement: in fact, as sample input data, we will use a (restricted) web graph.

In the long run, the possibility of a centralized (or locally distributed) execution of FC programs on graphs may allow the implementation of collective adaptive systems that exploit the IoT/edge/fog/cloud continuum. In fact, the same AP paradigm could be exploited for programming the far edge, constrained devices as well as the intermediate and most powerful nodes in the architecture, and this would simplify the dynamic migration of computations between different layers based on mutable conditions such as availability of resources and network infrastructures. In this paper, we devise a roadmap towards this aim, identifying the current paper's contribution as a first preliminary step.

The remainder of this paper is structured as follows. Section 2 presents the necessary background on aggregate programming, FCPP and the graph statistic application. Section 3 delineates a roadmap towards a IoT/edge/fog/cloud continuum through AP. Section 4 presents the implementation of the first step, allowing graph-based data processing in FCPP. Section 5 experimentally evalu-

ates the approach on graph statistic computation, and Section 6 concludes with final comments and remarks.

2 Background

2.1 Aggregate Programming and the FCPP DSL

AP [14, 32] is an approach for programming networks of devices by abstracting away from individual devices behaviour and focusing on the global, aggregate behaviour of the collection of all devices. It assumes only local communication between neighbour devices, and it is robust with respect to devices joining/leaving the network, or failing; thus supporting an *open dynamic topology*. Beside communicating with neighbours, the devices are capable to perform computations. In particular, every device performs periodically the same sequence of operations, with an usually steady rate, asynchronously from other devices:

1. retrieval of received messages,
2. computation of the program;
3. transmission of resulting messages.

In case devices are equipped with sensors/actuators, they may use sensor data and provide instructions to actuators during the program execution. In AP, we assume that all device execute the *same* program. Note that this assumption does not restrict the realisable behaviour, as every device may follow different branches of the same program, resulting in a radically different behaviour.

AP is formally backed by FC [12], a small functional language for expressing aggregate programs. Few concrete implementations of FC exist to date: Proto, Protelis [29], Scafi [20], FCPP [3]. In this paper, we will focus on the latter and most recent, which is structured as a C++ internal DSL (i.e., library). The syntax of aggregate functions in FCPP is given in Fig. 1. It should be noted that, since FCPP is a C++ library providing an internal DSL, *an FCPP program is a C++ program* (so all the features of C++ are available). For compactness of presentation, we restrict here to a subset of the language with sufficient expressiveness for later examples. In the formal syntax, we use $*$ to indicate an element that may be repeated multiple times (possibly zero).

An *aggregate function declaration* consists of keyword `FUN`, followed by the return type t and the function name d, followed by a parenthesized sequence of comma-separated arguments t x (prepended by the keyword `ARGS`), followed by *aggregate instructions* i (within brackets and after keyword `CODE`), followed by the *export description*, which lists the types that are used by the function in message-exchanging constructs.

Aggregate instructions always end with a *return* statement, reporting the function result. Before it, there may be a number of local variable declarations (assigning the result of an expression e to a variable x of type t), and for loops, which repeat an instruction i while increasing an integer index x until a condition e is met. The main types of *aggregate expressions* are:

aggregate function declaration

F ::= `FUN` t d(`ARGS`, t x∗) {`CODE` i} `FUN_EXPORT` d`_t` = `export_type<`t ∗ `>`;

aggregate instructions

i ::= `return` e; | t x = e; i | `for`(`LOOP`(x, ℓ); e; `++`x){i} i

aggregate expression

e ::= x | ℓ | t(e∗) | ue | e o e | p(e∗) | `node`.c(e∗) | f(`CALL`, e∗) | [&](t x∗)`->`t {i} | e ? e : e

type	aggregate function
t ::= T | bt | tt`<`t∗, ℓ ∗`>`	f ::= b | d

built-in aggregate functions

b ::= `old` | `nbr` | `oldnbr` | `spawn` | `self` | `mod_self` | `map_hood` | `fold_hood` | `mux`

Fig. 1. Syntax of FCPP aggregate functions.

- a *variable* identifier x, or a C++ *literal value* ℓ (e.g. an integer or floating-point number);
- an *unary operator* u (e.g. −, ∼, !, etc.) applied to e, or a *binary operator* e o e (e.g. +, ∗, etc.);
- a *pure function call* p(e∗), where p is a basic C++ function which does not depend on node information nor message exchanges;
- an *aggregate function call* f(`CALL`, e∗), where f can be either a defined aggregate function name d or an aggregate built-in function b (see below);
- a *conditional branching* expression e_{guard} ? $e_\top$: $e_\bot$, such that $e_\top$ is evaluated and returned if e_{guard} evaluates to `true`, while $e_\bot$ is evaluated and returned if e_{guard} evaluates to `false`.

Finally, several aggregate built-in functions are provided, which allow implicit message exchange and state preservation across rounds. In this paper, we will mention the following ones:

- `old(CALL,i,v)`, which returns the value passed for `v` to the function in the previous round, defaulting to `i` on the first call of this function;
- `nbr(CALL,i,v)`, which returns the *field* (i.e., collection of) values passed for `v` to the function in the previous rounds of neighbour devices (including the current device, defaulting to `i` for it on the first call of this function);
- `self(CALL, v)` given a field `v` returns the value in `v` for the current device;
- `fold_hood(CALL, f, v, i)` given a field `v` and a function `f`, reduces the field to a single value by starting from `i` and repeatedly applying function `f` to each element of the collection and to the current partial result.

It is worth observing that the FCPP syntax uses a number of macros (e.g., `CALL`, `CODE`, etc). These macros ensure that the aggregate context (i.e., the `node` object) is carried over throughout the program, also updating an internal representation of the stack trace for *alignment*. Thanks to alignment, the messages

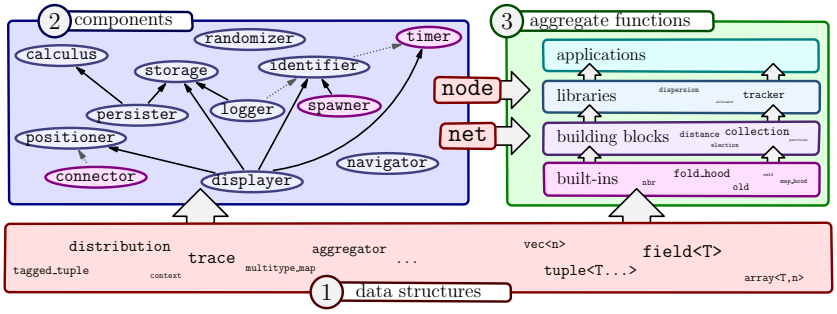

Fig. 2. The three main layers of the software architecture of FCPP: *data structures* for both other layers, and *components* which provide node and network abstractions to *aggregate functions*. Components that have been modified in this work are highlighted in magenta. Dependencies between components can be either *hard* (solid), for which the pointed component is always required as an ancestor of the other; or *soft* (dotted), for which the pointed component is required only in some settings.

(implicitly) originating from a `old` or `nbr` construct are matched in future rounds (on the same or different devices) only to the *same* construct, that is, a construct called in the same position in the program syntax and in the stack trace. This mechanism allows to freely compose functions, and use recursion, without risking interferences of messages between different parts of the program.

2.2 FCPP Library Architecture

FCPP is based on an extensible software architecture, at the core of which are *components*, that define abstractions for single devices (*node*) and overall network orchestration (*net*), the latter one being crucial in simulations and cloud-oriented applications. In an FCPP application, the two types *node* and *net* are obtained by combining a chosen sequence of components, providing the needed functionalities in a mixin-like fashion.

The FCPP library architecture is divided in three main conceptual layers represented in Fig. 2: *(i)* C++ data structures of general use; *(ii)* components; *(iii)* aggregate functions. The first layer comprises data structures needed by the second layer either for their internal implementation or for the external specification of their options, but also data structures designed for the third layer. Some components are sufficiently general purpose to be used across different domains, including simulations and deployments (calculus, randomizer, storage and timer). Others may be useful only in certain domains (displayer, navigator, persister and spawner), or come with *variations* for different domains, sharing a common interface (connector, identifier, logger and positioner). For example, for simulations the *connector* component is given as a `simulated_connector`, which exchanges messages as pointers between objects in memory, determining if connection is possible based on simulated positions. For deployments, a `hardware_connector` is given, which instead exchanges physical messages through

some provided networking interface. In order to handle processing of graph-based data, the timer and spawner components have been extended and a new variation of the connector component has been provided (cf. Section 4).

The basic structure of each component is as follows:

```
template <class... Ts>
struct my_functionality {
  template <class F, class P>
  struct component : public P {
    class node : public P::node { ... };
    class net : public P::net { ... };
  };
};
```

Thus, each component is a templated class with a variable number of type arguments, which are used to provide compile-time options to be used by the components to tune their behaviour. This options are empty templated types, wrapping the data of interest, such as `parallel<true>` to enable parallelism, or `connector<fixed<100>>` to specify that devices are to be connected within a fixed radius of 100 units. The outer class has a `component` template subclass with two type parameters: `P`, which represents the parent component composition, and `F`, the final outcome of the whole component composition, which is retrieved by the C++ template system thanks to the *Curiously Recursive Template Pattern* (CRTP, first introduced in [19]). Both the `node` and `net` classes are defined inside the component subclass to inherit from the corresponding classes in the parent composition `P`. The final outcome of the composition `F` may be used by those classes to mimic virtual-style calls that are resolved at compile-time.

The scenario originally supported by the first versions of FCPP is the simulation of distributed systems. Compared to the alternative implementations of FC (Protelis [29] and Scafi [33] with their simulator Alchemist [28]), it features additional simulation capabilities (3D environments, basic physics, probabilistic wireless connection models, fine-grained parallelism), while granting a significant reduction of the simulation cost in CPU time and memory usage, which comes with a corresponding speed-up of the development and test of new distributed algorithms.

2.3 Graph Statistics

Several techniques for collecting statistics from graph-based data have been investigated in the data mining community. In this section, we recall the statistics measuring centralities of the nodes of the graph considered in [9]. These are quite common and can be naturally implemented in FC. We will use them as a benchmark for the application of AP on graph algorithms.

Degree centrality is the historically first and conceptually simplest centrality measure. It is defined as the number of links incident upon a node (i.e. its degree): nodes with an higher number of links should be less prone to encounter network disconnections, so electing these nodes as communication hubs should

be more effective than electing nodes with a lower degree. Degree centrality is simple and efficient to calculate. However, compared with other centrality measures, degree centrality is usually the least effective. In fact, in approximately homogeneous situated networks – where nodes at the edge of the network have lower degrees and all other nodes have similar degrees – selecting the node with the highest degree would correspond to select a random node which is not at the network edge.

PageRank [26] is an instance of the broader class of *eigenvector centrality measures*. This centrality measure has been first introduced for the Google search engine and it is quite popular in the data mining community. According to PageRank, the centrality score r_i of a node i is defined as the fixed point of the system of equations:

$$r_i = (1 - \alpha) + \alpha \sum_{j \in \text{neigh}(i)} \frac{r_j}{\deg(j)}$$

where α is a parameter (usually set at 0.85 [18]), $\deg(j)$ is the degree of node j and $\text{neigh}(i)$ is the set of neighbour nodes j connected to i. PageRank has been proved effective on logical graphs as the web graph. It can be efficiently calculated by re-iterating the equations above for each node, starting from $r_i^0 = 1$.

Closeness Centrality and Harmonic Centrality are the most effective centrality measures that we consider [16]. They are both derivable from (variations of) the *neighbourhood function* of a graph, which is defined as follows.

Definition 1 (Neighbourhood Function). *Let $G = \langle V, E \rangle$ be a graph with n vertices and m edges. The* generalized individual neighbourhood function *$N_G(v, h, C)$, given $v \in V$, $h \geq 0$ and $C \subseteq V$, counts the number of vertices $u \in C$ which lie within distance h from v. In formulas, $N_G(v, h, C) = |\{u \in C : \ \text{dist}(v, u) \leq h\}|$.*

Many different questions – like graph similarity, vertex ranking, robustness monitoring, and network classification – can been answered by exploiting elaborations of the N_G values [16, 27]. Unfortunately, exact computation of N_G is impractical: it requires $O(nm)$ time in linear memory and $O(n^{2.38})$ time in quadratic memory.

Fast algorithms approximating N_G up to a desired precision have been developed. In particular Vigna et al. [16] improved the original algorithm by: (i) exploiting *HyperLogLog counters* (a more effective class of estimators) [23]; (ii) expressing the "counter unions" through a minimal number of broadword operations; and (iii) engineering refined parallelisation strategies. HyperLogLog counters maintain size estimates with asymptotic relative standard deviation $\sigma/\mu \leq 1.06/\sqrt{k}$, where k is a parameter, in $(1 + o(1)) \cdot k \cdot \log\log(n/k)$ bits of space. Moreover, updates are carried out through k independent "max" operations on $\log\log(n/k)$-sized words. As a result, given a fixed precision, N_G can be computed in $O(nh)$ time and $O(n \log\log n)$ memory. This enables to apply it on very large graphs like, e.g., the Facebook graph [13].

We are now ready to present closeness centrality and harmonic centrality.

Closeness centrality of a node i, denoted by c_i, is defined as the reciprocal of the total distance to other nodes. It can be computed in terms of the neighbourhood function by the following equation:

$$\frac{1}{c_i} = \sum_{j \neq i} \mathrm{dist}(i,j) = \sum_{h=1}^{D} h \left(N_G(i,h,V) - N_G(i,h-1,V) \right)$$

where D is the graph diameter (maximum distance between nodes in G).

Harmonic centrality of a node i, denoted by h_i, is defined as the sum of the reciprocals of distances to other nodes. It can be computed in terms of the neighbourhood function by the following equation:

$$h_i = \sum_{j \neq i} \frac{1}{\mathrm{dist}(i,j)} = \sum_{h=1}^{D} \frac{N_G(i,h,V) - N_G(i,h-1,V)}{h}$$

where D is the graph diameter (maximum distance between nodes in G).

Nodes with high closeness/harmonic centrality are connected to many other vertices through a small number of hops. So, they are best-suited to be elected as leaders for coordination mechanisms.

3 Roadmap

In this section, we describe a roadmap to make the AP paradigm applicable beyond the level of a network of constrained devices. The first step has already been taken, and will be described in more detail in the rest of this paper. The other three steps, up to a hybrid deployment where computations can be dynamically moved between the devices (far edge) and a central infrastructure (cloud), will require further research and development efforts. However, we can at least sketch some concrete lines of work that shall be followed for their realization.

Step 1: Data Processing Support. In order to exploit cloud-based resources and integrate the AP paradigm with them, it is first of all necessary to provide a centralized, abstract view of the network in terms of a graph of nodes and their connections, allowing AP to centrally process this graph-based data. In this context, the notion of neighbourhood, which is fundamental for AP, can be derived from the graph structure, instead of being implicitly determined by the physical vicinity of devices. Note that AP simulators such as Alchemist and the simulator embedded in FCPP also have the necessity to represent the network of devices in a centralized structure. However, they assume that *(i)* the nodes are deployed in a 2D or 3D euclidean space; that *(ii)* possible connections are computed from the physical positions; that *(iii)* round scheduling is constrained by energy saving needs; and that *(ii)* the simulator can take full control of the nodes (position, velocity, etc.).

In a centralized AP computation, however, some or all of that assumptions may fail. Nodes may not be deployed in a physical space, or their position may be

inaccessible due to lack of dedicated sensors. Connections between nodes should not be computed by the central application, but instead either read from a configuration file or induced by mirroring a physical deployment. Round scheduling should not be connected with energy saving needs, but instead tuned to get the best performance out of the available cloud resources. Finally, the amount of node control available in the central application may be limited.

In order to achieve this goals, we extended FCPP to be able to read and process locally available graph-based data, while allowing the schedule of rounds to be determined reactively in order to maximise performance. More details on this initial step and its implementation in FCPP are given in Sections 4 and 5.

Step 2: Multi-CPU Distribution. The centralized AP computations can of course benefit of multi-core architectures by associating the nodes of the graph (and thus their computations) to multiple threads. This is already possible in the current implementation, and was relatively easy to implement by simply protecting with locks the (short) critical sections where nodes exchange messages with neighbours.

However, high-performance centralized infrastructures are often based on NUMA (Non-Uniform Memory Access) architectures, where multiple CPUs have (preferred) access to local memories. In such scenarios, the shared-memory model needed by multi-thread applications is not applicable, and a message-passing model must be adopted to connect computations carried by different CPUs. A promising approach consists in the adoption of MPI (Message-Passing Interface), the de-facto standard for message-passing on high-performance parallel architectures, which defines the syntax and semantics of a rich set of library routines. Several open-source implementations of MPI are available, in particular for the C++ language used by the FCPP implementation of the AP paradigm.

The main challenges we envision for integrating MPI with FCPP are: the automatic partition of the set of graph nodes on different CPUs so as to reduce as much as possible the cost of message-passing between different memories (we shall evaluate the applicability of graph partitioning algorithms such as [25]); and a software architecture that makes as transparent as possible the difference between shared memory communications (that should continue to be exploited by nodes assigned to the same CPU) and message-passing communications.

Step 3: Dynamic Multi-CPU Distribution. Up to now, we have assumed that the graph representing the AP network is static, but in general this is not the case: nodes and their connections can be added and removed dynamically, to reflect changes in the structure of the underlying distributed computation. In fact, dynamism in graph structure is already supported by the single-CPU implementation, as links can be inserted or removed through dedicated methods, as will be discussed in Section 4.

In multi-CPU scenarios, it is therefore of great importance to implement online mechanisms in charge of deciding if and which nodes should migrate from one CPU to another, in order to accommodate the changes in graph structure, while keeping the load balanced among CPUs for better performance. In order to

implement this point, we predict that mainly two ingredients would be needed: a node migration mechanism, together with heuristic strategies guiding it.

Step 4: Hybrid and Mirror Systems. Until this point, the centralised AP-based system we proposed is described as fully logic, directly operating on graph-based data somehow available on the cloud. This data may have a purely logical origin as well, being for instance collected by a web crawler. However, it would also be crucial to consider data obtained by mirroring a physical network of IoT devices. In particular, we envision scenarios where virtual devices associated with physical IoT devices [22, 30] perform their computations directly on the cloud, possibly inter-operating with physical IoT devices directly running the same AP system without mirroring.

For instance, a group of physical devices deployed at location *L1* may directly execute a FC program by communicating point to point with one another, while indirectly interacting with physical devices deployed at another location *L2*. Those may delegate their executions of the FC program to the cloud, since too many rounds of computation and communication would be needed to reach convergence of their results, which is too resource demanding to be handled locally. Further virtual nodes may also be present in the mirrored network in the cloud, that are derived fully logically without any physical mirror device, to perform further heavy assistive tasks (e.g. federated learning computations [24]).

In order to allow the integration of such a system within FCPP, given the infrastructure available from the previous steps, it would be necessary to add a component ensuring proper mirroring of a virtual device in the cloud with a physical IoT device. That may also require some routing mechanism in place, in case the IoT device to be mirrored does not have direct internet connection. After such a connection can be established, mirroring may be ensured by an external daemon process synchronizing the graph-based data on the cloud used by the centralised AP system with the data obtained from sensors on the mirror IoT device (possibly including messages listened from its neighbourhood), with a given frequency depending on energy requirements.

Step 5: Dynamic Hybrid Systems. Similarly to the dynamic distribution of nodes to different CPUs, it may be useful to enhance hybrid edge-cloud AP systems with the possibility of dynamically migrating some computations from the edge to the cloud and vice-versa, depending on the weight of the required computation, as well as on the current availability of resources. Given that the previous steps are all met, this could be implemented by proper heuristics guiding a migration mechanism, as for step 3.

4 A First Step: Extending FCPP to work with Graphs

In this section, we outline the extensions made to the FCPP library in order to allow FC programs to process centralized graphs. This new feature effectively

constitutes a first step towards the goal of a self-organising edge-cloud application, as outlined in Section 3. Overall, these extensions will culminate into the definition of two new *component compositions*.

As mentioned in Section 2.2, an FCPP application is first specified through such a composition of components from the hierarchy shown in Figure 2, which are then customised further with suitable parameters. For example, FCPP defines a batch simulation application as:

```
DECLARE_COMBINE(batch_simulator,
  simulated_connector, navigator, simulated_positioner, timer,
  logger, storage, spawner, identifier, randomizer, calculus);
```

exploiting the `DECLARE_COMBINE` macro to combine into the `batch_simulator` type all the components listed as the remaining parameters. The order of the specified components induces their parent relations, and must comply with compile-time restrictions enforced by the components themselves (depicted in Figure 2). For example, in the `batch_simulator` combination, the `spawner` component has `identifier` as direct parent, and `randomizer` and `calculus` as further ancestors.

The new compositions we introduced are called `batch_data_processor` and `interactive_data_processor`, which differ from the existing `batch_simulator` in:

1. While the nodes of a simulation are situated in a 3D space, and thus have 3D position, velocity and acceleration vectors, the nodes of a graph do not have such attributes. This can be simply accommodated by omitting the `simulated_positioner` and `navigator` components from the mix.
2. In simulation, rounds are scheduled according to a programmatic policy. In data processing, we need rounds to be *reactively* scheduled when neighbours' values are updated. Since this feature may be relevant for classical simulated and deployed systems as well, we implemented it by extending the `timer` component in order to trigger reactive rounds after a message is received.
3. While in a simulation nodes are usually algorithmically generated across the simulated space, we need to generate them based on data read from a file. Since this feature could also be useful in 3D simulations, provided that position information is stored in the given file, we implemented it by extending the `spawner` component with options for file-based generation.
4. Finally, as the main structural difference, the neighbourhood of each node is not determined by the physical locations of nodes, but is given by the edges of the graph (also read from a separate file). This requires to introduce a new variant of the *connector* component, which we called `graph_connector`.

Based on this considerations, the mentioned combinations are defined as:

```
DECLARE_COMBINE(batch_data_processor,
  graph_connector, timer, scheduler, logger, storage,
  spawner, identifier, randomizer, calculus);
```

Note that this definition is remarkably similar to that of a batch simulation, i.e., we have been able to exploit several existing components. We also defined

a `interactive_data_processor` combination providing a graphical user interface through which the network situation can be inspected (based on the existing similar `interactive_simulator` composition):

```
DECLARE_COMBINE(interactive_data_processor,
  displayer, graph_connector, simulated_positioner, timer, scheduler,
  logger, storage, spawner, identifier, randomizer, calculus);
```

The main responsibility of the *spawner* component is that of creating nodes with unique identifiers exploiting the `identifier` component (which is its parent). Formerly, that had to happen by algorithmic generation of node parameters. We extended the component to provide exactly the same function when the underlying system is a graph read from the disk. In particular: (i) the `net` constructor of the component handles the option `nodesinput` which specifies the name of the file where the nodes of the graph are stored; and (ii) each row of the nodes file is parsed according to a list `node_attributes` of expected attributes, which need to contain every mandatory argument needed for node construction (such as the `uid` attribute). All file-generated nodes are created at the execution start, but their first round can be scheduled arbitrarily later exploiting the `start` construction argument given by the *timer* component.

The *connector* component is in charge of handling the exchange of messages between neighbours. The `graph_connector` replaces the `simulated_connector` used in simulations by providing exactly the same function when the underlying system is a graph read from the disk. In particular:

1. it provides `connect` and `disconnect` methods to the `node`, so that nodes of the graph can be connected at start and disconnected at end by the component, while allowing dynamic connections to be also established through the program logic;
2. connections can be considered as directed or undirected by simply setting the Boolean option `symmetric`;
3. the `net` constructor of the component handles the option `arcsinput` which specifies the name of the file where the initial arcs of the graph are stored;
4. each row of the arcs file is parsed expecting a pair of node identifiers; and an arc between the two nodes is created by calling the `connect` method on the first node with the second node as a parameter;
5. it provides functionality to send messages to the outgoing neighbours of a node, i.e., the nodes reachable with an outgoing arc in the graph.

The *timer* component is in charge of scheduling rounds both a-priori and in the program logic. We extended it by: (i) adding a parameter `reactive_time`, that for every node sets a delay for triggering a round after a message is received; and (ii) through the function `node.disable_send()` provided by the *connector* component, messages can be blocked avoiding triggering rounds in neighbours whenever the results of the algorithm at hand are stable. By setting a reactive time that is much shorter than the non-reactive scheduling of rounds, we can ensure that nodes are reactively processed until convergence before any non-reactive rounds are scheduled.

In performing the extensions just described, we had to overcome some notable issues: (i) in data processing, fine-grained parallelism is necessary, and thus the implementation had to be designed so to avoid both data races and deadlocks, which is not trivial for graph-like structures; and (ii) by allowing reactive rounds, we broke an assumption of FCPP that the next event on a node is scheduled right after an event is executed. The *identifier* component keeps a queue of node events, executing them (in parallel) as necessary. Formerly, this queue could be updated by adding the next events right after each event is executed. In order to handle reactive rounds, we had to add an alternative way of updating the event queue: the identifier checks at each message received whether the next event changes, updating the queue accordingly. In particular, this introduced a new component dependency, requiring the timer to appear as parent of the identifier in order for this interaction to be captured.

5 Experimental Evaluation

5.1 Implementation

We evaluated our approach by implementing a case study on the computation of graph statistics, focusing on the centrality measures presented in Section 2.3. These statistics have been implemented as FC programs expressed in the FCPP DSL described in Section 2.1, similarly as previously done in the Protelis DSL in [9]. Differently than there, we enhanced the HyperANF algorithm to not require an upper bound of the diameter in input, computing the neighbourhood function up to a variable depth, stopping whenever no further nodes are found in the last level. The resulting code is shown with explanatory comments in Figure 3.

The HyperANF algorithm is based on HyperLogLog counters, which we implemented in C++ with a `hyperloglog_counter` template class by mimicking the extremely efficient Java implementation described in [16]. The template parameters allow the specification of:

1. number of registers per HLL counter;
2. number of bits per register (defaulting to 4);
3. type of the counted data (defaulting to `size_t`, i.e., unsigned long integer);
4. hash function used to convert the counted data before HLL operations.

In particular, the implementation exploits *broadword programming* techniques to parallelize the union operations on all the registers contained in a word (e.g., 16 registers at a time, if the word is 64 bit and the register is 4 bits), thus significantly speeding-up the fundamental operations performed during the computation of N_G. In Figure 3, type `hll_t` is used for those counters, which is defined as a specific instantiation of the `hyperloglog_counter` template.

5.2 Results

In order to evaluate the effectiveness of FCPP on centralised graph-based data processing, we compared our approach with the state-of-the-art WebGraph-based implementation [17, 15] of the HyperANF algorithm computing harmonic

```
FUN tuple<real_t, real_t> hyperANF(ARGS) { CODE
    real_t h = 0; // harmonic count
    real_t c = 0; // closeness count
    bool u = true; // are results unchanged from previous round?
    hll_t l = hll_t(node.uid); // HLL counter of nodes up to a depth
    real_t ps = 0; // size at the previous depth
    real_t cs = l.size(); // size at the current depth
    // loop over depths until no further nodes discovered
    for (LOOP(depth, 1); ps < cs; ++depth) {
        ps = cs; // current size becomes previous size
        field<hll_t> nl = nbr(CALL, hll_t(), l); // neighbour HLLs
        u = self(CALL, nl) == l and u; // check for change
        // accumulate neighbour HLLs into l
        fold_hood(CALL, [&](hll_t const& x, nullptr_t){
            l.insert(x); // accumulate as side-effect
            return nullptr; // dummy return value
        }, std::move(nl), nullptr);
        cs = l.size(); // new current size
        h += (cs-ps)/depth; // update harmonic count
        c += (cs-ps)*depth; // update closeness count
    }
    // notify neighbours with messages only if changes occurred
    if (old(CALL, hll_t(), l) == l and u) node.disable_send();
    return tuple<real_t, real_t>(h, c); // return counts
}
FUN_EXPORT hyperANF_t = export_list<hll_t>; // types used in messages
```

Fig. 3. Implementation of the HyperANF algorithm in FCPP.

and closeness centrality [16]. As reference graph, we used the *cnr-2000* test set [1] of moderate size, and run 10 executions for both implementations, in order to account for variability in execution times.

On a MacBook Pro with 2,4GHz Intel Core i9 8 core CPU and 32GB 2667MHz RAM, the WebGraph implementation took from 28.263s to 36.919s to complete, with an average of 34.348s. On the other hand, the FCPP implementation took from 111.230s to 123.950s to complete, with an average of 115.066s. Even though the FCPP implementation was noticeably slower, requiring about 3$\times$ time to complete, it has other advantages that compensate for this difference: most notably, the generality of the approach, which translates into much lower software development costs. In fact, the WebGraph implementation is highly specific to the problem at hand, and low-level optimised for it, so that it cannot be easily modified to accommodate any other task: the codebase needed to implement HyperANF is very large and complex, requiring high development costs. The FCPP codebase is also large and complex, however, it is sufficiently generic so that any other graph-based problem could be easily formulated in order to be executed with it, without direct intervention on that codebase. The development

costs are thus limited to the AP formulation of the problem at hand, and are thus much lower as can be seen through the code snippet in Figure 3.

We also remark that optimisations reducing execution times are planned for future releases of the FCPP implementation. These are likely to reduce the performance gap, although we do not expect this gap to be fully compensated. As the FCPP implementation is subject to the constraint of being able to run *any* aggregate program, in a way that is *inter-operable* with deployed self-organising systems running the same software, problem-specific optimisations are not possible, restricting leeway for improvements.

6 Discussion and Conclusions

In the present paper we have extended the FCPP implementation of FC in order to support a new execution environment, namely high-performance, centralized computers. In particular, FCPP can now ingest large-scale graph structures and execute FC programs as if the nodes of the graph were distributed devices, and the graph arcs represented the proximity-based communication links. The centrality statistics, chosen as a benchmark to test the extension, have been coded naturally with the FCPP DSL, and have shown more than decent performances compared with a state-of-the-art, carefully crafted Java implementation [16]. This extension has been carried out as a first step towards an AP-based IoT/edge/fog/cloud continuum, while also delineating a roadmap of 5 milestones to reach it, together with the main associated challenges to be overcome.

The evaluation of the extension presents some notable limitations that may be addressed in future work. First, the single problem of graph statistics computation was considered: the evaluation may be enhanced by considering more graph-related problems, such as routing, maximum flow or minimum spanning tree estimation. Furthermore, we carried the evaluation on a relatively small graph on a laptop computer: more accurate benchmarks could be obtained by performing a similar computation on larger graphs on an high-performance computer. In fact, such systems are sometimes equipped with Graphics Processing Unit (GPU)s beside multi-core CPUs: in the present paper we supported only CPU-based systems, since the Single Instruction, Multiple Data (SIMD) model of GPUs imposes restrictions on the computations that deserve a separate, in-depth analysis in future works.

References

1. Web crawl of the italian cnr domain in year 2000. https://law.di.unimi.it/webdata/cnr-2000, accessed: 2022-05-26
2. Alongi, F., Ghielmetti, N., Pau, D., Terraneo, F., Fornaciari, W.: Tiny neural networks for environmental predictions: An integrated approach with miosix. In: IEEE International Conference on Smart Computing, (SMARTCOMP). pp. 350–355. IEEE (2020). https://doi.org/10.1109/SMARTCOMP50058.2020.00076

3. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). pp. 153–159. IEEE (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
4. Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M.: Optimal resilient distributed data collection in mobile edge environments. Comput. Electr. Eng. **96**(Part), 107580 (2021). https://doi.org/10.1016/j.compeleceng.2021.107580
5. Audrito, G., Casadei, R., Torta, G.: Fostering resilient execution of multi-agent plans through self-organisation. In: ACSOS Companion Volume. pp. 81–86. IEEE (2021). https://doi.org/10.1109/ACSOS-C52956.2021.00076
6. Audrito, G., Casadei, R., Torta, G.: Towards integration of multi-agent planning with self-organising collective processes. In: ACSOS Companion Volume. pp. 297–298. IEEE (2021). https://doi.org/10.1109/ACSOS-C52956.2021.00042
7. Audrito, G., Damiani, F., Giuda, G.M.D., Meschini, S., Pellegrini, L., Seghezzi, E., Tagliabue, L.C., Testa, L., Torta, G.: RM for users' safety and security in the built environment. In: VORTEX. pp. 13–16. ACM (2021). https://doi.org/10.1145/3464974.3468445
8. Audrito, G., Damiani, F., Stolz, V., Torta, G., Viroli, M.: Distributed runtime verification by past-ctl and the field calculus. J. Syst. Softw. **187**, 111251 (2022). https://doi.org/10.1016/j.jss.2022.111251
9. Audrito, G., Pianini, D., Damiani, F., Viroli, M.: Aggregate centrality measures for iot-based coordination. Science of Computer Programming **203**, 102584 (2021). https://doi.org/https://doi.org/10.1016/j.scico.2020.102584
10. Audrito, G., Rapetta, L., Torta, G.: Extensible 3d simulation of aggregated systems with FCPP. In: COORDINATION. Lecture Notes in Computer Science, vol. 13271, pp. 55–71. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_4
11. Audrito, G., Torta, G.: Towards aggregate monitoring of spatio-temporal properties. In: VORTEX. pp. 26–29. ACM (2021). https://doi.org/10.1145/3464974.3468448
12. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Log. **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
13. Backstrom, L., Boldi, P., Rosa, M., Ugander, J., Vigna, S.: Four degrees of separation. In: Web Science 2012, WebSci '12, Evanston, IL, USA - June 22 - 24, 2012. pp. 33–42 (2012). https://doi.org/10.1145/2380718.2380723
14. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
15. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: World Wide Web (WWW). pp. 587–596. ACM (2011). https://doi.org/10.1145/1963405.1963488
16. Boldi, P., Rosa, M., Vigna, S.: Hyperanf: approximating the neighbourhood function of very large graphs on a budget. In: World Wide Web (WWW). pp. 625–634 (2011). https://doi.org/10.1145/1963405.1963493
17. Boldi, P., Vigna, S.: The webgraph framework I: compression techniques. In: World Wide Web (WWW). pp. 595–602. ACM (2004). https://doi.org/10.1145/988672.988752
18. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Comput. Networks **30**(1-7), 107–117 (1998). https://doi.org/10.1016/S0169-7552(98)00110-X

19. Canning, P.S., Cook, W.R., Hill, W.L., Olthoff, W.G., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA). pp. 273–280. ACM (1989). https://doi.org/10.1145/99370.99392
20. Casadei, R., Viroli, M., Audrito, G., Damiani, F.: Fscafi : A core calculus for collective adaptive systems programming. In: ISoLA (2). Lecture Notes in Computer Science, vol. 12477, pp. 344–360. Springer (2020)
21. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artif. Intell. **97**, 104081 (2021). https://doi.org/10.1016/j.engappai.2020.104081
22. Datta, S.K., Bonnet, C.: An edge computing architecture integrating virtual iot devices. In: Global Conference on Consumer Electronics (GCCE). pp. 1–3 (2017). https://doi.org/10.1109/GCCE.2017.8229253
23. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: Analysis of Algorithms 2007 (AofA07). pp. 127–146 (2007)
24. Li, L., Fan, Y., Tse, M., Lin, K.Y.: A review of applications in federated learning. Computers & Industrial Engineering **149** (2020). https://doi.org/https://doi.org/10.1016/j.cie.2020.106854
25. Liu, X., Zhou, Y., Guan, X., Shen, C.: A feasible graph partition framework for parallel computing of big graph. Knowledge-Based Systems **134**, 228–239 (2017). https://doi.org/10.1016/j.knosys.2017.08.001
26. Page, L.: Method for node ranking in a linked database (Sep 4 2001), uS Patent 6,285,999
27. Palmer, C.R., Gibbons, P.B., Faloutsos, C.: ANF: a fast and scalable tool for data mining in massive graphs. In: International Conference on Knowledge Discovery and Data Mining (SIGKDD). pp. 81–90 (2002). https://doi.org/10.1145/775047.775059
28. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27
29. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Symposium on Applied Computing (SAC). pp. 1846–1853. ACM (2015). https://doi.org/10.1145/2695664.2695913
30. Sahlmann, K., Schwotzer, T.: Ontology-based virtual iot devices for edge computing. In: International Conference on the Internet of Things. Association for Computing Machinery (2018). https://doi.org/10.1145/3277593.3277597
31. Testa, L., Audrito, G., Damiani, F., Torta, G.: Aggregate processes as distributed adaptive services for the industrial internet of things. Pervasive and Mobile Computing **85** (2022). https://doi.org/10.1016/j.pmcj.2022.101658
32. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
33. Viroli, M., Casadei, R., Pianini, D.: Simulating large-scale aggregate mass with alchemist and scala. In: Federated Conference on Computer Science and Information Systems (FedCSIS). Annals of Computer Science and Information Systems, vol. 8, pp. 1495–1504. IEEE (2016). https://doi.org/10.15439/2016F407