Parallelising an Aggregate Programming Framework with Message-Passing Interface

Parallelising an Aggregate Programming Framework with Message-Passing Interface

Giorgio Audrito
Dipartimento di Informatica
Università di Torino
Turin, Italy
0000-0002-2319-0375

Alberto Riccardo Martinelli
Dipartimento di Informatica
Università di Torino
Turin, Italy
0000-0002-3707-7015

Gianluca Torta
Dipartimento di Informatica
Università di Torino
Turin, Italy
0000-0002-4276-7213

Abstract—**FCPP is an optimized C++ implementation of the Aggregate Programming (AP) paradigm for the implementation of distributed systems. Until now, it has been either deployed on networks of constrained *far edge* devices, or used for simulating AP systems on a single local computer. Recent work hints at a third possibility, namely adopting AP as a way to effectively program distributed algorithms, and execute them on a centralized high-performance hardware instead of a network of low-end computational nodes. This could also allow an integration of edge and cloud resources, where workload could be dynamically moved between the computational layers.**

In the present work, we describe the first extension of FCPP that supports the execution on a distributed network of high-end computational nodes (e.g., NUMA architectures, or small computer networks). The extension allows the mapping of a large simulation of a far edge system into such nodes basing on the MPI (Message-Passing Interface) standard, providing a first step towards an edge-cloud continuum for aggregate programs.

Index Terms—**distributed systems, programming languages, aggregate programming, high performance computing**

I. Introduction

In recent years, Aggregate Programming (AP) [1] has attracted significant attention as an innovative approach for the development of fully distributed systems. 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 domains such as swarm-based exploration, crowd safety management and monitoring [2]–[4], data routing and collection from sensor networks, dynamic multi-agent plan repair [5], [6].

The main implementations of AP [7]–[10] offer two largely orthogonal features: support for the constructs of the foundational language of AP, namely the Field Calculus (FC) [11]; and connection of FC programs with the execution environment where the distributed system is deployed. More recently, the FC has also been re-instantiated into the more expressive Exchange Calculus (XC) [12], [13].

This work is supported by the Spoke 1 “FutureHPC & BigData” of ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU; by project NODES, funded by the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036; by the Italian PRIN project “CommonWears” (2020HCWWLP); and by the Ateneo/CSP ex-post 2020 project NewEdge.

In particular, the *FCPP* library [9] implements XC as a C++ internal Domain-Specific Language (DSL). The main execution environment provided by FCPP consists of simulations that run on a single computer, simulating sets of nodes situated in space, their dynamics, and proximity-based communications between them. Several research works have used FCPP simulations in recent years, e.g., [2], [14], [15].

The FCPP library has also been extended to work in two physical execution environments, where nodes are Micro Controller Unit (MCU)-based boards communicating through wireless networks [16], [17].

More recently, a new direction for the application of FCPP has started to be explored [18], namely, the implementation and execution of distributed algorithms on high-performance, centralized computers such as the ones typically available in a cloud environment. They have proposed a roadmap going all the way to hybrid execution environments, where a single FCPP application runs partly on small embedded devices, and partly on a powerful central system. And as a first step, they have extended FCPP to execute distributed algorithms on graphs written in FC on a single machine.

In this research work, we add support for the distributed execution of batches of simulations of aggregate systems in FCPP, thus allowing to leverage massive computing resources in the cloud for the task of developing and planning programs for the far edge. Following the roadmap devised in [18], this is a further step that provides the basis to develop extensions supporting a true edge-cloud continuum.

II. Background

A. Aggregate Programming

The AP paradigm is a programming approach suitable for networks of devices, which abstracts away from the specific details of the individual devices, and focuses on the aggregate collective behavior of the whole network [1], [19]. As in a swarm, the devices can communicate with their neighbor devices, and can perform local sensing and actuating. Overall, every device asynchronously executes so called computation *rounds* at a (usually) steady rate, during which: it retrieves received messages; it computes a program (the *same* for all devices), performing local sensing and actuating as needed; and it transmits the messages resulting from the program aggregate function declaration

F::=`FUN` t d(`ARGS`, t x$*$) {`CODE` i$*$}

`FUN_EXPORT` d_t = `export_type`<$t*$>;

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

built-in aggregate functions

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

Fig. 1. Syntax of FCPP aggregate functions.

execution to its neighbors. Different devices, at different times, can follow different branches of the program, thus potentially exhibiting different behaviors.

FCPP [9], [10], the C++ framework whose parallelisation is the main focus of this paper, is one of the existing concrete implementations of FC [11], the formal language underpinning AP. Two other relevant implementations of FC are worth mentioning: Protelis [7] and Scafi [8], both based on the JVM and integrated with the Alchemist simulator [20].

In the rest of this section, we briefly present FC syntax and semantics as it is implemented within FCPP (see Fig. 1), while the next section will describe the architecture of FCPP. Compared to an ordinary C++ function, the declaration of an *aggregate function* requires to: prepend keyword `FUN` to the whole declaration; prepend keyword `ARGS` to the parenthesized sequence of comma-separated arguments; and prepend keyword `CODE` to the other instructions in the function body. Moreover, the declaration must be followed by the *export description*, containing the types that are used by the function in message-exchanging constructs.

The main types of *aggregate expressions*, include the usual *variable* identifiers, *literal values*, *unary operator* and *binary* operators, etc. Specific to the FCPP DSL are:

- *component function calls* `node`.c(e$*$), where c is a function provided by some component (components are described in the next section);
- *aggregate function calls* f(`CALL`, e$*$), where keyword `CALL` must be prepended to the rest of the arguments.

A crucial role is played by aggregate built-in functions in Fig. 1, which provide the constructs of FC in the FCPP DSL. In particular, function `old` (corresponding to the `rep` operator of FC) allows a device to keep track of its own status across different rounds, while function `nbr` (corresponding to the `nbr` and `share` operators of FC) allows communication of a device with neighbors. The main overloads of `rep` and `nbr` offered by FCPP are:

- `old`(`CALL`, v_0, v) with v_0, v of type t, returns the value fed as second argument v in the *previous round* of computation, defaulting to v_0 if no such value is available;
- `old`(`CALL`, v_0, f) computes the result of applying function $f : t \to t$ to the value returned by `old` at the previous round (using v_0 if no such value is available);
- `nbr`(`CALL`, v_0, v) with v_0, v of type t returns the *neighboring field*[1] of values fed as second argument v in the previous round of computation of the *neighbor devices*, defaulting to v_0 for the current node;
- `nbr`(`CALL`, v_0, f) computes the result of applying function f : `field`<t> $\to t$ to the neighboring field of values returned by `nbr` on the neighbor nodes at the previous round, using v_0 for the current node if no such previous value is available for it.

Other built-in aggregate functions worth mentioning here are:

- `fold_hood`(`CALL`, f, ϕ) which *folds* the values in the range of ϕ of type `field`<t> (i.e., a neighboring field) through the binary operator $f : (t, t) \to t$, reducing them to a single value;
- `mux`(`CALL`, c, e_1, e_2) evaluates condition c and expressions e_1, e_2; if c is *true*, it returns the value of e_1, otherwise it returns the value of e_2.

The special keywords (corresponding to C++ macros) that appear in the FCPP DSL syntax (`FUN`, `ARGS`, etc.) are used to ensure that the aggregate context (represented by the `node` object) is carried over throughout program execution. This is needed, among other things, to update an internal representation of the stack trace needed for the *alignment* of messages; in this way, messages (implicitly) originating from `old` and `nbr` are matched in future rounds (on the same or different devices) only to the *corresponding* calls, i.e., calls in the same position in the program syntax and in the stack trace. This mechanism allows to freely compose functions, and use recursion, without mixing-up messages between different parts of the program. As an illustration, consider again the `mux` built-in presented above: since both expressions e_1 and e_2 are evaluated, regardless of the value of condition c, if e.g., e_1 contains a call to `nbr`, such a call will be executed by all the nodes, so a device where c holds will consider as neighbors also devices where c does not hold (and vice versa). This is different from what happens, e.g., when the call to `nbr` is in a branch of a C++ `if` or ternary operator.

B. FCPP

The FCPP library architecture, shown in Fig. 2, consists of three conceptual layers: *(1)* C++ data structures of general use; *(2)* components; and *(3)* aggregate functions.

While the first layer defines data structures needed by the other layers, and the third layer defines the aggregate functions that implement FC (and several useful algorithms built on-top of them), it is worth giving some details about the *components* layer. As shown in Fig. 2, components define the fundamental abstractions *node* (representing single devices) and *net* (representing the overall network). Exploiting C++ template programming, several components can be combined in a mixin-like fashion to define the specific types of *node* and *net* for a given application. The purpose of the most relevant components is shown in Table I.

[1]For the present purpose, a neighboring field can be thought of as a map from neighbor device identifiers to associated values.

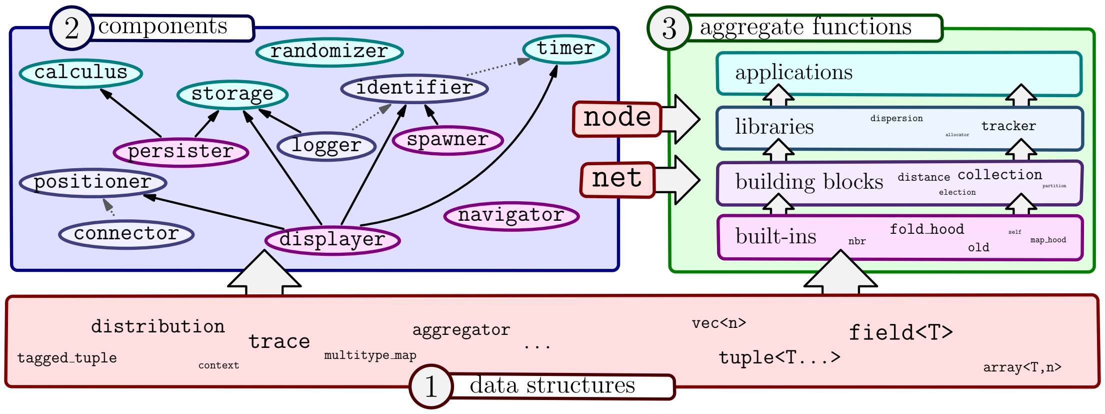

Fig. 2. The three main layers of the software architecture of FCPP: *data structures* for both other layers, and *components* which provide node and network abstractions to *aggregate functions*. Dependencies between components can be either *hard* (solid), for which the pointed component is always required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

TABLE I
MAIN FCPP COMPONENTS.

Component	Provides
`calculus`	allows usage of aggregate programming constructs
`connector`	handles periodic broadcasts of messages
`displayer`	provides a GUI for the whole network
`identifier`	gives access to nodes through their unique identifiers
`spawner`	automatically creates nodes in the `identifier`
`storage`	attaches tagged data to nodes or to the net object
`timer`	accesses and regulates scheduling of rounds

Some components come with *variations* for different scenarios, sharing a common interface. For example, the *connector* component has a variant `simulated_connector` used in simulations, a variant `hardware_connector` used in physical deployments, and a variant `graph_connector` for centralized processing of graphs.

As an example of a relevant combination of components, which is offered directly by FCPP, the *node* and *net* classes suitable for a batch simulation can be obtained by defining the `batch_simulator` type as follows:

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

exploiting the `DECLARE_COMBINE` macro to combine all the components listed as the remaining parameters.

C. Parallel and Distributed Computing

For improving the performance of an application, there are several approaches. One option is to invest in faster hardware, while another is to optimize the code to make better use of the available resources.

Modern computers often come equipped with multiple CPUs, each containing multiple physical and logical cores. To fully leverage this hardware, applications need to be designed with parallelism in mind. However, parallelizing code can be a challenging task, especially when the original code was written with a sequential mindset. Writing parallel code introduces additional complexities, such as the potential for deadlocks or resources starvation. Consequently, significant efforts have been made to simplify this process, ranging from low-level libraries like PThreads [21], OpenMP [22], OpenCL [23] and MPI to high-level skeleton-based approaches such as Skepu [24], SkeTo [25], TBB [26], and Fastflow [27].

In many real-world and scientific applications that involve heavy computations, a single computer's CPUs, cores, and memory may not be sufficient. In such cases, the application needs to be distributed across multiple nodes. Writing distributed computing code introduces additional complexities. The user must program communication over an unreliable network, further increasing the importance of fault-tolerant practices as more hardware is utilized, increasing the likelihood of hardware failures.

To facilitate common operations in distributed computing, the Message Passing Interface (MPI) provides a durable standard. MPI enables point-to-point communication for message exchanges. It also supports collective communication, which encapsulates common communication patterns between processes, such as Broadcast, Scatter, Gather, and more.

To evaluate the quality of parallelization, two commonly used measures are *strong scaling* and *weak scaling*.

Strong scaling refers to the capability of a software to solve a problem of fixed size more efficiently as the computing resources increase. It is closely tied to the concept of speedup in a program. The speedup of a parallel program, denoted as $S(N)$, is defined as the ratio between the time taken by the sequential program (t_s) and the time taken by the parallel program ($t_p(N)$) with a given number of processing elements (N). Ideally, the speedup should exhibit a linear relationship.

Amdahl's law [28] states that there is an upper bound to the speedup of a program. Let $p \in [0, 1]$ represent the fraction of time spent in the part of the code that can benefit from parallelization, while $s = 1 - p$ represents the fraction of time spent in the serial part of the code. Assuming an ideal speedup in the parallel part, the execution time on N processors is no better than $t_p(N) = s * t_s + p * t_s/N$.

Amdahl's law indicates that the maximum speedup, denoted as $S(N) = t_s/t_p(N) = t_s/(s * t_s + (1 - s) * t_s/N) = N/(1 + (N - 1) * s)$, is strongly influenced by the sequential fraction of the code. As N approaches infinity, $S(N)$ approaches $1/s$.

However, it's important to note that Amdahl's law doesn't account for all the overheads introduced by parallel implementations, such as communication, synchronization among different workers, or initialization of processes/threads. As a result, the actual performance of a program tends to be worse than predicted by this law.

In the investigation of weak scaling, the focus is on a variable problem size while maintaining a constant workload per computing resource. Gustafson's law [29], proposes a different approach by scaling the problem size to the number of processors rather than fixing the problem size. Instead of deriving the execution time of the parallel code from the sequential one like Amdahl's law, Gustafson's law aims to keep the time spent on each processor constant by fixing the problem size solved on each processor. As a result, the overall problem scales as the number of processors (N) increases. For a fully parallelizable code, the time required for a problem of size $O(N)$ to run on N processors remains constant, and weak scalability is usually easier to achieve than strong scalability.

III. Parallelising FCPP with MPI

A. Goal

In this research work, we added support for the distributed execution of batches of simulations of aggregate systems in FCPP [10], thus allowing to leverage massive computing resources in the cloud for the task of developing and planning programs for the far edge. Following the roadmap devised in [18], this is a step that provides the basis to develop further extensions supporting a true edge-cloud continuum.

Thus, with this work we had multiple goals in mind. First, we wanted to show that simulations in FCPP can be easily split across resources in the cloud, speeding up the production of *global* plots that are meant to guide the development of far edge applications. Second, we wanted to abstract the distribution details, allowing the final user to seamlessly use our extension without significant effort or prior knowledge of MPI processes. Finally, we wanted to provide a starting point that could be reused in future extensions, where (parts of) a far edge execution could be mirrored into a digital twin running in the cloud in order to optimize the allocation of resources.

B. User Interface

In order to provide the simplest interface for MPI parallel execution to the user, we mainly expanded:

- the `make.sh` build helper (based on CMake), by adding an `mpi` optional flag to enable MPI support;
- the `batch::run` function, by writing an overload that instead executes the processes on an MPI cluster of nodes.

The new overload is selected by passing a new *execution policy* (other than the previously existing *sequential*, *parallel*, *dynamic* policies), called *distributed execution*, which is a configuration struct with 4 fields: *(i)* the number of threads per node to use, *(ii)* the size of dynamic chunks, *(iii)* the fraction of simulations to be assigned in dynamic chunks rather than statically, and *(iv)* a boolean flag on whether the simulations should be (deterministically) shuffled. If `batch::run` is called without a policy, the default policy is *distributed execution* if MPI is enabled, otherwise is *dynamic execution*.

Overall, the provided extensions allow to run a large batch of simulations on an MPI cluster while requiring minimal effort from the user. By just adding the `mpi` flag in the build helper script, the previously existing target `run/spreading_collection_batch.cpp` of the FCPP sample project[2] will run on multiple nodes, specified in a file `hosts.txt`, provided that they have MPI installed.

C. Implementation

To parallelize the code we followed the methodology presented in [30], using the durable MPI standard for the communication across different computational nodes. We chose MPI over higher level libraries for several reasons. MPI is extensively used in HPC for distributed applications, and its implementations are highly mature and well-optimized. For our specific use case, MPI offers an optimal level of abstraction, effectively concealing the intricacies of various network architectures while maintaining satisfactory performance levels. Opting for a higher level tool wouldn't have been more advantageous, as it could potentially decrease performance and introduce additional dependencies. Another advantage of MPI is its versatility, as it can be employed on various network architectures, and there are multiple mature implementations (e.g. OpenMPI [31], MVAPICH [32], etc...) tailored to different purposes. As a result, the same MPI program can be executed on different network architectures while delivering state-of-the-art performance.

FCPP is written in modern C++, making it easy to identify the loops that need parallelization. By adhering to modern and well-established Software Engineering practices, FCPP avoids the misuse of global variables or functions with a high number of parameters, thus facilitating the parallelization process.

Originally, FCPP was a multithreaded code designed to run a set of simulations using a pool of threads to take advantage of a single modern multicore CPU. Since the need to run numerous simulations may exceed the computational capacity of a single node, we decided to distribute FCPP using MPI, enabling us to execute multiple simulations in parallel across multiple nodes, effectively reducing the overall runtime.

Thus, we spawn a multi-threaded MPI process per node. According to the *distributed execution* policy, the set of simulations for each of the N nodes is obtained first by assigning regularly one every N of the first part of simulations to each node, then by dynamically assigning the remaining simulations in chunks to nodes as they finish. In both phases, threads within each node are assigned single simulations dynamically.

Once the simulations on a node are completed, a plot structure is generated, containing an aggregation of the obtained

[2]https://github.com/fcpp/fcpp-sample-project

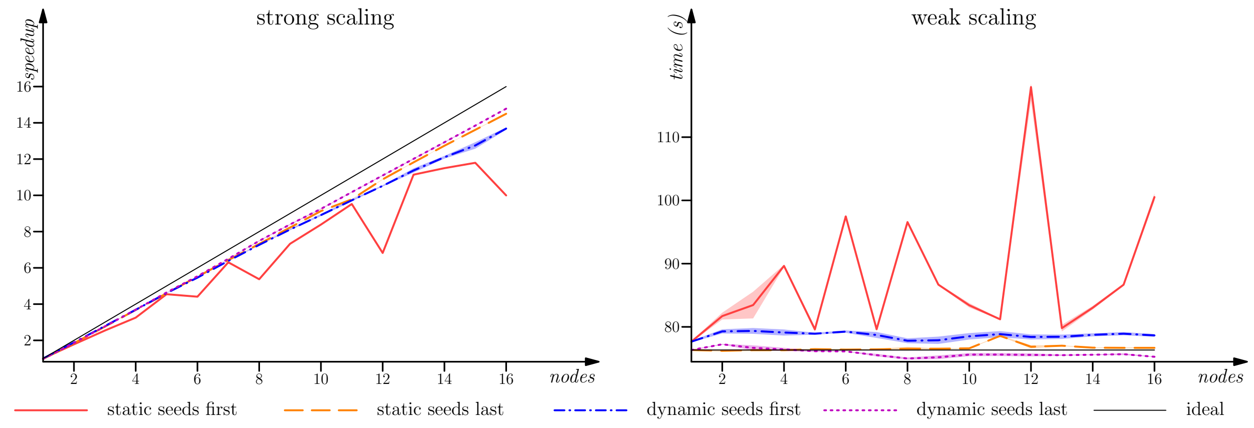

Fig. 3. Strong and weak scaling for different configurations and orderings, relative to an ideal baseline

data, and able to produce PDF plots. The plot structures generated by all nodes must be aggregated to create a single final plot. Thus, all MPI processes send their plots to a master process, which combines them to generate the final plot. While this step could potentially become a bottleneck, the typical size of plot structures is relatively small as they only contain aggregated information (a few kilobytes at most; e.g., 28 KB for the example in our evaluation). Thus, in our tests we did not observe a noticeable degradation in performance. In case larger plot structures could become relevant in the future, the resulting aggregation issue could still be easily solved by implementing an $O(\log N)$ global reduce operation on the plots, leveraging their associative and commutative nature.

IV. Evaluation

For our experiments we used up to 16 light nodes of the OCCAM cluster [3], with the following technical specs.

- CPU: 2xIntel Xeon E5-2680 v3, 12 core at 2.5Ghz
- RAM: 128GB/2133 (8 x 16 Gb)
- DISK: SSD 400GB SATA 1.8 inch.
- NET: IB 56Gb + 2x10Gb
- LAYER IB: 56 Gbps
- LAYER ETH10G: 10 Gbps
- TOPOLOGY IB: FAT-TREE

Each test consists in running the “spreading collection” batch of simulations from the FCPP sample project, using up to 16 nodes. In the simulated scenario, randomly moving devices compute a self-adaptive estimate of the network diameter, by a combination of information spreading and gathering. The batch of simulations measures the performance of the diameter estimation while varying random seeds as well as 25 levels of device speed, device density, network diameter in hops, and variance of the computation schedule.

We performed 5 runs for each test and calculated mean and standard deviation, comparing two different configurations of the distributed execution policy (fully static and fully dynamic), and two different orderings of simulations: ordered so that heavier simulations repeat every 24 simulations (seeds first), and a more mixed-up ordering (seeds last). In both configurations we disabled shuffling, and we set the dynamic chunk size to 1 (single simulations are being dynamically assigned to nodes), as we found shuffling and larger chunk sizes were not sufficiently beneficial for the system considered. As a baseline for strong and weak scaling, we used the version of FCPP before introducing MPI on a single node, assuming perfect scaling for more nodes (ideal line).

Fig. 3 shows strong and weak scaling (c.f. Sec. II-C) for the different policies and orderings. The plots show good scaling performance on all different configurations, with very low variance, and the centralized communications step does not seem to be a problem with the number of nodes used. Having the simulations in an even (seeds last) or uneven (seeds first) ordering has a strong impact, with *seeds last* reaching almost ideal scaling, and *seeds first* being almost twice as slow for 12 nodes. The difference in performance increases with the greatest common divisor (GCD) between the number of nodes and 24, so that 24 nodes would result in the maximum unevenness. Furthermore, the dynamic policy seems preferable as it always outperforms the static one: by a small margin for the favorable ordering, and by a larger margin for the unbalanced one.

V. Conclusions and Future Work

In this work, we added support for the distributed execution of batches of simulations in FCPP, allowing to leverage cloud resources for the planning of algorithms for the far edge. The support has been designed while ensuring seamless integration with existing FCPP projects, and requiring minimal effort from the final user. We validated our contribution on the main example in the FCPP sample project, checking both weak and strong scaling, obtaining good results. In [33] the Alchemist simulator is integrated with the MultiVeSta statistical analysis tool [34] to execute distributed simulations of algorithms written in Protelis [7]. While in general JVM-based implementations of AP are slower than FCPP, as shown

[3]https://c3s.unito.it

in [9], a deeper comparison with the distribution approach and performance of [34] is worth exploring in future work.

Following the roadmap devised in [18], this work is a first step towards the integration of edge and cloud resources. Even though at present FCPP does not yet support a true edge-cloud continuum, it could reach it by performing a few additional steps, that we plan to do in future work:

1) *Allowing the distribution with MPI of a single large-scale simulation.* This could be accomplished by applying similar techniques as those used in this paper to the main event cycle of FCPP, which is currently already multi-threaded (but not multi-CPU).
2) *Designing a cloud digital twin of a far edge system.* This could be accomplished by developing a "mirroring" component, exchanging the necessary data from far edge devices and a centralized digital twin of the system, that could run on the cloud thanks to the achievement in (1).
3) *Devising strategies for the dynamic offloading of computations.* By empowering the mirroring component with suitable policies, the computation could be dynamically moved from being executed on the edge device to being executed on the digital twin.

References

[1] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the Internet of Things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[2] G. Audrito, F. Damiani, V. Stolz, G. Torta, and M. Viroli, "Distributed runtime verification by past-ctl and the field calculus," *J. Syst. Softw.*, vol. 187, p. 111251, 2022.

[3] G. Audrito and G. Torta, "Towards aggregate monitoring of spatio-temporal properties," in *VORTEX*. ACM, 2021, pp. 26–29.

[4] G. Audrito, F. Damiani, G. M. D. Giuda, S. Meschini, L. Pellegrini, E. Seghezzi, L. C. Tagliabue, L. Testa, and G. Torta, "RM for users' safety and security in the built environment," in *VORTEX*. ACM, 2021, pp. 13–16.

[5] G. Audrito, R. Casadei, and G. Torta, "Fostering resilient execution of multi-agent plans through self-organisation," in *ACSOS Companion Volume*. IEEE, 2021, pp. 81–86.

[6] ——, "Towards integration of multi-agent planning with self-organising collective processes," in *ACSOS Companion Volume*. IEEE, 2021, pp. 297–298.

[7] D. Pianini, M. Viroli, and J. Beal, "Protelis: practical aggregate programming," in *Symposium on Applied Computing (SAC)*. ACM, 2015, pp. 1846–1853.

[8] R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "Fscafi : A core calculus for collective adaptive systems programming," in *ISoLA (2)*, ser. Lecture Notes in Computer Science, vol. 12477. Springer, 2020, pp. 344–360.

[9] G. Audrito, "FCPP: an efficient and extensible field calculus framework," in *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 153–159.

[10] G. Audrito, L. Rapetta, and G. Torta, "Extensible 3d simulation of aggregated systems with FCPP," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 13271. Springer, 2022, pp. 55–71.

[11] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM Trans. Comput. Log.*, vol. 20, no. 1, pp. 5:1–5:55, 2019.

[12] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, and M. Viroli, "Functional programming for distributed systems with XC," in *Proceedings of ECOOP 2022*, ser. LIPIcs, vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:28.

[13] G. Audrito, R. Casadei, F. Damiani, G. Torta, and M. Viroli, "Programming distributed collective processes for dynamic ensembles and collective tasks," in *Proceedings of COORDINATION 2023*, ser. Lecture Notes in Computer Science, vol. 13908. Springer, 2023, pp. 71–89.

[14] G. Audrito, R. Casadei, and G. Torta, "On the dynamic evolution of distributed computational aggregates," in *Proceedings of ACSOS 2022, Companion Volume*. IEEE, 2022, pp. 37–42.

[15] G. Audrito, F. Damiani, S. Rinaldi, L. C. Tagliabue, L. Testa, and G. Torta, "Aggregate programming for customized building management and users preference implementation," in *IoT Edge Solutions for Cognitive Buildings - Technology, Communications and Computing*. Springer, 2023, pp. 147–172.

[16] L. Testa, G. Audrito, F. Damiani, and G. Torta, "Aggregate processes as distributed adaptive services for the industrial internet of things," *Pervasive and Mobile Computing*, vol. 85, 2022.

[17] G. Audrito, F. Terraneo, and W. Fornaciari, "Fcpp+miosix: Scaling aggregate programming to embedded systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 3, pp. 869–880, 2023. [Online]. Available: https://doi.org/10.1109/TPDS.2022.3232633

[18] G. Audrito, F. Damiani, and G. Torta, "Bringing aggregate programming towards the cloud," in *Proceedings of ISoLA 2022, Part III*, ser. Lecture Notes in Computer Science, vol. 13703. Springer, 2022, pp. 301–317.

[19] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019.

[20] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *J. Simulation*, vol. 7, no. 3, pp. 202–215, 2013.

[21] D. R. Butenhof, *Programming with POSIX Threads*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[22] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann, "Parallel programming environment for openmp," *Sci. Program.*, vol. 9, no. 2,3, p. 143–161, 2001.

[23] *OpenCL*, Khronos Compute Working Group., 2023 (last accessed), https://www.khronos.org/opencl/.

[24] J. Enmyren and C. W. Kessler, "Skepu: A multi-backend skeleton programming library for multi-gpu systems," in *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications*, ser. HLPP '10. ACM, 2010, p. 5–14.

[25] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Proc. of the 1st Int. Conf. on Scalable Information Systems*, ser. InfoScale '06. ACM, 2006, p. 13–es.

[26] *Intel Threading Building Blocks*, Intel Corp., 2020 (last accessed), http://software.intel.com/en-us/intel-tbb/.

[27] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. Wiley-Blackwell, Jan. 2017, pp. 261–280.

[28] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). ACM, 1967, p. 483–485.

[29] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, p. 532–533, may 1988. [Online]. Available: https://doi.org/10.1145/42411.42415

[30] M. Aldinucci, V. Cesare, I. Colonnelli, A. R. Martinelli, G. Mittone, B. Cantalupo, C. Cavazzoni, and M. Drocco, "Practical parallelization of scientific applications with openmp, openacc and mpi," *Journal of Parallel and Distributed Computing*, vol. 157, pp. 13–29, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731521001295

[31] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2004, pp. 97–104.

[32] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, pp. 101–208, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093

[33] D. Pianini, S. Sebastio, and A. Vandin, "Distributed statistical analysis of complex systems modeled through a chemical metaphor," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 416–423.

[34] S. Sebastio, A. Vandin *et al.*, "Multivesta: Statistical model checking for discrete event simulators," in *VALUETOOLS*. ACM, 2013, pp. 310–315.