Refactoring an electric-market simulation software for massively parallel computations

Franco Seveso, Raúl Marichal, Ernesto Dufrechou, and Pablo Ezzatti

Instituto de Computación (INCO), Universidad de la República, Montevideo, Uruguay
{franco.seveso, rmarichal, edufrechou, pezzatti}@fing.edu.uy

Abstract. In the last two decades, Uruguay has been immersed in the process of significantly changing its energy generation matrix, especially by the introduction of wind and solar sources. In this context, SimSEE, a simulation and optimization software designed to help decision-making in generating and distributing electrical energy, is extensively used. The design of this tool is conceived for conventional CPUs and follows a sequential execution paradigm. This paper focuses on a refactoring of SimSEE that enables leveraging massively-parallel hardware platforms, seeking to adapt the tool for the increasing size and complexity of Uruguay's electric market. We extend our previous ideas about reorganizing the software architecture to exploit the parallelism in each time-step of SimSEE's simulation. In more detail, we present two variants following this parallelism pattern, a straightforward parallel version that requires replicating the used memory and a variant that implies limited performance restrictions but requires a minimal memory overhead.

Keywords: Coarse-grained parallelism · Electric energy generation · Stochastic Dynamic Programming · Memory Usage.

1 Introduction

In the last two decades, the Uruguayan electricity generation matrix experienced important changes, mainly due to the constant incorporation of new generation sources such as wind and solar farms [16]. This brings important challenges to efficiently using and distributing the available resources, making demand and generation capacity prediction necessary. The SimSEE (Electric Power Systems Simulator) is a software tool that allows users to make customized simulations of an electric energy generation system. Its principal purpose is to aid in the decision-making process, both in the long term (investment planning) and in the short term (system operation and market simulation) [7]. It was developed at the Universidad de la República, by engineers of the Instituto de Ingeniería Eléctrica (IIE) of the Facultad de Ingeniería (FING), between the years 2006 and 2008, maintaining a constant evolution up to now [10, 6]. The tool is tailored to represent the reality of Uruguay's electric market. Hence, its most intensive users are the Uruguayan public agency managing the electricity generation market, called ADME [3] and the Uruguayan public company that generates and distributes electric energy (UTE).

This work's motivation is to adapt the simulation tool to increase the number of actors (e.g., power sources) and complexity of the models while maintaining the accuracy levels and simulation runtime constraints, such as keeping bounded simulation times. For this purpose, we aim to leverage the computational power offered by modern massively parallel hardware platforms, such as heterogeneous servers equipped with GPUs. These platforms have shown an impressive evolution in the last decades and have become a vital piece of the HPC landscape [5, 4, 11].

This effort is an extension of [13], where we identify the SimSEE bottlenecks by evaluating different realistic cases and propose a new software architecture design for the SimSEE to exploit the massively parallel computations. Specifically, this work presents two different variants of SimSEE following the previously described parallelism pattern. First, a direct parallel version that requires replicating the used memory, and second, we design a new variant that implies concrete performance restrictions but requires a minimal memory overhead for each simulation trajectory computed in parallel. In other words, a parallel version that offers scalability in memory use.

The rest of the work is organized as follows. Section 2 synthesizes the arrived results of the previous work. In Section 3 we present different variations to implement the previously proposed and discussed parallel design. Next, in Section 4, the experimental evaluation results of the implementations are summarized. Finally, Section 5 presents our conclusions and some future lines of work.

2 The SimSEE and previous results

As we stated previously, SimSEE (Electric Power Systems Simulator) is a software tool that allows users to make customized simulations of an electric energy generation system. It is based on Stochastic Dynamic Programming techniques, it allows to simulate the contribution of multiple energy sources, including thermal, solar, hydro-electric or wind energy, to a specific electrical network. For this reason in the simulation different and random realizations of these stochastic processes, called trajectories, are executed, and the results are expressed in terms of the expected value or as distributions or probabilities of exceedance. Its principal purpose is to aid in the decision-making process both in the long term (investment planning) and in the short term (system operation and market simulation) [7]. It was developed at the UDELAR, by engineers of the Instituto de Ingeniería Eléctrica (IIE) of the Facultad de Ingeniería (FING) in the Pascal programming language [12].

In the previous work were designed and evaluated different realistic test cases, varying the scenarios between three classes: hourly, daily, and weekly (i.e., short, mid, and long-term). Since the SimSEE is a legacy complex system developed in

Pascal and is being used by ADME, the main goal of these tests was to exhibit the most resource-consuming procedures in the simulation routine and then, with a fine-grain approach, efficiently implement a parallel version of those procedures. This approach avoids re-implementing the entire simulation routine, a task of serious difficulty and resource demand. The experimental results showed that none of the procedures represent an important bottleneck since a single execution of these procedures is not demanding enough, deriving that the cost came from the number of calls or invoked. Nevertheless, we found that an important part of the simulation runtime is invested in Simplex resolution routines. In Figure 1, for example, we can see the proportion of Simplex-related operations (middle green rectangle), and Table 1 shows how many times some of these procedures are called into the simulation. Based on this, we propose a strategy to exploit parallelism by designing a new simulation scheme focused on a coarse-grain approach instead of including fine-grain parallelism.

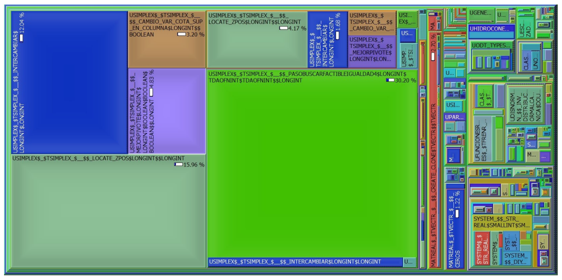

Fig. 1. Graphical representation of simulation call map with a daily playroom. The area of each rectangle is proportional to the runtime of the corresponding procedure.

Table 1. Top 5 procedures for the daily playroom simulation.

Procedure	Perc. (%)	CPU Cycles	Calls	Unit
`PASOBUSCARFACTIBLEIGUALDAD4`	29.0	3.349×10^{11}	1.348×10^{6}	`USIMPLEX`
`LOCATE_ZPOS`	20.3	2.344×10^{11}	5.991×10^{7}	`USIMPLEX`
`INTERCAMBIAR`	15.4	1.780×10^{11}	8.258×10^{6}	`USIMPLEX`
`MEJORPIVOTE`	6.1	6.991×10^{10}	5.972×10^{7}	`USIMPLEX`
`CAMBIO_VAR_COTA_SUP_EN_COLUMNA`	4.1	4.719×10^{10}	5.670×10^{10}	`USIMPLEX`

The simulation algorithm can be divided into two main loops. The first and outer loop iterates through the trajectories, line (3) in Algorithm 1, and for each of these trajectories, there is a second loop (5) that sets up and solves, for each time step (7), an optimization problem using the Simplex algorithm, represented by a matrix, based on the work of Rutishauser et al. [14]. Since each trajectory is independent, in [13] we redesigned the routine interchanging the loops and structures related to solving, for the different trajectories, the simplex matrices associated with each time step in parallel. Note the (parallel) **for** of trajectories in line 11 of the Algorithm 2, solving independent simplex matrices.

Algorithm 1 `Simular + cargarSala`

```
room = cargarSala(room_file)
// Simulation
prepare(room)
for Trajectories:
  Traj_Init(room)
  for time_steps:
    simplex_matrix = simplex(room)
    solve(simplex_matrix)
  end
end
```

Algorithm 2 `Simular' + cargarSalas`

```
1  rooms = cargarSalas(room_file)
2  // Simulation
3  for i in Trajectories:
4    prepare(rooms[i])
5    Traj_Init(rooms[i])
6  end
7  for time_steps:
8    for Trajectories:
9      spx_array[i] = simplex(rooms[i])
10   end
11   (parallel) for Trajectories:
12     solve(spx_array[i])
13   end
14 end
```

Pseudocode of the original SimSEE simulation scheme (Algorithm 1) and the reorganization proposed in [13] (Algorithm 2)

3 Proposal

As discussed in the previous section, the strategy behind the massively parallel version of the simulation is to process independent trajectories at the same time, solving in parallel multiple simplex matrices. To implement this design, the data associated with each trajectory must stay independent from the others. In other words, each trajectory needs its *playroom*, which is a data structure that holds all the information about the electrical system being simulated, including the state variables of each power source, such as the water level of hydroelectrical plants. In the original version, the playroom is created by the procedure `cargarSala` reading a configuration file associated with the room line by line. This room is instantiated once and used throughout the whole simulation. Therefore, the cost of the `cargarSala` in the original algorithm is constant for any number of trajectories, which is why it was not taken into account in the previous work evaluation. Depending on the room, this procedure can take several computation cycles, mainly conditioned to the number of entities and historical information. Then, in the simulation phase, as trajectories are run sequentially for each time step, the state variables of the playroom can be reset for each new trajectory. If trajectories run in parallel, sharing the state variables is impossible, so a certain degree of data replication is necessary.

In the following sections, we propose different implementations to load multiple playrooms and, considering the cost of in/out and file reading [9], we advance in implementing routines that use the playroom file once and create multiple rooms. On the other hand, it is important to highlight that memory usage multiplication is a very limiting strategy when trying to massively parallelize systems. In other words, one of the essential characteristics to reach in parallel patterns is the scalability in memory usage [9, 8]. We design, implement and evaluate a version where the different rooms share certain structures through pointers or references, using SimSEE native classes. Later we present another version with a simple structure to store the reference pointers for the shared units between the rooms, avoiding the list searches of the first.

3.1 Loading the playrooms for massively-parallel trajectories, `naive`

A straightforward strategy to address the problem of independent playrooms is to create a collection of rooms, with a size equal to the number of trajectories, by calling the procedure `cargarSala` multiple times. As an early result, Table 2 shows the outcome of this implementation. The table shows the elevated cost of instantiating these rooms, with linear growth of the elapsed time to load the rooms and simulation time due to the preparation of the rooms in the `Simular'` procedure.

Table 2. Elapsed time (ms) of room load (`cargarSala`) and simulation (`Simular`) comparison for both strategies in hourly playroom (see Section 4.2 for details about test cases and the runtime environment).

# of Trajectories	`cargarSala`	`Simular`	Loop of `cargarSala`	`Simular'`
256	4125	27828	968844	161094

For reasons previously described, we studied the procedure `cargarSala` in detail and evaluated the possibility of implementing a new procedure capable of instantiating multiple rooms efficiently.

3.2 Improving the playrooms replication, `base`

Although the previous implementation returns the expected collection, it needlessly repeats procedures when loading the different rooms. Since all rooms are equal and loaded from the same file, we can improve this implementation in different ways, such as refactoring some of the procedures or optimizing the access to the file for reading. Considering the above, we modified the strategy initially used to load the rooms, deriving this procedure in `cargarSalas`. Unlike `cargarSala`, this implementation reads the text file associated with the room once, line by line, and instantiates simultaneously many structures and units as rooms are needed.

Although `cargarSalas` is also a naive version, it allows fulfilling the task of loading multiple rooms, avoiding multiple file reads and the implied runtime overhead. However, this strategy implies a large memory usage since it instantiates every single structure for the room N times.

The memory used in the simulation is mainly given by the initialization of each room. For example, the sizes of the evaluated rooms (hourly, daily and weekly) result in around 90MB, 3.8MB, and 2.4MB, respectively, when mapped to Pascal objects. Therefore, simulating 1000 trajectories of a hourly playroom will require around 90GB of memory, making this `cargarSalas` implementation not scalable and unfeasible when the number of trajectories grows for devices with limited memory resources.

3.3 Sharing references to avoid memory allocations, `RefCat`

A reasonable conclusion from the previous discussion, can be that the main restriction for the inclusion of massively parallel techniques in the SimSEE is the multiplication of the memory usage. This motivated a detailed study of the units created in `cargarSalas`, looking for possible instances unchanged between the different trajectories. The *Dynamic Parameters record* or simply *record* are particular cases of a valid structure for this approach. Based on [15], the dynamic *records* can be defined as the system allowing the various playroom entities to change their parameters at runtime. A dynamic *record* consists of a start date, a periodicity, and a set of parameters depending on the type of entity that it belongs to. The actors or other entities that require parameters that may vary over time must specify in their *records* what those parameters are, and the system automatically updates them when the indicated date is reached, both in simulation and in optimization. Each *record* will be valid from its start date until another replaces it. In other words, if an actor has a single *record* at the beginning of the simulation, it will be valid for the entire simulation horizon. If the actor has a *record* at the beginning and another in the middle, the first *record* will be valid until half of the simulation, then it will be replaced by the second one that will follow until another replaces it or the simulation finishes. Basically, represent historical states of the entities, and remain invariant through the simulation.

In the particular case of the hourly room (VATES [1]), it has about 2000 *records* associated with the entities that must be instantiated when loading the room, which, in the worst case, implies having to instantiate $2000 \times NTrajectories$ *records*. The simplest solution would be for the same entities from the different rooms to point to a single instance of the *records* and somehow share them. The problem that arises is that many of these *records* have references or pointers to the entities within the same room. Due to this, it is impossible to directly share the *records* since these references are accessed when computing the variables to prepare, for example, the simplex matrix at each time step. So, to share the *records* it is necessary to have the information of the different references corresponding to each room and switch those references when computing with them. Figure 2 shows this problem; while the entities of different rooms must be independent, the *records* remain constant through the simulation (red box), and the main problem is the references to entities in the same room (yellow arrows).

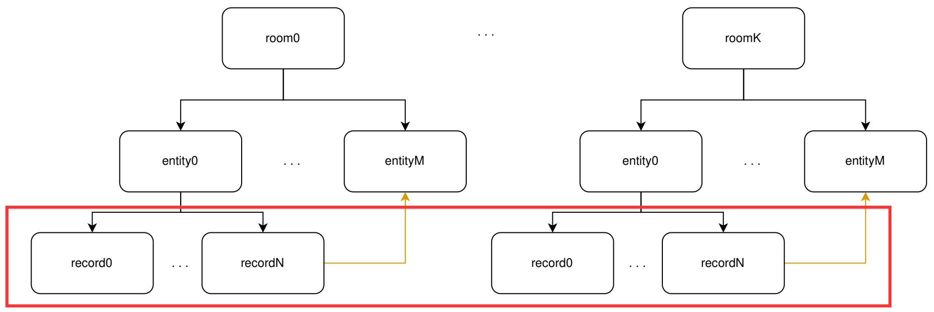

Fig. 2. Simplified scheme of multiple playrooms. Entities in different rooms must be different instances, but the files remain the same for the same entity in different rooms except by the references.

Considering the previously described situation, we implemented a “Reference Catalogue”, which is responsible for containing the information of references and maintaining the consistency of the ones for the room when the simulation needs to use it. The main idea is to map the excessive memory usage of rooms related to the *records*, to simple pointers stored in the “Reference Catalogue”, representing a critical memory usage reduction. The new simulation scheme sharing *records* is presented in the Algorithm 3, where the procedure `ChangeReferences` is added to set the correct references to entities of different rooms when a trajectory calculates and prepares, for example, the simplex matrix. Note that this procedure is called twice in the Algorithm 3, in first place (line 3), due to the preparation of the rooms, initializing the actors, sources and variables of the playrooms previous to the true simulation stage. The second call (line 9) in the simulation stage, is needed to set the correct references before the simplex creation for each time step.

Algorithm 3 `Simular'` sharing room's objects

```
rooms, catalogue = cargarSalas(room_file)
for i in Trajectories:
  ChangeReferences(rooms[i], catalogue)
  prepare(rooms[i])
  Traj_Init(rooms[i])
end
for time_steps:
  for Trajectories:
    ChangeReferences(rooms[i], catalogue)
    spx_array[i] = simplex(rooms[i])
  end
  (parallel) for Trajectories:
    solve(spx_array[i])
  end
end
```

3.4 Enhancing the access to shared references in the simulation, `RefDicc`

In the previously described strategy, when a reference needs to be changed, it is necessary to search for the new reference within the list of entities and the "Reference Catalogue" in simulation time. This operation implies a complexity proportional to the number of references and entities the room has, which can be substantial depending on the room. To avoid these searches, we propose to introduce a dictionary (a matrix of pointers), responsible for saving the already resolved references for all the rooms.

This strategy avoids the unnecessary task of iterating over the list of entities looking for a reference and then resolving it. By applying these changes, we convert the complexity of the operation that changes a reference to a constant order, significantly reducing the computation cost and its impact on the simulation. In other words, we sacrifice memory consumption to store the already solved references for each room to avoid repeated operations in simulation time.

4 Experimental evaluation

This section presents the experimental evaluation of our proposal.

4.1 Test cases

To evaluate how the proposed algorithm schemes perform, we vary the simulation scenarios between hourly, daily, and weekly playrooms, which are representative workloads of the SimSEE. Based on VATES [1] the hourly playroom runs to calculate the optimal energy dispatch in the following week and incorporates the forecasts of hydraulic contributions to the dams, the forecast of wind and solar generation, and the forecast of demand for every hour. It has 168 time steps, composed of three hydro generators with reservoirs and one without, twelve fuel-fired generators, one wind and one solar generator representing all the country's farms, and a CEGH source, that states for modeling multi-variable stochastic processes, working as a time series synthesizer having common characteristics with the available time series measures, with two hydrologic state variables.

The daily playrooms are mostly used for seasonal programming, with a CEGH source for the contribution to basins that also has two hydrologic state variables, and there are still three hydro generators with reservoirs. This kind of playroom is the most required by SimSEE users [2].

4.2 Runtime environment

This section contains the environment specification where all the executions and results proposed in this work were carried out. This environment has an 8-core processor Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 GB of RAM with Linux operating system.

4.3 Experimental results

The time measurements of the simulation were presented in scale to obtain an abstraction for the number of trajectories used, but in the same way, the computation time required by the simulation with the strategy of loading multiple rooms is still a bit distant from the original in terms of efficiency.

As mentioned in the previous section, Table 2 shows how the time of loading many rooms (repeating the file read) is significantly high, converting this into a non-viable implementation strategy, and in terms of memory use, it replicates all the rooms. The first proposed idea to mitigate this problem was reading the file once, with the implemented procedure `cargarSalas`. In the new idea, the runtime is not a problem, but the issue of memory usage keeps present.

Table 3. Peak memory usage (MB) for the different strategies simulating 100 trajectories of hourly, daily and weekly rooms.

	Original	Full independent	Reference Catalogue
Hourly	500	9560	970
Daily	401	780	750
Weekly	142	630	363

In this line, the Vates playroom is the worst case in memory usage due to its large size and number of *records*, but for the same reason, it represents an opportunity to see the profits of the optimization strategy of sharing *records* between the rooms. Table 3 reflects the results of this approach. It shows that the new variant that shares structures using the references catalogue improves the rooms' loading stage, using considerably less memory. On the other hand, for the hourly scenario, with 100 trajectories, the principal disadvantage of this strategy can be seen in the last columns of Table 4, where the computational cost is now transferred to the simulation, with the procedure `ChangeReferences` implemented with the a "Reference Catalogue", native system class mainly used for the playroom's load.

Table 4. Elapsed time (ms) of `Simular`, `Simular'` and `ChangeReferences` with 100 trajectories for Hourly, Daily and Weekly playroom, using the reference catalogue.

Playroom	`Simular`	`Simular'`	`ChangeReferences`
Hourly	7875	274406	261711
Daily	43996	180896	128596
Weekly	14929	36312	19856

Finally, the last version of the refactored SimSEE is evaluated. Specifically, Table 5 presents the results of applying the structure share techniques between rooms in terms of memory usage.

Table 5. Peak memory usage (MB) for the different strategies simulating 100 trajectories of hourly, daily and weekly rooms.

	Original	Full independent	Reference Catalogue	Reference Dictionary
Hourly	500	9560	970	1068
Daily	401	780	750	760
Weekly	142	630	363	407

The first observation is that, although the Reference Catalogue technique employs less memory to store the references that need to be changed between rooms (2K references for VATES), they are solved every time in simulation when a room needs to make a computation. This resulted in poor performance, negatively affecting the simulation, as shown Table 4. On the other hand, the strategy that uses a matrix of pointers to store the already solved references requires little more memory since it employs as many rows and columns as trajectories and references, respectively, to store the correct pointers. Moreover, it significantly reduces the overhead introduced to the simulation by the previous technique.

When it is not the worst case (for example, Daily Room) and the number of referenced *records* contained in the room is not so large, the efficiency of loading the rooms with this implementation of shared *records* is not as noticeable as the

Hourly room, but severely impacts on the simulation time. Like the previous one, the weekly playroom contains few *records*, and the improvement is not so noticeable. Despite this, both daily and weekly playrooms present very good performance loading rooms.

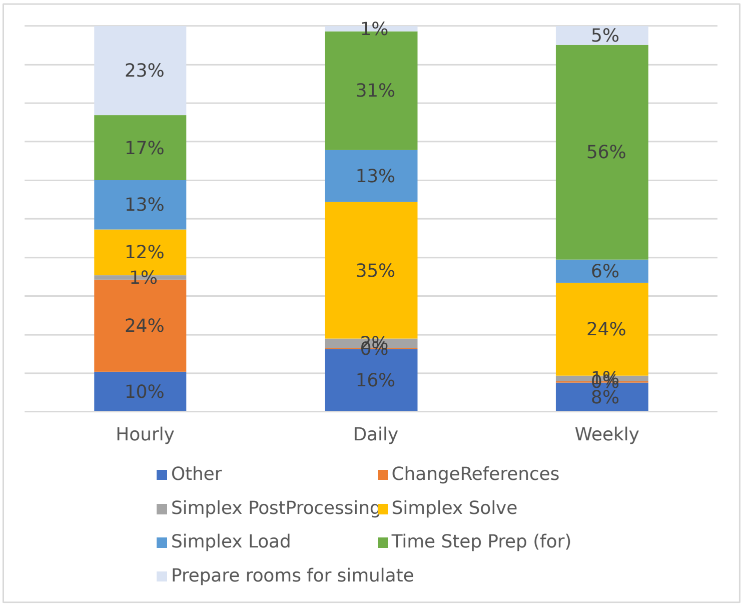

Fig. 3. Stages percents associated with the simulation of 100 trajectories sharing references with a dictionary, for hourly, daily and weekly playrooms.

Figure 3 shows the ratio between the simplex resolution time and the whole simulation time for the three evaluated scenarios. The first stage (Prepare rooms for simulation), which performs settings and initializations of the variables necessary for each trajectory during the simulation, consumes 23%, 1% or 5% of the total of the simulation for the hourly, daily, and weekly respectively. The second stage, as shown in Figure 3, is composed of “Time Step Preparation”, “ChangeReferences”, Loading, Solving and Post Processing Simplex, and “Others”. Those stages take the largest part of the time within the simulation. The final stage prints the results of all the simulations in an output file, which is almost negligible compared to the other stages from the computational point of view.

Carrying out an analysis of the simulation times with the different strategies, we can conclude that a large part of the difference between the times occurs in handling the references. This confirms that the last strategy to optimize simulation implementation, using a matrix of pointers instead of another complex structure as a Reference Catalogue, can lead to important savings.

Although this modification does not fully cover the time difference between the simulations, another important factor is that, in the first stage, certain variables are initialized, and memory is reserved for each room. These procedures have a heavy computational weight. Since in the initial version of the simulation, only one room is needed, these initializations are done once, whereas, in this version, they must be done as many times as rooms are needed. Line 3 in Algorithm 2 sets a lower bound to optimize the elapsed time of the first simulation stage. Considering this, the settings and initialization in the first stage and the ChangeReferences procedure in the second determine this implementation's overhead. Therefore, depending on the room, the overhead varies, allowing rooms with less overhead to be more efficient in the future.

In rooms where the time taken by solving the Simplexes is high, parallelizing the Simplexes solution in a massively parallel architecture (for example, a GPU) can save significantly more time than the overhead implied by our change in the simulation design, accelerating the original model.

Tables 6, 7 and 8 show, for each playroom (Hourly, Daily and Weekly), a comparison of the elapsed simulation times for the developed strategies.

Table 6. Simulation times (ms) for Hourly playroom (VATES) with different implementations.

# of Trajectories	Original	Refs. Catalogue	Refs. Dictionary
2	179	5432	295
4	337	11251	590
8	649	22749	1184
16	1308	43431	2401
32	2590	89255	4913
64	5131	175247	10066
100	7875	274406	15880
128	10280	351584	22119
256	20106	701891	45542

As seen in the previous tables, the implementation obtained after the different optimizations specified throughout this work is just above the sequential strategy, especially for rooms containing many *records*. Thus, as previously mentioned, the time it takes to solve the simplex in these rooms is constant in all the implementations, but unlike before, there is now an infrastructure that allows solving trajectories in parallel, creating the opportunity of solving many Simplexes in parallel using a GPU in the future.

5 Conclusion and future work

In this work, we have refactored a large legacy computational system to expose parallelism and create the opportunity of accelerating it using GPUs shortly,

Table 7. Simulation times (ms) for Daily playroom with different implementations.

# of Trajectories	Original	Refs. Catalogue	Refs. Dictionary
2	902	3567	933
4	1743	6934	1804
8	3574	13806	3673
16	7168	28041	7483
32	14138	56802	15209
64	28256	114822	31219
100	43996	180896	49113
128	56452	233024	63320
256	113594	480108	128958

Table 8. Simulation times (ms) for Weekly playroom with different implementations.

# of Trajectories	Original	Refs. Catalogue	Refs. Dictionary
2	312	714	315
4	616	1412	613
8	1205	2809	1212
16	2406	5585	2398
32	4794	11211	4847
64	9562	22780	9895
100	14929	36312	15657
128	19174	46688	20324
256	38091	94600	40868

extending our initial effort to introduce modern parallelism techniques on the SimSEE tool. Considering our previous results, we study the principal challenges and constraints to implementing the proposed massively-parallel version of the SimSEE.

Concretely, we successfully implemented and evaluated versions of the system that allow simulating multiple trajectories in parallel by instantiating as many playrooms as the number of trajectories. The evaluation involved different temporal scenarios, facing various problems for some of these cases. The principal difficulty was the memory scalability, i.e., the memory footprint related to the rooms' instantiation. Therefore, we propose multiple optimizations. In the first place, and common to all the strategies, we instantiate all the rooms by reading the playroom's file once, reducing unnecessary serialized I/O activity. The true optimizations focused on understanding playrooms' structures, identifying units that remain invariable through the different trajectories' room, to instantiate them once, for all rooms. We successfully implemented a version of the room's loading and simulation procedures, considerably more suitable for a massively-parallel implementation, with a scalable shared-memory variation, reducing 88% the memory footprint of the hourly case. The downside of this strategy is the introduction of certain overhead in the simulation routine. To reduce this overhead, we propose two different variations, one using a native structure implemented with classes and the other using a simple pointer matrix. We significantly reduced the overhead introduced by the first technique by sacrificing a very small percentage of memory. For example, in the hourly case, compared to the fist technique we improved the simulation time by a factor of 17×, by increasing memory usage by only 10%.

In future work, we intend to address the GPU parallelization of the Simplex solved for all the trajectories in each time step. Additionally, it is interesting to evaluate the parallelization of other stages of the simulation.

Acknowledgements The authors of this article were partially financed by project ANII FSE_1_2018_1_153060 Aceleración del SimSEE utilizando GPUs (SimSEE-MP).

References

1. ADME: VATES [Online] Available at http://latorre.adme.com.uy/vates/
2. ADME: Usos del SimSEE. https://www.simsee.org/simsee/usos.html, accessed: 2021-07-31
3. ADME: Administración del Mercado Eléctrico. https://adme.com.uy/ (2022), [Online; accessed 10-June-2022]
4. Baya, R., Porrini, C., Pedemonte, M., Ezzatti, P.: Task parallelism in the WRF model through computation offloading to many-core devices. In: Merelli, I., Liò, P., Kotenko, I.V. (eds.) 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018. pp. 596–600. IEEE Computer Society (2018). https://doi.org/10.1109/PDP2018.2018.00100, https://doi.org/10.1109/PDP2018.2018.00100
5. Bayá, R., Pedemonte, M., Gutiérrez Arce, A., Ezzatti, P.: An asynchronous computation architecture for enhancing the performance of the weather research and forecasting model. Concurrency and Computation: Practice and Experience **32**(19) (2020), www.scopus.com
6. Camacho, V., Chaer, R.: Hourly model of a combined cycle power plant for simsee. In: 2020 IEEE PES Transmission & Distribution Conference and Latin America (T&D LA). pp. 1–5. IEEE (2019)
7. Coppes, E., Tutté, C., Maciel, F., Forets, M., Cornalino, E., Chaer, R.: SimSEE Proyecto ANII FSE 2009 18 Mejoras a la plataforma SimSEE (2012), https://iie.fing.edu.uy/publicaciones/2012/CTMFCC12
8. CORDIS: REfactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach. https://cordis.europa.eu/project/id/644235 (2014), https://cordis.europa.eu/project/id/644235
9. Dennis, J., Loft, R.: Refactoring Scientific Applications for Massive Parallelism, pp. 539–556 (02 2011). https://doi.org/10.1007/978-3-642-11640-7˙16
10. Flieller, G., Chaer, R.: Introduction of ensemble based forecasts to the electricity dispatch simulator simsee. In: 2020 IEEE PES Transmission & Distribution Conference Latin America (T&D LA). pp. 1–6. IEEE (2019)

11. Igounet, P., Alfaro, P., Usera, G., Ezzatti, P.: GPU acceleration of the caffa3d.mb model. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) Computational Science and Its Applications - ICCSA 2012 - 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 7336, pp. 530–542. Springer (2012). https://doi.org/10.1007/978-3-642-31128-4"˙39, https://doi.org/10.1007/978-3-642-31128-4_39
12. Jensen, K., Wirth, N.: PASCAL user manual and report: ISO PASCAL standard. Springer Science & Business Media (2012)
13. Marichal, R., Vallejo, D., Dufrechou, E., Ezzatti, P.: Towards a massively-parallel version of the SimSEE. In: 2021 IEEE URUCON. IEEE (nov 2021). https://doi.org/10.1109/urucon53396.2021.9647142, https://doi.org/10.1109/URUCON53396.2021.9647142
14. Rutishauser, H., Gutknecht, M., Gautschi, W., Schwarz, H., Henrici, P., Läuchli, P.: Lectures on Numerical Mathematics. Birkhäuser Boston (1990)
15. Fichas Dinámicas, https://simsee.org/simsee/simsee/ayuda/fichas-parametros-dinamicos.htm
16. UTE: Wind Energy in Uruguay. https://portal.ute.com.uy/composicion-energetica-y-potencias (2022), [Online; accessed 18-June-2022]