Science & Technology Facilities Council

Technical Report
RAL-TR-2012-013

FLAME: an approach to the parallelisation of agent-based applications

LS Chin, DJ Worth, C Greenough, S Coakley, M Holcombe, M Kiran

August 2012

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358- 6254

FLAME: An Approach to the Parallelisation of Agent-Based Applications

L.S. Chin†, D.J. Worth†, C. Greenough†
S. Coakley‡, M. Holcome‡ and M. Kiran‡

August 10, 2012

Abstract

This report describes an approach to the parallelisation of agent-based applications. The software framework which forms the foundation of this work is the Flexible Large-scale Agent Modelling Environment - FLAME - which is being developed in a collaboration between the Computer Science Department of the University of Sheffield and the Software Engineering Group at STFC's Rutherford Appleton Laboratory.

FLAME is a generic program generator for agent-based simulations. Using a definition of the agent-based model and an C implementation of the agents state change functions, FLAME generates a complete ABM application. FLAME can generate versions of the application which can execute on both serial and parallel systems.

This report describes this process in detail and focuses on the techniques used within FLAME to parallelise the agent-based application. Some test results are presented which show some promising speed-ups of these applications.

The report concludes with some future developments of the FLAME functionality and architecture which should improve the parallel efficiency of FLAME generated applications.

† Software Engineering Group, STFC Rutherford Appleton Laboratory
‡ Computer Science Department, University of Sheffield

Keywords: agent-based modelling, parallelisation, optimisation

---

Email: {shawn.chin, christopher.greenough, david.worth}@stfc.ac.uk
{m.holcombe,s.coakley}@sheffield.ac.uk

Reports can be obtained from: http://epubs.stfc.ac.uk

Software Engineering Group
Computational Science & Engineering Department
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

In this report we describe in detail the parallel implementation of the FLAME - The Flexible large-scale Agent-based Modelling Environment. FLAME [1] has been developed in a collaboration between members of the Software Engineering Group at the STFC Rutherford Appleton Laboratory and the Computer Science Department at Sheffield University. This work was performed as part of the EU EURACE Project [2] which was a three-year project with the goal of investigating the use of agent-based modelling in economic and financial applications. The report builds on [3] which describes some initial developments of FLAME.

There are many agent-based modelling systems. A detailed survey of such programs, systems and frameworks is given by [4] and , [8]. Many of these systems are based on Java as their implementation language and although a good language for web-based and some communications applications, it is not the language of choice for high performance computing. Similarly there are relatively few agent systems that address the problem of scalable parallel simulations.

Before considering the parallelisation of FLAME it is worth considering the characteristics of a software system that make it worth taking the time and effort to parallelise. It should also be remembered that FLAME is not the applications program to be parallelised: FLAME is but an applications program generator. So it is the generated program that should exploit parallelise, not FLAME itself. FLAME takes a description of an agent-based model (the XMML definitions plus the associated function code) and generates a bespoke code for the particular application.

We are directing our attention to systems that are often referred to generically as *high performance clusters*. This hides the multitude of differences in the hardware architectures of these systems - the types of processors - single or multi-core, the communications network between processors - specialised integrated or commodity. Collectively these systems can be considered *Single Program Multiple Data - SPMD* architectures.

It is interesting to summaries some reason for parallelising an application:

- It is not possible to run the application on a single processor - memory requirement or run time too long.
- The application will be reused very frequently and has a very long execution time but the results are very important.
- The code easily decomposes into independent tasks which require little communication with other task.

However, in general terms it should also be remembered that some algorithms simply do not parallelise - there is insufficient independence in the component parts of a algorithm to utilise a multi-processor system. The algorithm just does not map onto the system of distributed processes exchanging information through a communications network.

As mentioned above the FLAME Framework is not the agent application - it is the application generator. Hence the approach to parallelisation of FLAME generated applications must rely on the underlying generic architecture of the generated application. Although this underlying architecture determines much of the potential parallelism in applications code might be utilised there, must be parallelism in the application.

It is clear that developing a scalable agent-based framework will be difficult - the underlying structure may well be determined by the framework but the application tasks - their computational size and communications characteristics are really unknown to the frameworkand will have a considerable impact of the parallelisation.

There are many other agent systems such as Netlogo [5], REPAST [6], MASON [7], SAMAS [9], JADE [10], SIMJADE [11], MACE3J [12] and SPADES [13] to name a few. Each have their own approach to implementation and each achieve varying levels of performance. Some been used to demonstrate scalable agent computing but these have been relatively small simulations.

The starting point of FLAME has been different from many these systems. High performance computing was thought to be essential to address large-scale simulations and thus the

implementation language was chosen to be C. This is the implementation language favoured by many of those programming HPC systems. This choice opens the implementation to many parallel programming tools and harnesses such as *MPI* - Message Passing Interface and *OpenMP* - Open Multi-Processing. These are the two main parallel processing tools used by the high performance computing community.

2 The FLAME architecture

We initially focus on the most basic characteristics of FLAME and ABM systems in general: that of communications - the exchange of information agent to agent and computational load - the work the agents perform.

The communication between agents is one of the most important characteristics of an agent-based application. There are various ways in which this communication can be implemented. In FLAME we have chosen to represent this communication and its associated data as a set of *message boards* on which agents can post messages and from which agents can read the messages. Figure 1 show this diagrammatically. We show two agent types: *Firms* and *Households*. The

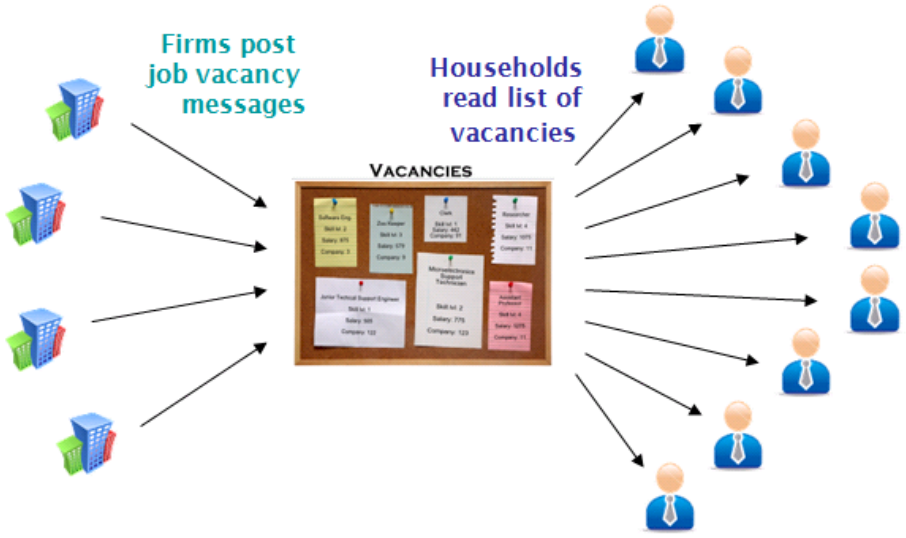

Figure 1: FLAME Message Boards

Firm agents require employees and communicate with Households about job opportunities by posting vacancy notices on the *Vacancies* message board. Households then read the information on the message board and select suitable job possibilities. In FLAME all the communications are through message boards and there are no direct agent-to-agent communications. FLAME provides the modeller with a message board API through which they can manage agent communications. The management of the message data and its access by agents are key operations within FLAME.

The second element of FLAME to be considered is the *computational load* - the work the agents perform. It is reasonable clear that an effective use of parallel system will be to distributed the computational load over all the processors. Figure 2 show the FLAME agent. It is an extension of Eilenburg's X-machines [14]. The important components of an agent are:

- The agent memory ($MEMORY$): this is local to the agent and can contain any form of agent-specific information;
- the agent states (S_i): these are the states each an agent can take depending on values in the agent memory or the condition of the overall model;

- the state transition functions (ϕ_i): these functions describe the rules by which agent can change state, and
- the FLAME message boards (In this example Message Boards X and Y): the primary communication mechanism.

The structure and number of each of these agent attributes is defined in the model description. Figure 2 shows transition function ϕ_1 reading (blue) and writing (red) to local memory and reading from message board X and writing to message board Y. There can be any number of

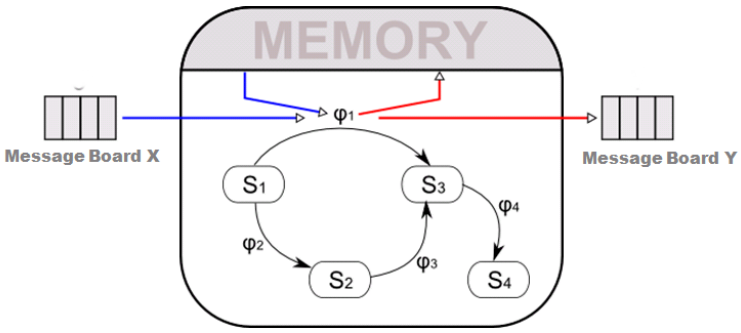

Figure 2: The FLAME agent

such reads and writes within a transition function and any number of potential agent states. However, in the design of FLAME we make one serious restriction on an agent's access to message boards. Within a single transition function an agent cannot read and write to the same message board: transition function ϕ_1 cannot read and write to message board X or Y.

3 Developing a FLAME model

Details of how to the develop a FLAME model are given the in the FLAME User Manual [15] but it sensible to give a brief overview here. FLAME has the two components: the model specification and the state transition functions. The model specification is defined in FLAME's agent specification language, XMML which is based on the XML standard and has its own set of XML tags. The state transition functions are implemented in C and use FLAME APIs to access agent memory and FLAME message boards. FLAME uses these files plus a set of templates to generate the application as a C program. This can then be compiled and link with the message board library using standard tools to produce an executable.

Figure 3 show the model development process. FLAME allows modellers to define their agent based systems and automatically generate efficient C code which can be compiled and executed on both serial and parallel systems. So the main elements of FLAME are: the FLAME `xparser` with associated templates and the message board library (*libmboard*). The modeller provides two input files: the Model XMML and the agents functions. These are parsed by the `xparser` and the results combined with the `xparser`'s template library to generate the application.

Using this approach the modeller can design a model that can be realised as a serial or a parallel program. Although for many models this naive approach might achieve reasonable parallel performance there are many pitfalls. To gain reasonable parallel performance in a very complex model the modeller will need to be aware of the impact of his choices on the performance of the model.

4 Parallisation of FLAME applications

The FLAME architecture has some inherently good characteristics that lend itself to parallelisation. However, because FLAME is an application generator it does not have a complete

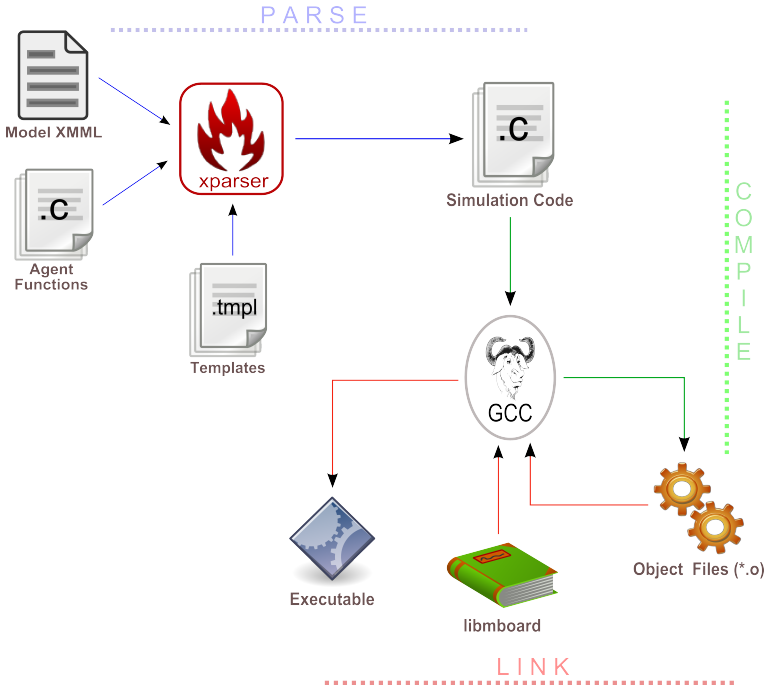

Figure 3: The structure of the FLAME Environment

understanding of the application it is generating.

Agent-based applications can be characterised as a set of communicating tasks. Although the agent population and their interactions can be specified *a priori* the computational load of each agent and the number of communications they perform are very difficult to determine without running the application on an example agent population.

However we clearly have a set of self-contained computational task - be it the agents or the state transition functions within them - and a vast amount of associated communications data. It is reasonable to propose distributing the agents - the computational load - across a processor array. Also, as FLAME uses a collection of message boards to facilitate inter-agent communication this data can also be distributed across the processors. As the majority of large high performance computing systems currently use a distributed memory model a Single Program Multiple Data (SPMD) paradigm is considered most appropriate for the FLAME. So in a nutshell, the parallelisation of FLAME utilises partitioned agent populations and distributed message boards linked through MPI communication. Figure 4 shows the difference between the serial and parallel implementation. The most significant operation in the parallel implementation is providing access to the message information required by agents on one node of the processor array which are stored on a remote node of the processor. The FLAME Message Board Library manages these data requests a synchronisation process. This synchronisation essentially ensures that local agents have all the message information they need as the simulation progresses. Coupled with this process of distribution there two main areas that specific attention: initial and dynamic load balancing and communications strategy for message board synchronisation.

Initial load balancing is not too difficult: we have a population of agents, of various complexities, to which we can assign relative weights and so in the most general case the agents can be distributed over the available processors using the weights. This may well give an initial load balance but makes no account of the possible communication patterns of the agent population.

As the simulation develops the numbers of agents in the population may change and adversely affect the load balance of the processors. It is a very interesting and difficult problem to gauge whether the additional work (computation and communication) involved in remedying a load

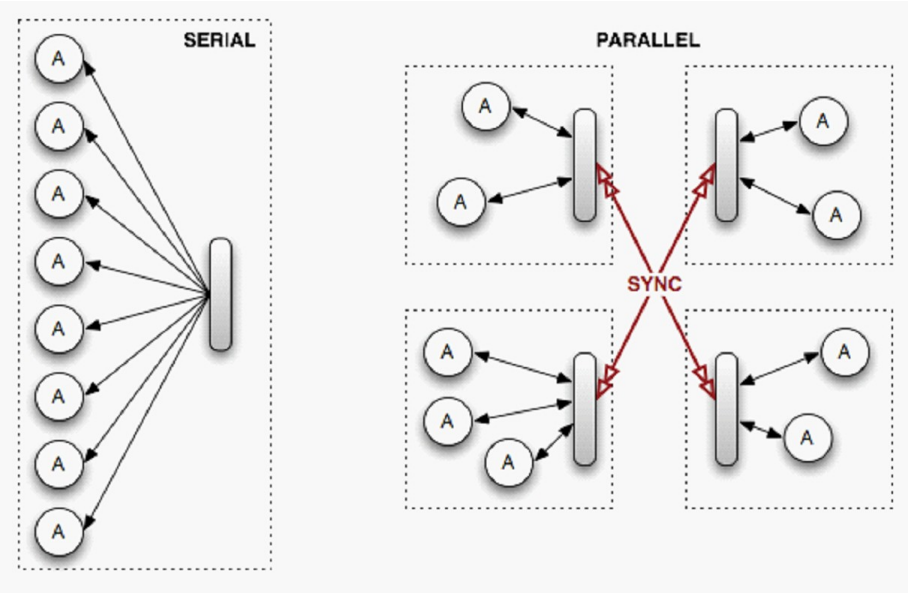

Figure 4: Serial and Parallel Message Boards

imbalance is worth the gain. Given that the goal of any dynamic re-organising of the agents is to reduce the elapsed time of the overall simulation, determining whether a process of dynamically re-balancing the population will contribute to this is very problematic. It may well be that a slight load in-balance will have no significant effect of the wall clock time of the simulation.

The patterns and volumes of communication for the population will have a considerable impact on the performance and parallel efficiency of the simulation. In general, agents are rather light-weight in terms of computational load. Where all agents can and do communicate with all others through the message boards the communications load within and across processors will be great. Fortunately communications within a processor are generally efficient. However across processors this communication can dominate the application. FLAME manges the communication between agents thought the Message Board Library, which uses MPI to communicate between processors. The Message Board Library attempts to minimise this communication overhead by overlapping the computational load of the agents with the communication and minimising the amount of data being transferred. In a model in which there is a fully connected and communicating agent population, the inter-agent interaction may not be local but long range. This could lead to many-to-many, inter-node communication which can drastically impact the scalability and simulation time. FLAME addresses this situation by providing the modeller with tools to filter selectively the information being transferred between processors.

Where the agents have some form of locality the initial distribution of agents can make use of this information in placing agents on processing nodes. During the simulation agents can be dynamically re-distributed to maintain computational load balance. However given the light-weight computational nature of many agent types the effect of dynamically re-distributing agents on the grounds of their communications load may well turn out to be more important than considering computational load.

5 The Message Board Library

As described above the distribute of the message boards is one element of the parallelisaton of FLAME applications. It is the job of Message Board Library to manage these message boards: their creation and deletion, synchronisation and access to message information.

The Message Board Library is decoupled from the FLAME framework and implemented as a separate library. This provided the flexibility to experiment with different parallelisation strate-

gies while minimising the impact on current users of the FLAME framework. The Message

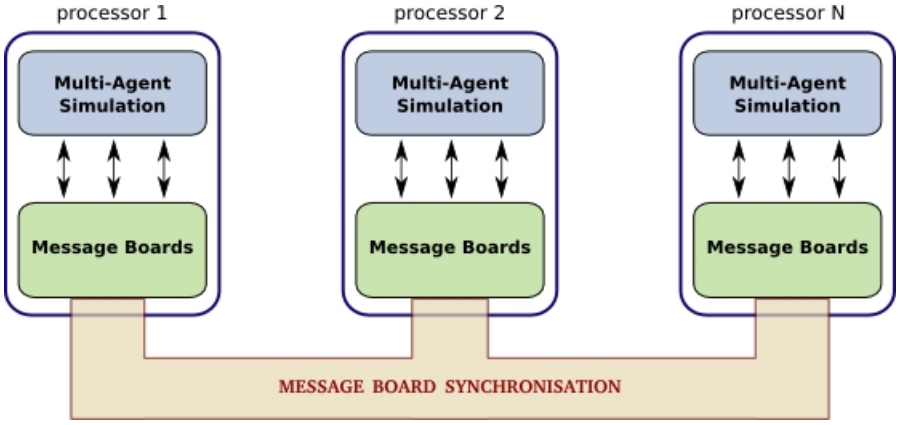

Figure 5: Parallelisation of FLAME using distributed Message Boards

Board Library (*libmboard*) can be built as a set of static libraries which are linked to users' simulation object files to create serial and parallel executables. This setup enables users to maintain a common source base for both serial and parallel simulations, thus simplifying code management and testing. It also provides *libmboard* developers with the facility to quickly switch between different library implementations without having to constantly recompile the test program. *libmboard* uses the Message Passing Interface (MPI) to communicate between processors,

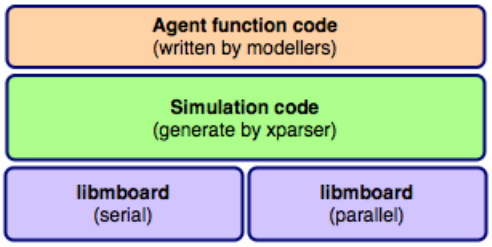

Figure 6: Users can create either serial or parallel executables by linking their object files to the appropriate *libmboard* library

and POSIX threads (pthreads) to manage a separate thread for handling data management and inter-process communication. The details of these components will be discussed in later sections (Section 5.4 and Section 5.6 and also in the full *libmboard* Reference Manual [16].

5.1 *libmboard* API

All functionality provided by the Message Board Library is accessible via the *libmboard* Application Program Interface (API) which defines a set of routines and opaque datatypes. Through the API, the full functionality of the library is made available without exposing the internal implementation. With this separation between interface and implementation, the complete communication strategy and data representation system can be replaced without the users being affected.

The *libmboard* API is intended for use within the FLAME generated simulation code. It provides the functionality for creating, managing and accessing Message Boards. Implementation details such as internal data representations, memory management and communication strategies are transparent to API users and therefore can be replaced or improved without affecting FLAME developers and end-users.

Modellers are provided with model-specific routines that hide the complexity of managing

boards and packaging data into suitable datatypes. Figure 7 shows an example of this whereby an agent's message add request gets translated into a *libmboard* API call.

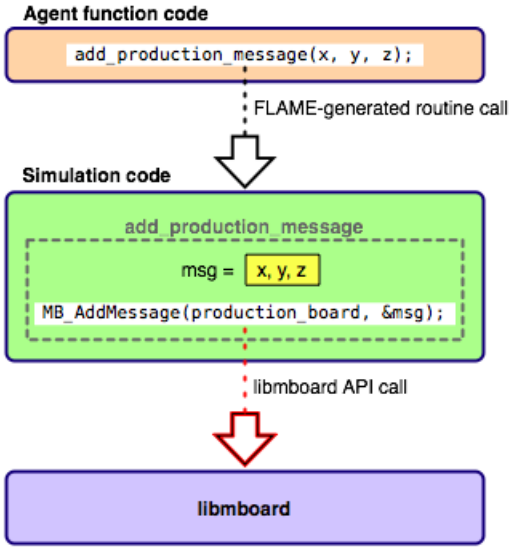

Figure 7: Modellers use FLAME-generated routines which get translated to the appropriate *libmboard* API call

5.2 The Message Boards

Message Boards are essentially distributed data structures that can be read from and written to by agents on all processing nodes. Up to 4096 different Message Boards can be created, whereby each Board is created to store message objects of a specified size.

Message Boards are created using the `MB_Create()` routine. This routine will return a Board Handle (of type `MBt_Board`) which can be used to reference the Board when performing further operations.

Once a Board is created, messages can be added to it using the `MB_AddMessage()` routine. During an add, the message data is duplicated and stored in the Board, allowing the calling code to reuse or deallocate the original message data. The cloning of data greatly simplifies the usage of the API and protects the internal data representation from accidental corruption.

Messages added to the Board are immediately available to all agents within the local processing node, or, after a sync (see Section 5.4), across all processing nodes.

The API provides further routines from emptying (`MB_Clear()`) and deleting (`MB_Delete()`) Boards.

5.3 Message Board Iterators

Iterators are opaque objects used for traversing Message Board content. They provide *libmboard* users access to messages while isolating them from the internal data representation of Boards. It also aids in enforcing the rule that Boards should be not modified in any way by a read access. With this rule in place, multiple Iterators can be created to traverse the Board independently using different access patterns.

Iterators are created against a Board using the `MB_Iterator_Create()` routine. Upon creation, the Iterator generates a list of the available messages within the Board and places a cursor at the first entry. When an Iterator is created, it essentially creates a snapshot of the content of a local Board. Any data added to the Board after the creation of the Iterator will not be available, while emptying or deleting the Board will invalidate the Iterator.

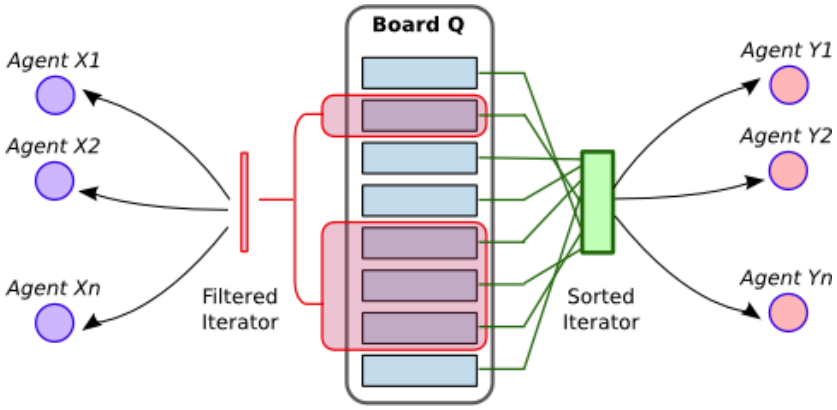

Figure 8: Using specialised Iterators to achieve different access patterns

Data traversal is performed by repeated calls to `MB_Iterator_GetMessage()`. This routine returns a copy of the data from a single message, then moves the cursor to the next item. Once the cursor moves beyond the last item, further calls to `MB_Iterator_GetMessage()` will return a `NULL` pointer indicating that iteration has ended. `MB_Iterator_Rewind()` can be used to move the cursor back to the first item to allow for reuse of the Iterator.

The *libmboard* API also provides several routines for creating specialised Iterators which allow users to traverse subsets of the Board content in a customised order. These routines accept user-defined sort and/or compare functions that are used to control the selection of messages and their order in the Iterator. These specialised Iterators are traversed just like a standard Iterator, using `MB_Iterator_GetMessage()`.

Other Iterator routines include `MB_Iterator_Randomise()` for randomising Iterator entries, and `MB_Iterator_Delete()` for deleting an Iterator.

5.4 Message Board Synchronisation

Synchronisation of Boards involves the propagation of message data such that agents farmed out across the different processing nodes have a unified view of the collective content through their local Boards. This is vital to ensure that the simulation is coherent even though instances of the Board are distributed across different processing nodes.

Synchronisation of Boards is performed in two stages – the initial Board synchronisation request, and the actual completion of synchronisation.

The initial Board synchronisation request is performed using `MB_SyncStart()`. This routine locks the Board and adds it to the *Synchronisation Request Queue* (discussed in Section 5.6.1). The calling code is then immediately given back control and is free to proceed with other tasks that do not require access to the Board in question.

When read access to the Board is required, the synchronisation process must first be completed using `MB_SyncComplete()`. This routine is blocking, and will wait for the Board to be unlocked before returning control to the calling code. Alternatively, there is also the `MB_SyncTest()` routine which is non-blocking and returns immediately with the completion status of the synchronisation. The calling code can immediately access the Board if the returned status is `MB_TRUE`, or, if the returned status is `MB_FALSE`, proceed with other tasks and repeat the test at a later stage.

The sequence diagram in Figure 9 depicts how a board synchronisation request may take place. The process of actually synchronising and unlocking the board is performed concurrently in the background by the *Communication Thread*. This is discussed further in Section 5.6.

To make the most of the concurrent nature of *libmboard*, calls to `MB_SyncStart()` should be placed as early as possible within the code (immediately after the final message has been written) such that more work can be scheduled before `MB_SyncComplete()` is called. This will

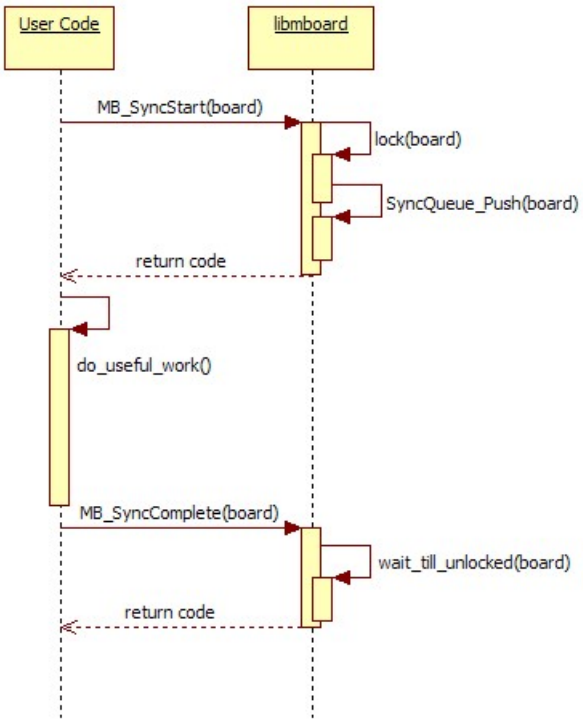

Figure 9: Other work can be scheduled during board synchronisation to hide the overheads of communication

maximise the amount of overlap between useful computation and the synchronisation overheads.

5.5 Message Board synchronisation optimisation

In its simplest form, a Board synchronisation may be implemented as a full replication of all messages within each local Board. This method is rather straightforward to implement, and would indeed serve its purpose of providing every agent access to the collective Board content. Unfortunately, it is also extremely expensive in terms of communication and memory requirements, and would therefore defeat the purpose of running in parallel (considering the key reasons behind parallelisation are to reduce elapsed time and per-node memory usage).

libmboard overcomes this problem by tagging each message with the IDs of processing nodes that require read access to it (see Figure 10). With the tagging in place, full data replication is avoided by only propagating the relevant messages to each processing node.

In order to tag messages, each processing node would need to indicate the selection of messages that it requires. This facility is provided by the `MB_Function_Assign()` routine which assigns used-defined filter functions and their associated parameters to each Board. At the start of each synchronisation the parameters are gathered from all remote nodes and message tagging is performed.

Within the FLAME framework, the filter functions are generated automatically based on the `<filter>` tag that modellers assign to the input of each agent function, while the function parameters are computed at run-time based on an analysis of agent memory.

Obviously, if the `<filter>` tags are not used, filter functions will not be assigned to Boards and the synchronisation process will fall back to a full data replication.

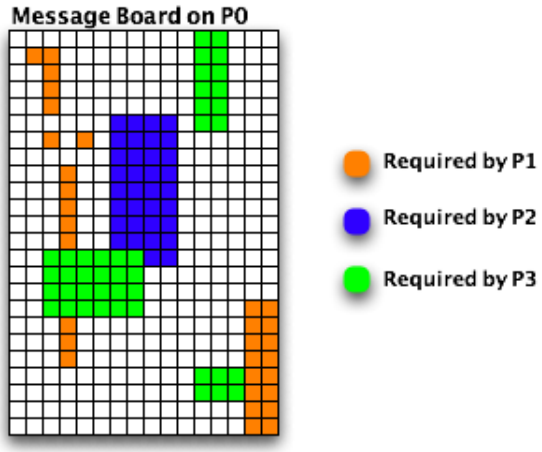

Figure 10: Full data replication is avoided by only propagating tagged messages to the relevant nodes

5.6 The Communication Thread

During the initialisation of the Message Board environment, *libmboard* starts up a separate thread (the *Communication Thread*) to handle the synchronisation of distributed boards.

Apart from potentially making better use of multi-core processors, delegating communication and memory intensive operations to a separate thread allows us to minimise the effective overheads by performing them concurrently with the main simulation and thus overlapping the Board synchronisation time with that of useful computation.

Once the *Communication Thread* is started, it goes into a continuous loop of processing two control queues – the Synchronisation Request Queue (*Sync Queue*), and the Pending Communication Queue (*Comm Queue*).

The loop is interrupted whenever both queues are empty, whereby the thread sleeps till it receives a signal from the parent thread, or quits when the termination flag is set. This is depicted by the Activity Diagram in Figure 11.

5.6.1 The Sync Queue

The *Sync Queue* is the interface between the *Communication Thread* and the parent thread, and acts as a staging point for a Board's synchronisation requests. Boards that need to be synchronised are locked by the main thread and pushed into the *Sync Queue* for further action by the *Communication Thread*. Access to the queue is protected by the mutex lock mechanism provided by the *pthreads* API.

When the *Sync Queue* is processed, all Boards within the queue are placed in an appropriate state[1] and moved into the *Comm Queue*. The individual Board will remain locked until it has passed through the *Comm Queue*.

5.6.2 The Comm Queue

The *Comm Queue* manages the list of Boards that are in the midst of synchronisation. Each Board within the *Comm Queue* is assigned a state based on the synchronisation stage it is in. Whenever the *Comm Queue* is processed, the *Communication Thread* iterates through the list of Boards and executes the transition function associated with the state of each Board.

Boards that reach their `END` state are removed from the *Comm Queue* and unlocked to indicate that synchronisation is complete. The unlocking of the board would be picked up by the parent

[1] depending on whether filter functions and parameters have been assigned to the Board

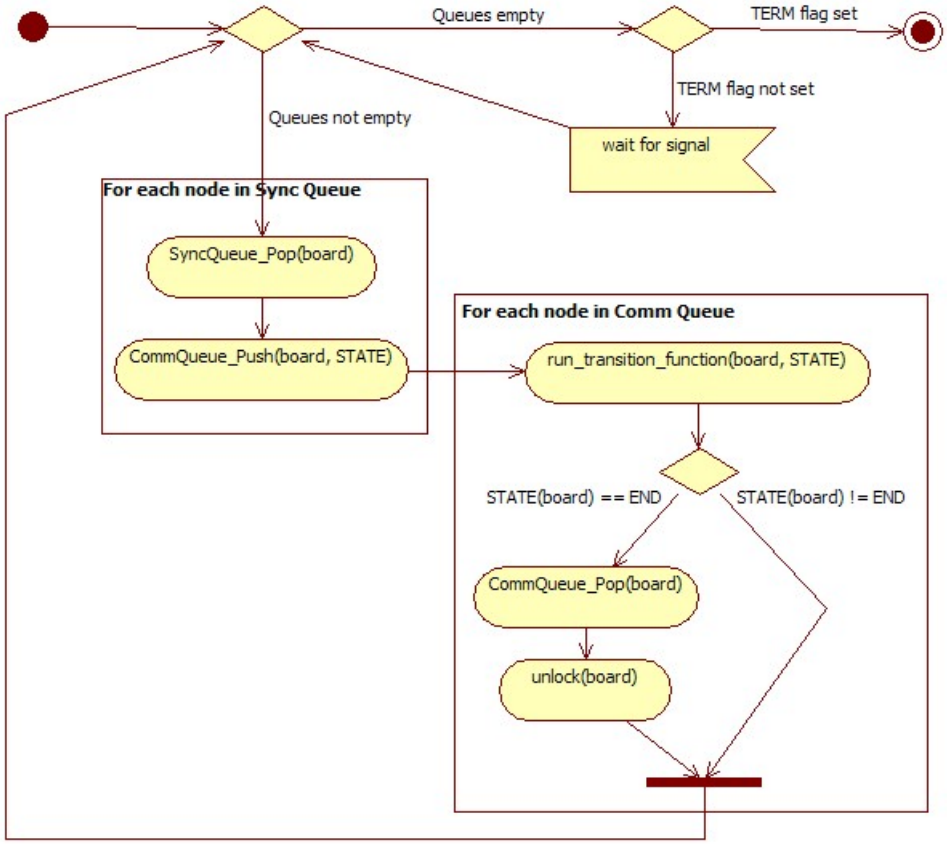

Figure 11: Activity diagram for Communication Thread

thread when the appropriate *libmboard* API routines are called (refer back to Figure 9 for further clarification).

5.6.3 Stages of synchronisation

The synchronisation process of a Board is split into stages such that inter-process communication (non-blocking MPI sends and receives) spans across at least two state transition functions – one to initialise the non-blocking send/receive operation, and the others to test and complete the communication.

For example, when a Board is in the `PRE_PROPAGATION` state (see Board synchronisation state diagram in Figure 12), the `InitPropagation()` transition function will prepare the necessary buffers, issue a series of non-blocking MPI sends and receives, and transition the Board to the `PROPAGATION` state without waiting for the communication to complete. During the next iteration, the *Communication Thread* would come back to the same board and either transition it to the next state if all communication has completed, or maintain the state if there are still pending communications.

By ensuring that transition functions never involve idle waits for events or specific conditions, the *Communication Thread* can continuously cycle through all pending synchronisations and perform the transition of each Board as they are ready.

5.7 Use of *pthreads* with FLAME Message Boards

During the design stage of *libmboard*, several complications were identified when considering the use of *pthreads* for implementing the *Communication Thread*. Some of the issues that were recognise are:

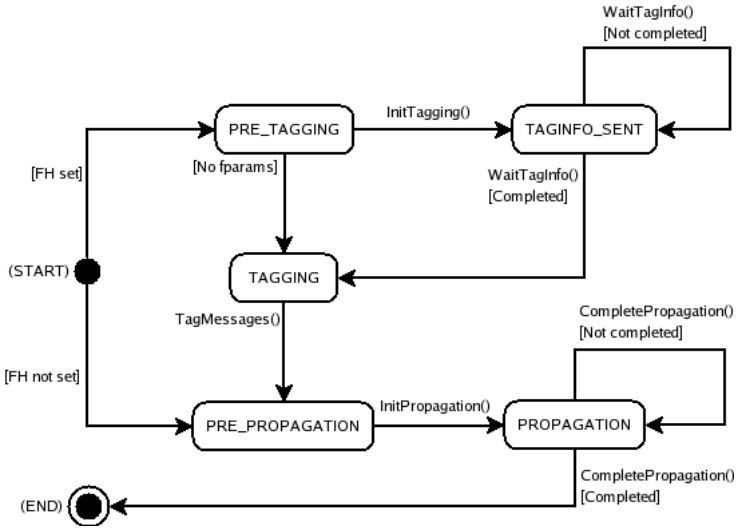

Figure 12: State diagram for processing Comm Queue nodes

- Portability – some platforms do not natively support *pthreads*. For example, *pthreads* are not supported on the IBM Bluegene/L system. They are also not available natively on Microsoft Windows Operating Systems. However, *pthreads* support has been introduced in the latest generation of Bluegene (Bluegene/P), and on Windows, *pthreads* can be used through Cygwin or by using third-party libraries.

- Thread-safety – Not all MPI Libraries currently available are thread-safe. This made the development of *libmboard* much harder as this introduces an additional limitation that the parent thread (which includes the calling code and the API routines themselves) must not issue MPI calls when the Communication Thread is also issuing a call or has any outstanding MPI communication. Extensive testing had to be performed using different compilers and MPI Libraries to ensure that *libmboard* functions correctly.

- Process mapping – Mixed-mode programming (Multi-threaded + MPI) makes the launching of jobs on multi-core SMP clusters more complicated. Choosing the most efficient mapping of threads and MPI tasks to the various cores of different affinity is not immediately obvious. Additionally, once a reasonable mapping is decided upon, conveying it in a request to the different job schedulers when launching jobs on shared clusters could prove to be a challenge.

It was eventually decided that the possibilities brought about by using a threaded model outweighed the potential drawbacks.

5.8 Use of `<filter>` in the FLAME Message Boards

As mentioned previously in Section 5.5, *libmboard* relies on the filter functions assigned to Boards to reduce communication and avoid full data replication by tagging messages. This in turn relies on the FLAME framework providing the relevant information gleaned from `<filter>` tags in the model definition.

The `<filter>` tags provide additional information to the FLAME framework about the information required buy agents. This information is use in two ways: firstly in constructing iterators (see Section 5.3) to reduce the number of messages presented to an agent for processing and secondly they are use by the Message Board synchronisation routines to reduce the volume of data being gathered from other nodes.

The analysis and use of this information is performed by the FLAME framework which generates suitable call to the *libmboards* API. The use and combination of message filters requires

developing some complex algorithms and strategies. In the current release of FLAME although the iterator can use any form of filter only *constant* data is use in the synchronisation filters.

6 Data Partitioning

As described above in general terms, parallelisation in FLAME has been introduced through distributed message boards and distributed agent populations. Hence at the start of any simulation the agent population must be distributed over the available processors.

As achieving some form of load balance - each processor performing a similar work load - is important in reducing the elapsed time of a simulation, the initial distribution of the population should attempt to achieve this. However such an initial distribution can only be based on the information provided in the XMML models files and the associated user provided C code. In the current version of FLAME there is little useful information provided.

Although achieving a load balance over the processors is important in reducing elapsed time, reducing inter-processor communication is equally if not more important in agent-based applications. Deriving information on the communications load of an agent population can only be achieved whilst the application is executing although some information can be derived from the XML and C code.

Two basic methods of static partitioning have been developed: partitioning based on a separator and *round robin* partitioning. Both are implemented within FLAME and the user can request one or the other with command line flags when the model is run.

6.1 Separator Partitioning

Separator partitioning distributes the agents amongst the partitions based on one or more memory variables of the agent. Every agent must have these variables for this to work and the variables can be either discrete or continuous numerical values. The most obvious example is distributing agents using their position (x, y, z co-ordinates) which gives a good initial distribution in cases where communication is between near neighbours. This is already implemented in FLAME due to the framework's initial field of application, biological systems, and is referred to as *geometric partitioning*. Other examples of separators could be considered but the overall

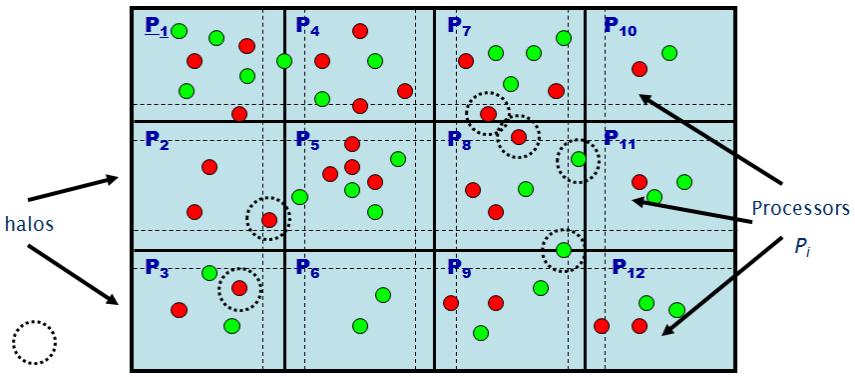

Figure 13: Geometric partitioning of agents

aim is to get those agents that will generate a lot of communication with each other on to the same partition - something that is not always feasible.

6.2 Round Robin Partitioning

This is the simplest form of partitioning in which agents are distributed, one at a time to each partition in turn. No account is taken of behaviour during the simulation but this may be the only way of partitioning if the agents have no common memory variables. This type of

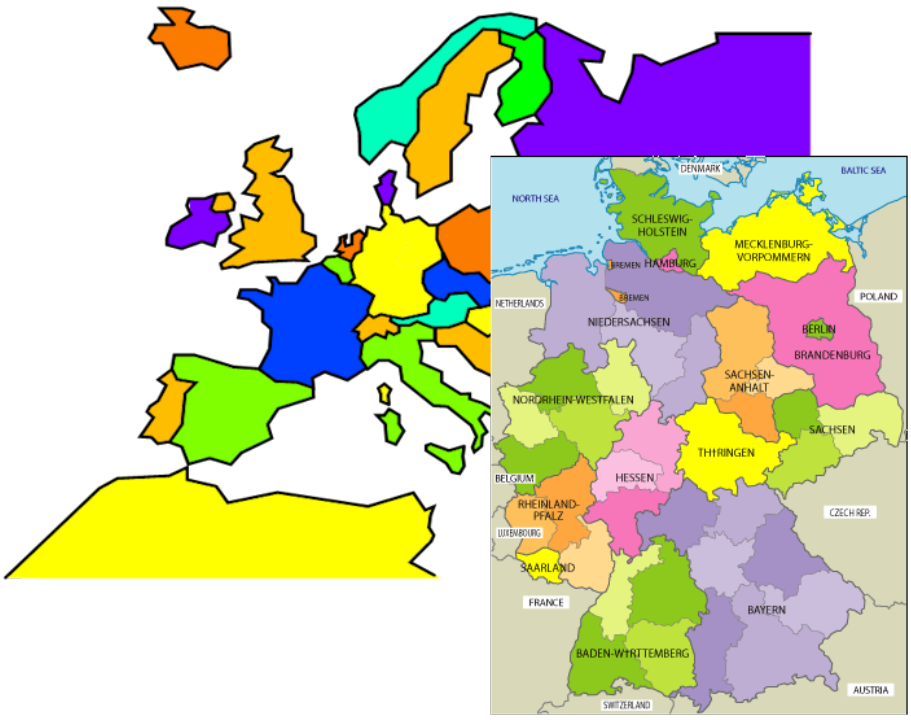

Figure 14: Partitioning based on a country identifier

partitioning is implemented in FLAME. It is possible to extend this method by using the agent

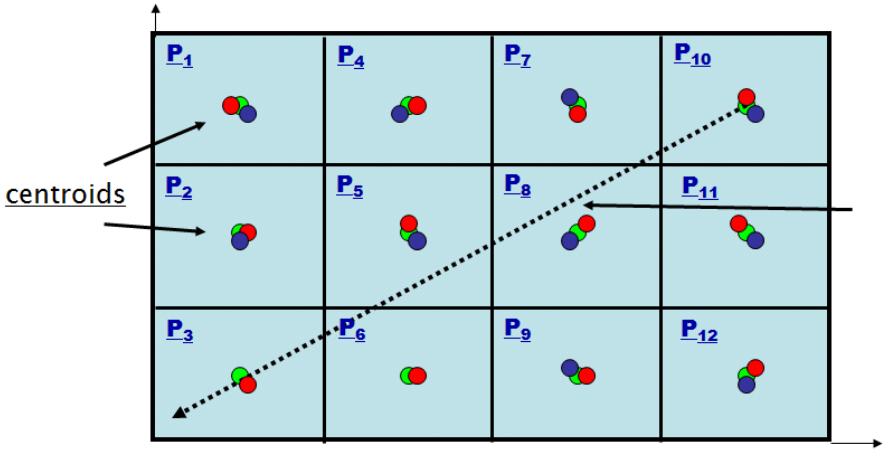

Figure 15: Round Robin distribution of agents

type as a discriminator. Agents of a particular type would be allocated to one partition (or set of partitions) on a round robin basis if it were known (or envisaged) that agents communicated with other agents of their own type more than any other type.

7 Static and Dynamic Analysis Tools

There are many elements to the testing and assessment of an application and this is all the harder in the case of FLAME as FLAME is a program generator. We have not only to verify that FLAME generates *correct* code as defined in the FLAME model definition but also that the generated code is also *correct*.

A number of static and dynamic analysis tools have been developed to help perform this type

of consistency analysis and verification. The example outputs are taken from the EURACE Model [2]

FLAME_Analyses : a static analysis of the FLAME model which gives detailed information on the components of a model: agent, function and messages types, their number and sizes, a static communications table,and a weighted communications table. An example of the outputs from FLAME_Analyses are given in Tables 1, 2 and 3.

```
Agent list:
----------
(  260 Bytes) (M: 7, 7) Bank
( Dyn Memory) (M: 5, 6) Mall
( Dyn Memory) (M: 3, 5) IGFirm
( Dyn Memory) (M:19,16) Household
( Dyn Memory) (M:11, 7) Government
( Dyn Memory) (M:20,29) Firm
( Dyn Memory) (M: 4, 3) Eurostat
( Dyn Memory) (M: 3, 2) Clearinghouse
( Dyn Memory) (M: 6, 1) Central_Bank
```

Table 1: : Agent list with internal memory descriptions

In Table 1 we have a list of all the agents present in the model and an indication of the internal memory state: `Dyn Memory` indicates that the agents internal memory is dynamic. The expressions `(M:7,7)`, as in the case of the `Bank` agent indicates that this agent receives 7 message types and sends 7 message types. For the `Clearinghouse` agent 3 message types are received and two types are sent. Note that these numbers refer to the message types, not the number of messages as this will be population and model state dependent.

```
* Bank
     |--( bank_identity )-------------------------> Firm (daily,0)
     |--( dividend_per_share )--------------------> Household (monthly,1)
     |--( accountInterest )-----------------------> Household (daily,0)
     |--( loan_conditions )-----------------------> Firm (daily,0)
     |--( bank_interest_payment )-----------------> Central_Bank (daily,0)
     |--( tax_payment )---------------------------> Government (monthly,0)
     |--( bank_to_central_bank_account_update )---> Central_Bank (daily,0)
   F <------------------( policy_announcement )--| Government (yearly,1)
     <---------------------------( policy_rate )--| Central_Bank (monthly,1)
   F <-------------------------( loan_request )--| Firm (daily,0)
   F <-----------------------( loan_acceptance )--| Firm (daily,0)
     <---------------------------( installment )--| Firm (daily,0)
     <---------------------------( bankruptcy )--| Firm, Firm (daily,0)
   F <------------------( bank_account_update )--| Firm, IGFirm, Household (daily,0)
```

Table 2: : Message activity of Bank agent including timing and filtering

In Table 2 the arrows (`-->`) show the direction of the messages and the `F` indicates that the message is filtered. At the end of each line the entry `(daily,0)`, for example indicates the period and phase of a message.

The final example of the output from the analysis tool is the inter-agent communications table show in Table 3. The table displays the possible inter-agent communication based on the communications defined in the model definition. Although this has been weighted using the message size as a measure of a message's volume the table takes no account of the numbers of particular agents in the population. Despite this it gives some indication of the likely communications traffic between agent types which can be used to inform any agent distribution process in a parallel implementation of the model.

		Message Destination								
		0	1	2	3	4	5	6	7	8
Firm	0	0.000	0.000	3.065	1.533	1.571	10.000	2.299	1.571	4.598
Central_Bank	1	0.000	0.000	0.000	0.000	0.038	0.000	0.000	0.000	0.038
Clearinghouse	2	2.299	0.000	0.000	0.000	1.533	0.766	0.000	0.000	0.000
IGFirm	3	1.533	0.000	0.000	0.000	0.038	0.038	0.000	0.000	0.766
Government	4	0.003	2.299	1.533	0.003	0.000	0.808	0.000	0.000	0.003
Household	5	6.935	0.000	0.766	0.000	2.337	0.000	1.533	0.038	0.766
Mall	6	0.766	0.000	0.000	0.000	0.000	3.065	0.000	0.038	0.000
Eurostat	7	0.038	0.038	0.000	0.000	0.805	0.766	0.000	0.000	0.000
Bank	8	1.533	1.533	0.000	0.000	0.038	0.805	0.000	0.000	0.000

Table 3: : Weighted Inter-agent Message Analysis

In this example it is clear that *Firm* agents have considerable communications with *Household* agents and vica versa. However it should be noted that the information flow is not symmetric. Given the population sizes of each agent type is possible to determine the potentially dominant communications and hence determine possible strategies to minimise the communications overheads in a parallel simulation.

FLAME_Consistency : a static consistency checker which compares the XMML definition with C code and ensures that the number and usage of messages is consistent. The script scans all the XMML and C code looking for message definitions and where messages are being passed. The tool checks that all messages that are sent by agents can be received. The consistency of the data being sent and received is not checked. This checking is included in the compilation and building of the application.

The Message Monitoring Package : The *MM* package is a dynamic library designed to monitor message traffic in the simulation. It is a set of additional directives included in the FLAME templates which are embedded in the application code that monitor all message traffic and writes to an SQL database. The database can be post-processed to assess the message traffic in the model. It also gathers information on the agent population in the simulation and the records of all function calls.

The Timer Package : The *Timer* package is used to measure elapsed CPU time for functions and message board synchronisations during a simulation. Knowing which functions take the longest time has helped to narrow the application of more detailed profiling tools such as `gprof` allowing for quicker identification of problems and possible solutions. Analysis of message board synchronisation times has shown that the message board implementation has provided excellent overlap of communication and computation.

FLAME_Analyses and FLAME_Consistency are python scripts that process the FLAME model definition and C-code files and the MMP and TP are libraries included in the FLAME application at build time to produce run time output.

8 Performance of the FLAME Framework

Two of the fundamental design features of FLAME are that all communications between agents takes place through a specified message board - a message repository - and that these message boards are distributed over the processing nodes of the parallel system. These message boards can be considered the data load and the agents themselves the computational load. However, although there may be some imbalance in computational load, with a reasonable initial data - agents - distribution, any imbalance should be small. The crucial element in the parallel

implementation of FLAME is the distribution of the message boards over the system and their synchronisation.

It is clearly essential that the FLAME infrastructure does not impose a high overhead on the application. The *Timer* package has been use to estimate the overhead of the FLAME infrastructure. The most important task performed by the FLAME framework is message and message board management. Although applications make use of a variety of message board function - writing and reading messages from message boards - the message board synchronisation process is potentially the most costly.

Table 4 gives details on the time taken by the FLAME framework to perform this synchronisation for part of the EURACE Model [2]. Each row in the table gives the synchronisation time as a percentage of the total elapsed time per iteration for the most significant message boards. The columns show the basic synchronisation time with and without overlapping. Even when combined it is clear the all the message board synchronisations are only taking 5% of the total run time. Although we would expect to reduce this through some more detailed optimisation of the synchronisation algorithms and code, 5% is considered to be an acceptable overhead.

	No overlap		Overlap	
Message board	Node 0	Node 1	Node 0	Node 1
order	3.3	2.9	3.0	2.2
loan_conditions	0.1	0.2	-	-
info_firm	< 0.1	< 0.1	-	-
order_status	< 0.1	< 0.1	< 0.1	< 0.1
bank_account_update	< 0.1	-	< 0.1	< 0.1
eurostat_send_macrodata	-	1.2	-	-
vacancies2	-	-	< 0.1	-
capital_good_request	-	-	< 0.1	< 0.1
capital_good_delivery	-	-	-	< 0.1

Table 4: Percentage of total time spent synchronising top nine message boards

In the design of FLAME care has been taken to overlap communication and computation so that, as far as possible, agent functions are not waiting for data during their activation. This has been achieved by FLAME analysing the sending and receiving of messages defined in the model XMML file to push the start of communication (call to `MB_SyncStart()`) as high as possible in the call tree (i.e. just after all messages of a particular type have been sent to the message board) and the wait for communication to complete (call to `MB_SyncComplete()`) as low as possible (just before the messages are required).

To illustrate the effect of overlapping communication and computation a special version of FLAME that placed the calls to `MB_SyncStart` and `MB_SyncComplete` as successive operations was run alongside the usual version. The timer package was used to time the `MB_SyncComplete` functions in both cases and the results are given for the 5 longest elapsed times in Figure 16. Runs were for a population of around 25,000 agents distributed over 2 processing nodes and the simulations ran for 40 iterations.

It is clear that the `eurostat_send_macrodata` message board is a significant beneficiary of overlapping. On the top right bar chart (Node 1 with no overlapping) this message board takes a significant time to synchronise. This cost disappears in the overlapped version (bottom right). By a careful analysis of the model's state graph the messages sent by the Eurostat agent are sent very early in an iteration and read by the Government agents very late in the iteration. This gives the system plenty of time to complete the synchronisation process. In the same way it is clear that the `order` message board benefits less since order messages are required very soon after they are sent.

Part of the optimisation process performed by FLAME is the re-organise tasks in the model's

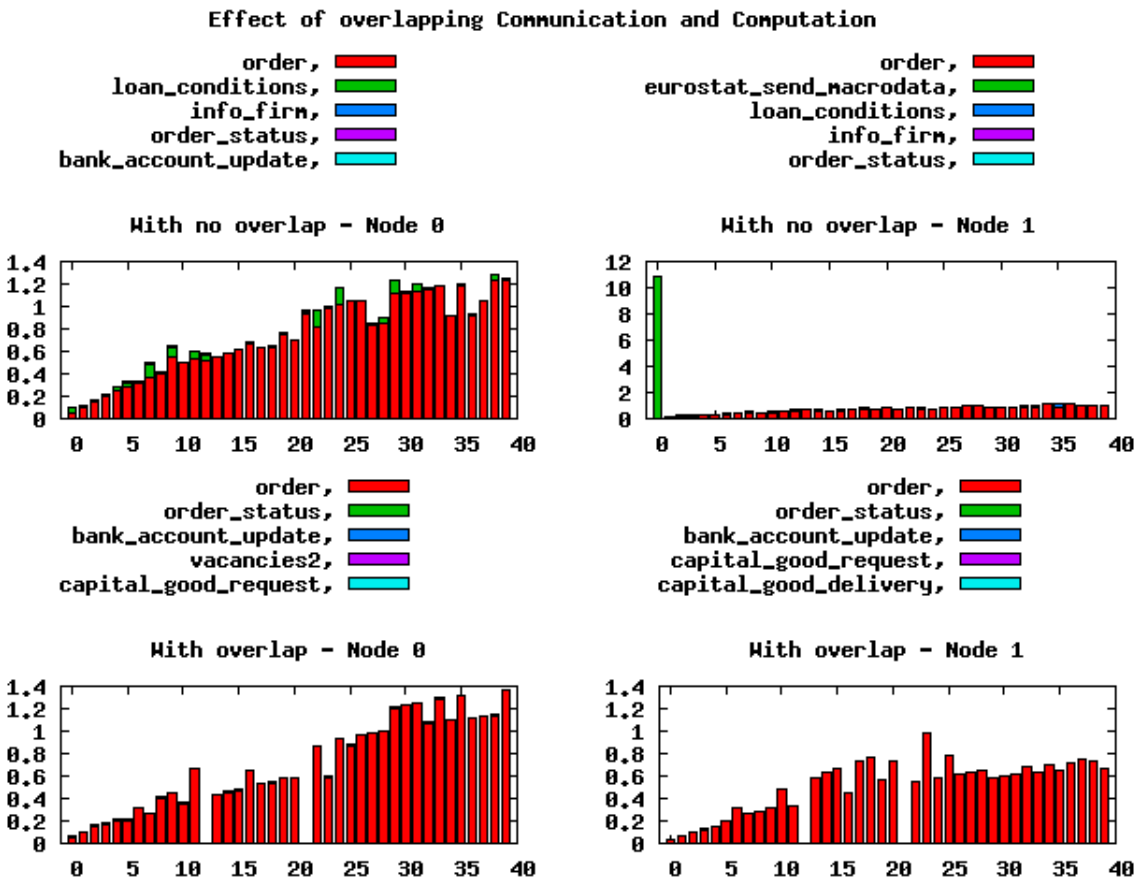

Figure 16: Elapsed times (s) for message board synchronisations with an without communication/computation overlapping

dependency graph in order to maximise the separation of the start and completion of the message board synchronisations.

9 Testing serial and parallel implementations

Verification and validation of FLAME and its parallel implementation is again made difficult by its nature. We must verify and validation FLAME itself and we must also verify in some way the application generated by FLAME. It should be noted that FLAME has two distinct parts: the `xparser` which generates the application from the models XMML and C code files and *The Message Board Library - libmboard* which is the underlying infrastructure that manages the inter-agent communications. *libmboard* also provides an application to any parallel hardware through the MPI message passing interface.

libmboard has is own set of unit tests and test programs. It has been developed using an agile test driven methodology. These are documented in the *libmboard* documentation. Similarly the `xparser` has its own set of tests which are detailed in other reports.

For the developers we need to verify that FLAME is generating the model specified in the XMML and C code and that the execution of the generated application is *correct*. Throughout the project we have gather a number of test examples which help verify the FLAME implementation. These test examples are model definitions and their associated C code.

We have started to provide a set of *simple* problems that enable us to do this. They need to be simple for the only way of checking that the code in correct is by very careful walk throughs. Their main characteristics being they exercise the FLAME infrastructure and we can determine the expected results of the simulation. By using these types of simple models we are able to

verify that both the serial and parallel versions of the FLAME generated applications are *correct* - in as much that they produce the expected results.

It is important to ensure that applications generated by the FLAME framework execute *correctly* in both their serial and parallel modes. Because of the stochastic nature of the agent-based approach to modelling it is unrealistic to expect complex simulations to following exactly the solution path although general trends should be similar. However for some simple applications we can expect the serial and parallel implementations to produce exactly the same results throughout the simulation. Such example applications can be used to verify the correctness of both the serial and parallel implementations.

The *Circles Model* is one such application. The *Circles* agent is very simple. It can be viewed a small elastic circular ring of specified radius. Each agent will interact with its neighbours by deforming their boundaries and responding to a simple 'spring' like force. So given a sufficient simulation time any initial distribution of agents will tend to a steady state as the agents move away from each other until they are fully separated. Each agent has in memory its position, x,

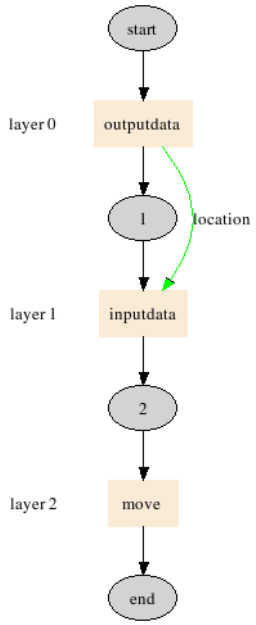

Figure 17: The *Circles* agent dependency graph

y, the current forces acting on it fx, fy and its *radius* of influence. The agent has three states: *outputdata*, *inputdata* and *move*. The agents communicate via a single message board, *location*, which holds the agent *id* and position. Figure 17 shows the dependency graph of the Circles agent. The force function use in the model to calculate the change of position is:

$$F^x_{uv} = \sum_{(u,v)} k_{uv}[d(p_u, p_v) - l_{uv}]\frac{x_u - x_v}{d(p_u, p_v)} \tag{1}$$

where k_{uv} is the *spring* strength $d(p_u, p_v)$ the distance between the two agent centres and x_u and x_v the x position coordinates of the centres. l_{uv} is the resting (undeformed) separation distance between agent centres. There will be a similar express for the F^y_{uv} term.

Given the simplicity of the agent it is possible to determine the final result of a number of ideal models. A set of simple test models and problems have been developed based on the *Circles* agent. Each test has a `model.xmml` file and a set of initial data (`0.xml`).

Test 1 : Model: single *Circles* agent type; Initial population of no agents.

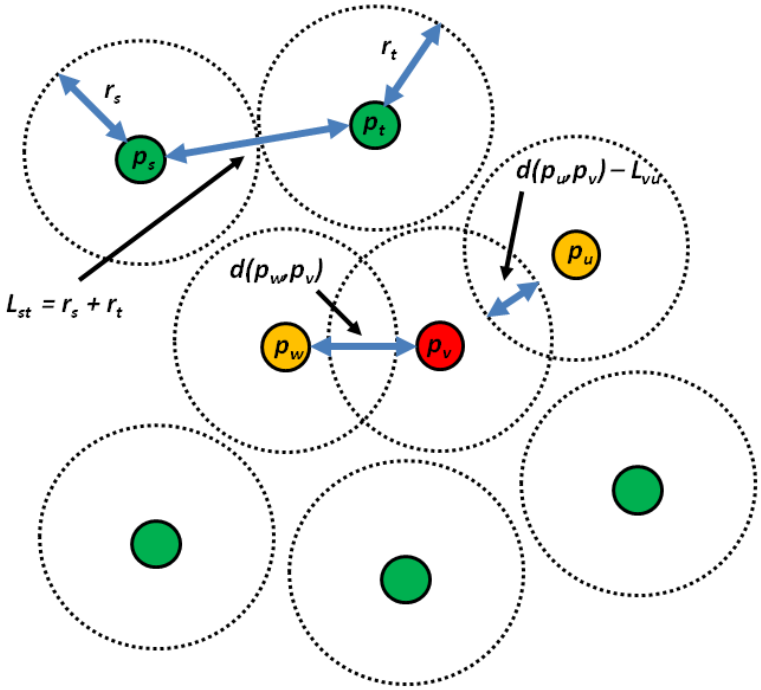

Figure 18: Particle interactions

Test 2 : Model: single *Circles* agent type; Initial population of one agent at (0,0).

Test 3 : Model: Two *Circles* agent type; Initial population of agents at (-1,0) and (+,0).

Test 4 : Model: Four *Circles* agent type; Initial population of one agent at ($\pm$1,$\pm$1).

Test 5 : Model: Four *Circles* agent type; Initial population of one agent at (0,$\pm$1) and ($\pm$1,0).

Test 6 : Model: Four *Circles* agent type; Initial population of one agent at random positions.

In each of these models the expected results can be specified and therefore they provide a very simple check of the implementation.

The *Circles* agent also provides a good mechanism to check the parallel implementation against the serial. Such is the nature of the model, the positions of the agents at each iteration of the simulation is independent of the order of calculation. As the order of calculation can not be easily prescribed in the parallel simulation we can use this characteristic to test the validity of the parallel implementation against the serial. We would expect to get the identical positions for each agent at every iteration of the simulation.

In the general testing of FLAME using these test models the results produced were as expected. The final initial test was to use a larger population of *Circles* model. Figure 19 shows the initial data of a circles simulation for a population of 2000 agents and their positions after 500 iterations. As expected we can see that the population gradually spreads out towards and steady state during the simulation. Identical results were reproduce by both the serial and parallel versions of the model once the random seeds had been set to the same values.

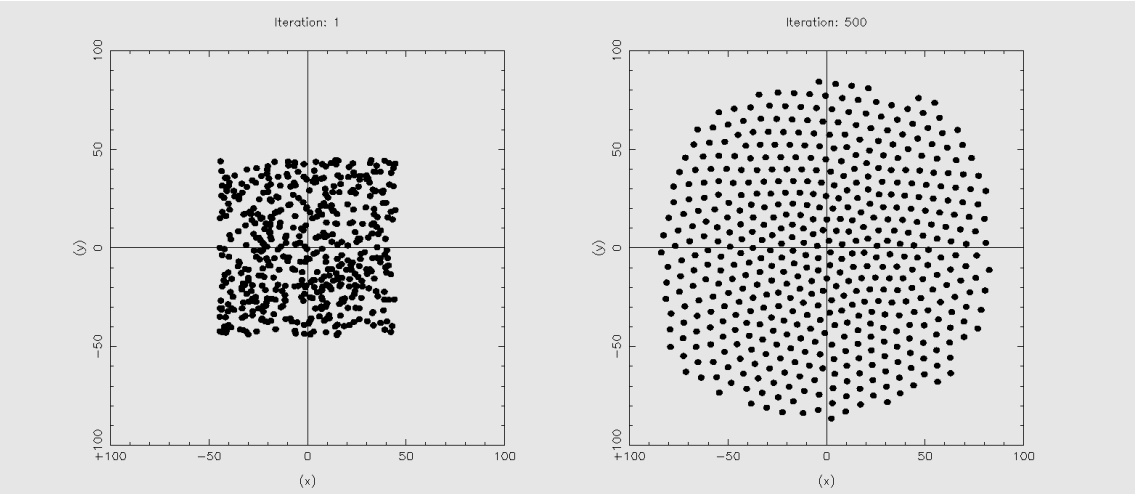

Figure 19: Positions for *Circles* agent population: Initial (left) and after 500 iterations (right)

10 Parallel Performance

The final section of this report considers the parallel performance of FLAME. We will only give some overall results from a number of computational experiments. In general we fix the initial agent population size and note the overall execution time as the number of processors increases. Five different models were used in this exercise. These models are characterised in Tabel 5. The experiments were performed on a variety of distributed memory machines and in

Model	Agents	Messages	Population
Circles	1	1	10^5
C@S	3	9	124,000
Labour Market	4	10	110,101
Bielefeld	4	29	4,3100
EURACE	9	62	101,044

Table 5: Details of Initial Benchmark Models

each case the initial agent population was partitioned to produce a reasonable load balance over the processors.

We will not go into the detail of the hardware architectures of the systems used save to say that they are generally very different and have different interconnects and balances.

Figure 20 shows the results for the basic test model - *Circles*. The agent population was partitioned using the *geometric* method which is appropriate for the type of position dependent agent. The *Circles* model was run on five different HPC systems with the number of processors ranging from 1 to 100. The graph presents the time taken to perform a single iteration of the model. As can be seen for this population of 10^5 agents there are considerable improvements in performance up to around 40 processors. From that point the improvements in performance decrease rapidly.

The other test examples were all developed during the EURACE project and are a gradual development of an economic model. The second test example is a simple two-agent model which simulates

The final test model comes from the EURACE project and is the most complex FLAME model so far constructed. It contains 9 agents types and has been run with populations of over 500,000 agents.

Analysis of parallel performance includes profiling the agent functions as for the serial case but also investigating how the EURACE application performs as the number of processes is

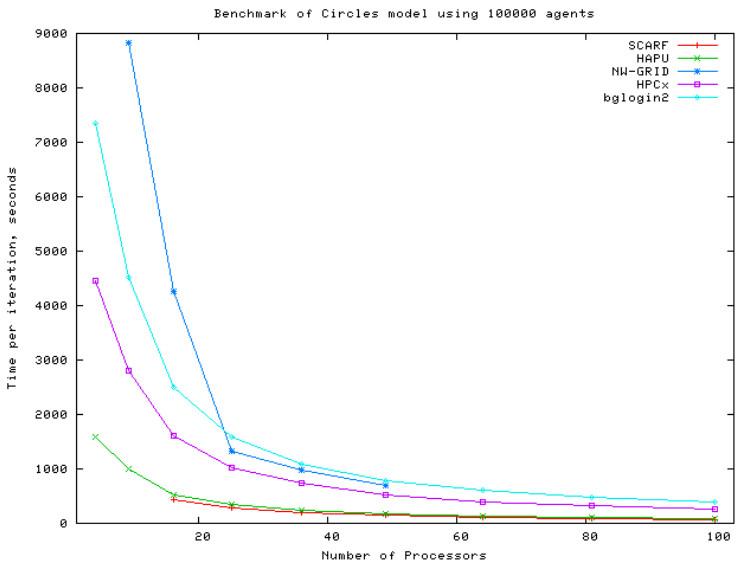

Figure 20: Results of *Circles* model

increased for a fixed initial population.

In general the results showed promising parallel speedup using up to 40 processors.

11 Conclusions

In this report we have described in detail the parallel implementation of the Flexible Large-scale Agent-based Modelling Environment (FLAME) and its assessment together with some benchmarking results using the *Circles* and EURACE models. We have also demonstrated FLAMEs use on populations ranging from a few hundreds of agents, through tens of thousands to, in one case, a million agents. In some of these simulations the parallel implementation of FLAME has shown reasonable scalability and parallel efficiency but in other the results have been disappointing.

Communication and synchronisation between computational modes is seen as one major difficulty in the parallel performance in general. These are being addressed in a re-design of FLAME [17] by developing more sophisticate filtering functions within the framework together with better neighbour representations and searching.

An important goal of the EURACE project has been to perform, in parallel, a large simulation using the EURACE Model. The project achieved this to a degree: the model has been defined, important parameters have been values, a method of generating agent populations implemented and a parallel implementation of the EURACE model can be generated by FLAME. Using these steps serial and parallel simulations of the EURACE Model have been perform. In this process a detail assessment of the FLAME generated code, the serial and parallel implementations and the EURACE Model have been performed. Message counts, function times and synchronisation times are a few of the measures that have been used together with a detail static analysis of the model to identify the performance deficiencies in both the FLAME framework and the EURACE model.

All this analysis has lead to improvements in FLAME which in general have improved its computational performance. However the presence of substantial serial components in any of these models has resulted in very poor parallel scalability. It is well known that parallel speedup is limited by the serial faction of a code - this is Amdah's Law. The analyses performed on the EURACE Model have shown that the singleton agents have a significant impact of the parallel performance.

In the re-design of FLAME we are addressing the problems presented by singleton agents

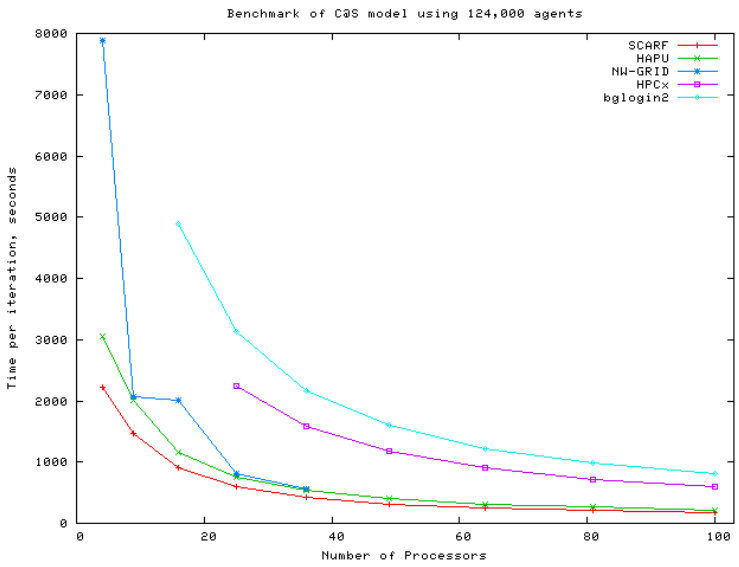

Figure 21: C@S Model

through a more fine grain decomposition of the agents into tasks which form the basis a new approach to scheduling the agent state changes. The initial results from this re-design and re-implementation should be report over the next few months.

Acknowledgements

Much of this work was carried out in conjunction with the EURACE project (EU IST FP6 STREP grant 035086) which is a consortium lead by S. Cincotti (Universit di Genova), H. Dawid (Universitaet Bielefeld), C. Deissenberg (Universit de la Mediterranee), K. Erkan (TUBITAK National Research Institute of Electronics and Cryptology), M. Gallegati (Universit‘ Politecnica delle Marche), M. Holcombe (University of Sheffield), M. Marchesi (Universit‘ di Cagliari), C. Greenough (STFC - Rutherford Appleton Laboratory).

References

[1] Coakley (2005) ”Formal Software Architecture for Agent-Based Modelling in Biology”, PhD Thesis, University of Sheffield

[2] EURACE (2006) ”Agent-based software platform for European economic policy design with heterogeneous interacting agents”, EU IST Sixth Framework Programme.

[3] C. Greenough, DJ Worth, LS Chin, M. Holcome and S Coakley (2009), Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework, Rutherford Appleton Laboratory Technical Report RAL-TR-2009-022, Jul 2009

[4] Mangina (2002) ”Review of software products for multi-agent systems”, Agent-Link, http://www.AgentLink.org, July 2002

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

[6] Repast home page - http://repast.sourceforge.net/.

[7] MASON home page - http://cs.gmu.edu/Beclab/projects/mason/.

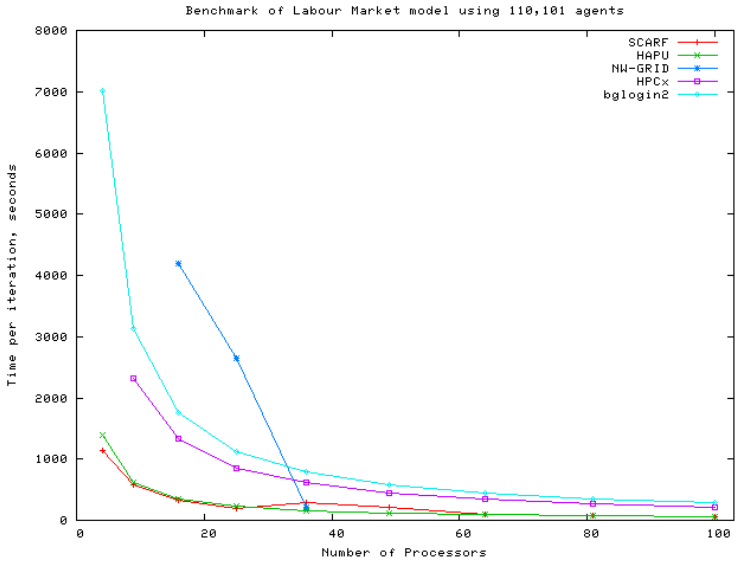

Figure 22: Labour market Model

[8] R.J. Allen (2009) Survey of Agent Based Modelling and Simulation Tools, STFC Daresbury Laboratory Technical Report

[9] A Chaturvedi, J Chi *et al* (2004) SAMAS: Scalable Architecture for Multiresolution Agent-Based Simulation. In: M. Bubak et al. (eds.): ICCS 2004,LNCS 3038, Springer.

[10] `http://jade.tilab.com/`

[11] D. Pawlaszczyk1 and I. J. Timm (2007) "A Hybrid Time Management Approach to Agent-Based Simulation" Lecture Notes in Computer Science, Springer Berlin, ISSN 0302-9743, Volume 4314

[12] L. Gasser and K. Kakugawa (2002) "MACE3J: fast flexible distributed simulation of large, large-grain multi-agent systems" International Conference on Autonomous Agents, Bologna, Italy

[13] P. Riley (2003) "SPADES a system for parallel-agent, discrete-event simulation" AI Magazine, Volume 24 , Issue 2

[14] S. Eilenberg (1974) Automata, Languages and Machines, Vol. A. Academic Press, London.

[15] User Manual - FLAME?

[16] LS Chin (2008) "libmboard Reference Manual (Version pre-0.1.5)", August 2008.

[17] LS Chin, DJ worth, C Greenough, S Coakley, M Holcombe and M Gheorghe (2012) "Redesigning FLAME for better parallel performance", to be published as a Rutherford Appleton Laboratory Technical Report.

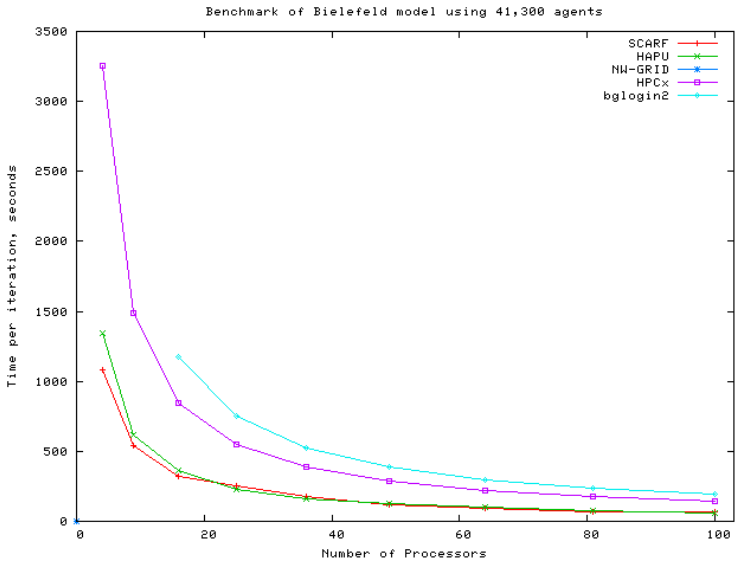

Figure 23: Bielefeld Model

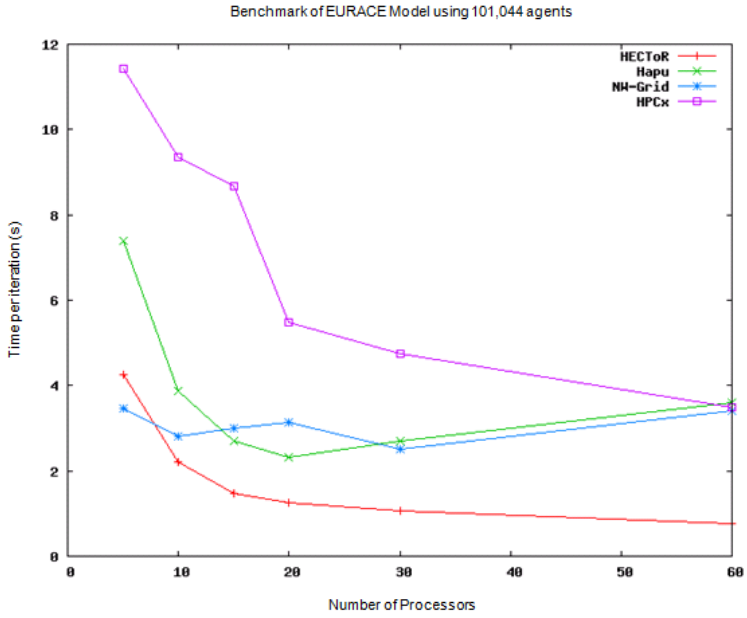

Figure 24: EURACE Model