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

D8.4: Porting of the software platform to parallel computer

Due date of deliverable:
31/05/2009
Actual submission date:
30/09/2009

Start date of project: September 1st 2006 Duration: 39 months

Organisation name of lead contractor for this deliverable
Science and Technology Facilities Council - SFTC (formerly CCLRC)

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	
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)	X

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.

This report updates the ealier deliverable on porting FLAME to parallel systems - Deliverable D1.4 - and presents the developments of FLAME and the porting of the Full Integrated EURACE Model during the final year of the project.

Some of the initial results from the initial benchmarking activities have been included in the Appendix so as to provide a complete record of the FLAME developments.

In the first report the results presented demonstrated that the parallel implementation of FLAME worked 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. During the final year significant developments have been made to FLAME and the underlying Message Board Library that significantly improved both its serial and parallel performance. The filtering available within FLAME has been enriched and its utilisation within the framework improved.

The report presents the developments of FLAME and its associated analysis tools during the final period which including a model consistency check, a initial validation tool and various static and dynamic analysis tools to help the assessment of model performance.

The final section of the report describes the results achieved in benchmarking the EURACE Model at the time of writing. Although largist populations can be simulated - in the order of 30,000 agents - we have not been able to achieve the 100,000s that was the potential goal of the project.

1 Introduction

In this report we describe the parallel implementation of the FLAME Framework [13]. It updates the material presented in EURACE Delieverable D1.4 and presents specifically the developments of FLAME over the last period plus the results from assessing FLAME itself and the Full Integrated EURACE Model (FIEM).

We cover some of the reasons for parallelisation and also give a detailed description of the approach being adopted. We give some details on the performance assessment tools that have been developed to aid the model developer in developing his model. We present some results from some performance assessments we have made of both the FLAME Framework using the EURACE Model and of the EURACE Model itseld.

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 parallel simulations.

Before considering the parallelisation of FLAME it is worth considering the characteristics of a software system than make it worth taking the time and effort to parallelise and the types of systems we are directing our efforts toward - for there are many different types of parallel system. It should also be remeber that FLAME is not the applications program to be parallelised. FLAME is but an applications program generator. It is this program that should exploit parallelisation. As describe in other papers and reports FLAME take 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 attension 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-code, the communications network between processors - specialised integrated or commodity. Collectively these systems can be considered *Single Program Multiple Data - SPMD* architectures. Some characteristics of when to parallelise an application on these types of systems are:

- 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

In general terms it should also be remembered that some algorithms simply do not parallelise - there is insufficient independance 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. Often these types of algorithms are term *closely coupled* or *fine grained.*

As mentioned above the FLAME Framework is not the application - it is the application generator and hence the approach to parallelisation of FLAME generated applications must rely on the underlying architecture imposed on 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. We will discuss later the approach taken in FLAME to exploit parallelism.

From the list above and a knowledge of current system architectures locality and communication are key elements in achieving parallel perforance. However even if an application has many potentially independ task that could be executed in parallel the balance between the computational and communicatons load will greatly effect any parallel performance. These ideas can be expressed in terms of the granularity of application: what are the relative sizes of the computational and communications load each of these independent task.

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 this opens the implementation to 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.

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI remains the dominant model used in high-performance computing today.

OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Although MPI will be used in the parallel implementation of FLAME many high performance systems use multi-core nodes - dual or even quad core processors. On these types of system OpenMP can provide an additional way of expoiting any potential parallelism in an application.

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. Because FLAME is an application generator it does not have a full understanding of the application it is generating. Agent-based applications could 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 code.

We will not address the dynamic re-configuring of an agent population at run time. Initial experiments have shown that this is a very complex problem where there are many trade offs to be considered. In this effort to have an automatically generated parallel implementation of a FLAME application we focus on the most basic characteristic of FLAME and its agents: that of communications - agent to agent. This communication between agents is implemented within

FLAME as a set of *message boards* on which agents post messages (information) and from which agents can read the messages (information). There is one message board per message type and FLAME manages all the users interactions with the message boards through a Message Board API.

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 can drastically impact the scalability and simulation time. However, in many applications of FLAME, 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.

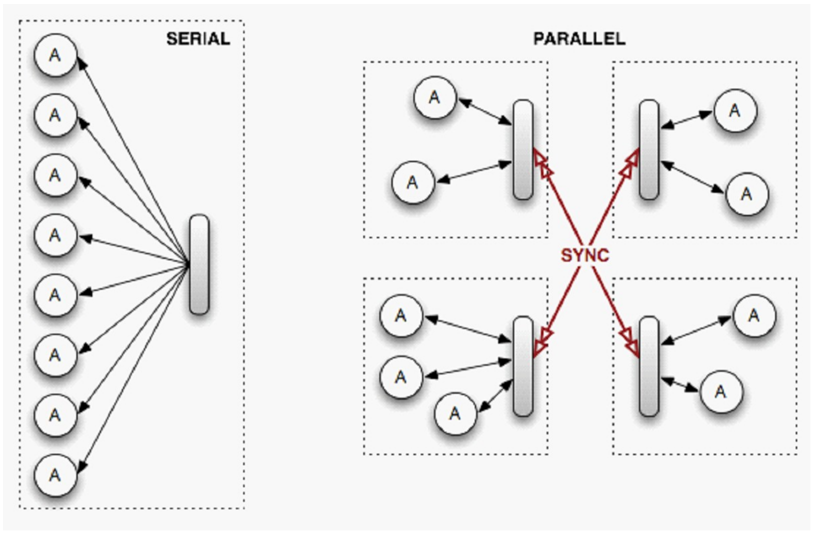


Figure 1: Serial and Parallel Message Boards

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 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.

3.2 The Message Board library

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 strategies while minimising the impact on current users of the FLAME framework.

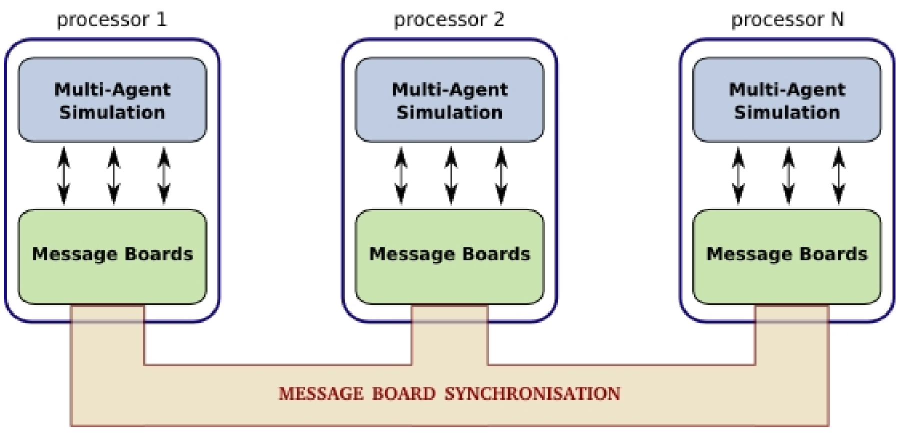


Figure 2: Parallelisation of FLAME using distributed Message Boards

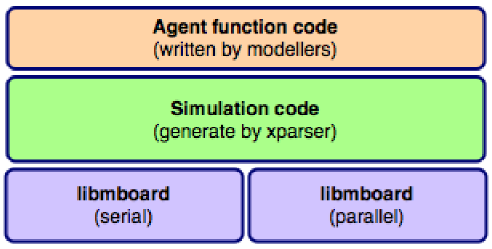


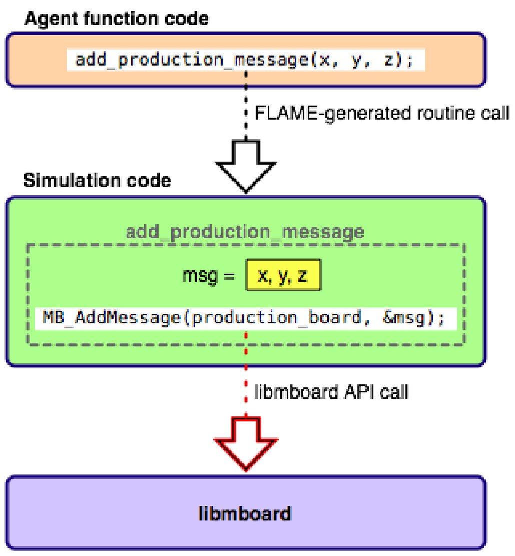
Figure 3: Users can create either serial or parallel executables by linking their object files to the appropriate *libmboard* library

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.



Agent function code
add_production_message(x, y, z);
FLAME-generated routine call
Simulation code
add_production_message
msg =
x, y, z
MB_AddMessage(production_board, &msg);
libmboard API call
libmboard

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

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.

3.3.1 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 3.3.3), across all processing nodes.

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

3.3.2 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.

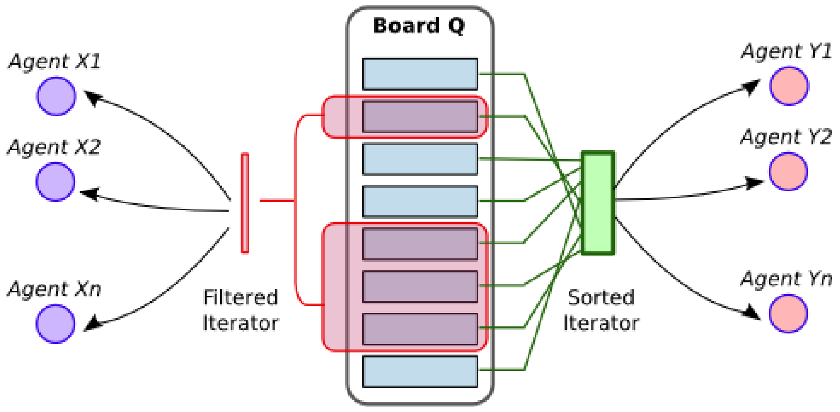


Figure 5: Using specialised Iterators to achieve different access patterns

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 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 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 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.

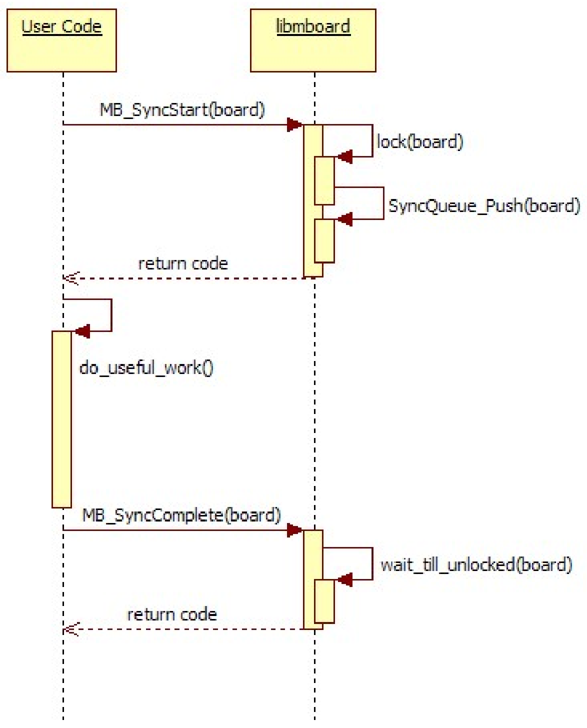


Figure 6: Other work can be scheduled during board synchronisation to hide the overheads of communication

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

3.3.4 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 7). With the tagging in place, full data replication is avoided by only propagating the relevant messages to each processing node.

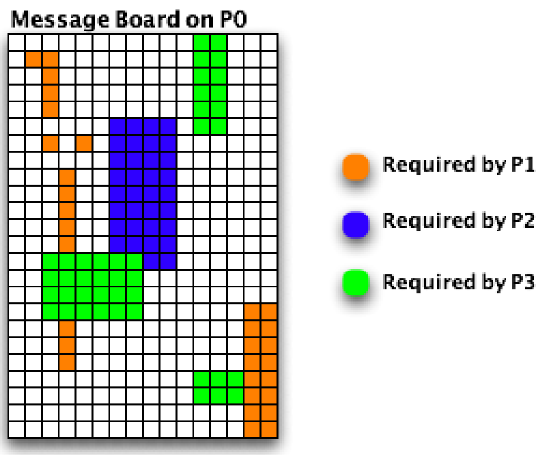


Figure 7: Full data replication is avoided by only propagating tagged messages to the relevant nodes

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.

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

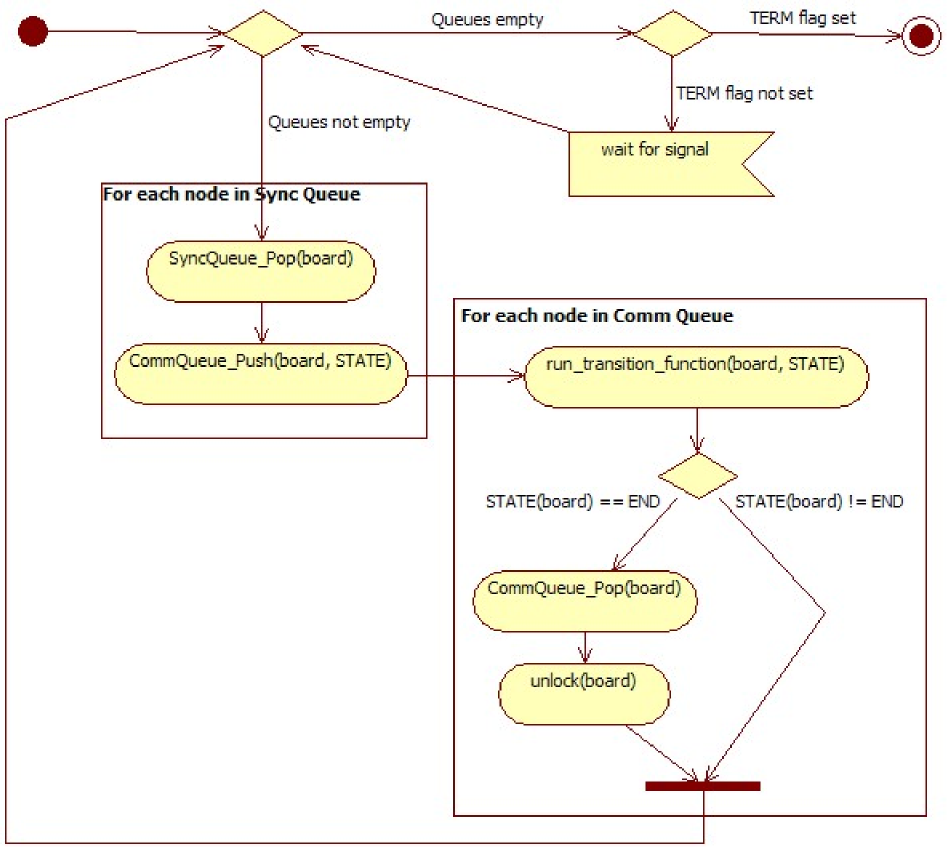


Figure 8: Activity diagram for Communication Thread

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.

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).

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

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.

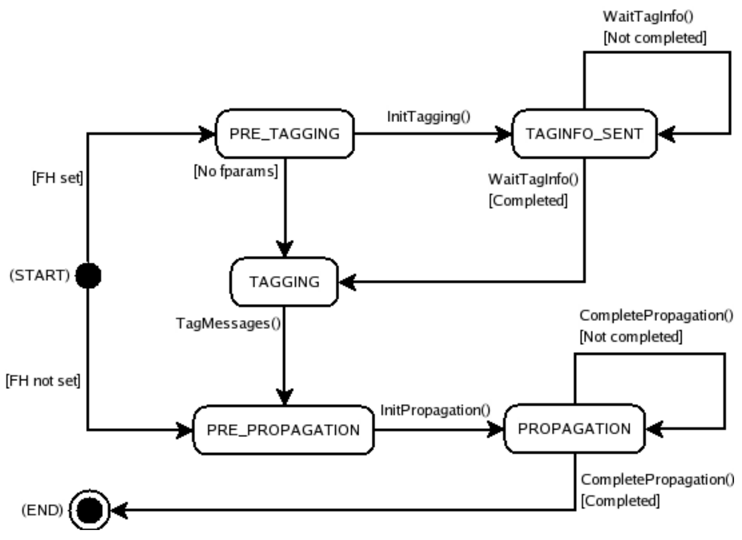


Figure 9: State diagram for processing Comm Queue nodes

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.

3.5 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:

- 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 evebtually decided that the possibilities brought about by using a threaded model outweighed the potential drawbacks.

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

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.

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 constructuing iterators (see Section 3.3.2) to reduce the number of messages presented to an agent for processing and secondly they are use by the Message Board sychronisation 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. This will be extended in future releases.

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. Both are implemented within FLAME and the user can request one or the other with command line flags when the model is run.

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 Pre-partitioning

In a parallel application FLAME can accept pre-partitioned initial data in separate files, one file per process. Each process reads one file thereby reducing the time taken reading initial data, which may be significant for very large populations. The data can be partitioned in any way but the user must remember that FLAME does not provide any redistribute the agents if the initial distribution causes a load imbalance.

In EURACE the cloning of intial data (see Section 5.1) can easily produce pre-partitioned initial data with one or more regions on one processor.

5 Initial Data Generation

5.1 Introduction to Cloning

As the EURACE model became more and more complicated so did the relationships between agents. These realtionships are both inter- and intra- regional and provide a complex set of constraints for the initial population of agents. As modellers and computer scientists demanded larger populations it was found that the Tubitak population GUI could not provide populations quickly enough and ran into memory problems even on very well specified machines. For this reason a simpler method of creating large populations was designed, that of *cloning* one population many times replacing agent ids as required. The popGUI still has a roll, that of providing an easy way of specifying the population characteristics for one region, including statistical distributions for agent memory variables and inter-relationships between them. This data is stored in a `pop` file that is used to initiate the cloning process.

A variety of implementations were tried, `bash` scripts using `sed/awk`, python code and finally and most successfully C code that could run in serial or in parallel. The seed population for cloning has one region and so there are intially no inter-regional interactions but it is possible for these to occur in the transient stage of the simulation if the model is set up correctly. The lack of these interactions means that cloning is a perfectly parallel algorithm.

Cloning from a single region also has the benefit that it is possible to provide pre-partitioned input data for the parallel implementation of the EURACE model. For example, clone one region to give 16 regions and input files for 1, 2, 4, 8 and 16 processes can be easily produced.

The cloning process relies on a special `xml` file in which any agent ids are marked with `REPLACE_ID_n` where `n` is the id of the agent in the region being cloned. This `xml` file is produced from a `pop` file by the `instantiate.py` script written by Tubitak and using code from the population GUI. The script is invoked as

```
python instantiate.py -r 0_bench_oct_12.pop 0_markers_oct_12.xml
```

Actually cloning the region is illustrated by the following example:

```
./clone.sh 0_markers_oct_12.xml 8 tmp -r 2
```

This clones `0_markers_oct_12.xml` creating 2 input files each containing 8 regions. The output files are put into a directory called `16R_2P` indicating that this is input for a 16 region run on 2 processes.

It is possible to control which agents are cloned by listing their names in the file `agent_list.txt`.

5.2 Implementation of Cloning

The cloning application runs from the commandline using a `bash` script to control the cloning and organisation of output files, `sed` to perform some ad-hoc text manipulation of output files and a C application to do the computationaly intensive work of reading the input file and writing an output file. Both serial and parallel (MPI) versions of the C code are available.

The serial code works by repeatedly calling it with arguments that are the increment to make in id numbers between regions and a number (0 based) of this particular clone. The id increment is set to the number of agents in the original region since there can be no more agents in any cloned region. Running the code in paralle requires only the increment value to be given as each process can get the clone number as its node id by calling `MPI_Comm_rank`.

Source code is available from the EURACE Subversion repository:

http://ccpforge.cse.rl.ac.uk/svn/eurace/models/utils/cloning

Using these tools we are able to generate large data sets from a small seed population.

6 Job Submission with ExpGUI

An Experimental GUI (ExpGUI) has been developed by Tubitak to enable users to define a series of experiments based on a single population definition (`pop` file) created by the population GUI. The user can specify the number of instantiations from the population definition, ranges of *environmental* variables (such as income tax rate) and finally the number of runs for each combination of parameter values. The ExpGUI will then execute all theses runs on the local machine, or the user can choose one input file to be run.

It was thought at first that the job submission bash shell scripts described in Appendix E could be used by the experimental GUI for remote working, even on a Windows machine using Cygwin or MSYS. A later decision to make the Windows GUIs work natively meant that this could not happen and so an alternative implementation has been necessary. By making use of paramiko (`http://www.lag.net/paramiko/`) an implementation of the SSH2 protocol for python, the remote execution and copying of files has been implemented to run natively on all platforms. Scripts that run on the remote machine have been left as bash scripts as all the compute clusters run some version of Linux or Unix and the existing remote host configuration files can be used.

At present from the experiment GUI the user can:

- choose the remote host
- specify a serial or parallel job
- specify the number of processes for a parallel job
- run the job
- check on job status (if remote system uses batch processing)
- retrieve results.

Resource constraints have meant that it has not been possible to implement the full requirements of Projects containing Jobs within the experimental GUI.

7 Tools for FLAME and Model Assessment

7.1 Static and Dynamic Analysis Tools

EURACE has developed a very complex model in which there are many agents and many communications. The nested model directories contain 11 subdirectories and ∼50 XMML and C-code files. Checking the consistency of the model is a very difficult task. Although FLAME's parser will check the validity of *xml* within the context of the DDT of FLAME tags, checking that messages are used in a consistent way is difficult.

There are many elements to the testing and assessment of an application. This is the harder in the case of FLAME as FLAME is a program generator. The project has agreed development standards for all software developed which includes FLAME and any C code component of the EURACE Model. These are detailed in other reports and on the EURACE Wiki. 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*.

Although there is a Unit Testing suite for EURACE its target is not FLAME - its has own set of test examples. Its target is the FLAME model functions. It addresses the verification and consistency of agent function calls once the model has been parsed by the `xparser` using the model XMML files.

However these *unit test* do not check the consistency of the whole XMML model. A number of static and dynamic analysis tools have been developed to perform these types of analyses. These tools include the following analyses:

analyses_mode.py : a static analysis of the FLAME model which gives detailed information on the components of a model: agent, function and messages types, number and sizes, a static communications table, a weighted communications table.

check_message_consistency.py : a static consistency checker which compares the XMML definition with C code and ensures that the number and usage of messages is consistent.

The MM Package : The *MM* package is a dynamic to monitor message traffic in the simulation. It is a set additional directives included in the FLAME Templates which embedded in the application code that monitor all message traffic and outputs to an SQL data base.

The data base can be post processed by the developed 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 Time Package : The *Timer* package described in Appendix D has been used to measure elapsed CPU time for functions and message board synchronisations. 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.

7.2 FLAME Verification

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. In Appendix F we have described some of these very basic models that are used in verifying the FLAME generation process. As you will see these are very simple problems. There 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.

7.3 Model Validation

Validating the outputs of any simulation code generated by FLAME is in itself difficult. This will really require mining the outputs of the application and making comparisons with analytic or observer results. There are very few tools that can perform this task. A simple model validation tool. The `sim_validator` uses an SQL data base of the simulations results to which it applies a set of *simulation rules.*

The default file is `eurace.rules`. A snippet from the rules file:

```
::VARIABLES

#Balance sheet:  Firm

firm_payment_account = Firm(payment_account)

firm_cum_revenue = Firm(cum_revenue)

...
```

```
#RULE VERIFIED
#Firm:
abs( firm_payment_account + firm_total_value_local_inventory +
firm_total_value_capital_stock - firm_total_debt - firm_equity)< PRECISION
```

In this snippet a number of variables are define: `firm_payment_account` for example is a reference to a Firm's memory variable `payment_account`. Any number of variables can be defined. In the results select composite rules can be defined using these variables.

The tool applies these rules to the output of the simulation and reports any violations. There are currently 18 such rules in the EURACE validation file. The definition of these rules takes careful consideration by the model developers.

8 Performance of the FLAME Framework

FLAME is a agent-based modelling framework which generates an applications program. It manages the communications between agents through the Message Board Library and provide mechanisms in the model definition file (XMML) and the through the FAME API for the application developer to definite and optimise his application.

By design FLAME will generate both serial and parallel versions of an application. They both use the functions of the *libmboard* to manage messages. It is *libmboard* that performs the mapping of the application onto a parallel system. The parallel implementation of FLAME seeks to use an SPMD paradigm - each node of the parallel system is running essentially the same *program*.

However given the nature of agent-based modelling and the FLAME implementation each nodal program could perform a very different sequence of instructions and function activations as it traverses its part of the state space.

The two 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. In FLAME both these are distributed over the computational nodes.

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 1 gives details on the time taken by the FLAME framework to perform this synchronisation. Each row in the table gives the synchronisation time as a percentage of the total elapsed time per iteration for the top five message boards in each group. The two left-hand columns of this show the basic synchronisation time. 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.

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.

	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 1: Percentage of total time spent synchronising top nine message boards

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 10. Runs were for a 16 region problem on 2 nodes for 40 iterations.

It is clear that the `eurostat_send_macrodata` message board is a significant beneficiary of overlapping. Looking at the state graph for the model shows that the messages are sent by the Eurostat agent very early in an iteration and read by the Government agents very late in the iteration.

The `order` message board benefits less since order messages are required very soon after they are sent (again from the state graph).

9 Performance of EURACE Model

9.1 Assessment and Benchmarking for the EURACE Model

In this section we consider the benchmarking of the full EURACE Model as described in the other deliverables of the project. Before presenting the results of the benchmarks it is useful to consider what we are measuring and why. The goal of most performance analyses is to identify sections or elements of a program that are taking significant resourses - computational time - with the aim of optimising these to reduce the total elapsed time of the simulation. Our purpose is no different from this expect that we will be considering both serial and parallel performance.

It goes without saying that there is little point assessing the parallel performance of a code that performing poorly in serial mode. Most of the components of the code will be executed in both modes although there may be re-organisation of the control flow. So we start will an analysis of the serial version of the EURACE Model.

When assessing parallel performance two basic laws influence the developer: Amdalh's Law and Gustafson's Law. Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized.

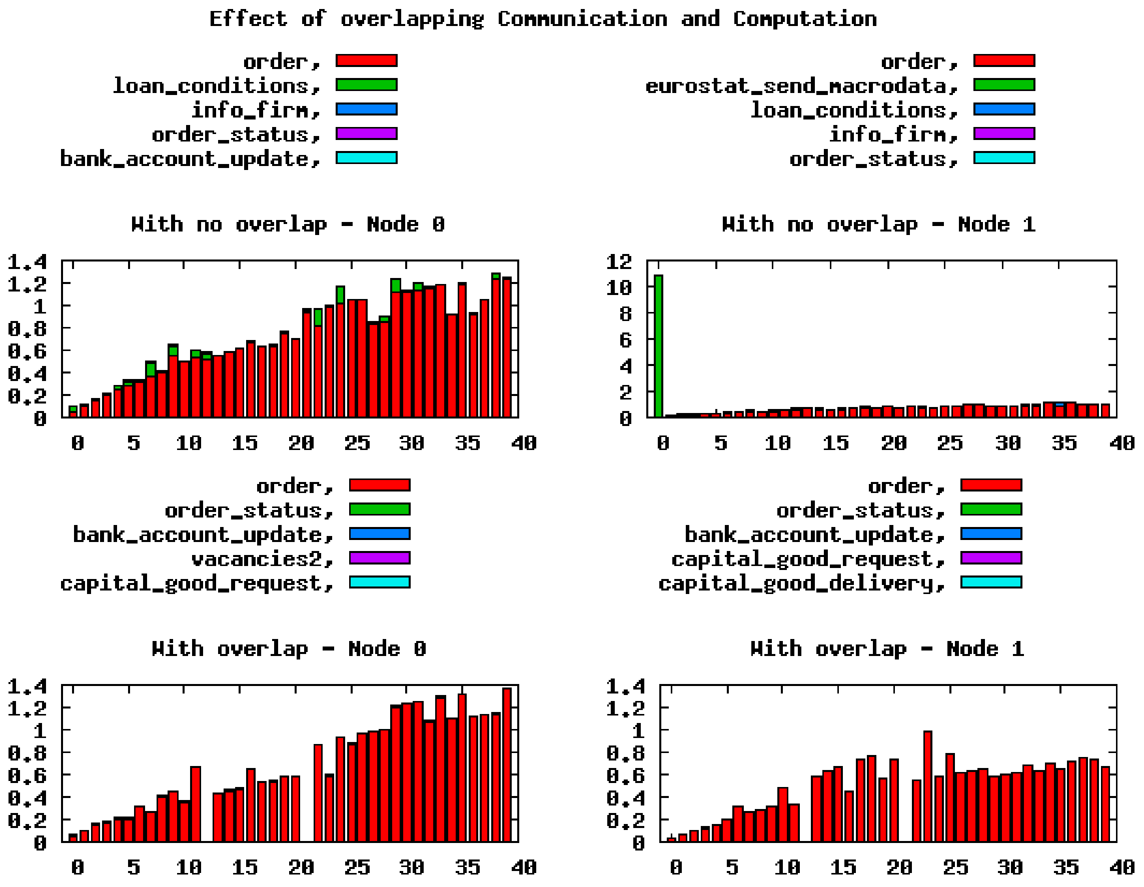


Figure 10: Elapsed times (s) for message board synchronisations with an without communication/computation overlapping

Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1-P)$ is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is

$$\frac{1}{(1-P)+\frac{P}{N}}. \tag{1}$$

In the limit, as N tends to infinity, the maximum speedup tends to $1/(1-P)$. In practice, performance to price ratio falls rapidly as N is increased once there is even a small component of $(1-P)$.

As an example, if P is 90%, then $(1-P)$ is 10%, and the problem can be speed up by a maximum of a factor of 10, no matter how large the value of N used. For this reason, it has been often said, parallel computing is only useful for either small numbers of processors, or problems with very high values of P: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce the component $(1P)$ to the smallest possible value.

Fortunately this is not the end of parallel computing. It must be noted that this was for a **fixed size** application. Gustafson's Law paints a different picture. It states that any sufficiently large problem can be efficiently parallelized and the speedup that can be gained is:

$$S(N) = N - \alpha \cdot (N-1) \tag{2}$$

where N is the number of processors, S is the speedup, and α the non-parallelizable part of the process i.e. the serial part.

Gustafson's law addresses the shortcomings of Amdahl's law, which cannot scale to match availability of computing power as the machine size increases. It removes the fixed problem size or fixed computation load on the parallel processors: instead, he proposed a fixed time concept which leads to scaled speed up for larger problem sizes (i.e. weak or soft scaling).

Amdahl's law is based on fixed workload or fixed problem size (i.e. strong or hard scaling). It implies that the sequential part of a program does not change with respect to machine size (i.e, the number of processors). However the parallel part is evenly distributed by N processors.

In both of these formalisations of potential parallel speedup to proportional of serial activity has a significant effect of the potential parallel speedup. However Gustason's Law gives the hope that for a *sufficiently large* problem parallel performance can be demonstrated.

Although neither of these models completely characterises the exectly situation - it is very difficult to estimate the serial part of a parallel code - it is clear that the serial part has a significant effect on the speed up of any application. So in the assessment of the EURACE Model we will focus on identifying the serial part.

We have developed a number of analysis tools that have allowed us to benchmark any FLAME model and assess its serial and parallel performance. We have used a number of versions of the EURACE Model in this benchmarking - the model has been under constant development.

9.2 Serial Performance Analysis

This section comprises tables of agent function CPU times recorded with the timer package as the EURACE integrated model has evolved in recent revisions. The code was compiled with debugging and profiling turned on which will have an effect on the absolute timings but not on the proportion of time spent in each routine. At the start of this analysis (Table 5) one

Agent type	**Number of agents**
National	
Government	1
Central_Bank	1
Clearinghouse	1
Eurostat	1
IGFirm	1
Regional	
Mall	1
Bank	2
Firm	80
Household	1600

Table 2: Distribution of agent population (small)

`clearinghouse` agent function took up 72% of the run time. When this function was profiled with the GNU `gprof` tool (see Figure 11) it was found that the time was being taken buy the routine `newPrice`. After reviewing the algorithm implemented in `newPrice` ways were found to improved it in a number of ways which would lead to a halving of the number of calls to `aggregateDemand` - a core routine of `newPrice`. During a second phase the message counts table - Table 9.2 was generated, This shows the number of message of a particular type that have passed through the system board system. From the timings it is cleat that `clearinghouse` agent is a serious bottleneck in the simulation. This is re-enforced by Table 9.2 as the `order` message

Writing			Reading	
Message Name	**Counts**		**Message Name**	**Counts**
order	57557		order	2935407
bank_account_update	1551		quality_price_info_1	13700
job_application	666		info_firm	7850
job_application2	552		accountInterest	6000
order_status	337		dividend_per_share	3000
accepted_consumption_1	274		bank_account_update	1551
consumption_request_1	274		job_application	666
tax_payment	62		vacancies	666
hh_subsidy_notification	60		job_application2	552
hh_transfer_notification	60		vacancies2	552

Table 3: Initial top ten counts of message Reads and Write (population: 1688 agents)

is the most used message type and this is the primary input message for the `clearinghouse` agent.

Considerable time was taken in understanding these restuls and various improvements to the algorithms used by the`clearinghouse` agent were suggested to the modellers. These improvements were passed on to the modellers and were incroporated into a new version of the code.

Table 9.2 show the message counts after these improvements had been implemented. These

Writing			Reading	
Message Name	**Counts**		**Message Name**	**Counts**
order	11929		quality_price_info_1	4480
job_application	3704		order	11929
job_application2	3408		dividend_per_share	6400
bank_account_update	1681		job_application	3704
vaaccepted_consumption_1	306		cancies	3704
consumption_request_1	306		job_application2	3408
tax_payment	84		vacancies2	3408
infoAssetCH	81		accountInterest	3200
hh_transfer_notification	80		bank_account_update	1681
info_firm	80		info_firm	880

Table 4: Optimisaed top ten counts of message Reads and Write (population: 1688 agents)

changes significantly lowered the time taken by the clearinghouse and this was enough to bring the second placed function in Table 5 to the top in Table 6.

```
ClearingHouse_receive_orders_and_run (73.5%, called 40 times)
  +----emptyClearing
  +----receiveOrderOnAsset
  | +----setOrder
  | +----isBuyOrder
  | +----addBuyOrder
  | | +----addOrder
  | +----isSellOrder
  | +----addSellOrder
  |   +----setAsSellOrder
  |   | +----getOrderQuantity
  |   +----addOrder
  +----computeAssetPrice (72%, called 2040 times)
  | +----setClearingMechanism
  | +----lastPrice
  | +----runClearing (72%, called 2040 times)
  | | +----buyOrders
  | | +----sellOrders
  | | +----sortOrders
  | | +----newPrice (71.2%, called 2040 times)
  | | | +----aggregateDemand (69.25%, called 4,495,474 times)
  | | | +----aggregateSupply (2%,     called 4,495,474 times)
  | | +----aggregateDemand
  | | +----aggregateSupply
  | | +----ordersMacthing
  | | | +----addOrder
  | | +----rationing
  | |   +----randomize
  | |   +----removeZeroOrders
  | +----addPrice
  | +----addVolume
  +----sendOrderStatus
    +----buyOrders
    +----formedPrice
    +----sellOrders
```

Figure 11: Call graph for `ClearingHouse_receive_orders_and_run` showing work done in most expensive call path

Function	State from	State to	Time (s)	%
ClearingHouse_receive_orders_and_run	RECEIVEDINFOSTOCK	COMPUTEDPRICES	82.81	72
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	25.32	22
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	2.06	1.7
Household_bond_beliefs_formation	BOND_BELIEF_FORMATION	SEND_ORDERS	0.44	0.38
Household_rank_and_buy_goods_1	09	09b	0.42	0.36
Firm_read_job_applications_send_ job_offer_or_rejection	03	04	0.37	0.32
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	0.16	0.14
Household_receive_dividends	06	06b	0.11	<0.1
Household_receive_info_interest_from_bank	Household_received_coupons	SELECTSTRATEGY	0.09	<0.1
Household_send_account_update	15	16	0.09	<0.1

Table 5: Revision **2701**, Serial, 40 iterations, small population. Total run time 1:55[m:s]

Function	State from	State to	Time (s)	%
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	245.78	61
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	46.44	11
ClearingHouse_receive_orders_and_run	RECEIVEDINFOSTOCK	COMPUTEDPRICES	41.76	10
Household_bond_beliefs_formation	BOND_BELIEF_FORMATION	SEND_ORDERS	4.98	1.2
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	1.48	0.3
Household_rank_and_buy_goods_1	09	09b	1.04	0.28
Household_rank_and_buy_goods_2	11	12	0.9	0.22
Household_receive_dividends	06	06b	0.8	0.20
Household_receive_info_interest_from_bank	Household_received_coupons	SELECTSTRATEGY	0.79	0.20
Firm_read_job_applications_send_ job_offer_or_rejection	03	04	0.62	0.15

Table 6: Revision **2723**, Serial, 240 iterations, small population. Total run time 6:42[m:s]

Function	State from	State to	Time (s)	%
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	63.8842	33.3
ClearingHouse_receive_orders_and_run	RECEIVEDINFOSTOCK	COMPUTEDPRICES	59.7536	31.1
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	8.82473	4.6
Household_rank_and_buy_goods_1	09	09b	5.23093	2.7
Firm_read_job_applications_send_job_offer_or_rejection	03	04	1.41734	0.7
Household_receive_dividends	06	06b	1.38099	0.7
Household_receive_info_interest_from_bank	Household_received_coupons	SELECTSTRATEGY	1.14297	0.6
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	1.11446	0.6
Household_receive_data	Household_Start_Policy_Data	Household_Start_Financial_Market_Role	0.619365	0.3
Household_send_account_update	15	16	0.607822	0.3

Table 7: Revision **2772**, Serial, 240 iterations, small population. Total run time 3:12[m:s]

Function	State from	State to	Time (s)	%
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	77.95	37.5
ClearingHouse_receive_orders_and_run	RECEIVEDINFOSTOCK	COMPUTEDPRICES	54.51	26.2
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	9.83	4.7
Household_rank_and_buy_goods_1	09	09b	3.91	1.9
Household_receive_dividends	06	06b	1.47	0.7
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	1.34	0.6
Firm_read_job_applications_send_job_offer_or_rejection	03	04	0.98	0.5
Household_select_strategy	SELECTSTRATEGY	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	0.8	0.4
Household_receive_info_interest_from_bank	Household_received_coupons	REVISE_PORTFOLIO	0.76	0.4
Household_send_account_update	15	16	0.64	0.3

Table 8: Revision **2802**, Serial, 240 iterations, small population. Total run time 3:28[m:s]

This types of iterative analysis and optimisation has been used to improve the performance the EURACE model.

9.3 Parallel Performance Analysis

In Appendix B the results from the initial parallel benchmarks are described. These models are characterised in Tabel 9. In general the results showed promising parallel speedup using up to 40 processors. However as the number of message used by a model - as in the case of the

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 9: Details of Initial Benchmark Models

Bielefeld model - some irratic behaviour was observed. Table 10 gives the characteristics of the full integrated EURACE Model. It is most important to note that the number of message types is 62 - a great many more than previous models.

Model	Agents	Messages	Population
EURACE	9	62	30,000

Table 10: Details of the EURACE Model

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 increased for a fixed initial population. The profiling of agent functions is shown in Tables 11 and 12 and it is clear that the clearinghouse agent function for finding the correct market price for assets is dominant. Not only that; because there is only one clearinghouse it is a serious **serial bottleneck**. All agents have to wait for it to complete its calculations before they can carry on. The household function for sending orders is not a bottleneck since the agents are distributed round the processes by FLAME.

Although FLAME makes great efforts to maximise the time available to synchronise the message boards, as explained in Section , this will only have an effect if the function in question has potential parallel tasks. Unfortunately the Clearing House, and particular when computing the correct market price for assets, requires asset and order information from all Firm and Household agents before it is able to perform the calculation. This generates a serial bottleneck which will have a serious effect on performance. This is reflected in the timings in Table 12 - the Clear House is top of the list. All other agents - and processors - much wait until the Clear House completes its work - which is around 35% of the runtime.

Although this is a serious problem in terms of performance it is still possible to perform runs of the EURACE model with larger populations. Using a 16 region model with 26948 agents runs have been carried out on large parallel machines available to STFC, namely HAPU, NW-Grid and Hector. Details of these machines can be found in Appendix A.

Function	State from	State to	Time (s)	%
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	2083.3	14.2
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	211.144	1.4
order			137.024	0.9
Household_receive_dividends	06	06b	104.891	0.7
Household_receive_data	Household_Start_Policy_Data	Household_Start_Financial_Market_Role	43.6602	0.3
Household_receive_info_interest_from_bank	Household_received_coupons	SELECTSTRATEGY	37.1621	0.3
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	34.5611	0.2
Firm_read_stock_transactions	0003	End_Firm_Financial_Role	17.7424	0.1
Household_rank_and_buy_goods_1	09	09b	16.1013	0.1
Household_send_account_update	15	16	16.049	0.1

Table 11: Revision **2754**, Parallel, 240 iterations, 16 region population. Total run time 244:25[m:s]. Node 0

Function	State from	State to	Time (s)	%
ClearingHouse_receive_orders_and_run	RECEIVEDINFOSTOCK	COMPUTEDPRICES	5125.29	35.0
Household_send_orders	SEND_ORDERS	WAITORDERSTATUS	2067.41	14.1
Household_stock_beliefs_formation	CHOOSE_TO_UPDATE_BELIEFS_OR_NOT	BOND_BELIEF_FORMATION	222.105	1.5
Household_receive_dividends	06	06b	104.267	0.7
order			75.312	0.5
Household_receive_data	Household_Start_Policy_Data	Household_Start_Financial_Market_Role	43.5033	0.3
Household_receive_info_interest_from_bank	Household_received_coupons	SELECTSTRATEGY	37.0695	0.3
Household_update_its_portfolio	WAITORDERSTATUS	Household_Start_Labour_Role	34.4162	0.2
Household_rank_and_buy_goods_1	09	09b	16.4965	0.1
Firm_read_stock_transactions	0003	End_Firm_Financial_Role	15.9953	0.1

Table 12: Revision **2754**, Parallel, 240 iterations, 16 region population. Total run time 244:25[m:s]. Node 1

The graph of average iteration time (seconds) against number of processes is shown in Figure 12. Some interesting features can be seen in this graph: it can be observed that Hector shows generally good speedup, the NW-Grid machine shows a little but HAPU shows some seriously eratic behaviour. The iteration times are given in Table 13.

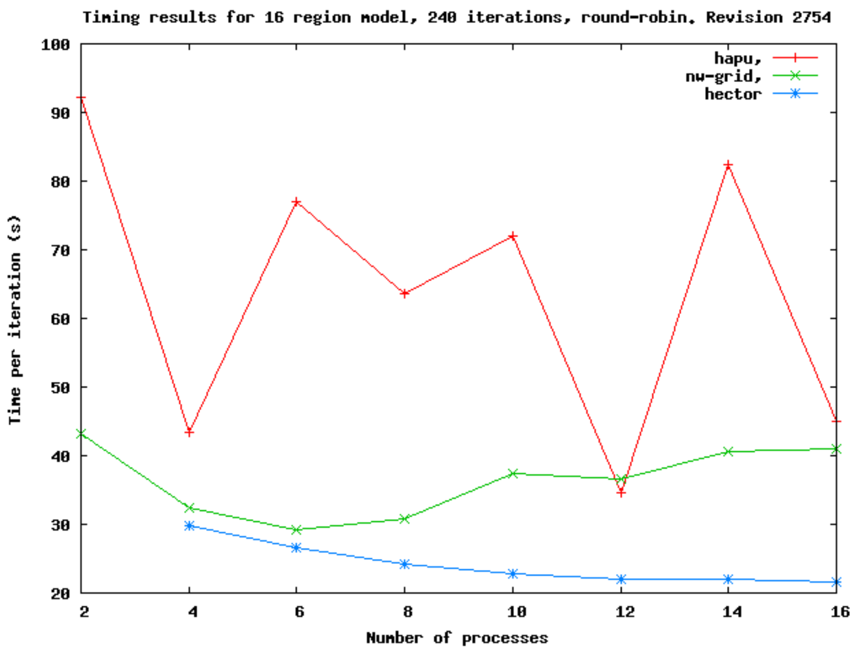


Figure 12: Average iteration times (s) 240 iterations of 16 region EURACE model

Num processes	HAPU	NW-Grid	Hector
2	92.3	43.2	-
4	43.3	32.4	29.8
6	76.9	29.3	26.6
8	63.6	30.8	24.1
10	72.1	37.3	22.9
12	34.6	36.5	22.0
14	82.5	40.5	22.1
16	45.0	41.0	21.7

Table 13: Average iteration times for 16 region EURACE model

Although the HECToR and NW-GRID results show some parallel performance it is disappointing. The HAPU results show some very strange behaviour which can only really be explained by defficiencies in the EURACE Model algorithms. During the assessment of the model it has been shown to be very sensitive to changes in parameter values: sometimes failing with zero divides but more often or not terminating in an infinite loop. The model in very complex and it is very difficult to debug. Programming conventions and testing proceedures - as defined in the project - have help elliminate many problems but as one might expect there will be residual bugs.

Throughout the assessment critical algorithms have been review and improvements made but there are still many processes that are not particularly robust. Given the very complex interactions between agents one could not expect every computational situation to have been covered. Only significant more testing and detailed evaluation and detailed assessment of the EURACE algorithm will lead to a robust simulator.

The results from running the EURACE Model in parallel have been very disappointing. The stability of the model has been very poor and as are these intial results.

As mentioned earlier the EURACE Model is being continually improve and developed so the results in this report are only a snapshot of the status at the time of writing.

10 Conclusion

In this report we have described the parallel implementation of the FLAME framework and its assessment together with some benchmarking results using the EURACE Model. We have also demonstrated FLAMEs use in a number of EURACE related simulations in addition to complete EURACE model on populations ranging from a few hundereds 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.

An important goal of the project has been to perform, in parallel, a large simulation using the EURACE Model. The project has achieve 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 sychronisation times are a few of the measures that have been used together with a detail static analysis of the model to identify the performance defficiencies in both the FLAME framework and the EURACE model.

All this analysis has lead to improvements in FLAME and the EURACE Model which in general have improved its computational performance. However the presence of substanial serial components in any model 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 - in particularly the Clearing House - have a significant impact of the parallel performance of the model. These types of potential problem were understood - fine grained tasks - at the start of the project and the modeller took steps to avoid them. The Clearing House was thought necessary to the architecture of the EURACE Model and although different strategies were tested to reduce its effect there was little that could be achieved. The Clearing House and any other serial bottleneck will compromise the parallel performance of the application.

Although at the end of EURACE we have not achieved the *optimum* solution to these problems we have at least advanced the current state of the art in the parallel implementation of agent-based simulations in the context of the FLAME Framework.

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.

A Parallel Computing Systems Used In EURACE

A number of the partners in the EURACE project have their own parallel systems. The FLAME Framework and the EURACE Model have been ported to these systems. STFC has a large number of different parallel computing systems which it has made available to the project and used in testing the EURACE Model. Each of these machine has a different hardware architecture and software infrastructure.

Unit	GREQAM	TUBITAK	UNIBI
Processor Type	Intel Xeon 5140 (Dual Core)	Intel Xeon E5355 (Quad Core)	Intel Xeon 5160 (Dual Core)
Total Cores	4 (2 x 2)	4 (1 x 4)	4 (2 x 2)
Total Memory	4GB (4 x 1GB)	16GB (4 x 4GB)	2GB
Memory per core	1GB	4GB	512MB
Total Storage	146GB (2 x 73GB)	292GB (2 x 146GB)	219GB (3 x 73GB)
Usable Storage	73GB (RAID 1)	?	?
Operating System	Windows XP Pro x64	?	Red Hat Enterprise Linux 4
MPI Library[1]	MPI 1	MPI 1	MPI 1

HPCx : The HPCx platform is currently number 43 in the 28th Top 500 Supercomputer list (Nov 2006). It is based on the IBM pSeries 575 system, and has a total of 2560 processors. The HPCx system uses IBM eServer 575 nodes for the compute and IBM eServer 575 nodes for login and disk I/O. Each eServer node contains 16 processors. At present there are two service nodes. The main HPCx service provides 160 nodes for compute jobs for users, giving a total of 2560 processors. There is a separate partition of 12 nodes reserved for certain projects. The peak computational power of the HPCx system is 15.3 Tflops peak.

SCARF : SCARF is a compute cluster run by the STFC's e-Science Centre (HPC services group). It uses the RedHat Enterprise Linux 4.4 Operating System, and provides the LSF scheduler and SCALI-MPI. SCARF has the following hardware specification:360 2.2GHz AMD Opteron processor cores, 1.296TB total memory, Gigabit networking and Myrinet M3F-PCIXD-2 low latency interconnect.

HAPU : HAPU is an HP Cluster Platform 4000 based on Redhat Enterprise Linux 4. It has 128 x 2.4GHz Opteron cores, with 2Gb memory per core, and a Voltaire InfiniBand interconnect.

NW-GRID : The NW-GRID Cluster comprises three compute racks, with each rack containing 32 SUN x4100 nodes. Each node contains 2 Dual Core 2.4Ghz Opterons with 8GB of memory. That brings the total processor count to 192 Dual-Core Opterons (384 processor cores).

MANO : MANO is an IBM Blue Gene/L machine. It comprises 1024 nodes of dual-core 700MHz PowerPC chips with the second cpu usually dedicated i/o and communications. The frontend (or login) node is a p5-520Q with 4x1.5GHz processors, 16GB RAM and running SuSE Linux Enterprise Server 9 and this is supplemented with an identical service node for system control. GPFS is provided through two p5-505 servers each with 2x1.5GHz processors and 4GB RAM.

bglogin2 : is a single frame of a IBM Blue Gene/P machine. A standard Blue Gene/P configuration will house 4,096 processors per rack. Four 850 MHz PowerPC 450 processors are integrated on each Blue Gene/P chip. It is at least seven times more energy efficient than any other supercomputer, accomplished by using many small, low-power chips connected through five specialized networks.

Hector : is a Cray XT4 scalar supercomputer. The XT4 comprises 1416 compute blades, each of which has 4 quad-core processor sockets. This amounts to a total of 22,656 cores, each of which acts as a single CPU. The processor is an AMD 2.3 GHz Opteron. Each quad-core socket shares 8 GB of memory, giving a total of 45.3 TB over the whole XT4 system. The theoretical peak performance of the system is 208 Tflops. There are also 24 service blades, each with 2 dual-core processor sockets. They act as login nodes and controllers for I/O and for the network. In addition there is a Cray vector Blackwidow part of the system which includes 28 vector compute nodes; each node has 4 Cray vector processors, making 112 processors in all. Each processor is capable of 25.6 Gflops, giving a peak performance of 2.87 Tflops. Each 4-processor node shares 32 GB of memory.

B Initial Benchmarks Results

The approach to the initial benchmarking of FLAME 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 14: Details of Initial 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.

B.1 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 15 and Figure 13.

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

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 15: Execution Times for 10^5 Circles

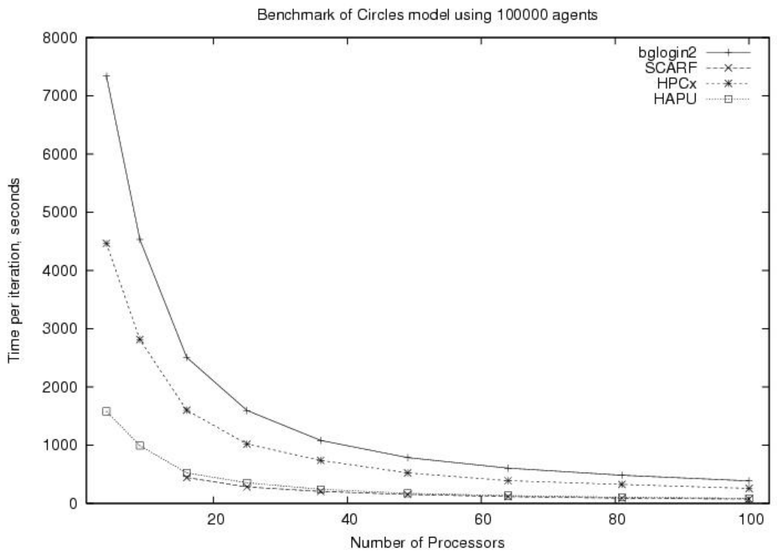


Figure 13: Graph of iteration times

range of systems used. The variations between systems being attributed to memory, architecture and communications hardware differences.

B.2 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 16: Execution Times for C@S Model

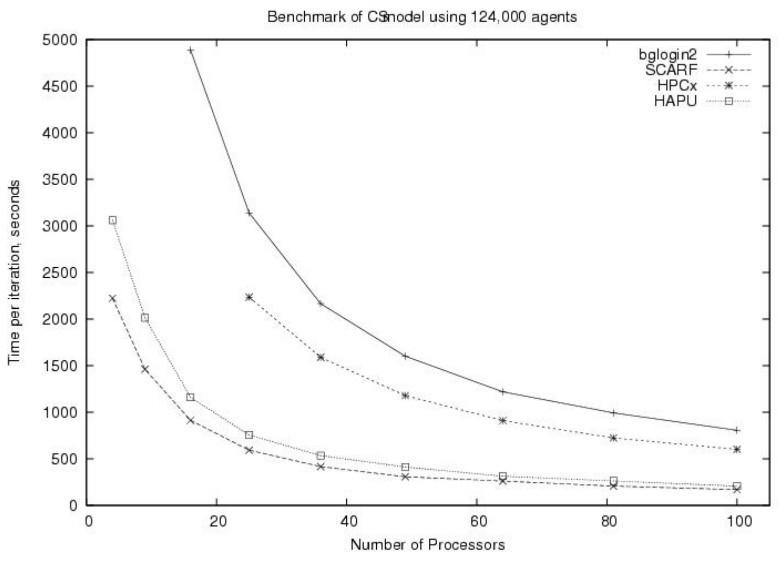


Figure 14: Graph of C@S Model execution times

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

B.3 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.

B.4 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 17: Execution Times for Labour Market Model

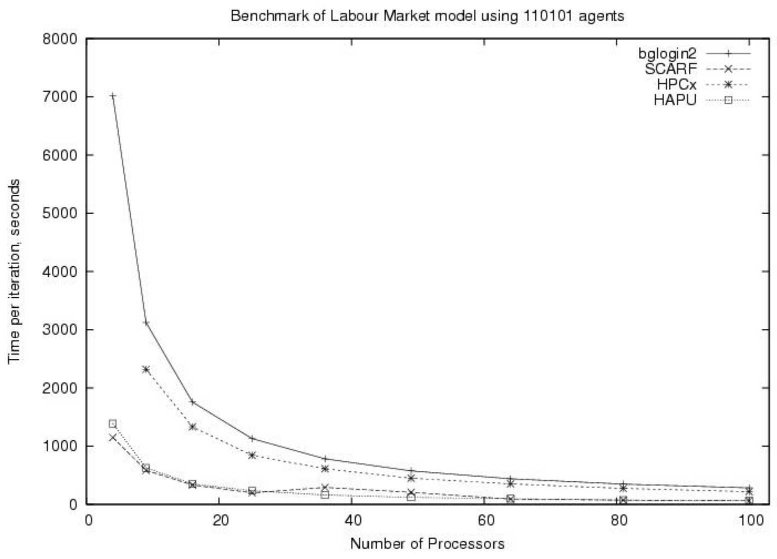


Figure 15: Graph of Labour Market Model iteration times

B.5 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 18: Execution Times for Bielefeld Model

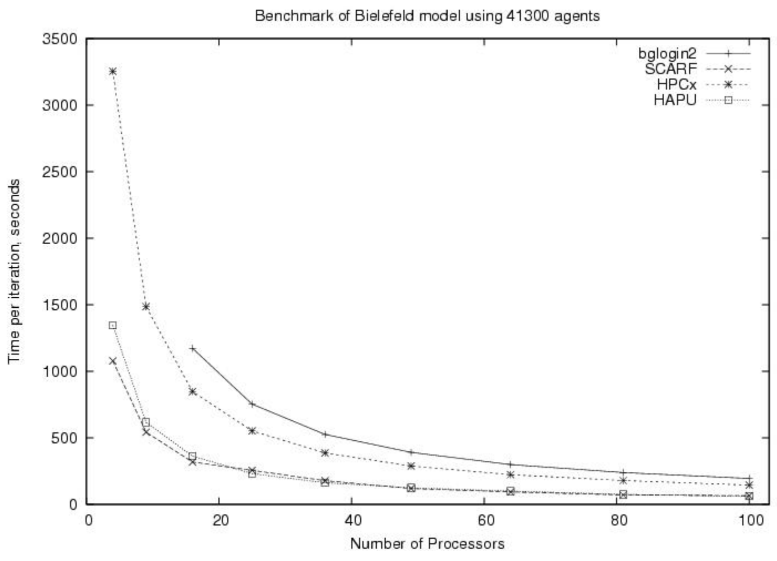


Figure 16: Graph of Bielefeld Model iteration times

C 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 17 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.

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

- elapsed time for various parts of a FLAME run,

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 19: Details of Current EURACE Models

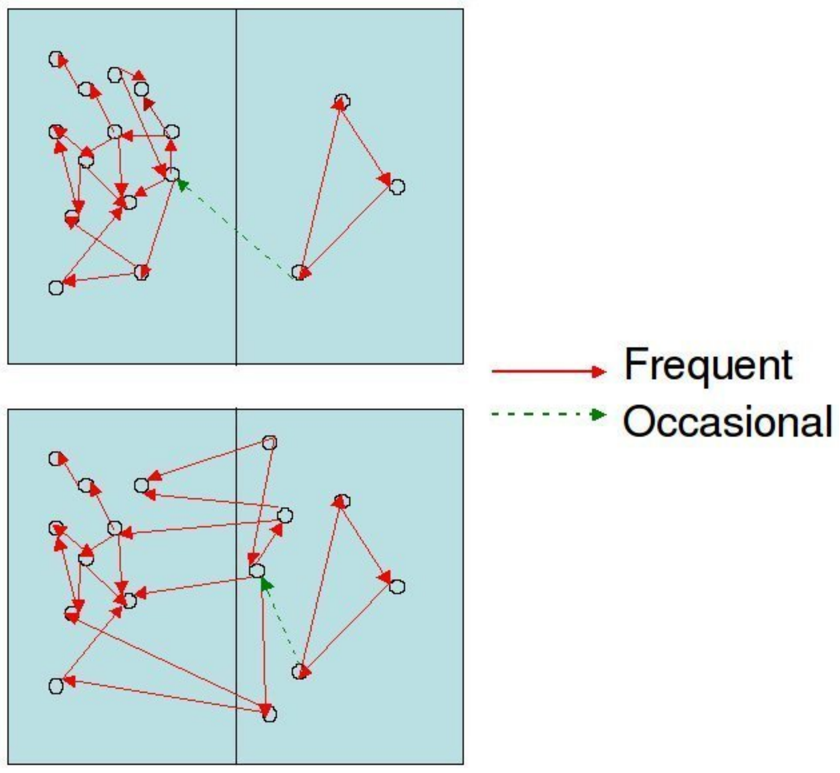


Figure 17: Illustrating some problems of load balancing

- 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.

This timer package has proved extremely useful during analysis of the model during its development, allowing STFC to place timers around calls to all agent functions and messageboard synchronisations. The elapsed time for each function and synchronisation for each iteration can then be printed and the analysis of the top 10 longest elapsed times over the whole simulation performed. The timers are implemented to work correctly for parallel applications so that timings of each function on each process can be recorded and analysed to look for load imbalances.

D 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].

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 18 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 19 shows that the elapsed time increased even though the work done by the agents decreased.

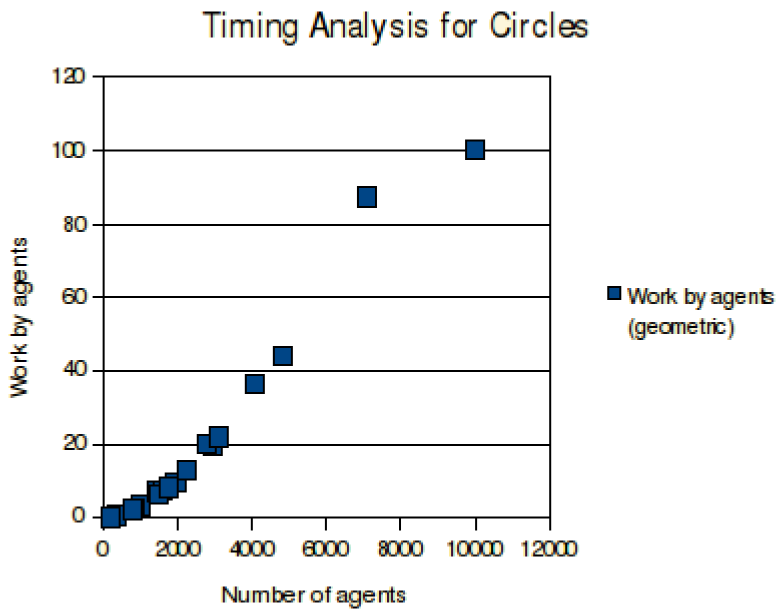


Figure 18: Timings results from Circles model

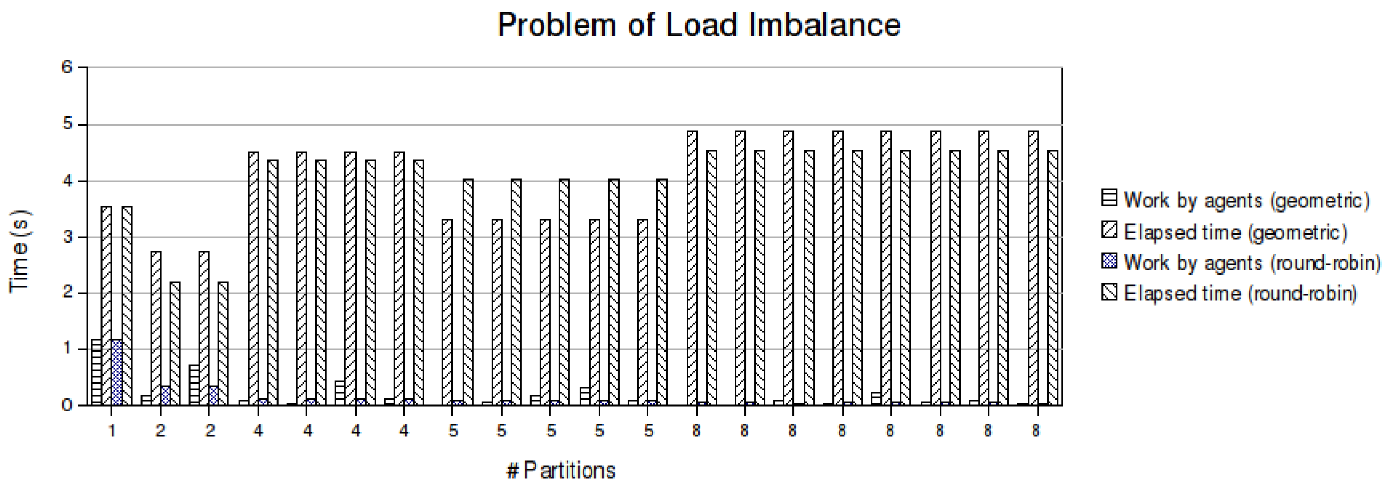


Figure 19: Illustration of problems with load balancing with Circles model

A further example, this time from the full EURACE model, shows the top 5 elapsed times for each iteration for the first 40 iterations: see Figure 20.

E Remote Job Submission

E.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

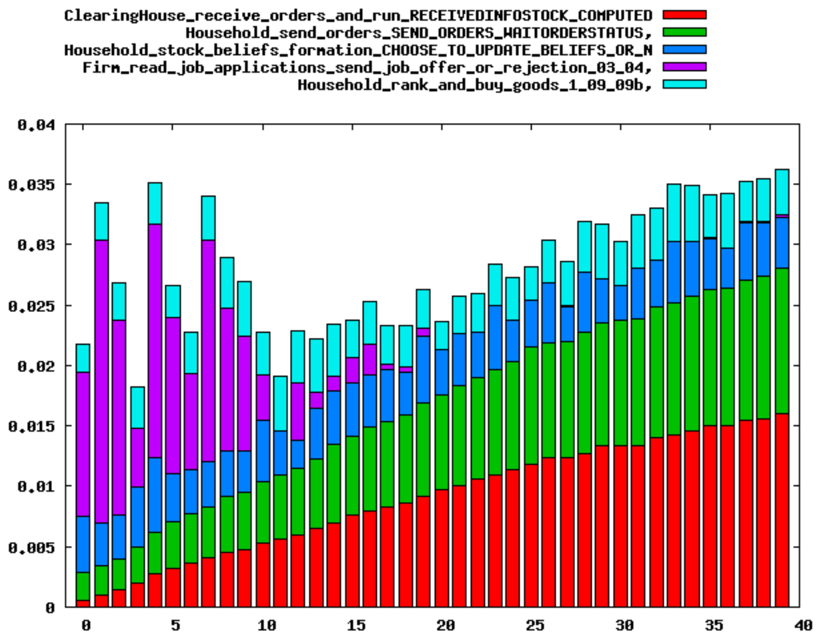


Figure 20: Example function timing results for EURACE Model

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 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.

E.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.

E.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.

E.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.

E.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.

E.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.

E.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.

E.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.

F FLAME Verification

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 (±1,±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.