Food and Ecological Systems Modelling Journal 5: e117593 (2024)
DOI: https://doi.org/10.3897/fmj.5.117593

Methods

Managing large and complex population operations with agent-based models: The ALMaSS Population_Manager

Christopher John Topping[1], Xiaodong Duan[1]

1 *Social-Ecological Systems Simulation Centre, Department of Ecoscience, Aarhus University, Aarhus, Denmark*
Corresponding author: Christopher John Topping (cjt@ecos.au.dk)

Abstract

We describe the method used to manage large and complicated populations of autonomous agent-based animal models in the Animal Landscape and Man Simulation System, as implemented in the ALMaSS Population_Manager class. Using three examples, we show how this approach facilitates the representation of populations in a mixed serial and parallel computer model, contributes to the efficiency of code operation and allows detailed behavioural representations to be modelled.

Key words: ALMaSS, Agent-based model, multi-threaded software, scheduling, simulation

OPEN ACCESS

Academic editor: Matthias Filter
Received: 19 December 2023
Accepted: 13 February 2024
Published: 6 March 2024

Citation: Topping CJ, Duan X (2024) Managing large and complex population operations with agent-based models: The ALMaSS Population_Manager. Food and Ecological Systems Modelling Journal 5: e117593. https://doi.org/10.3897/fmj.5.117593

Introduction

The Animal Landscape and Man Simulation System (Topping et al. 2003) has been continuously developed since the late 1990s (Topping 2022). At its core are two main components: the landscape simulation and the population managers. These components have never been fully presented despite being essentially unchanged in core design since the early ALMaSS versions until now. This article provides a description and some indications of the functionality of the basic approach to managing populations and an updated description of the model.

The role of the population manager in ALMaSS is to hold and manipulate lists of entities (typically agents) that are the focus of a particular model. It is responsible for scheduling actions by agents or other population objects and reporting the population status.

ALMaSS is written in C++ and uses a solid object-oriented approach in its design, which is used to tailor instances of the population manager to specific purposes, such as a skylark population manager or a ladybird population manager. The concept used is that all approaches common to all population managers are defined in the base class Population_Manager and those that are specific to a particular case are defined in descendant classes that inherit the base class code. Inheritance is a pillar of object-oriented programming. The Population_Manager class structure also uses polymorphism, whereby inherited code can be altered in descendent classes. For example, the DoFirst method described below is re-implemented differently for every population manager class, allowing each specific case to exhibit its own behaviour. The use of the

Population_Manager class and its descendants is an example of abstraction and encapsulation. As such, the details of the workings of the class are kept within itself, hiding this and the associated data from other parts of the code. This design means that the Population_Manager class is a fully functional entity with an interface to the rest of the ALMaSS simulation.

Interestingly, although potentially highly influential on the outcome, the management of populations of agents in population models seems to have received scant attention in literature. Weimer et al. (2019) note that in agent-based models, a random sequence is the norm, but without explanation and that books and introductory texts either ignore the issue of scheduling entirely (e.g. Bonabeau (2002); Macal (2016)) or apply a random scheduling (e.g. North and Macal (2007)). Older models sometimes also used a fixed schedule order (for example, DeAngelis et al. (1980)).

Some model systems do provide methods for altering scheduling, such as Repast Symphony (North et al. 2013), which includes a Scheduler class to manage lists of agents. Still, the consequences of this are rarely considered. The scheduling methods used affect important characteristics, such as the simulation's efficiency, the algorithm's flexibility and adaptability and even the outcome of the simulation itself. For ALMaSS, these are described here. New types of multi-agent systems are also emerging, such as Microsoft's AutoGen[1] and crewAI[2] for managing AI agents. However, here, the number of agents is low and the focus is on specifying the interface between agents rather than managing large numbers.

As ALMaSS is a framework for modelling the base Population_Manager classes, it must be constructed to allow maximum expression in the specific models that are derived from it. This expression was attempted using a standard structure that can be adapted using an object-oriented approach in descendent classes. To exemplify this, we provide a minimalist ALMaSS species model to explore the consequences of the population management methods used.

Methods

The primary code entity described here is the Population_Manager class and its design. The Population_Manager is the main controlling structure for modelling the populations of agents within ALMaSS. It is presented below, followed by examples demonstrating the utility of the design implemented in ALMaSS.

Population_Manager derived classes

The Population_Manager class is itself derived from the Population_Manager_Base. The latter contains all the information needed by the agent-based and any other potential type of population manager. The Population_Manager class inherits all aspects of the base class; thus, the functionality is described below as belonging to the descendent class, but where noted below, it is implemented in this base class.

The Population_Manager is never actually used to instantiate an object; instead, it is always used to create a derived class object, for example, Skylark_Population_Manger, which is then implemented for use. The derived

1 https://microsoft.github.io/autogen/blog/2023/12/01/AutoGenStudio/

2 https://github.com/joaomdmoura/crewAI

class inherits all the Population_Manager functionality. It adds model-specific information, such as the number and type of life stages and all necessary input/output to manage the simulations for the species under consideration. This added functionality includes any species-specific output formats used for this species only. This class structure allows further differentiation for future and more specific models. For example, the class hierarchy for beetles currently comprises three levels (Fig. 1), with all common beetle input/output (e.g. an array for calculating daily day-degrees for egg development) residing in a Beetle_Population_Manager and specific information in descendent classes (e.g. ladybird specific reproductive management). Each derived class may include a large degree of specialised functionality; however, this article only covers the use of the generic capabilities of the base Population_Manager class.

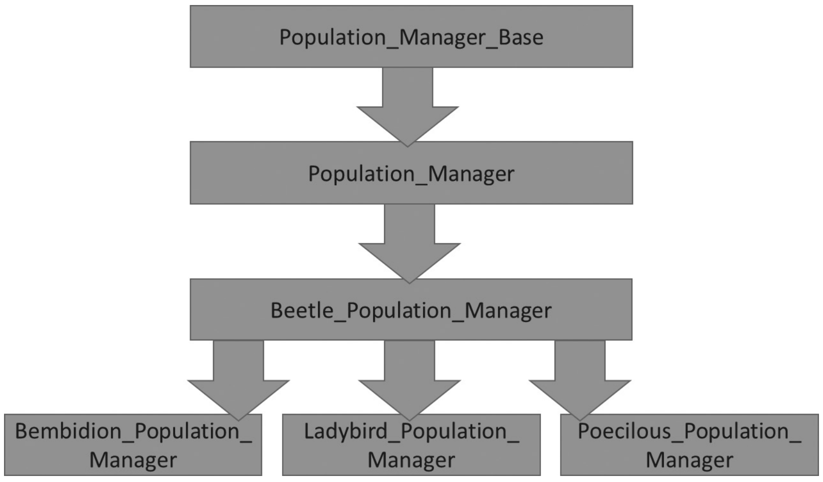
Population_Manager_Base
Population_Manager
Beetle_Population_Manager
Bembidion_Population_Manager
Ladybird_Population_Manager
Poecilous_Population_Manager

Figure 1. The current class hierarchy for beetle population managers, starting with the parent class Population_Manager_Base.

Population manager design

The Population_Manager class structure

Class data structures

The core of this class is a list of all individuals present in the population, which is the most important data structure used. In the original ALMaSS design, the population manager held all individuals in agent-based models in a data structure with a variable number of entries, one for each life stage represented as a separate class in the model. This means that this data structure must be traversed serially in scheduling and other operations and, for simulations with many organisms, this is inefficient. However, a relatively easy option exists for managing these lists in parallel with modern CPUs and code libraries. This option is now implemented in the current version of the population manager code.

Instead of having a single list for each object type, the list of objects is now created as a set of lists, one for each thread. These are denoted as 'TheSubArrays' within the program code. 'TheSubArrays' is a vector of vectors of forward_list. Vectors are arrays that can change in size during run time and are randomly accessed, thereby allowing the number of lists to be altered at run time to match the number of threads. A forward_list is a C++ container for a singly-linked list (each object points to the next), which is an efficient way to manage a list of objects that can be added to by pushing objects on to the end of the list. These two flexible structures mean that the size of the population can be determined at run-time and can be easily manipulated by descendant classes.

The first dimension of 'TheSubArrays' represents the types of population entities managed by the population manager, typically the life stages of the modelled organism. This approach allows descendant classes to change the number of life stages modelled by changing the first dimension. The second dimension of 'TheSubArrays' is the thread number, enabling parallel running when more than one thread is used. If only a single thread is used, the simulation is run sequentially. Thread-based parallel computing was implemented using OpenMP (Chandra 2001). The third dimension of 'TheSubArrays' is the list of individuals at each life stage manipulated by each thread. The length of this list will change with time during the simulation as individuals die or are born.

Scheduling

The key feature of Population_Manager is the method used to schedule and manage simulated individuals' behaviour. The top-level ALMaSS simulation loop is called from the main scheduling loop, which can be set to loop as many times as necessary to simulate the desired length of the time step (usually one day or 10 minutes in the models created to date). For each time step, the period is separated into 'pseudo time', which shares some components of sequential real time, but also allows complex time scheduling of activities, further detailed below. Before each time step can start, some specific fixed actions are needed. The first is to delete the objects of dead individuals from the previous time step and distribute the live individuals evenly to the threads, which results in the release of their allocated memories and optimises the use of the threads. The following action indicates that all individuals are ready to start the time step by a simple flag set for each individual. This flag is critical to the following scheduling process. The flag is called StepDone and is set to false for all objects in 'TheSubArrays'. Before any output is carried out, the final action is to carry out any required ordering of the lists in TheSubArrays. This reordering is typically to randomise the order of the lists, but can be set to order, based on characteristics such as location. Different orderings have different purposes and randomising the list is the best way to ensure that the order of individuals' actions does not affect the outcome, thus avoiding concurrency issues (Topping et al. 1999).

Following the initial set of actions, the main scheduling process is started. It consists of a higher-level sequence containing an iterative loop. The scheduling process includes three main methods: BeginStep, Step and EndStep, each preceded and followed by a customisable 'Do' method (Fig. 2). Parallel running occurs in all three methods for each life stage. However, these can be changed to run serially by setting the number of used threads to one. There is no parallel running across life stages because this is a design feature of the behavioural models (see the Skylark foraging example below). The four 'Do' methods are customisable by descendent classes and have no functionality in the Population_Manager classes. In descendent classes, they can be used for manipulating the lists, the environment or input-output as needed. In the example below ('*Theoretical1*'), only the DoFirst method is used. However, more complex models may require implementing all four methods.

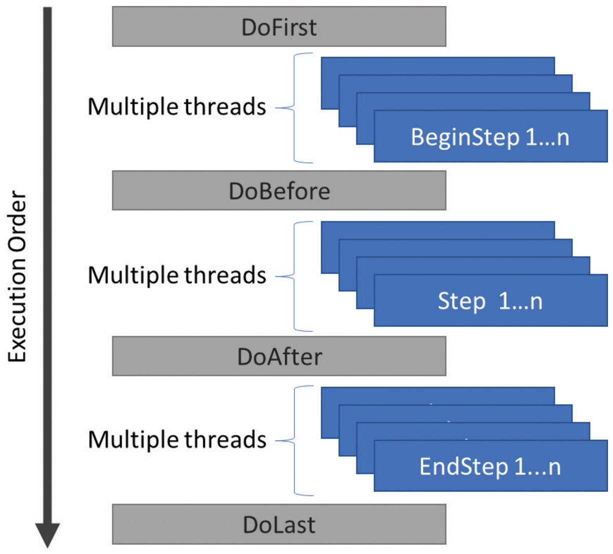

Figure 2. The time step processes. The three parts of the time step (BeginStep, Step, EndStep) process can run in multithreaded mode for each object 1 to n, extant at that time and are separated by customisable methods for reporting or list management by the Population_Manager class.

'BeginStep' and 'EndStep' have similar structures. Each iterates through all objects in parallel, depending on the number of threads and calls their 'BeginStep' or 'EndStep' methods, respectively. This process allows one type of behaviour or sets of behaviour to be selected at the beginning and end of a time step. 'Step' differs in that for this process, each object is called once following the order of objects in 'TheSubArrays'; it is also performed in parallel, depending on the number of threads used. This procedure is repeated for each object that did not set the flag 'StepDone' to true. Thus, all objects that did not signal that they finished the 'Step' process are called in the order of objects in 'TheSubArrays'. The Step completes when all objects signal that 'StepDone' is equal to true. The current implementation of the step process requires only one iterative sub-process, which is assigned to Step, a pragmatic decision, based on need. To date, there have been no situations where 'BeginStep' and 'EndStep' needed to be iterative.

The emergent behaviour from the overall time step processes is that behaviour can be scheduled for different life stages differently, such that complex interactions between life stages can be simulated. It also implements a state machine for the 'Step' process, allowing highly flexible behaviour sets to be simulated. A final touch to the iterative process is that an individual object can force another to return to the Step code by altering its 'StepDone' flag. This "event" facility is particularly useful for closely-connected objects, such as pairs or family groups.

Combining scheduling and parallel processing

Several agents in the same life stage run simultaneously when the individuals are looped inside the three-step methods; this is the critical procedure to accelerate the simulation running. However, this can cause problems when more than one individual accesses the same resources. The issue here is the simultaneous use of a resource by multiple agents. In this case, both may assume that they obtain it. These situations would result in serious program bugs and be challenging to track. To avoid this, we introduce a "guard map" for the landscape by segmenting the landscape into non-overlapping grids. Each grid has a “guard flag” to ensure that only one individual can perform location-related activities in a single grid. Before an individual tries to perform an activity in a specific location, it must possess the “guard flag” for that location. This “guard flag” stops other agents from carrying out behaviour here and, once the behaviour is completed, the flag is released. If the behaviour is triggered from the ‘BeginStep’ or ‘EndStep’ functions, the agent will simply wait and try the action again later in that time-step section when the flag is released. An enforced wait is needed because only one loop is executed in the ‘BeginStep’ and ‘EndStep’ functions. In the ‘Step’ function, waiting is unnecessary *per se* since it can be skipped automatically and the next individual will be called. Since the ‘Step’ function is iteratively called until all agents report ‘StepDone’ is true, the agent will automatically wait and try again in the next loop. Loops will continue until all individuals’ ‘StepDone’ flags are “true” in the ‘Step’ function, allowing all agents to complete their actions and release the guards.

Results

Population manager utility

Before-step actions

This example illustrates the use of before-step actions. These can do nothing, randomise the list or sort based on the x- or y-coordinate (this last option is for single-threaded execution only). To demonstrate the necessity of this functionality and the need for care in selecting the order of execution, we created a very simple ALMaSS individual-based model, ‘*Theoretical1*’, which represents animals competing for a limited resource. The model represents competition for food, which is provided to match the needs, based on the number of individuals (each gets 1.0 units). However, they will forage randomly and get 0.5–1.6 food units daily. Food is translated directly to growth. Each day, each individual selects another at random. If it is 20% larger than the other individual, it can eat that individual and grow 0.5% of the eaten individual’s size. Fig. 3 shows the results of running this scenario for 36 months with BeforeStepActions set to do nothing or to randomise the execution order. With ‘do nothing’ selected as the before-step action, the fixed ordering of execution provides a significant advantage to the first animals to feed since they can grow faster and eventually eat the others. This results in a higher maximum size, a lower minimum size and a declining population size. However, with ‘randomise’ selected as the before-step action, the randomised order of execution is enough to ensure that no individual gains sufficient advantage to eat another. Neither one version nor the other is necessarily right; this depends on what the model should represent, but this example shows the importance of considering the order of execution in a model. Outcomes can be considerably altered by choosing the wrong representation.

Skylark foraging

Here, we demonstrate the utility of the step design using the skylark model (Topping and Odderskaer 2004; Topping et al. 2013). This example also serves to explain the use of ‘pseudo-time-steps’. The problem of the skylark relates

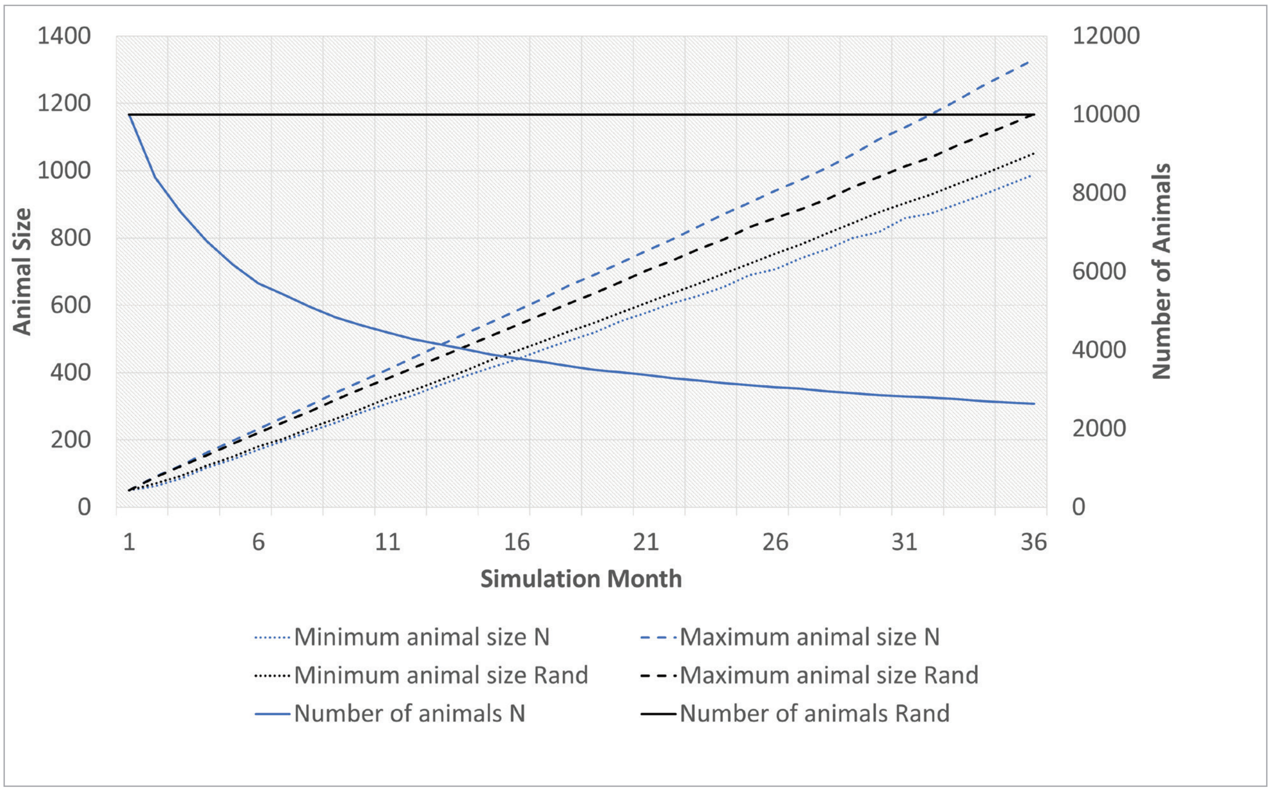

Figure 3. Change in the maximum and minimum sizes and population numbers for two scenarios using the Theoretical1 species, N = do nothing, Rand = randomise the execution order.

to foraging and the timing of activities, including multiple life stages and their interactions. Skylarks are territorial and their food is foraged from the area surrounding the territory. However, the amount of forage (insects) changes daily depending on the vegetation's state and the weather. For a time-step, the events that are needed to be scheduled are:

a. Assess the food availability around the territory
b. Male foraging
c. Male energetic costs filled
d. Male feeding chicks
e. Female foraging
f. Female energetic costs filled
g. Female feeding chicks
h. Chick growth

In the real world, all these types of behaviour are integrated over the day as a series of ongoing activities. Since the time-step of the model is one day, this integration is impossible. Still, it is possible to replicate the overall time-step behaviour using the BeginStep, Step, EndStep algorithm.

To manage the behaviour, the first action is to order the processes logically so that no process depends on another that follows it. In this case, the order can be defined as a – h above. Thus, the male's first activity is to determine the forage amount available (which is also communicated to the female). He then collects the forage, removes his energetic needs and finally allots portions to each chick. All of this behaviour is carried out in the BeginStep. In the Step, the female uses the information gathered by the male to determine the forage available, subtracts her needs and allots food to the chicks. In the EndStep, the chicks use the information on allotted forage and then determine their daily growth. As noted, these activities are integrated over the day in the real world. Using this method and viewed outside of the time-step, this integration is also the case. However, the separation of the time-step into 'pseudo-time' allows this integration to be carried out computationally efficiently. The alternative would be complicated and require multiple iterations through the life stages as each carries out part of the time-step, thus incurring a higher overall computational cost.

The multithreading running

For the third example, we aim to illustrate the impact of varying thread counts on the execution time. The '*Theoretical2*' example involves the creation of 10,000 animal objects. For each time-step, the population experiences 1040 new-born animals with 1000 dead animals, resulting in a gradually expanding population. Each animal is composed of a data structure containing 1,000 real number values. Within the Step function, 10,000 calculations, each squaring a double value, are performed for each animal and no other functionalities are associated with '*Theoretical2*' animals. All the simulations were run using the same ten by ten km landscape and each run had a duration of 5 years. The ALMaSS simulations require the landscape, but, in this simple example, the landscape does not affect the difference in the run time since Theoetical2 does not interact with it. The run time depends entirely on the number of threads and the population size. The run time for different thread counts is presented in Table 1. Employing multithreading yields a notable reduction in run time, with an improvement of 63% when using five threads. Leveraging more threads further accelerates the processing speed, although the rate of improvement gradually diminishes. A computing node with 2x Intel® Xeon® Gold 6130 CPU (X86-64) was used to run the simulations in Table 1. Each CPU has 16 physical cores, supporting 32 threads with Hyperthreading. The base frequency is 2.10GHz. The number of used threads was set by OMP_NUM_THREADS using OpenMP. The ALMaSS code was compiled using GCC12 using C++17 standard on Ubuntu 22.04.

Discussion

Since its release, ALMaSS and the Population_Manager class have been used to simulate a wide range of species ranging from highly numerous beetles, where simulations are recorded with > 50 million concurrent agents. Species modelled include beetles, spiders, newts, skylarks, hares, roe deer with highly detailed 10-minute time-step behaviour (Jepsen and Topping 2004) and even people (Williams et al. 2018). In the roe deer model, the time-step facility was used to allow communication between mother and young and simulate bouts of lactation. One

Table 1. Running time with different thread numbers. Each CPU core supports two threads.

Number of Threads	1	5	10	20	40
Running time in second	443	164	129	112	103
Relative to running time with one thread	1	0.37	0.29	0.25	0.23

of the latest models under development combines complexity and numbers to simulate every individual in a honey-bee colony (Duan et al. 2022). The single iterative Step function has been enough to model all the time-step complexity needed for all these species. However, there is no technical reason that both the BeginStep and EndStep could not be managed in the same way if needed, permitting even more detailed and interactive behaviour to be modelled for future species.

'*Theoretical1*' is a minimal implementation of an ALMaSS agent-based model. This publication provides the code to show how this model was implemented and provides the basis for developing new models (Appendix 1). Although ALMaSS is designed to generate realistic species representations, it can also be used to create 'toy models' such as the one used here. Although these models are usually used to understand the general properties of systems, they can also be used to develop tests of code or behaviour in isolation from other complicating factors. Here, in effect, we used the model to test the consequences of execution order in the absence of other dynamics. Although this is known to affect cellular automata models (Fatès 2014) and agent-based models (Weimer et al. 2019), there appear to be very few published data on effects.

The results of the simple food scenario raise some interesting points. During testing, it was clear that the precise results depend on the size of the benefit of the rate of growth when eating another individual and, therefore, the chance of a single individual diverging enough from the rest of the individuals to be able to start to eat them. This model is somewhat similar to the seminal work on individual-based models modelling wide-mouth bass in a fish tank (Deangelis et al. 1980). This model was probably the first to demonstrate how individual variation could lead to positive feedback. However, as suggested previously (Topping et al. 1999), the precise outcome of this variation may depend on how the simulation is set up. Here, we showed that randomising the order of execution to prevent one individual from gaining an arbitrary advantage over the others can remove the positive feedback, resulting in no important individual variations. This result indicates that care needs to be taken in interpreting the application of an execution order when designing and building agent-based models. However, the precise effect achieved depends on the parameter values and strength of underlying dynamics. In this case, if the benefit from eating another animal were increased, it became difficult to remove the effect using a randomised order alone. Whether these effects are important in more realistic population simulations in landscapes is debatable. In this simple model, there is no spatial constraint on interactions, which will strongly reduce the observed effects whilst also causing feedback loops of their own, for example, interactions in local populations may ignore global dynamics. Despite this caveat, it is important to be aware of the potential for artefact creation if the execution order is not handled correctly.

ALMaSS run times are often very long and CPU development has been geared towards expanding the number of processing cores rather than continuing the increase in CPU speed seen in the 1990s to early 2000s. Hence, to increase efficiency in exploiting modern CPU architecture, ALMaSS needed to address the potential to use multiple cores. '*Theoretical2*' represents a minimalistic implementation of the ALMaSS agent-based model to demonstrate the potency of multithreading capabilities. The flexibility inherent in its multithreading implementation is noteworthy. By configuring the thread count to 1, the model transforms into a sequential computation equivalent to the original ALMaSS implementation. As the thread count increases, the speed enhancement increase is reduced. This phenomenon might be attributed to the synchronisation overheads associated with managing threads, which aligns with the complexity of the simulation process. This result also suggests that complex models could benefit more from multithreaded code than initially anticipated since these models are usually computationally heavier compared to the cost of managing multi-threads.

Factors that affect this change in efficiency include the guard-map size and step-code complexity. The guard-map resolution will determine the frequency of thread locking in space; hence, the finer this can be, the fewer locks are needed. However, the guard-map grid size cannot be too small since interactions may occur beyond it, especially in the case of broad area interactions (e.g. a density-dependent calculation working on an area larger than 1 m^2). Step code complexity will also affect the optimal thread number needed. In an extreme case where the step code is empty, it will result in a simple loop; in this case, the optimal thread number will always be one. Hence, we cannot yet provide precise guidelines regarding the best thread number to choose; it will depend on the simulation being run.

As noted above with the guard-map, when implementing multithreaded code in an ALMaSS model, some care is needed to ensure that key activities where individuals interact with the same resource cannot occur during parallel processing tasks, such as ladybirds eating aphids in the same location. We have created the guard-map to control access to spatially-related resources, which includes other agents. Functions are provided to claim a guard, based on the individual's location when it is doing location-related activities, for example, moving to a new location or eating resources from a location and release the guard when the individual finishes its' location-related activities. However, there is one case in which this method cannot protect the multithreaded code. The guard-map fails if an agent can kill another agent of the same type at the same location. For example, two adult ladybirds, one eating the other. The possibility of this kind of interaction needs to be considered in new models and multithreaded code should be disabled if this is the case.

Conclusions

The ALMaSS Population_Manager class has stood the test of time for more than 20 years and has not had a major update until now. The current implementation introduces the new multithreaded code feature for efficiency, which provides significant gains in processing time with minimal increase in memory footprint.

The scheduling of ALMaSS agents provides the flexibility to describe complex behaviour and interactions between agents. However, how the scheduling should be implemented for a specific model should be an active choice to avoid the potential for artificial bias in the outcomes.

Additional information

Conflict of interest

The authors have declared that no competing interests exist.

Ethical statement

No ethical statement was reported.

Funding

This work was partially supported by DeiC National HPC (UCloud, GenomeDK, LUMI) (g.a. DeiC-AU-N1-000025, DeiC-AU-L5-0011 and DeiC-AU-N2-2023015) and by the EcoStack project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement no. 773554.

Author contributions

Conceptualization: CJT. Formal analysis: XD. Methodology: CJT. Software: XD, CJT. Writing - original draft: CJT. Writing - review and editing: XD.

Author ORCIDs

Christopher John Topping https://orcid.org/0000-0003-0874-7603
Xiaodong Duan https://orcid.org/0000-0003-2345-4155

Data availability

All of the data that support the findings of this study are available in the main text or Supplementary Information.

References

Bonabeau E (2002) Agent-based modeling: methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences (PNAS) 99(Suppl 3): 7280–7287. https://doi.org/10.1073/pnas.082080899

Chandra R (2001) Parallel programming in OpenMP. Morgan kaufmann, 240 pp.

Deangelis DL, Cox DK, Coutant CC (1980) Cannibalism and Size Dispersal in Young-of-the-Year Largemouth Bass - Experiment and Model. Ecological Modelling 8: 133–148. https://doi.org/10.1016/0304-3800(80)90033-2

Duan X, Wallis D, Hatjina F, Simon-Delso N, Bruun Jensen A, Topping CJ (2022) ApisRAM Formal Model Description. EFSA Supporting Publications 19: 7184E. https://doi.org/10.2903/sp.efsa.2022.EN-7184

Fatès N (2013) A Guided Tour of Asynchronous Cellular Automata. In: Kari J, Kutrib M, Malcher A (Eds) Cellular Automata and Discrete Complex Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–30. https://doi.org/10.1007/978-3-642-40867-0_2

Jepsen JU, Topping CJ (2004) Modelling roe deer (*Capreolus capreolus*) in a gradient of forest fragmentation: behavioural plasticity and choice of cover. Canadian Journal of Zoology-Revue Canadienne De Zoologie 82: 1528–1541. https://doi.org/10.1139/z04-131

Macal CM (2016) Everything you need to know about agent-based modelling and simulation. Journal of Simulation 10: 144–156. https://doi.org/10.1057/jos.2016.7

North MJ, Collier NT, Ozik J, Tatara ER, Macal CM, Bragen M, Sydelko P (2013) Complex adaptive systems modeling with Repast Simphony. Complex Adaptive Systems Modeling 1: 3. https://doi.org/10.1186/2194-3206-1-3

North MJ, Macal CM (2007) Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780195172119.001.0001

Topping CJ (2022) The Animal Landscape and Man Simulation System (ALMaSS): a history, design, and philosophy. Research Ideas and Outcomes 8: e89919. https://doi.org/10.3897/rio.8.e89919

Topping C, Hansen T, Jensen T, Jepsen J, Nikolajsen F, Odderskaer P (2003) ALMaSS, an agent-based model for animals in temperate European landscapes. Ecological Modelling 167: 65–82. https://doi.org/10.1016/S0304-3800(03)00173-X

Topping C, Odderskaer P (2004) Modeling the influence of temporal and spatial factors on the assessment of impacts of pesticides on skylarks. Environmental toxicology and chemistry 23: 509–520. https://doi.org/10.1897/02-524a

Topping C, Rehder M, Mayoh B (1999) VIOLA: a new visual programming language designed for the rapid development of interacting agent systems. Acta biotheoretica 47: 129–140. https://doi.org/10.1023/A:1002070223107

Topping CJ, Odderskaer P, Kahlert J (2013) Modelling Skylarks (*Alauda arvensis*) to Predict Impacts of Changes in Land Management and Policy: Development and Testing of an Agent-Based Model. PLOS ONE 8(6): e65803. https://doi.org/10.1371/journal.pone.0065803

Weimer CW, Miller JO, Hill RR, Hodson DD (2019) Agent Scheduling in Opinion Dynamics: A Taxonomy and Comparison Using Generalized Models. Jasss-the Journal of Artificial Societies and Social Simulation 22(4): 5. https://doi.org/10.18564/jasss.4065

Williams JH, Topping CJ, Dalby L, Clausen KK, Madsen J (2018) Where to go goose hunting? Using pattern-oriented modeling to better understand human decision processes. Human Dimensions of Wildlife 23: 533–551. https://doi.org/10.1080/10871209.2018.1509249

Appendix 1

The source code and the running directory are publicly available at:

https://gitlab.com/ALMaSS/almass_methodology.git

See also Suppl. material 1 containing the Population_Manager code documentation.

Supplementary material 1

The code documentation for the ALMaSS Population_Manager class

Authors: Christopher John Topping, Xiaodong Duan

Data type: pdf

Explanation note: The doxygen generated documentation from the ALMaSS code for the Population_Manager class.

Link: https://doi.org/https://doi.org/10.3897/fmj.5.117593.suppl1