Parallel ABM for Electricity Distribution Grids: a Case Study

Fanny Boulaire, Mark Utting and Robin Drogemuller

Queensland University of Technology
2 George Street
Brisbane, Queensland 4000
{Fanny.Boulaire, Mark.Utting, Robin.Drogemuller}@qut.edu.au

Abstract. This paper introduces a parallel implementation of an agent-based model applied to electricity distribution grids. A fine-grained shared memory parallel implementation is presented, detailing the way the agents are grouped and executed on a multi-threaded machine, as well as the way the model is built (in a composable manner) which is an aid to the parallelisation. Current results show a medium level speedup of 2.6, but improvements are expected by incorporating newer distributed or parallel ABM schedulers into this implementation. While domain-specific, this parallel algorithm can be applied to similarly structured ABMs (directed acyclic graphs).

Keywords: electricity distribution grid, network, composable ABM.

1 Introduction

Agent-based models (ABMs) have been used for diverse applications in many domains for their ability to capture the actions and interactions of the agents composing a complex system so that the overall behaviour of the system can be assessed [1]. This paper describes an ABM applied to the electricity domain. The aim of the project this work belongs to is to develop a planning tool for optimal investment strategy of electricity distribution networks over large areas and long planning horizons. A mix of renewable energy, energy storage and other new technologies are compared with traditional practices to assess their impact on the operation of the distribution network.

Traditionally, electricity has been generated at large centralised power stations and then distributed to consumers over a wired network. The introduction of renewable energy systems at various levels within the transmission/distribution system adds complexity to the system due to injection of electrical power in locations that were not initially designed to receive power, mismatches between time of availability and time of demand and reduced control over the phase of generated power. Ergon Energy [2], the industry partner in this project, provides electrical power to approximately 700,000 customers over an area exceeding 1,000,000 km^2. The Ergon network consists of approximately 150,000 km of power lines and 1,000,000 power poles, togeth-

er with associated substations and transformers, and has a total asset value of AU$10.6B. Ergon are using the software presented in this paper to support network infrastructure investment decisions, looking for the most appropriate combination of technologies and practices to reduce overall costs.

An Ergon user interacts with the software by selecting nodes within the network structure to be analysed against various scenarios, observing the impact these changes within the network might have on the overall system. A user may analyse a single network topology against a range of user demand predictions or, inversely model user demand predictions against a range of potential network configurations. Initially the system represented the infrastructure of the distribution grid, the attachment points to the transmission grid, solar and wind renewable energy sources as well as emergency diesel generation and the different types of consumers. During development it was recognised that battery storage and the demands of electric vehicles will be needed.

This tool is therefore being built to answer a wide range of questions, which can arise as time passes, taking into account both the technical and economic constraints of the system, and doing so in an integrated manner. Consequently, the tool was built in a composable manner to ensure its flexibility and extensibility. The wide range of physical asset types within the system, the wide range of potential configurations within each asset type, the need for communication of deep, complex issues between the software engineers and domain experts and a desire for flexibility into the future all indicated that an ABM approach had desirable characteristics. ABM was chosen as the modelling technique mainly because the evolution of the electricity grid is unknown due to the many new technologies being employed; in this case the past is no predictor of the future. Also, the ability to describe the agents at various scales, both over space and time, was important to capture the different system characteristics accurately and dynamically.

As the model grew, adding more agents of varied types, the simulation slowed down, leading to the need for parallelization. A parallel implementation of the agent scheduling was done to speed up the execution time and because the structure of the model could be broken down into independent tasks with known common patterns. This paper presents this parallel implementation using a fine-grained shared memory parallelism performed on a multi-core computer, detailing the technical implementation dependent on the overall tool architecture.

The first part of this paper describes the overall architecture of the tool, built in a composable manner, which defines the requirements of the implementation. Then, the description of the parallel implementation within that software framework is given and performances of the parallel implementation are discussed.

2 ABM planning tool architecture

This section briefly describes the overall architecture of the ABM planning tool to provide an understanding of the constraints on the agent-based model, which in turn influences the parallel execution of the agents.

2.1 Composition of the model using plugins

Because the tool was built with the need for evolution in mind, so that many questions could be answered and these in an integrated way, a composable approach was taken. For this, OSGI (formerly Open Services Gateway Initiative) [3], a specification that enables writing modular software, was chosen as the underlying technology for the software. As described by its community of users, it "*reduces complexity by providing a modular architecture for today's large-scale distributed systems as well as small, embedded applications*". This specification has been used by the Eclipse Community, whose plugins are OSGI bundles [4]. The tool presented in this paper is built using Eclipse plugins which are our unit of modularity.

The breakdown of the software in plugins is done over different layers. First, one main plugin is defined, called MODAM (MODular Agent Model), which can be seen as the core of the whole framework. It contains the schemas that define the different extensions, which are used by the different plugins to support interconnection [5]. It also contains the schedulers (sequential and parallel), and ensures the automatic creation of the objects and agents from the plugins using factories.

The different plugins contributing to the creation of the ABM are defined according to their logical meaning. A plugin can be defined for a given analysis type, for example the amount of electricity a solar panel can produce. Within each of these logical units further breakdowns can be performed, to contain different logical or functional units. So far, 3 types of functional units have been distinguished within a logical one. These can be seen through the different extensions that are defined in the MODAM plugin: asset factory, agent factory and data provider. Details about the software architecture can be found in [6]. Using solar panels as an example, the asset factory creates the solar panel assets with their physical characteristics (capacity, derate factor…), the agent factory the behaviour of the solar panel according to a photovoltaic (PV) output model dependent on the weather conditions, and the data provider reading the weather information at the location of the solar panel, being an input to the PV model.

A distinction has been made between assets and agents; this is further explained below.

2.2 Integration of assets and agents over separate layers

When looking at an implementation of an agent in an ABM, it is normally mainly defined through its behaviour in relation to its environment and other agents, using only the characteristics that are required to determine the agent's action. That is, the object's characteristics and behaviours are combined in one single class and only the necessary attributes are modelled. Also, the agents are often instantiated in a central class where the relationships amongst given agents are set, as well as their scheduling for the simulation. Such an approach can be sufficient when building a model that has well-defined boundaries. However, when planning on extending the functionality of a model, such an approach can be restraining, as reuse is limited because of this tight coupling and access to the code is required to extend the model.

For our ABM implementation, a distinction has been made between the characteristics and the actions of our agents. **Fig. 1** shows a schematic representation of the two distinct layers that compose our model implementation: the asset and the agent layer. In the context of this paper, an 'asset' is the object that contains the physical aspects, and an 'agent' holds only the behaviour. The two are however linked, and the behaviours can be informed using the asset's characteristics.

Fig. 1 shows that the different elements composing the simulation are coming from different factories that are in separate plugins. Often there is a one-to-one relationship between an asset and an agent, but it is not required. Indeed, as shown on the figure, an agent representing the behaviour of a single asset can be built combining multiple behaviours – this is the case for a premise agent, which is the combination of electricity consumption and demand-side management behaviours which can come from two distinct plugins and modified as more information becomes available to define the behaviour. It is not necessary to load both behaviours for all scenarios. This architecture also supports the loading of alternative behaviours, for example consumption, against a single implementation of demand-side management. Conversely, a single behaviour can be assigned to a group of assets, which is the case when doing a load flow analysis for example.

Finally, it can be seen that while many plugins have been used to implement the model, only one integrated network graph has been created and it runs as one single agent-based model – the one on the Agent Layer.

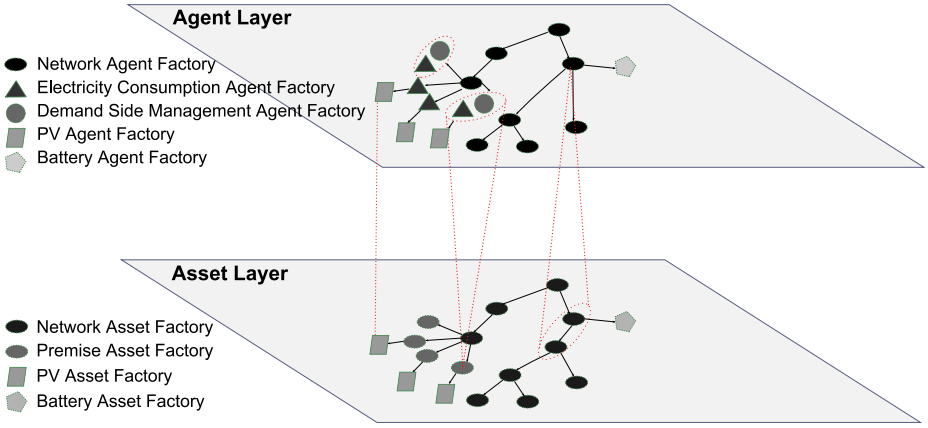

Fig. 1. Schematic representation of the ABM architecture.

3 Parallel ABM for the electricity distribution grid

Having defined the structure of the agent-based model implementation, this section describes its parallel implementation, which is a fine-grained parallelism on a single shared memory computer. The following sections describe the different aspects relating to the parallelization of the agent-based model execution:

- Ordering of the agents at simulation setup
- Parallel execution of the agents at runtime

It should be noted that the application presented in this paper is not a generic solution for parallel ABM; it is related to the specific case study presented here. However, because the nature of the problem is a graph structure, whose execution benefits from parallelisation, it can be applied to other applications of similar nature. Also, a distributed ABM could be built on top of this implementation, but this is outside of the scope of this paper.

3.1 Ordering of the agents at simulation setup

Because of the dependence of some agents on other agents that define their relationships and interactions, the ordering of the agents' execution is extremely important. Due to the way the agent-based model is built, this ordering needs to follow the architecture constraints. This means that two types of ordering needed to be considered: ordering within the same plugin, and ordering across plugins.

Because plugins can be used independently of one other, it was necessary to have agents ordered within the plugin in which they are defined. This ordering is handled within the code where the developer has access to the agents themselves through their relationships to one another. Because it is a graph structure, this was handled using '*Before'* and '*After'* attributes, to define a strict *partial order* over the agents. A partial order is a transitive relation that is irreflexive and antisymmetric, i.e. corresponds to a directed acyclic graph which is what is represented in our system.

The second type of ordering regards those agents that are dependent on other agents defined in another plugin. This cannot be defined in advance in either plugin, since it is possible for the plugins to be developed independently so neither one knows about the other. This type of ordering is a bit more sophisticated as it requires ordering agents from at least 2 sets, where within each of these sets, agents are also ordered. For this, partial ordering of the agents is again used. An example of this is given in **Fig. 2**. It shows 3 plugins, 2 of which can be run in parallel (Plugin A and B) with the third one requiring having its agents called after both of them. In each of them we have 3 agents; plugin A and C have their agents ordered sequentially, and plugin B has 2 agents that are ordered sequentially (b_1 and b_2) and one that can be called anytime (b_3). The ordering between plugin B and plugin C means that all agents in plugin B must finish their step methods before any agents in plugin C can start.

The ordering of the agents is set through an argument in the command line, using "-order" and a combination of the plugins. In the example given in **Fig. 2**, the ordering argument looks like:

-order = (*td*(PluginA) || *bu*(PluginB)); PluginC

Where *bu* stands for bottom up, and *td* for top down ordering; || shows that PluginA and PluginB are to be ordered in parallel, and the semi-colon (;) is to show sequential ordering.

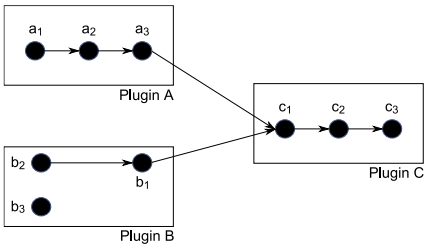

Fig. 2. – Example of ordering of agents within plugins and amongst plugins.

3.2 Parallel execution of the agents at runtime

Given the nature of the problem, each agent in the graph has to wait for one or more agents to be executed before it can proceed with its own action. A barrier to efficient and speedy processing is therefore a blocking condition where an agent is waiting on others' execution. Indeed, early versions of our parallel ABM treated each agent as a separate task, but profiling showed that this resulted in too much overhead synchronising between hundreds of thousands of small tasks.

So, in order to speed up the execution of the ABM, the partial order from the previous section is broken down into many small independent groups on which the calculations are performed. The size of these groups can be varied by the user using the parameter *–ThreadGroupSize* on the command line. Then these groups of agents can be executed in parallel on *n* threads, specified by the Java command line argument *–Threads*. Details of these two phases are as follows:

1. **Setting up of the queue** - Grouping is done at the beginning of a simulation, and whenever new agents are created, by traversing the partial order, using a depth-first search. These groups are defined and queued in a linear order, with each group recording any previous groups that it depends upon. Mathematically, the groups form equivalence classes over the agents and the partial order over the groups is derived from, or is an abstraction of, the partial order over the agents. These group sizes are less than or equal to the number specified by *ThreadGroupSize*, depending on the configuration of the graph – e.g. when an agent has multiple parents.
2. **Execution of the agents –** Execution is done repeatedly for each time step. Within each time step, it is also repeated until the whole graph has been executed.
   (a) Depending on the number of threads (*n*), the *n* first available groups of agents, or task, in the pool are executed in parallel. The calculations are performed within each of these groups and assigned to the top node that can be used in the next task.
   (b) As soon as a thread becomes free, it accesses the next available task in the pool and checks its dependencies to see if they have finished processing. If they have, the step method of the group of agents is executed.

Fig. 3 illustrates this algorithm using a small extract of a network with two main branches – one starting with a feeder line L1 and another one with the feeder line L2,

that then join at a bus B4. Under that bus, 5 houses are represented, some of which have solar panels installed.

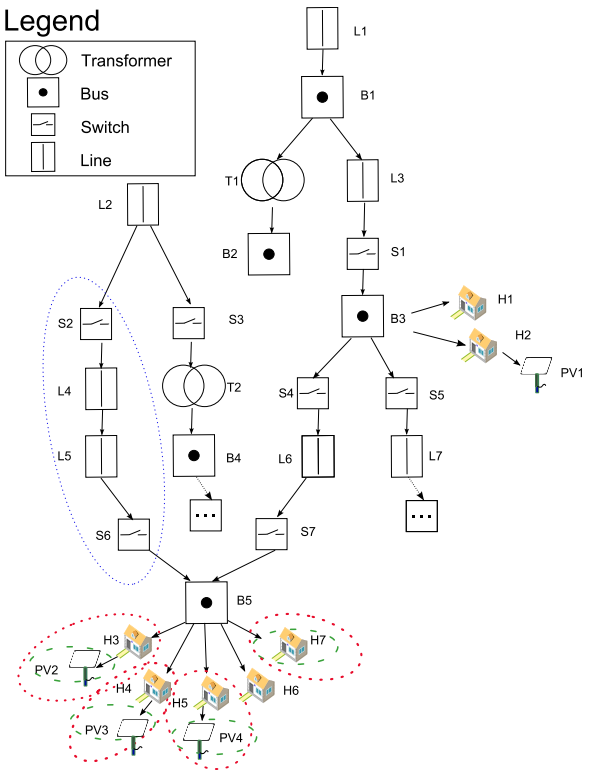

Fig. 3. Small extract of a distribution network, implemented with a parallel ABM.

Following the description above, each agent would have been ordered during the simulation setup in the module manager, indicating the order in which the agents need to be run. This ordering can be either top-down or bottom-up depending on the user's instructions.

From this ordering, the pool would have been created, first placing the agents at the leaves of the branches, i.e. those that do not have any dependency on other agents. From the example given in **Fig. 3**, with a *ThreadGroupSize* of 1, agents PV1, PV2, PV3, PV4, H1, H6 and H7 would be at the beginning of the queue followed by H2, H3, H4, H5, and so on. With a *ThreadGroupSize* of 4, we would have at the top of the pool, [PV2, H3], [PV3, H4], [PV4, H5], H6 and H7, followed by B5, and then [S6, L5, L4, S2] and [S7, L6, S4]. These groups contain fewer agents than 4 because they all impact on B5 which has multiple parents.

Following this, depending on the number of threads (n), the n agents are taken from the queue and their step method executed. As the agents' execution is finished, the thread is freed and the following task within the pool is then executed.

4 Results and Discussion

Following the parallelised implementation described above, simulations were performed on an i7-2600 CPU (4 cores + hyper threading) machine. The test case was for a medium-size town in Queensland, containing 75,910 assets of different natures (premises, lines, transformers…), and each asset had exactly one agent in this example. Parameter sweeps were performed and showed that using $n+1$ threads (where n is number of cores) and *ThreadGroupSize*=1000 would give the fastest runs. Using group sizes of 1000, **Fig. 4** shows the relative speed up obtained for numbers of threads varying from 1 to 10. The *ideal* (dashed) line shows the ideal performance if each of the available processors were fully utilised; from 4 to 8 cores, the slope is lower because hyper-threading typically gives less speedup than real hardware cores [7]. It can be seen here that a speed up of 2.6 could be attained when using 8 threads (green line). It can be noted on this graph that there are 3 additional series titled *start*, *step* and *stop*. These correspond to the three phases that have been distinguished within a step method of an agent execution; *step* corresponds to the calculations that are performed after obtaining the information (*start*) that is required to perform its decision and before it is sent (*stop*) to the next agent or stored in memory when requested by an agent. The *step* series shows the most improvement, which is the most important as it requires the most computing.

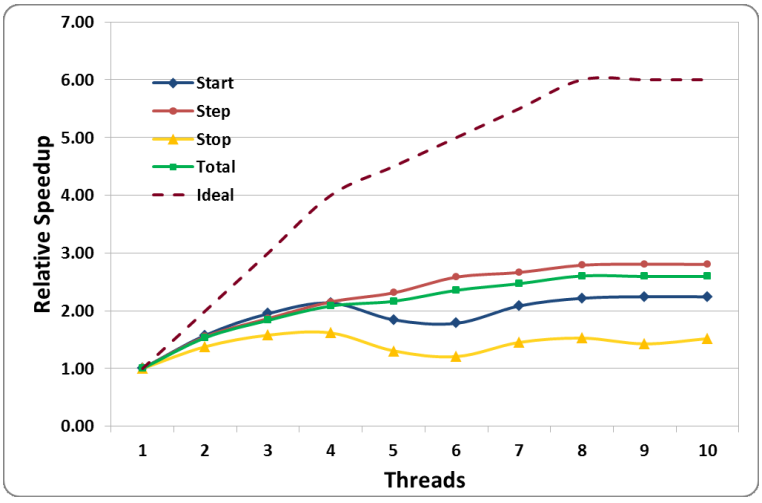

Fig. 4. Relative speedup of the parallel implementation of ABM model for the electricity grid.

The speedup obtained here is application-specific, as it depends on the relations between the agents and the granularity of their step methods. Our current scheduling algorithm, which is an NP-hard scheduling problem, gives medium speedup levels. It is however an improvement from other tested algorithms. There is on-going research to further improve the speedup.

An important point is the deterministic execution of the code. The outputs of the simulation are identical whether the simulation is run in parallel or sequentially which is an important feature. In some other implementations of parallel ABMs, determinism is not required, and even sometimes not desirable. In this case determinism is essential because of the impact of order of agent execution, which can be defined as a directed acyclic graph.

The i7-2600 CPU with 4 cores and hyper threading is being used to test the parallel algorithms due to the ability to constrain the entire execution environment. The full system is run on an SGI Altix XE Cluster with 128 nodes and 1924 x 64bit Intel Xeon Cores with a maximum 101.8 TeraFlops using single precision mode and 15,264 GigaBytes (GB) of main memory. Unfortunately, exclusive access to the cluster for experimentation purposes is not possible.

5 Related Work

Many applications are available to support the development of agent-based models ([8]; [9]). More recently, some of them have been further developed to support parallel and distributed ABMs, as the need for faster execution times and support of larger and larger models has arisen. This is the case for D-MASON [10], which is the parallel and distributed version of MASON, or Repast-HPC (High Performance Computing) [11] for Repast Simphony. Other platforms such as JADE [12] which is a middleware for distributed multi-agent application based on peer-to-peer communication architecture are also available. Most research on parallelizing ABMs has focussed on distributed computers, but there has been some research on multicore implementations, such as the study of spatial interactions by Gong et al. [13]. They achieve high speedups from 1-32 cores, but do not ensure determinism, so all their agents can be executed in parallel (with a small amount of locking required to ensure data integrity), which is essentially an embarrassingly parallel scenario. In contrast, we preserve determinism and our applications typically have complex and deep dependency graphs between agents that constrain the possible parallelism.

MASON was initially selected as the ABM engine for our software platform for its ease in separating the engine (which we were interested in) from the rest of the platform as well as for its execution speed. However, it was later replaced by our own implementation of a scheduler in Java. At the time the tool was being developed, D-MASON was not available and its availability has only recently come to the knowledge of the authors of this paper. Consequently, the work presented here has not been tried using D-MASON. D-MASON implements distributed ABMs rather than the shared memory parallelism discussed herein. This fine-grained parallelism is designed to speed up the simulation of networks of closely connected agents with frequent communication, while distributed ABM approaches are more suited for loosely connected sets of agents with less communication. Distributed ABM is complementary to the methods presented here. A combined approach will be tested in the future.

6 Conclusion

A parallel implementation of an agent-based model for planning the electricity grid has been presented. This implementation, which is application-specific, was based on the structure of the agent-based model, reduced to a directed acyclic graph. Agents were ordered, and then grouped respecting the ordering, to be executed in parallel over different threads. Results showed a relative speedup of 2.6, using an i7-2600 CPU (4 cores + hyper threading) machine, which further research will aim at improving. While this fine-grained shared memory parallelism is specific to the structure of this ABM, it can be applied to other ABMs with similar agent structures.

This application was built using the MODAM framework [6] which aims to answer diverse questions applied to the electricity grid. This software framework, which supports flexible and extensible models, was presented to provide an understanding of the structure of the model. This framework has two schedulers (sequential and parallel) which have been implemented in-house. The use of other schedulers from distributed and parallel ABM tools, such as D-MASON and Repast-HPC, will be investigated in future research to see if the speed of the simulations can be further improved.

7 References

1. North, M.J., Macal, C.M.: Managing Business Complexity. Oxford University Press (2007)
2. http://www.ergon.com.au/about-us/company-information/corporate-profile, accessed on 02/06/2013
3. http://www.osgi.org/Main/HomePage, accessed on 01/02/2013
4. http://www.vogella.com/articles/OSGi/article.html, accessed on 24/09/2012
5. EclipseZone, http://www.eclipsezone.com/eclipse/forums/t97608.rhtml, accessed on 13/02/213
6. Boulaire, F.A., Utting, M., Drogemuller, R.: MODAM: A MODular Agent-based Modelling Framework. International Conference on Software Engineering (ICSE), San Fransisco, CA, USA (2013)
7. Intel Corporation, http://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application/, accessed on 26/07/2013
8. Berryman, M.: Review of Software Platforms for Agent Based Models. In: Department of Defense, D.S.T.O. (ed.), (2008)
9. Jackson, S.K., Railsback, S.F., Lytinen, S.L.: Agent-based Simulation Platforms: Review and Development Recommendations. Simulation 82, 609-623 (2006)
10. Cordasco, G., Chiara, R.D., Raia, F., Scarano, V., Spagnuolo, C., Vicidomini, L.: Designing computational steering facilities for distributed agent based simulations. Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation, pp. 385-390. ACM, Montreal, Quebec, Canada (2013)
11. Collier, N.: Repast HPC Manual. (2012)
12. F. Bellifemine, G. Caire, A. Poggi, Rimassa, G.: JADE - A White Paper. exp in search of innovation 3, (2003)
13. Gong, Z.Y., Tang, W.W., Bennett, D.A., Thill, J.C.: Parallel agent-based simulation of individual-level spatial interactions within a multicore computing environment. International Journal of Geographical Information Science 27, 1152-1170 (2013)