Project no.
035086
Project acronym
EURACE
Project title
An Agent-Based software platform for European economic policy design with heterogeneous interacting agents: new insights from a bottom up approach to economic modelling and simulation

Instrument: STREP

Thematic Priority: IST FET PROACTIVE INITIATIVE "SIMULATING EMERGENT PROPERTIES IN COMPLEX SYSTEMS

Deliverable reference number and title
D1.4: Porting of agent models to parallel computers
Due date of deliverable:
31/08/2008
Actual submission date:
31/08/2008

Start date of project: September 1^{st} 2006 Duration: 36 months

Organisation name of lead contractor for this deliverable
STFC Rutherford Appleton Laboratory - STFC

Revision 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contents

Abstract

Making use of high performance computers in agent-based simulation is a complex and difficult task. This report describes the approach being explored within the EURACE project to exploit parallel computing technology in large-scale agent-based simulations involving millions of agents. The underlying software is the FLAME framework and the report will give an overview of the features and use of FLAME and discuss the implementation of techniques that attempt to exploit large parallel computing systems.

Some early simulation results will be presented together with a discussion of the requirements for efficient parallel simulations and the performance of the current implementation. Although these results demonstrate that the parallel implementation works and is portable between a number of systems the efficiency is quite poor as the full implementation of message filtering is not yet complete and filtering is not yet fully exploited within the EURACE models.

Finally the report also gives an indication of the work planned for the final year of the project which is focused around improving the parallel performance of the basic implementation. The optimisation and effective use of the message filtering will be an important element of this work.

1 Introduction

In this report we describe the parallel implementation of the FLAME Framework [13]. We cover some of the reasons for parallelisation and also give a detailed description of the approach being adopted. We also present some initial results from a set of benchmarks which hope to capture the computation and communications loads inherent in the EURACE application.

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

Before considering the current parallelisation of FLAME it is worth considering the characteristics of a software system than make it worth taking the time and effort to parallelise. Some characteristics of when to parallelise:

- Code is practically incapable of running on one computer, memory requirements too great, run time too long
- Code will be reused frequently - parallelisation is a large investment
- Data structures are simple, calculations are local, easy to communicate and synchronize between processors

The converse should also be considered. When not to parallelise a code:

- Code will only be used once (or infrequently) - An efficient parallel code takes time to develop!
- Current performance is acceptable and execution time is short
- There will be frequent and significant code changes

It should also be remembered that some algorithms simply do not parallelise.

It is clear that developing a scalable agent-based framework will be difficult. As mentioned above there are few examples none of which attempt to utilise the power of high performance systems such the Cray XT4 or the IBM BlueGene or the multitude of Beowulf type systems being offered by vendors. Agent systems such as SAMAS [7],JADE [3], SIMJADE [4], MACE3J [2] and SPADES [1] have been used to demonstrate scalable agent computing but these have been relatively small simulations.

The starting point of FLAME is different from these systems - high performance computing was thought to be essential and thus the implementation language is C.

2 General Parallel Implementation of FLAME

The FLAME architecture has some inherently good characteristics that lend itself to parallelisation. Unfortunately, it also has a number of bad characteristics. In the context of the EURACE project we believe that the good out-weighs the bad with the flexibility, size, and time to solution being foremost among the good.

In a fully connected and communicating agent population, interaction may not be local but long range leading to many-to-many, inter-node communication which drastically impacts the scalability and simulation time. However, in many applications of FLAME (including EURACE), we have seen that there is sufficient locality (that can be taken advantage of) to consider parallelisation taking into account the general population sizes.

The use of simple read/write, single-type message boards allows the framework implementer to divide the agent population and their associated communications areas. This division could be based on any number of parameters or separators but the simplest to appreciate is position or locality. If, as in EURACE, agents are people or companies for example, they will have locality defined either as location or by some group topology. It is also reasonable to assume that the dominant communications in both scenarios will be with neighbouring agents.

As explained above, FLAME uses a collection of message boards to facilitate inter-agent communication. 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 architecture. The parallelisation of FLAME utilises partitioned agent populations and distributed message boards linked through MPI communication. Figure 1 shows the difference between the serial and parallel implementation.

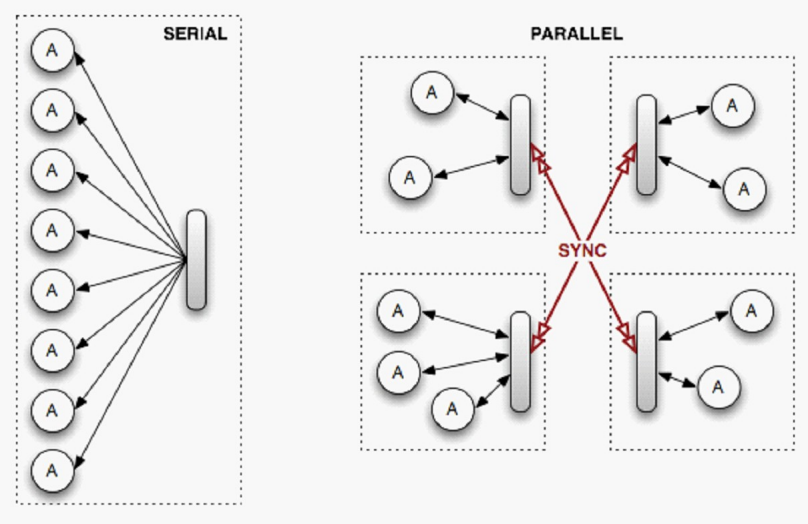

Figure 1: Serial and Parallel Message Boards

The most significant operation in the parallel implementation is providing the message information required by agents on one node of the processor array but stored on a remote node of the processor. The FLAME Message Board Library manages these data requests by using a set of predefined message filters to limit the message movement. This process could be considered a synchronisation of the local message boards within an iteration of the simulation. This synchronisation essentially ensures that local agents have the message information they need as the simulation progresses.

An additional advantage of implementing parallelism in FLAME through the Message Board Library is that development of the FLAME framework and the message board algorithms can continue independently to a great extend as the Message Board API defines the interface between the two elements of the code. This should enable the message board routines to be developed and optimised without major re-engineering of the framework.

The two main areas of algorithmic and technical development needed to achieve an effective parallel implementation are load balancing and communications strategy.

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 reference to 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 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. These problems are under investigation.

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 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. Within FLAME communication between agents is managed by the Message Board Library, which uses MPI to communicate between processors. The Message Board Library implementation attempts to minimise this communication overhead by overlapping the computational load of the agents with the communication.

Where the agents have some form of locality the initial distribution of agents makes 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.

Within the EURACE Project a parallel version of FLAME has been developed using these ideas and the sections below discuss some of the initial results in performing parallel simulations.

3 Detailed Description of Parallelisation

3.1 Overview

Within a FLAME simulation, every agent only interacts with its environment via the reading and writing of messages to a collection of Message Boards. This makes the Message Board component an ideal candidate for enabling parallelism.

Using distributed Message Boards, agents can be farmed out across multiple processing nodes and simulated in parallel while coherency of the simulation is maintained through a unified view of the distributed Boards.

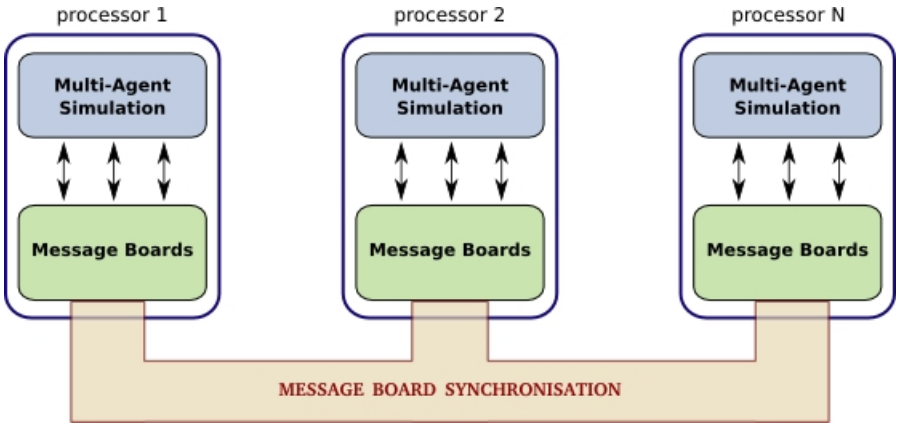

Figure 2: Parallelisation of FLAME using distributed Message Boards

3.2 The Message Board library

In the recent code release, the Message Board was decoupled from the FLAME framework and implemented as a separate library. This provided the flexibility to experiment with different parallelisation strategies while minimising the impact on current users of the FLAME framework.

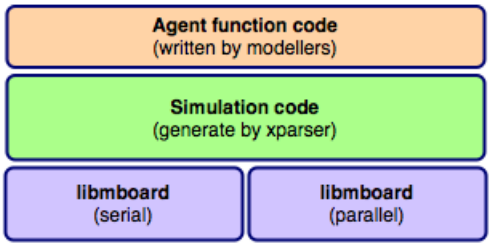

Figure 3: Users can create either serial or parallel executables by linking their object files to the appropriate *libmboard* library

The Message 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.

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. This level of flexibility is crucial in the EURACE project as *libmboard* is being developed in tandem with modelling activities that require a working implementation.

Within the FLAME framework, the *libmboard* API is used only by the framework-generated routines and not directly by the modellers. Modellers are provided with model-specific routines that hide the complexity of managing boards and packaging data into suitable datatypes. Figure 4 shows an example of this whereby an agent's message add request gets translated into a *libmboard* API call.

libmboard uses the Message Passing Interface (MPI) to communicate between processors, and POSIX threads (pthreads) to manage a separate thread for handling data management and inter-process communication. Further details are available in Section 3.3.3 and Section 3.4.

3.3 The *libmboard* API

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.

In the following sections we discuss the use of Boards, Iterators and Synchronisation. For further details, refer to the *libmboard* Reference Manual [26] which includes the full API specification and usage example.

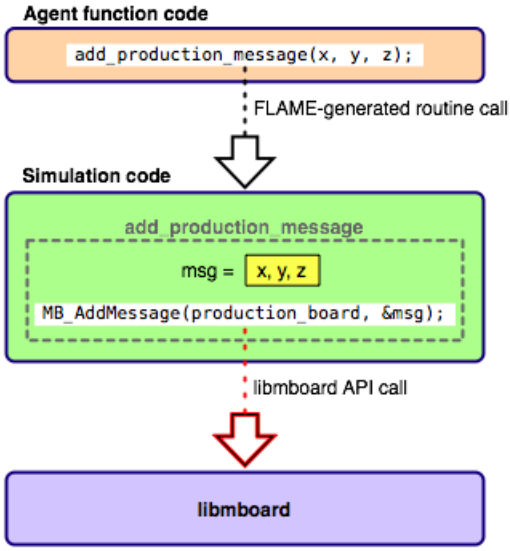

Figure 4: Modellers use FLAME-generated routines which get translated to the appropriate *libmboard* API call

3.3.1 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 3.3.3), across all processing nodes.

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

3.3.2 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.

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

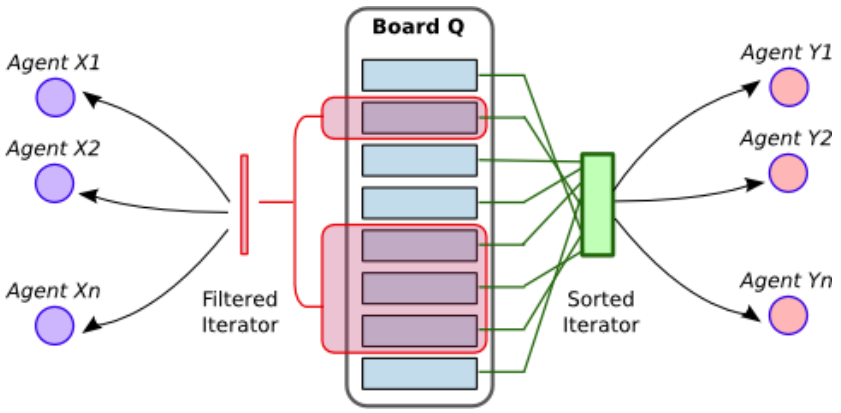

Figure 5: Using specialised Iterators to achieve different access patterns

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.

3.3.3 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 3.4.1). 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 6 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 3.4.

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

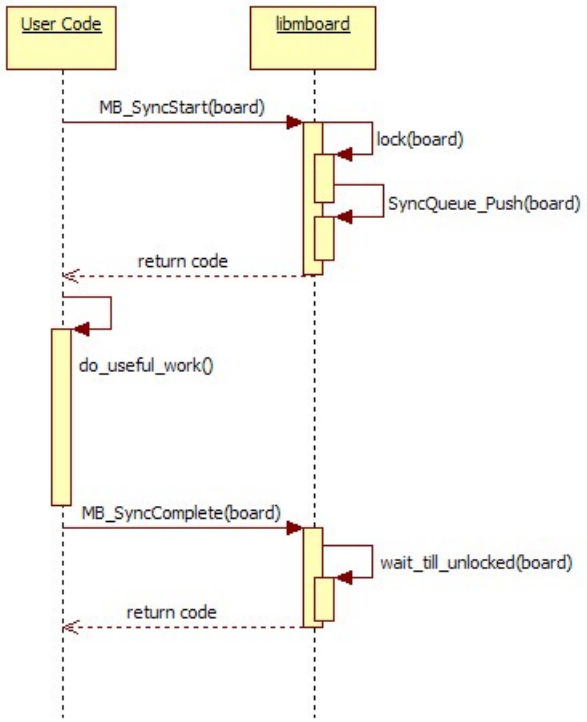

Figure 6: Other work can be scheduled during board synchronisation to hide the overheads of communication

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

3.3.4 Sync optimisation based on Message Tagging

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 7). 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.

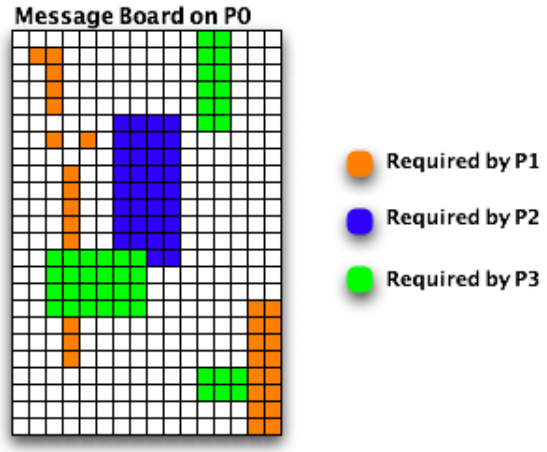

Figure 7: Full data replication is avoided by only propagating tagged messages to the relevant nodes

3.4 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 8.

3.4.1 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*.

3.4.2 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.

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

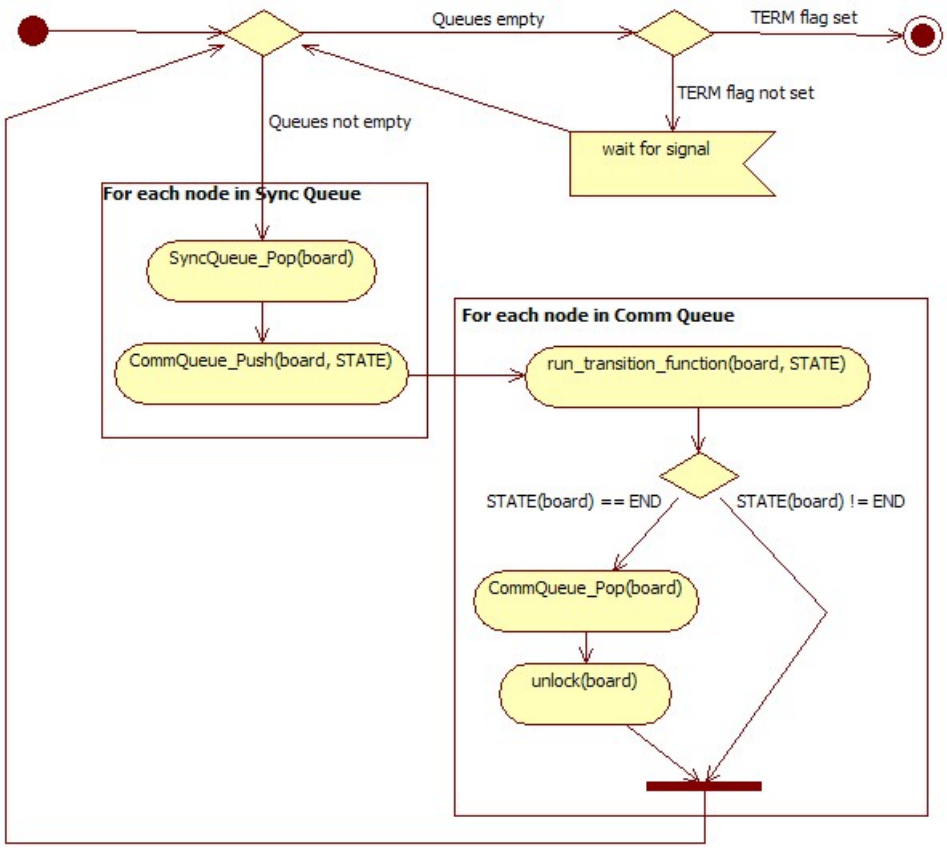

Figure 8: Activity diagram for Communication Thread

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 thread when the appropriate *libmboard* API routines are called (refer back to Figure 6 for further clarification).

3.4.3 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 9), 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.

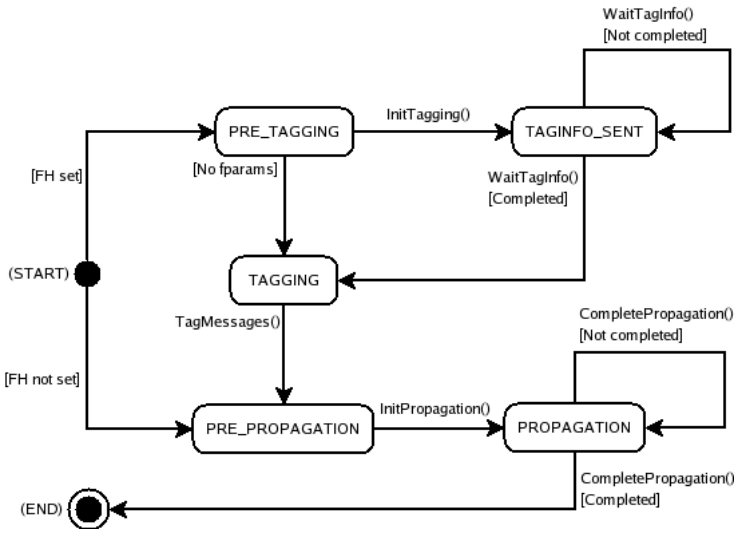

Figure 9: State diagram for processing Comm Queue nodes

3.5 Issues

3.5.1 Use of *pthreads*

During the design stage of *libmboard*, several complications were identified when considering the use of *pthreads* for implementing the *Communication Thread*. It was eventually decided that the possibilities brought about by using a threaded model outweighed the potential drawbacks. Some of the issues that were recognise are:

- 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.

3.5.2 Ensuring uniform assignment of Handles

All opaque objects (Boards, Iterators, and Registered Functions) are assigned integer-based Handles upon creation. To enable collective operations (such as board synchronisation), instances of the same object must be allocated the same Handle on all processing nodes. Coordinating the assignment of Handles to object instances across all nodes would therefore require additional

communication and possibly some form of processing barrier to ensure all nodes arrive at the same point in the code before proceeding. The overhead that this would incur is non-trivial, especially for more substantial simulations where numerous objects need to be created across large numbers of nodes.

In the interest of performance, we address this problem by delegating some of the responsibility to the FLAME framework. The framework is expected to generate simulation code that issues object creation calls in the same order on all nodes, while *libmboard* ensures that Handles are issued in the same sequence on all nodes.

libmboard provides a debug version of all libraries which performs additional checks to ensure that this condition, among many others, is met. FLAME framework developers and modellers are therefore expected to use the debug version for all development and validation work. The production version, which has all these checks removed, should only be used once the simulation code has been tested and validated.

3.5.3 `<filter>` tags not widely used

As mentioned previously in Section 3.3.4, *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.

However, the `<filter>` tag was introduced quite recently along with the new specification of XMML and is therefore yet to be widely adopted in models. While this does not affect the correctness of the simulation, the full potential of the Message Board library will not be realised. We therefore expect the performance of the current versions of FLAME generated models to be worse than previous versions (as seen in the benchmark results listed in Section 8.1).

Additionally, without any available models (of realistic size and complexity) making full use of *libmboard*, there is a lack of empirical data that can be used for analysing and optimising the performance of the library.

This issue will be addressed in the near future by actively encouraging the uses of the `<filter>` tag in models and optimising the translation of the tags to API calls within the FLAME framework.

3.5.4 Temporary buffers for MPI routines

Every MPI send and receive operation requires a contiguous block of memory as its buffer. These buffers need to be allocated and maintained throughout the duration of the operation. As *libmboard* is capable of synchronising numerous Boards simultaneously, the amount of buffer space that is required could lead to substantial overheads in terms of memory usage and time taken to allocate and deallocate memory.

This problem is partially alleviated by the fact that memory allocation and deallocation are performed by the *Communication Thread* and the elapsed time could therefore be overlapped by that of useful computation.

Further research and optimisation are in the pipeline to improve memory usage. These include implementing an adaptive synchronising algorithm that selects the best communication pattern based on memory usage analysis, and improved agent distribution that minimises communication.

4 Initial Data Partitioning

4.1 Introduction

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.

4.2 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 region or country id, but the overall 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.

4.3 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 partitioning is implemented in FLAME.

It is possible to extend this method by using the agent 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. (This could also be seen as a special form of separator partitioning.)

4.4 Other Benefits

As well as hoping to improve the load balance, partitioning of the initial data will mean that each node in the compute system will have to read only its data. In the current version of FLAME one node has to read all the data, decide on the partitioning and then distribute the data with which agents can be partitioned. Then all the nodes have to read all the initial data picking out only those agents that fall within their partition.

5 Detail on Dynamic Load Balancing

5.1 Dynamic Load Balancing Overview

The overall aim of parallelising the FLAME framework is to reduce the wall clock time for running a simulation. This relies on efficient parallelisation of communication and keeping the work load balanced between computing nodes. The message board library addresses the first of these and dynamic load balancing addresses the second.

To illustrate the problems in getting load balancing right, the diagram in Figure 10 shows agents on two nodes and their communication patterns. The top portion shows an unbalanced number of agents but the frequent communication is internal to each node with only occasional communication between the nodes. If agents are moved in an attempt to balance the load then frequent communication between nodes is introduced (lower portion of figure) which could mean a large increase in communication time and hence wall clock time. This example shows that measurement of communication between nodes must be part of the load balancing algorithm as well as elapsed time for various parts of the framework.

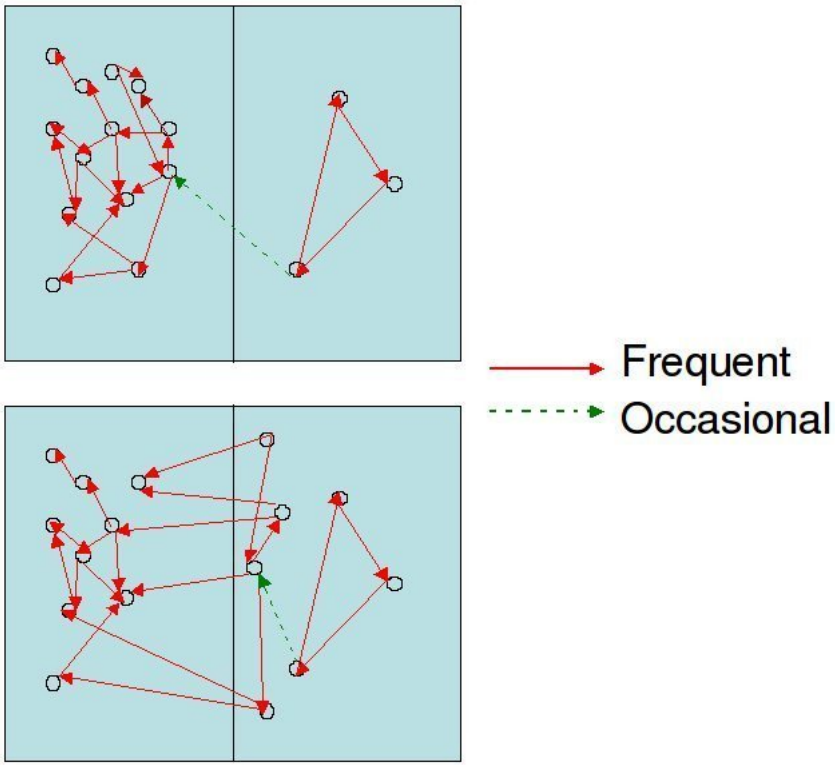

Figure 10: Illustrating some problems of load balancing

The dynamic load balancing library will therefore have to track data from two sources:

- elapsed time for various parts of a FLAME run,
- data communication pattern and volume between nodes.

The timing data will show problems with computational load balance, where some nodes are idle and others working; and the communication data will show imbalance in the message flow between nodes, where some nodes send a lot of data and others very little. Timing data will also show whether it is the computation by the agents or time for communication that

dominates the simulation. Attention can then be focused on improving load balance or improving communication balance as required.

An iteration of the model in FLAME proceeds in stages as agents move from one state to the next, therefore it will be necessary to look at data from each stage as well as the overall data for one iteration. If there is always an imbalance in load or communication for the same subset of nodes for all stages then it is easy to say that redistributing agents is necessary. However if the imbalance shows up for different nodes at different stages then the strategy for load balancing is more difficult to decide upon.

Agents within FLAME are allowed to have dynamic memory variables (i.e. their size is not determined when the model is compiled but during the simulation) and so writing code to pack an agent's memory ready for transfer to another node is very difficult and execution of that code will be a large overhead in the execution of the model. As a consequence it is not envisaged that agents will be moved at every stage of an iteration. The dynamic load balancing framework will monitor the imbalances and will only decide to move agents if an imbalance is "too great". Experiments with the EURACE models will help determine what is meant by "too great" and what strategies can be adopted when seeking to achieve the best distribution of agents.

Since agents do not communicate directly with other agents but via message boards, the communication data will be the volume of message data sent from one node to another. This does not allow for identification of individual agents that may be causing problems but, by analysing the messages that are sent by a certain type of agent at the time of the imbalance it may be possible to identify a set of agents of a certain type whose redistribution could help.

Implementation has started with a timer which allows developers to insert timing into any section of FLAME, from the framework itself to the user's model functions. The details of the implementation are described in the next section.

5.2 The *timer* Package

Timers will be used to measure the elapsed CPU time for portions of the running code and this data will be used as input to the load balancing strategy used in the FLAME framework. The initial requirements against which the timer package was implemented are given below.

- Can have multiple timers running simultaneously
- Timers can be identified individually
- Functions to start/stop/reset a named timer
- Function to get elapsed time from a named timer
- Definition of a set of timers.
- Functions to get statistics from a set of timer
- Turn timing on/off during program execution

We have implemented all the functionality for individual timers and a set of unit tests and an example program using the timers. Code is stored under Subversion source code control in the FLAME project on the CCPForge site.

User documentation is supplied in [25].

5.3 Timing Results

We have demonstrated the use of timers in the simple circles model by timing the work done by agents as the number of partitions over which the agents are distributed increases. The agents were not uniformly distributed in space and so, with geometric partitioning, some partitions will have more agents than others. The time taken by the agents on each node is plotted against the number of agents on the node in Figure 11 and we can see that there is a direct relation between the number of agents and the work done on each node. From this we can conclude that distributing the agents equally over the nodes will give a good load balance.

There is a cautionary note to be struck however. The timing data for circles has illustrated the problem identified in the overview, namely that naively giving equal numbers of agents to each node without taking into account communication can lead to worse performance. Comparing elapsed time for geometric and round robin for 5 partitions in Figure 12 shows that the elapsed time increased even though the work done by the agents decreased.

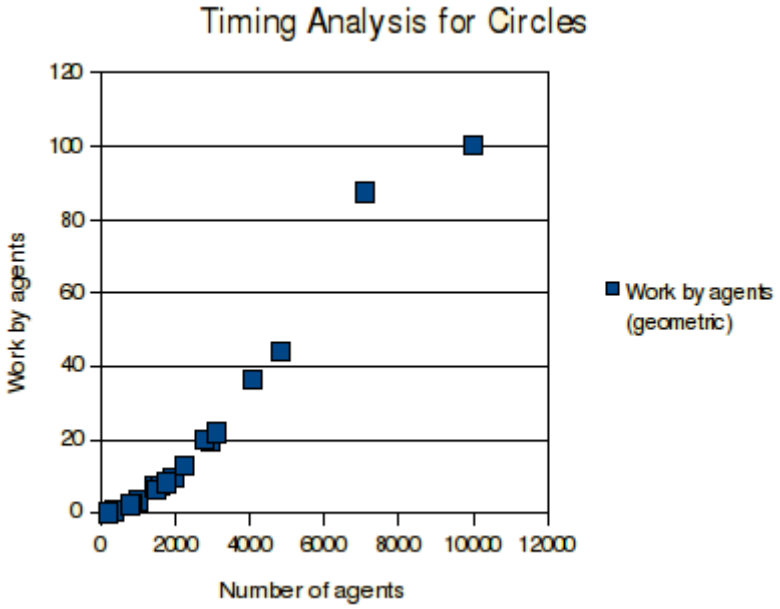

Figure 11: Timings results from Circles model

6 Remote Job Submission

6.1 Introduction

A job submission system for FLAME jobs has been designed and implemented so that it is easy to run EURACE on remote (parallel) machines. It will rely on the user knowing details of how to connect to the remote machine and details of the job scheduling software it uses. This data will be stored in the appropriate format in a machine configuration file. The sequence of steps for job submission was drawn up in discussion with TUBITAK and is as follows:

1. **Check authentication**. Does the user provided information allow a log in to the target machine? Return code for success/failure.

2. **Check FLAME version**. Is the required version of FLAME available on the target machine? If not copy files onto target and install. Return code for success/failure of

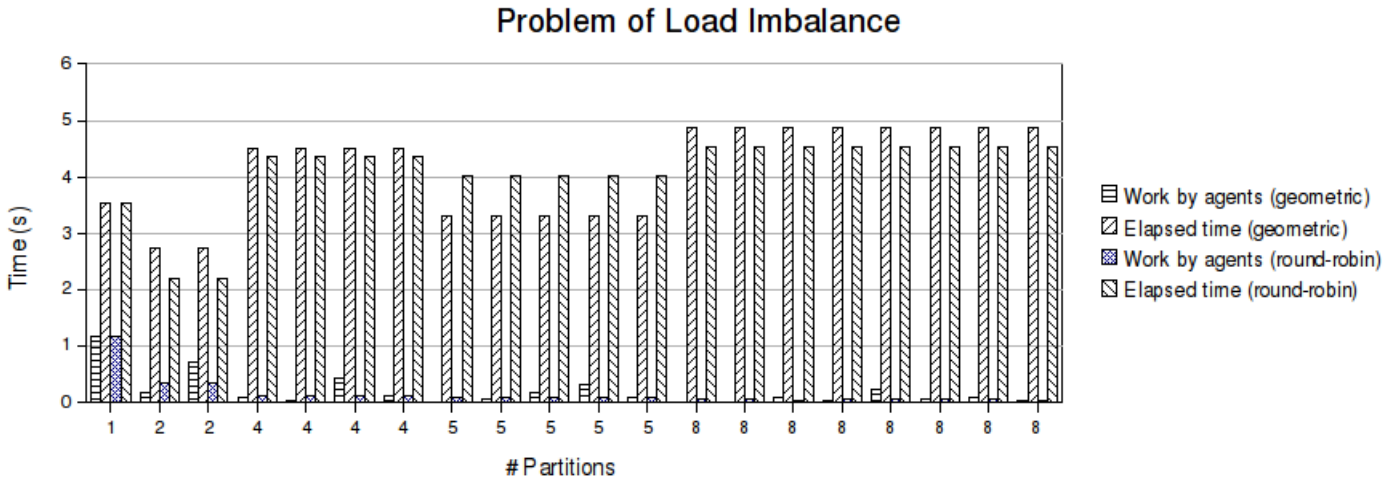

Figure 12: Illustration of problems with load balancing with Circles model

installation or success if installed. Could be some output text to say when FLAME has to be installed.

3. **Create a project**. Send the model XMML file and C code. Parse and compile the model. Return code for parse failure/compilation failure/success. Success return code is the project id.

4. **Submit job**. Send the 0.xml file(s) and project id. Submit the job according to data in the machine's configuration file. Return code code for success/failure. Success return code is the job id.

5. **Query job status**. Send project and job id. Return code pending/done/running/failed.

6. **Query status of all jobs in project**. Send project id. Return code is whether data is returned or not. Return text could be job id, status for each job.

7. **Query status of all jobs in all projects**. Return code is whether data is returned or not. Return text could be job id, status for each job.

8. **Get results**. Send project and job id. Copy results back and gather if parallel. Return code for success/failure.

The details for each of these steps are given later.

Connection to remote machines will be via `ssh` a standard secure connection mechanism which encrypts data between machines, or `gsissh` a grid-enabled version of ssh (part of Globus `http://www.globus.org`) which requires the user to have a grid (X.509) certificate. The scripts will work best if the user arranges for login authentication without a password. For ssh this means generating a public/private key pair (see the Authentication section of `ssh` manual and the `ssh-keygen` manual for details). The public key should be copied to the remote machine and then, using `ssh-agent` as shown below, the operations can be carried out without further authentication.

```
# Get the environment variables for ssh-agent
ssh-agent > file
# Set the variables
```

```
. ./file
# Add the private key to this session. Will require pass phrase for ssh key
ssh-add
# Run the job submission you want. As an example I have put in a simple ssh
ssh user@remote.machine.ac.uk
# Kill the ssh-agent session
ssh-agent -k
```

For gsissh the process is different. The local Grid computing community will have details on obtaining and using a Grid certificate and possibly be able to give advice on installing enough of Globus to use gsissh. It is beyond the scope of this document to go further.

6.2 Authentication

This will check whether the user and remote machine data given in the configuration file allow a log in to the remote machine. Comparing the hostname of the remote machine with that in the configuration file will indicate whether the log in was successful or not.

6.3 Check FLAME

Check for the xparser executable on the remote machine, assuming that its presence means that necessary libraries (such as the message board library) are therefore present. First look in the `$PATH` environment variable and if not found then look in a known directory where a previous check may have installed the parser. If the parser is found then check the version against that version required by the user. If the version is correct the script finishes.

If the parser is not found or the version is incorrect then the script copies the source for the parser and associated libraries to the remote machine and builds and installs them in a known directory.

6.4 Create Project

A project comprises the XMML file and C code for a model and then jobs are added to projects by giving the initial data, number of iterations and number of partitions for a FLAME run. The project is created by giving a directory on the local machine where the XMML and C code files can be found and they are copied to the remote machine. The xparser is run on the copied data and the resulting C code compiled. Errors from the parsing or compilation stages are reported where necessary and when the project has been successfully created it is given a project id that is returned to the user.

6.5 Submit Job

The initial data file is copied to the remote machine and the run initiated for the user defined number of iterations and number of partitions. The job should be assigned to a particular project so the system knows what model is to be run. Typically large parallel machines use some form of job scheduler to ensure users get a fair share of the machine and details of how to submit jobs to the scheduler should be provided by the user. These details go in the configuration file. When the job is scheduled on the remote machine the script returns a job id for the user to use later in queries.

It is possible to run jobs interactively, that is the script starts the job and waits until it is complete before returning.

6.6 Query Job Status

There are a number of variations for job status query, the simplest being querying the status of one job from one project. The information returned to the user will indicate whether the job has finished, is waiting to be started by the job scheduler, is running or has failed. The progress of a job can be monitored by comparing the names of output files (currently named `node<node-id>-<iteration>.xml`) against the number of iterations required for the job.

This basic query can be extended to retrieve the status for all jobs in a project and all jobs in all projects.

6.7 Get Results

When a job is complete the results will be left on the remote machine and will need to be copied to the local machine. For a parallel run each node writes its results to separate files, one for each iteration. Current analysis techniques require the results files from each node to be amalgamated into one file for each iteration. There is a script to do this and further scripts based on XSLT (a language for transforming XML documents into other XML documents, see `http://www.w3.org/TR/xslt.html`) can be written to extract information about particular variables of interest.

6.8 Implementation

Since the job submission ideas described above require a lot of work with `ssh/scp` and other command line utilities the job submission system has been implemented as a number of (bash) shell scripts each performing one of the tasks described above. Integration with a GUI which TUBITAK have indicated will be written in Python is simple using the `os.system` module. The script results can be redirected to a file and the file parsed by the GUI to extract useful information to display to the user.

Data on projects and jobs will be stored in a file on the remote system so that project and job ids will be unique and details about the parameters for a particular job can be retrieved at a later time, e.g. for a job status query and retrieving the results. For simplicity this is a text file at present.

7 Testing: Functional and Portability

7.1 Unit testing of the Message Board Library

As the stability and correctness of the Message Board Library (*libmboard*) is crucial for running FLAME models, extensive testing and validation steps have been incorporated into the development and maintenance workflow.

We adopted the Test Driven Development approach whereby unit tests for every API routine and supporting components are written (based on pre-determined specifications) before the actual code. The relevant code is then written and validated by the tests. CUnit, a unit testing framework for C code has been used for this task as recommended by the UNICA unit.

Tests used during development are also reused and packaged as test suites for:

- Regression Tests – All tests are executed before each code change is committed to the repository to ensure that nothing has been broken by the modification.
- Portability Tests – All tests are executed on a wide range of platforms before each code release.

- Installation Tests – The tests are provided along with the source code so user can validate the compilation on their platform before usage.

We also include an example simulation code which implements all features provided by the *libmboard* API. This serves as a usage example as well as an additional test designed to exercise *libmboard* routines in a more realistic scenario. This test is executed frequently on different platforms, especially before each code release.

7.2 Unit testing of Timer Package

During development of the timer package component of the dynamic load balancing library the functionality of the package has been tested using unit tests, based on CUnit as for the message board library. The tests implemented are:

- Single timer creation. Test that the first timer has correct handle and elapsed time.
- Multiple timer creation. Create 3 timers and check the handles and elapsed time are correct.
- Multiple timer start. Check that starting a timer while still running has no effect.
- Timer stop. Check that stopping a timer really does stop it.
- Timer reset. Check that resetting a timer sets its elapsed time to zero.
- Multiple timer stop. Check that stopping an already stopped time has no effect.

An example program to illustrate the use of a timer is also provided.

7.3 Testing serial and parallel implementations

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 has a position in two-dimensional space and a radius of influence. Each agent will react to its neighbours within its interaction radius repulsively. So given a sufficient simulation time the initial distribution of agents will tend to a field of uniformly spaced agents. Each agent has x, y, fx, fy and $radius$ in its memory and has three states: outputdata, inputdata and move. The agents communicate via a single message board, *location*, which holds the agent *id* and position. 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.

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,±1) and (±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.

8 Benchmark Problems

8.1 Approach to Benchmarking

Our approach to benchmarking has been incremental: starting from a very simple model and then gradually increasing the complexity. The benchmarks run so far are part of the assessment of the current parallel implementation. They also serve as a useful way of ensuring that FLAME and its generated applications are portable over a wide range of hardware and operating systems.

Model	Agents	Messages	Population
Circles	1	1	10^5
C@S	3	9	124,000
Labour Market	4	10	110,101
Bielefeld	4	29	43100

Table 1: Details of Current Benchmark Models

The starting populations have been generated using the initial population generator developed by STFC. The ratio of agent numbers in each population was retained from the original values.

Each benchmark has been run on a variety of HPC systems available to STFC using a range of process numbers: 4 9 16 32 49 64 81 and 100. The results presented show how the elapsed time per iteration varies with number of processors. In these experiments a round-robin initial distribution has been used for the Labour Market and Bielefeld models, while geometric partitioning has been used for the Circles and C@S models.

8.2 The Circles Model

The Circles agent is very simple. It has a position in two-dimensional space and a radius of influence. Each agent will react to its neighbours within its interaction radius repulsively. So given a sufficient simulation time the initial distribution of agents will tend to a field of uniformly spaced agents.

Each agent has x, y, fx, fy and *radius* in its memory and has three states: outputdata, inputdata and move. The agents communicate via a single message board, *location*, which holds the agent *id* and position.

The Circles problem is very simple but allows us an initial assessment of the performance of the parallelisation within FLAME. The simulation was started with a populations of 10^5 agents and experiments performed using from 4 to 100 processors. The averaged results are shown in Table 2 and Figure 13.

Processors	SCARF	HAPU	HPCx	bglogin2
4	-	1581.15	4464.95	7339.35
9	-	992.35	2813.93	4530.96
16	443.07	524.47	1600.15	2507.14
25	281.30	353.53	1019.06	1595.62
36	205.61	242.17	739.12	1082.74
49	154.03	173.25	523.46	786.60
64	116.13	134.08	390.13	605.77
81	88.83	105.35	325.52	484.19
100	75.20	87.49	256.59	386.71

Table 2: Execution Times for 10^5 Circles

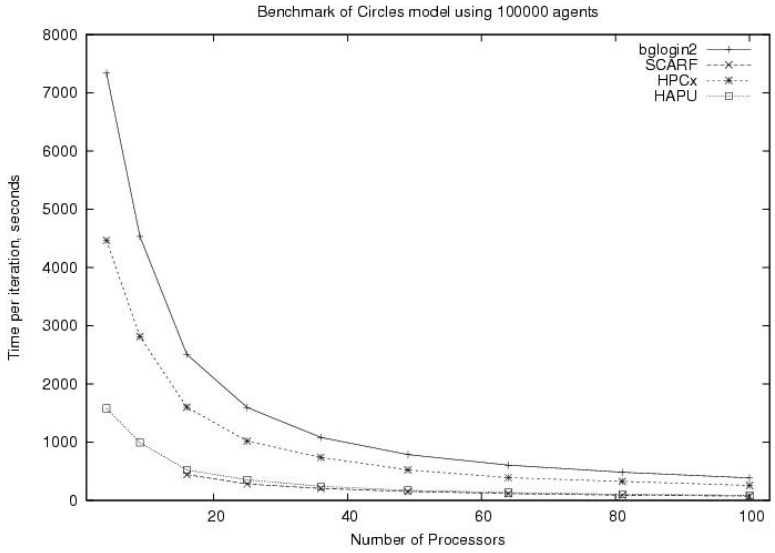

Figure 13: Graph of iteration times

The results indicate that this simulation benefits from using 30 to 50 processors after which the performance benefits flatten. It is interesting to note that this is essentially similar across the range of systems used. The variations between systems being attributed to memory, architecture and communications hardware differences.

8.3 The C@S Model

The C@S model was the first economic model to be implemented in FLAME by the EURACE Project. It is based on work detailed in Delli Gatti *et al.* [24] where an economy is populated by a finite number of *firms*, *workers/consumers* and *banks*. The acronym C@S stands for *Complex Adaptive Trivial System*.

This provides an initial economic model for testing FLAME. The EURACE version of C@S contains models for consumption goods, labour services and credit services. The population is a mix of agents: *Malls*, *Firms* and *People*. Each of these has different states and communicates with other agents in the population through 9 message types.

As the agents in the C@S Model have some positional/location data and the communication is localised, the initial distribution of agents to processors, as in the Circles Model, can be based

on location. This helps reduce cross-processor communication.

The initial population contained: 20000 firms, 100000 people and 4000 malls (124000 agents in total).

Processors	SCARF	HAPU	HPCx	bglogin2
4	2223.86	3062.12	-	-
9	1462.14	2014.56	-	-
16	913.52	1159.32	-	4888.57
25	592.44	755.84	2235.92	3138.71
36	416.84	534.62	1589.53	2165.96
49	307.15	411.32	1178.06	1600.83
64	260.53	313.39	910.23	1218.58
81	207.25	261.33	723.87	992.43
100	169.46	207.52	601.24	806.16

Table 3: Execution Times for C@S Model

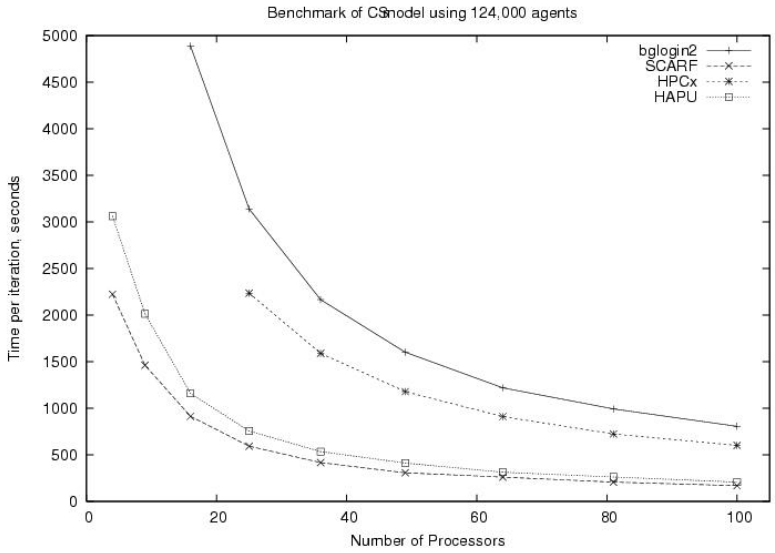

Figure 14: Graph of C@S Model execution times

The results show a potential reduction of the elapsed time of the simulation when using up to 30 processors.

8.4 Initial Labour Market

This model was first model based on the work of the EURACE project. The model represented a very simplified labour market. It contains four agent types: *Firm*, *Household*, *Market Research* and *Eurostat* and 10 message types.

8.5 Bielefeld Labour Market

This model is a refinement of the Initial Labour Market. It too contains four agent types, *Firm*, *Household*, *Mall* and *Investment Goods Producer* and has 27 message types.

Processors	SCARF	HAPU	HPCx	bglogin2
4	1149.21	1388.55	-	7014.64
9	585.19	627.45	2317.21	3120.96
16	334.17	352.51	1332.30	1755.93
25	198.85	233.35	841.33	1129.25
36	291.00	160.97	612.14	782.70
49	206.82	119.66	449.91	574.50
64	90.67	97.81	352.29	440.36
81	72.51	74.81	273.78	349.31
100	60.79	61.34	218.11	284.29

Table 4: Execution Times for Labour Market Model

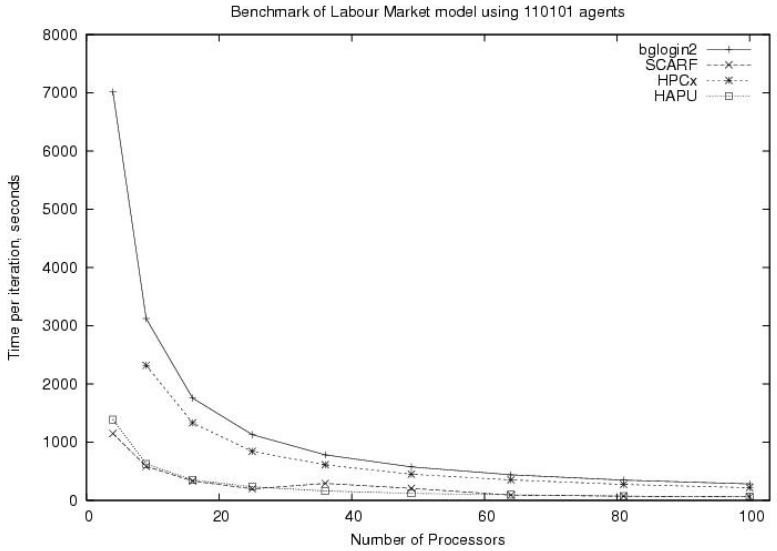

Figure 15: Graph of Labour Market Model iteration times

8.6 EURACE Models

During the development of the EURACE Model a number of domain specific models have been developed. These models were then integrated into the EURACE Model. Three domain specific models were developed: Credit Market, Labour Market and Financial Market. Each of these and the combined EURACE Model are the major economic models developed by EURACE. As part of the development of FLAME these models have been used to test the FLAME application generation and the framework infrastructure. In particular they have been very useful in testing the parallel implementation of FLAME. Although the initial agent populations is these models are very small they do encapsulate the full range and complexity of the EURACE model and to that end they are a very useful testing resource.

All these models have been successfully parsed, compile and executed in both serial and parallel on some of our target HPC machines.

Processors	SCARF	HAPU	HPCx	bglogin2
4	1078.51	1344.86	3253.83	-
9	542.11	616.52	1485.97	-
16	318.55	362.18	847.84	1173.15
25	256.06	231.78	552.62	753.00
36	178.75	162.90	386.80	525.79
49	119.29	124.79	288.37	390.83
64	93.39	99.91	222.95	299.56
81	71.06	75.00	179.58	239.38
100	65.59	61.18	144.97	194.95

Table 5: Execution Times for Bielefeld Model

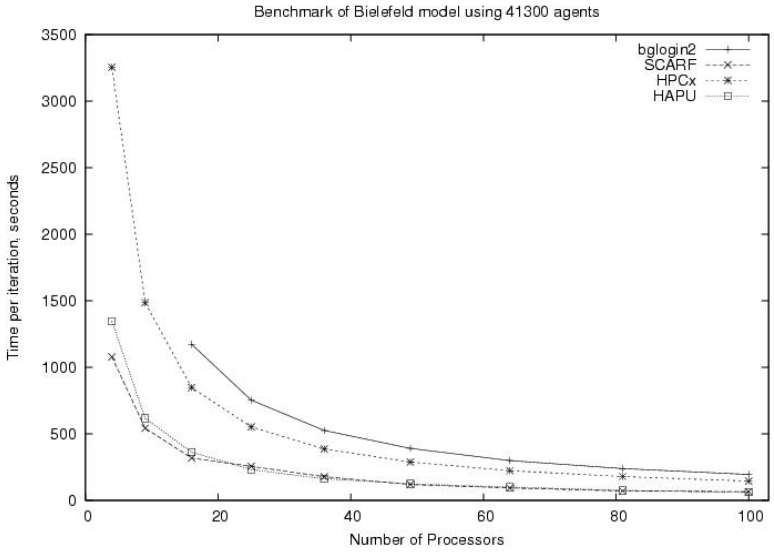

Figure 16: Graph of Bielefeld Model iteration times

Model	Agents	Messages	Population
Financial Market	4	6	1104
Labour Market	7	45	1236
Credit Market	3	12	110
EURACE Model	9	54	2029

Table 6: Details of Current EURACE Models

9 Future Development and Optimisation

9.1 Message Board Library

The Message Board Library is a very new addition to the project. It currently contains the core infrastructure and a very basic communication strategy that, while functional, requires additional tuning and optimisation. The next step in the development would therefore include further research and innovation geared towards a more efficient and scalable performance.

Some of the activities that are in the pipeline are:

- Assist in the parsing of `<filter>` tags within the FLAME parser to ensure Filter Functions are used optimally.
- Accelerate the adoption of `<filter>` tags in models so we can obtain benchmarks results that are more indicative of actual performance.
- Analyse the performance of the library, and optimise the communication routines.
- Profile the memory usage of the library, and improve the memory management and utilisation.
- Allow the tuning of performance to specific platforms by identifying and exposing relevant parameters.

9.2 Partitioning and Dynamic Load Balancing

The present implementation includes a *timer* that has already been used to time sections of the FLAME framework and agent functions. In addition to timing data load balancing requires knowledge of the communication patterns in a particular model and the next phase will be to design and implement a *communication data collector*. With this in place and a working EURACE model (or at least working isolated market models) the performance of simulations can be assessed and a start can be made on designing strategies for balancing computation and communication with a resulting reduction in simulation time. The data collected will also help with partitioning the initial agent population when multiple compute nodes are used.

9.3 Job Submission

The next step for the job submission scripts is to integrate them with the GUI in discussion with TUBITAK and extend the range of machines and job schedulers on which the submission system has been tested. There may be refinements of the scripts and functionality as the whole EURACE system comes together.

10 Conclusion

In this report we have described the parallel implementation of the FLAME framework and demonstrated its use in a number of EURACE related simulations including one of the complete EURACE model. The new message board library abstracts away from the FLAME framework all interactions with the message boards and provides an appropriate API for the FLAME developer.

Implementing the message functions in this way will enable the developers to improve the efficiency of the parallel implementation without affecting the interface to the FLAME framework.

The current implementation has been tested on a variety of models ranging form the simple *Circles* model to the current *complete* EURACE model. Some benchmarking of the FLAME application has been performed and the process of optimisation started.

Two areas that will require significant development have been noted:

Message board optimisation: We are aware that there are a variety of ways in which the current message board implementation can be improved. The first of these is a more considered approach to defining *Filters* and *Filter* variables in the FLAME models to represent any *locality* in the models. This will lead on to an optimisation of how they are used within the FLAME framework.

Dynamic load balancing: The nature of a model might well change during a simulation. The computational or communications load of an agent may change leading to an imbalance in process usage. Although a simulation may tolerate a certain level of imbalance there will be level at which the elapsed time of the simulation will deteriorate. In this case dynamic re-balancing of the processor loads may help.

Both of these activities are very much research areas but there is very little literature currently published.

Although at the end of EURACE we will not have achieved the *optimum* solution to these problems we hope to have at least advanced the current state of the art.

References

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

[2] 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

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

[4] 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

[5] T Takahashi, H Mizuta (2006) "Efficient Agent-Based Simulation Framework for Multi-Node Supercomputers", Simulation Conference, 2006. WSC 06. Proceedings of the Winter Volume , Issue , 3-6 Dec

[6] D Pawlaszczyk (2006) "Scalable Multi Agent Based Simulation - Considering Efficient Simulation of Transport Logistics Networks" 12th ASIM Conference - Simulation in Production and Logistics

[7] 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.

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

[9] Tesfatison (2006) "Agent-based computational economics: a constructive approach to economic theory" in Handbook for Computational Economics, Vol 2, Noth-Holland

[10] Finin et al (1994) "KQML as an Agent Communication Language", The Proceedings of the Third International Conference on Information and Knowledge Management

[11] Gregory et al (2001) "Computing Microbial Interactions and Communications in Real Life", 4th International Conference on Information Processing in Cells and Tissues

[12] Noble (2002) "Modeling the heart-from genes to cells to the whole organ", Science

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

[14] Walker et al (2004) "Agent-based computational modeling of wounded epithelial cell monolayers", IEEE Transactions in NanoBioscience

[15] Walker et al (2004) "The Epitheliome: Agent-Based Modelling Of The Social Behaviour Of Cells", Biosystems

[16] Pogson et al (2006) "Formal Agent-Based Modelling of Intracellular Chemical Reactions", to appear in Biosystems

[17] Qwarnstrom et al (2006) "Predictive agent-based NFkB modelling - involvement of the actin cytoskeleton in pathway control", Submitted

[18] Jackson et al (2004) "Trial geometry gives polarity to ant foraging networks", Nature

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

[20] Holcombe (1998) "X-machines a basis for dynamic system specification", Software Engineering Journal

[21] Kefalas et al (2003) "Communicating X-machines: From Theory to Practice", Lecture Notes in Computer Science

[22] Kefalas et al (2003) "Simulation and verification of P systems through communicating X-machines", Biosystems

[23] Eleftherakis et al (2003) "An agile formal development methodology", Proceedings of the First South-East European Workshop on Formal Methods

[24] Delli Gatti et al (2006), "Emergent Macroeconomics An Agent-based Approach to Business Fluctuations", submitted.

[25] D Worth (2008), "Dynamic Load Balancing Library - Timer User API, Version 0.0.1", August 2008.

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