2012 IEEE 14th International Conference on High Performance Computing and Communications

Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework

Simon Coakley, Marian Gheorghe, Mike Holcombe
Department of Computer Science
University of Sheffield
Sheffield, UK
Email: s.coakley@sheffield.ac.uk

Shawn Chin, David Worth, Chris Greenough
Software Engineering Group
STFC Rutherford Appleton Laboratory
Didcot, UK
Email: shawn.chin@stfc.ac.uk

***Abstract*—This paper describes the design of an agent-based modeling framework for high performance computing. Rather than a collection of methods that require parallel programming expertise the framework presented allows modelers to concentrate on the model while the framework handles the efficient execution of simulations. The framework uses a state machine based representation of agents that allows a statically calculated optimal ordering of agent execution and parallel communication routines. Some experiments with the current implementation and the results of using a simple communication dominant model for benchmarking performance are reported. The model with half a million agents is used to show that a parallel efficiency of above 80% is achievable when distributed over 432 processors. Future improvements are discussed including data dependency analysis, vector operations over agents, and dynamic task scheduling.**

***Keywords*-agent-based modeling; high performance computing;**

I. Introduction

The Flexible Large-scale Agent Modeling Environment (FLAME), http://www.flame.ac.uk, is a template driven framework for agent-based modeling (ABM) on parallel architectures. There are many agent-based simulation frameworks available but their execution model does not permit efficient and distributed simulation which is becoming more important with large scale economic [1] and molecular biology agent models [2], where either the execution time or memory requirements outstrip single machine capabilities. This paper discusses the issues for creating large scale agent based frameworks focusing on the current FLAME framework, its limitations and ideas for its improvement.

II. Review of parallel agent-based modelling frameworks

Popular agent-based modelling implementations have used synchronous agent updates where updated agent values are available to other agents immediately. These frameworks use user defined agent update schedulers, either where agents are added and updated one by one (MASON [3], Repast [4] and Swarm [5]) or all agents are asked to update and this is achieved one by one (NetLogo [6]). This means they lose the inherent parallelism of asynchronous agent update strategies like that which are used for cellular automata based models.

A. Parallelism

Some of these popular frameworks have tried to retroactively introduce parallelism:

With MASON (which uses Java) it is possible to create parallel schedulers as threads to run on multi-core machines but this makes it possible for agents on different schedulers to access the same parts of a model simultaneously, and introduce unforeseen errors.

Repast (which uses Java) has an expert-focused HPC version [7] (which uses C++). The HPC version uses schedulers on each memory independent process. Each scheduler updates its local agents. Copies of agents and any updates of these copies from other processes can be scheduled. Management of scheduling and coordination of process data synchronisation is handled by the user. The HPC version provides some built-in functions for users to utilise but users still need to schedule and coordinate the parallel aspects themselves.

NetLogo (which uses Scala and Java) can only run models in serial. BehaviourSpace is an extension that allows multiple runs of a model to be run in parallel on a multi-core machine.

Swarm (which uses Objective C) does not have any parallel capabilities.

Other agent based frameworks which mention parallel capabilities include:

EcoLab [8] (which uses C++), uses serialisable agents that can be sent to processes as a copy when needed. This is similar to the model used by Repast HPC version.

Cougaar [9] (which uses Java) is an Agent Application platform and not an agent based modelling framework. Cougaar provides a built-in bulletin board for agents to subscribe to and receive any notifications and is aimed at real-time applications, not ABM simulations.

ABM++ [10] (which uses C++) is an ABM toolkit for developing models on distributed memory architectures. ABM++ provides an interface similar to the model used by

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.79

Repast HPC where objects can be serialised and moved between distributed compute nodes. A synchronisation method is provided, and both time-stepped and distributed discrete event time update mechanisms are provided for modelers to utilise.

One of the main reasons that it has been hard to retrofit parallelism to these popular ABM frameworks is that agents can directly access other agent values: MASON directly accesses other agents via a spacial or network class; NetLogo directly accesses other agents by creating *agentsets*; Repast directly accesses other agents via built in methods; and Swarm directly accesses other agents via the built-in getNeighbor method. Therefore agents cannot be easily partitioned over memory independent processes and the whole of agent memory has to be copied to each process that needs to access it like the solutions created by Repast HPC, EcoLab and ABM++.

Some work has been done to execute ABM on GPGPUs which have been cellular automata based [11], [12] or have been an extension of FLAME on the GPGPU [13]. All have shown exceptional speed increases but are very limited in the scope and complexity of the models they can run. This is due the smaller amount of memory available and the homogeneous nature of the agents used in models.

III. Overview of current FLAME framework

The current FLAME framework has been designed from the start to be a inherently parallel ABM framework which does not require the user to have any expertise in parallel computation.

FLAME has evolved based on the requirements of different projects starting from a position aware framework used for biological models to a position-agnostic framework driven by a static scheduler and message board library catering to economic models.

FLAME generates simulation code (in C) by parsing a description of a model from a marked up XML document and applying the data to code template files. The program that does the parsing is called *xparser* which also (statically) calculates the optimal execution order of agent functions. The resulting simulation code is compiled with agent functions written in C, supplied by the modeler, and the FLAME communication library, called Message Board, to produce the simulation program, see Figure 1. All parallel features are auto-generated by the framework and do not need any effort by modellers to utilise.

A. Agent Definition

Agents are defined based on the concept of Communicating Stream X-Machines [14] (CSXM); each agent is represented by a acyclic state machine that characterises the behaviour of the agent per iteration, see Figure 2.

Each state transition function has access to the internal memory of the agent, as well as input and output streams of information.

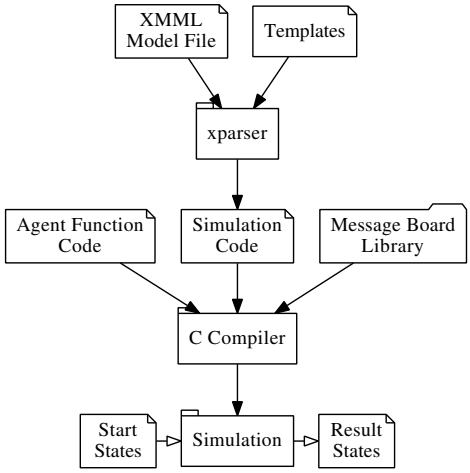

Figure 1. FLAME Templating Model

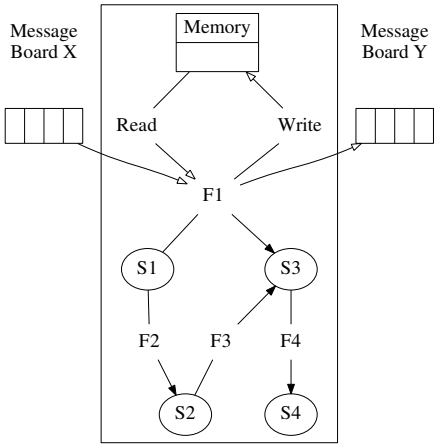

Figure 2. Agent described using the Communicating Stream X-Machines Model

To define an agent, modelers specify a set of state transition functions to transition an agent from one state to another. When linked together, these transition functions and their associated states form the acyclic state machine that represents the behaviour of the agent.

Agent instances are represented as a structure containing the internal memory of the agent. Agent transition functions take in this memory structure and update the values, effectively transitioning the agent instance to the next state ready to be consumed by the next function. The execution of a transition function is repeated for all agent instances of the associated type in the relevant state. Once all the functions have been called (in the correct order so as to meet dependencies) an iteration (time step) of the simulation is complete. To provide transition functions with access to the full memory structure, agent memory is stored as a list of structures.

A simulation is started by providing the initial state of each agent memory and once all agents have reached their final state the resultant agent memory for the iteration is written out, see Figure 1.

B. Agent Communication

In FLAME the input and output streams take the form of message boards. Each message board handles the messages of a single message type. Since message boards are the only means in which agents communicate with other agents, this makes the agent model inherently parallel. Each agent can be handled independently as long as the input message board contains the expected messages.

An example model can be seen in Figure 3 with two agent types (A and B) and two message types (X and Y) with their associated message boards. The first function of agent A sends messages of type X (writes to message board X) and the first function of agent B receives messages of type X (reads from message board X). A similar exchange happens with message board Y and associated agent functions.

C. Paralliseable Model Execution

The simulation can therefore be parallised by distributing agents across disparate processing nodes and synchronising the message boards to ensure that all agents see the same set of messages.

For efficiency, agents are not allowed to read and write to the same board from the same transition function. This avoids the need to synchronise the boards on every single write operation and the requirement to update agents one by one. Message boards can only be written to and then read from once per iteration step and they are wiped after each iteration so that each iteration is a self contained simulation run of the model.

The synchronisation of a board is initiated the moment all writes have been completed, in effect each agent function that outputs the type of message associated with the board has been executed. The framework achieves this by calling the *sync_start* function provided by the Message Board library. This function is non-blocking and the synchronisation process is performed in a background thread. The framework is then free to execute other functions that do not depend on the board in question. It is possible for multiple boards to be synchronised concurrently.

Before executing agent functions that read messages from a board the framework calls a *sync_complete* function provided by the Message Board library. The function checks the status of the synchronisation process, and returns immediately if the synchronisation is complete. However, if synchronisation is still in progress the function blocks until completion.

To restrict the amount of data synchronisation it is an advantage to know if a processing node requires a copy of a message for its local agents to read.

When FLAME was originally aimed at biological models [15], which used Cartesian space, the partitioning of agents over processing nodes was achieved by partitioning the simulation space. Each message sent was given an interaction radius for the distance agents had to be to be able to read the message. Each node knew the simulation space controlled by other nodes and so could filter messages that would not affect agents on certain nodes.

FLAME was then used for location-agnostic economic models [1] where the filtering of messages depended on a multitude of factors. For example networked regions, salary, cost, etc. Instead of a built-in Cartesian interaction radius filter the filtering of messages was opened up to the modeler. A model could now include, for each input to a transition function, a modeler defined filter. The partitioning of agents then became non-trivial, as it depended on a multi-layered communication network linking different agent types, regions and dynamic agent memory. As such a simple round-robin approach was taken. To limit the amount of message synchronisation between nodes, for each message board, the agent filter data on each node are amalgamated and then sent to all the remaining nodes. The remaining nodes can then use the filter data to filter messages before they are sent to the originating node.

D. Scheduling and Performance

An important aspect of attaining good performance is to completely hide the communication cost by scheduling as much computation as possible between functions that write messages and those that read them. The communication bottleneck for ABM is usually high as each agent is continually communicating with other agents.

The scheduling of functions is handled by the framework and is currently done statically. This is done by the *xparser* program which parses agent definitions in a marked up XML document then produces a directed acyclic graph representing the dependency graph of transition functions. The model seen in Figure 3 has the dependency graph seen in Figure 4. It can be seen that message board X depends on the first function of agent A and so on down through the dependencies.

Each agent in a model will have its own function dependency graph and they are coupled together by dependencies on message boards. Every message board will be a descendent of functions that write to it, and a parent to those that read messages from it.

Using the function dependency graph, the *xparser* can schedule the execution of agent transition functions such that message producers are scheduled as early as possible and message consumers as late as possible. This maximises the amount of computation being performed while the synchronisation process is in flight. This process is totally automated and is directed by the dependencies determined by the state-based model provided by the user.

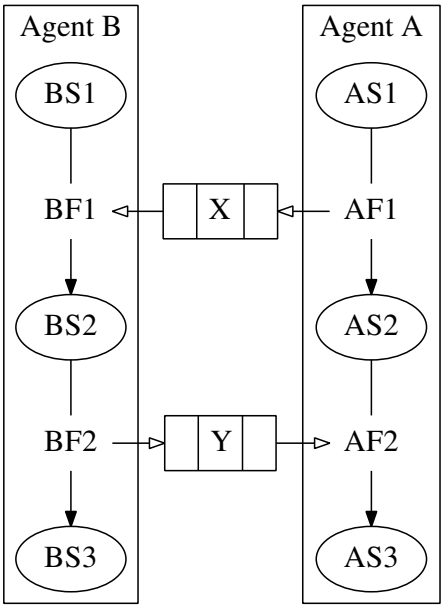

Figure 3. Example Model Definition

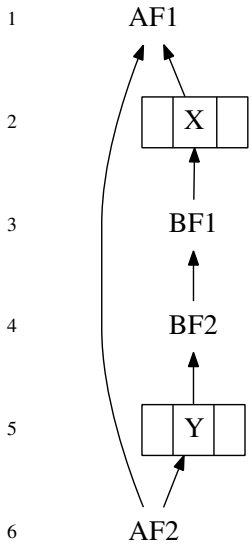

Figure 4. Dependency Graph of Example Model

IV. ABM FRAMEWORK COMPARISONS

Large scale agent-based models require large amount of memory as well as computing power. HPCs are able to distribute the computational power and memory across nodes but to effectively make use of this architecture requires specialist knowledge that some ABM frameworks try to solve.

Current HPC ABM frameworks, Repast HPC, EcoLab, ABM++ provide in-built methods for modellers to use but this still requires expert knowledge of HPCs to use. The problem is that they all allow direct agent-to-agent memory access. This means agent updates are synchronous and requires them to copy the whole of agent memory between processing nodes when an agent from one node needs to access an agent from another node. The capabilities for this are provided but the scheduling of agent updates and the synchronisation of data between nodes must be managed by the modeller.

For computational power GPGPUs have provided exceptional speed ups [11], [13]. By having to execute the same functions at the same time ABM GPGPU frameworks use asynchronous agent updates. This is because they are either cellular automata based or are an extension of the FLAME model for GPGPUs. This though means that they are better suited for homogeneous agent models, where there is only one type of agent with the same functionality. GPGPUs are also restricted by their amount of on-board memory they have direct access to. This can be solved by using multiple GPGPUs but you then get the same memory independent node problem as with HPCs.

Rather than providing in-build methods for modelers to program their own simulation programs, the FLAME framework provides a way to design agent models using a formal modelling technique. By breaking down agents into Communicating Stream X-Machines the agent-based model is broken down into constitute components. This provides some of the following strengths of this approach.

By defining agent transition functions as communicating via streams of messages, in contrast to other HPC ABM frameworks which allow direct agent-to-agent access, there is not a requirement to copy the whole of agent memory between nodes.

By defining each agent separately using state ordered transition functions with defined incoming and outgoing communication the execution of agent updates and handling of communication routines can be automatically calculated by the framework. This allows the integration of new components into agents by adding new states and transitions to the state machine representation. The scheduling of agent updates is then automatically recalculated for the new definition.

By decomposing models into CSXM it is also possible to use formal testing methods [16], [17] to test the components and the complete system.

An example of a large scale model that uses FLAME is the EURACE economic model [1]. It is too large and heterogeneous to fully fit a GPGPU architecture and the communication between agents would be too complicated to manually calculate when agent memory needs to be copied between nodes. The size of agent memory required for a EURACE agent, about 1 MB, would alone would make this prohibitive as 1000s of agents would need to be copied multiple times across multiple nodes every time step.

Using the state machine based approach to define agents, economic experts working on different areas of the EURACE model created their own agent states and transition functions and they were merged together when needed. The model, in total, required 9 agent types, 185 agent functions and 103 different message types (boards). Because FLAME automatically handles the scheduler, the plug and play (introduction and removal) of agents and agent functionality is easily managed and does not require any user interaction. This also made testing of the model easier as it was simple to remove parts of the model to be replaced by dummy parts that did not affect the parts being tested.

V. Current FLAME Benchmarks

To give some example of the current ability of FLAME to execute agent models on a HPC a simple model was used to predict performance.

The model used for benchmarking is the *Circles* model - a simplified molecular dynamics problem where each agent represents a spherical object that moves in a 2D Euclidean space based on the repulsive or attractive forces acting on it from other agents.

This model is ideal for benchmarking the performance of FLAME because:

- It is simple to understand, with little to go wrong
- It has minimal computational load
- It uses all-to-all agent communication (no message filters), so the model is communication dominant
- Message sends are immediately followed by receives so there is no opportunity for communication-computation overlaps
- Load balance is easily achieved using round-robin partitioning

The all-to-all agent communication represents a worst-case scenario and so this provides a lower bound for expected performance for a simple model[1]. In practice, various techniques can be applied to substantially improve performance; this includes the use of filters to reduce the amount of communication and the scheduling of computation in between message sends and receives to overlap useful computation with background communication tasks (message synchronisation).

The machine used for benchmarking is a 432 core Fujitsu PRIMERGY blade server with 36 nodes, each with 2 Intel Xeon X5670 CPUs. Each CPU contain 6 cores each capable of 2 simultaneous threads (HyperThreading).

The *number of procs* from now on refers to the number of cores (or the number of MPI tasks/processes, which is equivalent). Due to limited access to the machine, each benchmark configuration is repeated only four times and in the few cases where the timings were drastically different, the runs were repeated and the slowest ones discarded.This is therefore by no means a through benchmark and serves as a rough indicator of potential performance.

[1]More complex models may involve simultaneous synchronisations of multiple message boards, variable computational load, and various opportunities to reduce and hide communication overheads; this makes it a lot harder to predict performance

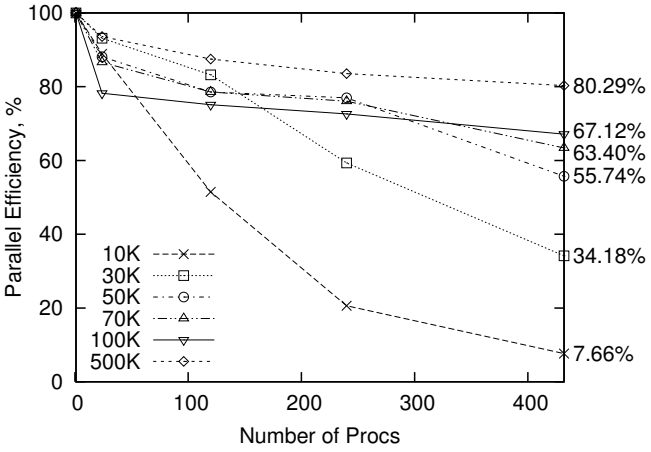

Figure 5. Comparing the parallel efficiency for different population sizes

To determine the weak scaling[2] of the model, the benchmark was repeated for different population sizes starting from 10,000 agents. The population size was stopped at 500,000 agents due to the prohibitively long single processor runs.

Figure 5 shows that scalability improves remarkably as the population size increases. Using 432 processes, the 10k agent run has a parallel efficiency of only 7.66% while the 500k agent run was able to maintain a parallel efficiency above 80%. This shows in a worst-case scenario the communication routines of FLAME, when given a suitably large population size, can be very efficient.

By plotting the changes in iteration time for different process counts (see Figure 6), a definite trend is seen in the timings whereby the graph starts flattening off at about 120 processes for all population sizes. Of course, with the limited number of points on the graph it is difficult to tell where the actual inflection point is and how it changes with population size.

Further benchmarks are planned to investigate the usage of message filters on the *Circles* model and to also create a location agnostic test model. This set of benchmark models will inform any future improvements make to FLAME.

VI. Limitations of the current FLAME model

The current FLAME model has provided a way to define agents such that the framework can automatically execute a simulation as efficiently as possible on a HPC without any expertise required from the modeller. The current framework

[2]Defined as how the iteration time varies with the number of processes for a fixed problem size per process

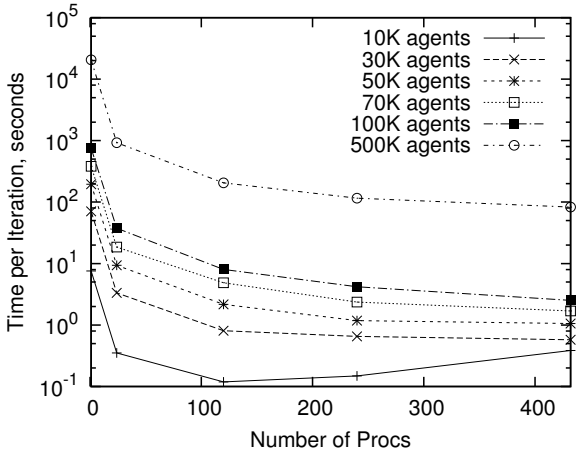

Figure 6. Time per iteration for different population sizes

has some self-imposed limitations that can be removed and are discussed in this section.

Agent functions have read-write access to all variables within agent memory, which seems sensible at first, but in hindsight is the cause of (or a contributing factor to) some of the limitations of the current FLAME execution model.

A. Data Granularity

Because each agent function can potentially write to all agent memory variables, the smallest unit of data is the whole agent instance. Data partitioning for parallel execution has to therefore be done at the agent level. Because there are computational costs (transition functions) and communication overheads (message access) tied to each agent, determining an optimum partitioning strategy is not straight forward.

For example, optimising for a balanced utilisation and computational load by equally distributing agents across nodes can lead to excessive overheads due to all-to-all synchronisation of all message boards. While grouping agents by type to reduce communication load can affect scalability due to uneven load as well as limit the simulation sizes due to insufficient memory capacity in a heavily populated node.

B. Execution Path

Currently the memory access requirements of each transition function are not known to the framework. The *xparser* therefore cannot make any assumptions about the dependencies between the functions and has to rely solely on the state diagram defined by the modelers.

More often than not, this leads to an execution graph that is mostly sequential with very few concurrent paths. Such a graph would be tall and narrow, and expresses very little parallelism.

For example from the example model if we knew the second function of agent B did not need to read a memory variable written by the first function of agent B then there would be no dependency from the second to the first function. Therefore the dependency graph seen in Figure 4 could be made to be short and wide, see Figure 7, which is better for parallelism as more computation can be scheduled at the same time.

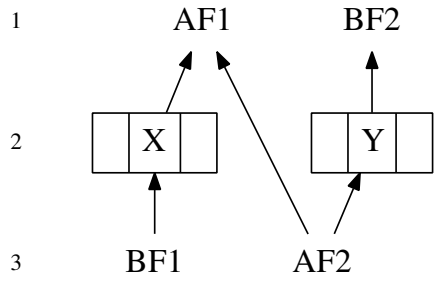

Figure 7. Updated Dependency Graph of Example Model

VII. Future Improvements of FLAME

This section discusses some of the approaches that are intended to be explored in order to maximise the parallelism within the FLAME framework.

A. Data Dependency Analysis

To improve the parallel performance of the FLAME framework, one needs to extract as much concurrency as possible from a simulation. This involves breaking the simulation down into more parallelisable units then scheduling their execution in a manner which fully utilises all resources available to the execution environment.

One way to extract more concurrency over the current framework is to have the memory access requirement of each agent transition function explicitly defined by modelers. With this additional information along with the state transition graph of the agent, one can build a more accurate view of the dependencies between the different transition functions.

Each agent memory variable can then be treated as an independent entity, and each write to the variable is seen as a transformation of the variable to a new version. Keeping track of memory reads and writes allows us to determine which functions can be run concurrently, and which ones need to be run in sequence to ensure that the correct versions of memory variables are accessed.

If a sequence of transition functions all read the same memory variables and never update the values, they are all reading the same version of data and therefore have no dependencies on each other. These functions can be executed concurrently (assuming there is no conflicting access to message boards). The same goes for functions which access different subsets of agent memory.

However, if a transition function has write access to a memory variable, then subsequent transition functions are considered to depend on the new version of the memory variable and therefore have to wait till that information is available.

B. Vector Operations

With the changes introduced in the previous subsection, transition functions can be treated as operations on a pre-defined set of independent variables. Since all the agents of the same type have the same set of transition functions and memory structure, we can effectively treat the transition function as an operation on long vectors where each vector element corresponds to an agent instance.

This shift in paradigm brings about many desirable features:

- The granularity of data has been reduced from an agent instance to a single memory variable. This allows us more flexibility in the storage structure, and a more fine-grained approach to data and task decomposition.
- Operations on long vectors are potentially more efficient and can better utilise memory and caches. They are also more amenable to different parallel programming paradigms, e.g. SIMD, task farming, stream processing, etc.
- Check-pointing and migration of data can be done more efficiently - the elements in the long vectors are of equal size and contiguous in memory so data packing and buffering is no longer required. Furthermore, using the new dependency graph, data can be written to disk in stages as the final version becomes available.
- Transition functions can be treated as independent tasks that can be executed in any order as long as its inputs are available. This opens up many opportunities for optimisation including dynamic scheduling of tasks, multiple levels of parallelism, etc. This is discussed in the following section.

C. Dynamic Task Scheduling

The function dependency directed acyclic graph (DAG) generated based on the analysis of memory reads and writes would implicitly encode the data dependencies. As long as the function dependencies are met, each function would effectively be accessing the correct versions of memory and messages. This greatly simplifies the job of managing dependencies and ensuring the correctness of the simulation.

Instead of converting the DAG into a static sequence of function calls as done in the current FLAME implementation, the DAG can be represented as a list of tasks to be consumed by a dynamic scheduler at runtime.

The use of a dynamic scheduler will allow the simulation to adapt to different runtime conditions and the variations in computational and communication loads that can occur in agent based simulations. Furthermore, a common runtime code can be used for all models. This leads to less code generation for each model thus improving the testability and maintainability of the framework.

During the simulation, tasks are added to a queue as they become available (dependencies met) and the scheduler selects tasks for execution based on priority levels assigned to each task. Once all tasks have been executed, the iteration is complete and the whole process is repeated for the next iteration.

Each entry in the task list could contain the following information:

- Task identifier: a unique handle for each task
- Task type: a label to determine which queue the task belongs to (covered below)
- Dependency list: references list of tasks that must be completed before this task can be executed
- Priority level: determines the priority of this task should there be more than one task in the queue

1) Using Task Priority to Optimise Resource Utilisation: The *priority level* indicates the urgency of each task. It assists the scheduler in determining which task from the queue should be executed first.

The priority level can be assigned based on many criteria, for example:

- Sub-tree weight - a task that has many dependants should be scheduled as early as possible.
- Task type - if there is only one queue, then the priority mechanism can be used to ensure urgent tasks such as message syncs are launched first, and non-urgent tasks (such as check pointing) are only slotted in to fill the gaps.
- Estimated run-time - if profiling information is available from previous iterations, we can predict a task's runtime and weight it accordingly.
- Vector length - if profiling information is not available, we can use the size of the input vector length as an initial estimate for weighing the task.

The priority levels should be recalculated periodically based on the runtime statistics collected during the previous iterations. The job of recalculating the priority level can itself be wrapped up as a task that is managed by the scheduler. The same goes for other framework level operations that require significant use of available resources. This ensures that none of the resources are oversubscribed.

2) Using Multiple Queues for Managing Different Resources: Assigning a *task type* allows the opportunity to support multiple task queues. Each queue can be assigned to different resources that can be managed independently. For example, we may choose to have separate queues for disk I/O heavy tasks (for data check-pointing), communication tasks (message syncs), and computation tasks (execution of agent functions). In addition, different computation resources that can operate independently (CPU, GPUs, and

other accelerators) can each have their own individual queue and be managed separately.

3) Using Slots to Control the Number of Concurrent Tasks Running on Each Resource: The scheduler queue is designed to ration the use of a particular resource type; there may be more than one instance of each resource (multiple CPU cores), or the resource may be able to handle several tasks simultaneously.

To take this into account, each queue is assigned one or more execution slots which it can fulfil. At runtime, the scheduler will attempt to maximise the use of resources by keeping every slot filled with running tasks, replacing each completed task with a new tasks from the associated queue. The total number of execution threads during a simulation would therefore be the number of slots, plus the execution threads of the framework runtime (maybe one or more).

VIII. Conclusion

Compared to popular ABM implementations that utilise synchronous agent updates and direct agent-to-agent access, the FLAME framework has the ability to automatically decompose agent based simulations into a dependency graph of agent functions and communications that can be distributed across processing nodes. Because FLAME handles the scheduler rather than the user the ordering of events can be make to be as efficient as possible and automatically fit the architecture it is on. By regenerating the schedule each time the framework can easily handle the addition and removal of parts of a model. This is important with large models as different parts can be created by different teams and have to be able to be integrated easily. Now that this has been achieved with the current version of FLAME there are many strategies that can now be utilised to further the efficiency of simulations and their use of any available resources. These include: a dependency graph generated by analysing the memory accesses of each agent function; a new execution model that will enable various opportunities including more efficient data structures and better resource utilisation using dynamical task scheduling.

Acknowledgement

This work has been funded by EPSRC Grants EP/I030654/1 and EP/I030301/1.

References

[1] C. Deissenberg, S. van der Hoog, and H. Dawid, "EURACE: a massively parallel agent-based model of the European economy," *Applied Mathematics and Computation*, vol. 204, no. 2, pp. 541–552, October 2008.

[2] M. Holcombe, S. Adra, M. Bicak, S. Chin, S. Coakley, A. Graham, J. Green, C. Greenough, D. Jackson, M. Kiran, S. MacNeil, A. Maleki-Dizaji, P. McMinn, M. Pogson, R. Poole, E. Qwarnstrom, F. Ratnieks, M. Rolfe, R. Smallwood, T. Sun, and D. Worth, "Modelling complex biological systems using an agent-based approach," *Itegrative Biology*, vol. 4, pp. 53–64, 2012.

[3] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A multi-agent simulation environment," *Simulation: Transactions of the society for Modeling and Simulation International*, vol. 82, no. 7, pp. 517–527, 2005.

[4] M. North, N. Collier, and J. Vos, "Experiences creating three implementations of the Repast agent modeling toolkit," *ACM Transactions on Modeling and Computer Simulation*, vol. 16, no. 1, pp. 1–25, January 2006.

[5] N. Minar, R. Burkhart, C. Langton, and M. Askenazi, "The Swarm simulation system: a toolkit for building multi-agent simulations," Santa Fe Institute, Working Paper 96-06-042, 1996.

[6] Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. (1999) NetLogo. [Online]. Available: http://ccl.northwestern.edu/netlogo/

[7] N. Collier and M. North, "Repast SC++: A platform for large-scale agent-based modeling," in *Large-Scale Computing Techniques for Complex System Simulations*, W. Dubitzky, K. Kurowski, and B. Schott, Eds. Wiley, 2011.

[8] R. Standish and R. Leow, "EcoLab: Agent based modeling for C++ programmers," *Proceedings SwarmFest 2003*, 2003.

[9] A. Helsinger, M. Thome, and T. Wright, "Cougaar: a scalable, distributed multi-agent architecture," *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, vol. 2, pp. 1910–1917, October 2004.

[10] RTI International, Durham, North Carolina. (2009) ABM++. [Online]. Available: http://parrot-farm.net/ABM++/

[11] M. Lysenko and R. M. D'Souza, "A framework for megascale agent based model simulations on graphics processing units," *Journal of Artificial Societies and Social Simulation*, vol. 11, no. 4, p. 10, 2008.

[12] B. G. Aaby, K. S. Perumalla, and S. K. Seal, "Efficient simulation of agent-based models on multi-gpu and multi-core clusters," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '10, 2010, pp. 29:1–29:10.

[13] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, February 2010.

[14] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan, "Communicating stream X-machines systems are no more than X-machines," *Journal of Universal Computer Science*, vol. 5, pp. 494–507, 1999.

[15] T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood, and S. MacNeil, "An integrated systems biology approach to understanding the rules of keratinocyte colony formation," *J. R. Soc. Interface*, vol. 4, pp. 1077–1092, 2007.

[16] F. Ipate, "Complete deterministic stream X-machine testing," *Formal Aspects of Computing*, vol. 16, pp. 374–386, 2004.

[17] R. M. Hierons and M. Harman, "Testing conformance to a quasi-non-deterministic stream X-machine," *Formal Aspects of Computing*, vol. 12, pp. 423–442, 2000.