Science & Technology Facilities Council

Technical Report
RAL-TR-2012-006

The EURACE agent-based economic model: benchmarking, assessment and optimization

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

April 2012

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

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

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

ISSN 1358- 6254

The EURACE Agent-Based Economic Model: Benchmarking, Assessment and Optimization

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

April 2012

Abstract

This report describes the authors' experiences in developing and optimising the serial and parallel versions of a very large complex agent-based economic model. The EURACE Model was developed as part of the EU Framework 6 project EURACE using the Flexible Large-scale Agent Modelling Environment - FLAME - which has been developed in a collaboration between the Computer Science Department of the University of Sheffield and the Software Engineering Group at the STFC Rutherford Appleton Laboratory.

The report contains a brief description of FLAME, which is an applications program generator for agent-based simulations and its parallel implementation. However the bulk of the report is concerned with the process of developing and optimising the serial and parallel versions of the EURACE model. This includes a description of the tools developed by the project to aid this development and how they were used in debugging the EURACE Model.

The final sections of the report describe some of the large-scale benchmark simulations performed with a study of the effects of serial sections on the overall parallel performance of such a model.

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

Keywords: agent-based modelling, economic modelling, parallelisation, optimisation, performance analysis

---

Email: {shawn.chin, christopher.greenough, david.worth}@stfc.ac.uk

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

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

Contents

1 Introduction

The report builds on a previous report [1] which describes the initial developments of the parallel framework and some initial performance results with simple agent-based models. Although in this report we give a short summary of FLAME and its parallelisation, the bulk of the report is concerned with the optimisation of the serial and parallel versions of the EURACE Economic Model and the tools and types of analysis used in the optimisation.

After a short summary of FLAME and its parallelisation we consider the assessment of the overheads and parallel performance of the FLAME infrastructure. In particular we attempt to monitor the overheads of the message management through the message board library and the overlapping of communication and computation achieved by multi-threading. The report then considers both the serial and parallel optimisation of the EURACE model and considers the different tools developed to verify and monitor the performance of the model.

The final sections give the results of some computational experiments and an assessment of the effect of serial sections of the code on the parallel performance.

The initial development of FLAME was driven by Simon Coakley [4] in the Computer Science Department at Sheffield University. It is now developed jointly with the Software Engineering Group at the STFC Rutherford Appleton Laboratory. Details can be found on the FLAME Web site `http://www.flame.ac.uk`.

This work was performed as part of the EU funded project EURACE Project [2] (STREP No 035086) which was a three-year project investigating the use of agent-based modelling in economic and financial systems.

2 FLAME - The Flexible Large-scale Agent Modelling Environment

FLAME (The Flexible Large-scale Agent Modelling Environment) is what it says - it is an environment for developing agent-based applications. FLAME develops the ideas of Kefalas *et al.* [6] which describes a formal basis for the development of an agent-based simulation framework using the concept of a communicating X-machine.

FLAME has a model specification language, XMML (based on the XML standard) and a set of tools to compile the agent-based system. FLAME uses a template library to drive the code generation and can produce both serial and parallel programs. The main elements a FLAME model are: the XMML model definition and the agent transition functions (provide as C code).

The modeller provides a description of his model and the functions that define the operations, communications and changes of state of the agents and FLAME generates the applications program. Figure 1 shows the structure of the FLAME environment. The model file is parsed by the `xparser` and the results combined with the transition functions through the `xparser`'s template library to generate the application.

Full details of the theoretical background to FLAME and the X-Machine approach to agent-based modelling is given in other various reports and papers [1, 4, 5].

3 General Parallel Implementation of FLAME

We will not go into detail concerning the parallelisation of FLAME in this report. A detailed description can be found in [3]. A brief summary will be sufficient to appreciate the approach and its impact of the performance on FLAME models. Although the FLAME architecture has some inherently good characteristics that lend themselves to parallelisation, it also has a number of bad characteristics. Because FLAME is a simple application generator it does not have a full understanding of the application it is generating. Although the agent population and their interactions can be specified *a priori* the computational load of each agent and the nature of

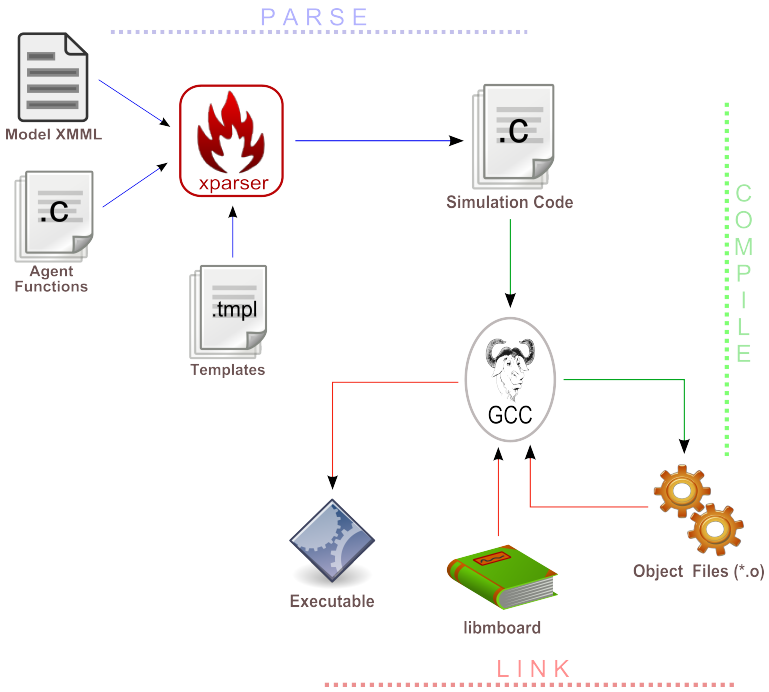

Figure 1: The structure of the FLAME Environment

the communications they perform are very difficult to determine without running the code.

A basic characteristic of FLAME and its agents are those of activation (state changes) and communication (conceptually agent to agent, but not in implementation). This could generally be considered a bad characteristic as the communication happens often and the communicating tasks are computational light-weight so the potential parallelism could be seen as very fine grained. It is not quite that simple as the computational weight of each transition function could vary greatly.

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. There is one message board per message type and FLAME manages all the interactions with the message boards through a Message Board API. The use of simple read/write, single-type message boards allows FLAME to divide the agent population and their associated communications areas providing a high level parallelisation strategy. 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 may be reasonable to assume that the dominant communications in both scenarios will be with neighbouring agents.

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 2 shows graphically the difference between the serial and parallel architecture of FLAME. The most significant operation in the parallel implementation of FLAME 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.

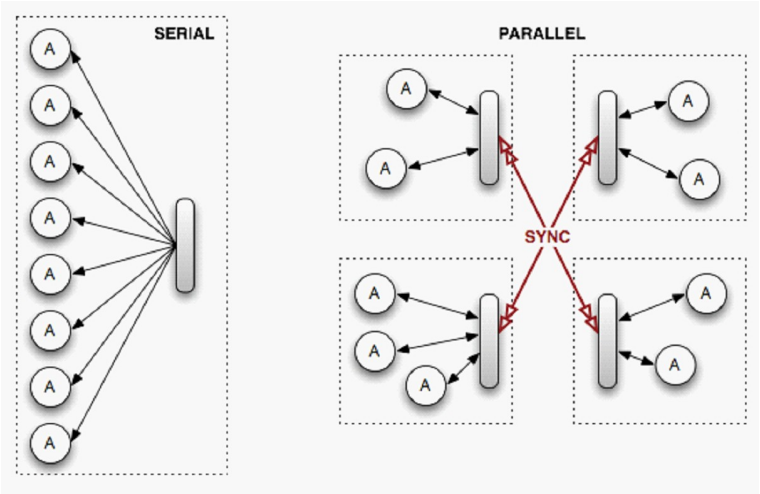

Figure 2: Serial and Parallel Message Boards

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; but 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. 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. Exploiting this locality is key to gaining parallel performance and as with all parallel applications it is the serial component of the application that quickly degrades this performance.

4 The EURACE Economic and Financial Model

The complete EURACE model is an extremely large and complex model made up of eight modules. Each module is the FLAME implementation of a particular economic role for different agents. The list of modules is:

1. Labour market;
2. Consumption goods market;
3. Investment goods market;
4. Credit market;
5. Financial market;
6. Firms' financial management role;
7. Government;
8. Eurostat.

Each module implements a set of functions for all those agents playing a certain role. All the agents (firms, households, etc.) may be involved in different economic activities (i.e. markets) and are characterised by different actions or functions. These actions are grouped into distinct

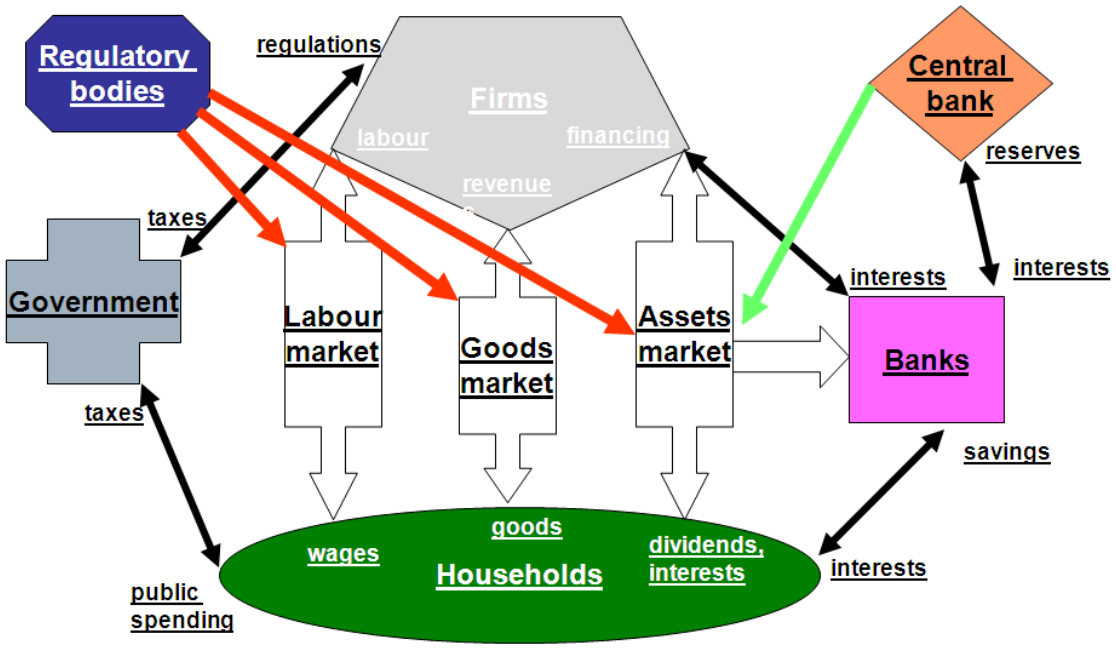

Figure 3: The EURACE Economic Model

roles and roles are define the interfaces between agents and markets. For example, firms selling goods are playing a role in the consumption goods market using a set of functions related to that role. However, when they are asking for loans, they are working in the credit market role using the set of functions that this role requires. The last two modules (7-8) deal with roles played by one single agent respectively.

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 1: Distribution of agent population

The final EURACE model is made up of 9 agent types (see Table 1). The proportions of the agents are uneven and this will effect the distribution of the agents over the available processors in a parallel simulation. It is also worth noting that there are a number of *national* agents - there will only be one of these in the whole population. These singleton agents are an essential part of the model but they could be computational bottlenecks. Although these singleton agents have been classified as *national* in a model containing many countries some of them (*Central Bank* and *Euostat*)could be super-national.

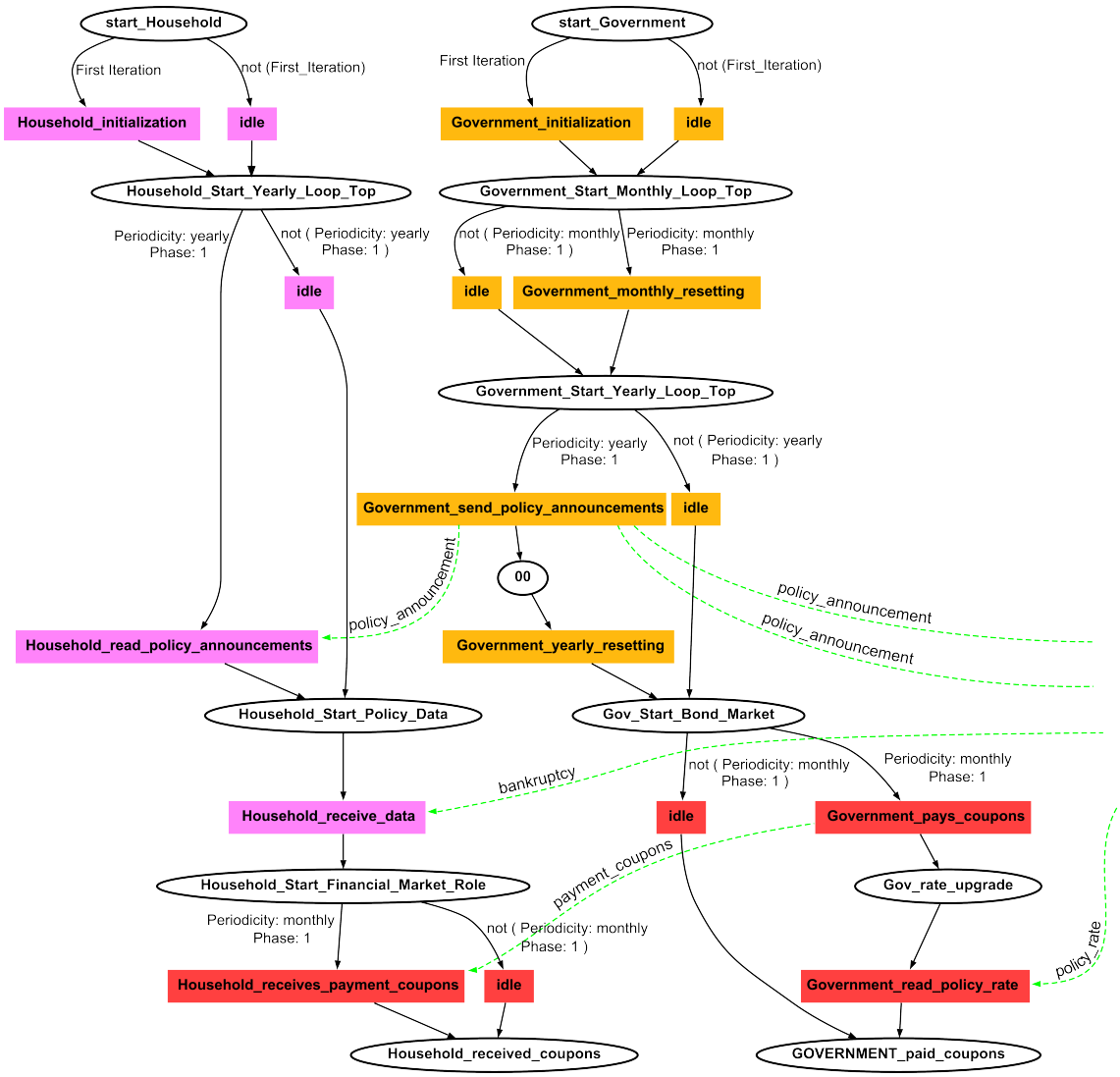

Figure 4: Portion of EURACE state-graph

What is not reflected in these diagrams and tables are the message types used by the model - within the EURACE Model there are 62 message types of varying size. Figure 4 shows a small portion of the EURACE state-graph illustrating the start of the Household and Government agents. The boxes indicate the agent functions and the ovals the state names. Associated with some of the state names there are conditional branches, for example at the states *start_Household* and *start_Government*. These conditionals are due to the periodicity of the functions but, in general these conditional branches can involve more general statements based on the agents' memory. The lines connecting two agent strands in the portion indicate *communication* between agents (albeit through the message board mechanism of FLAME). For example consider the function *Government_pays_coupons* near the bottom of the *Government* agent strand. This functions post messages to the *payment_coupons* message board which the function *Household_receives_payment_coupons* receives by reading from the same message board.

5 Tools for FLAME and Model Assessment

5.1 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, for although FLAME's parser will check the validity of the model within the context of the DTD (Data Type Definitions) 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 and this is all the harder in the case of FLAME as FLAME is a program generator. The EURACE project had 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 - which has its own set of test examples - rather it targets the EURACE model functions. The unit testing framework 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 this type of consistency analysis as we will now describe.

FLAME_Analyses : a static analysis of the FLAME model which gives detailed information on the components of a model: agent, function and messages types, their number and sizes, a static communications table,and a weighted communications table. An example of the outputs from FLAME_Analyses are given in Tables 2, 3 and 4.

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

Table 2: : Agent list with internal memory descriptions

In Table 2 we have a list of all the agents present in the model and an indication of the internal memory state: `Dyn Memory` indicates that the agents internal memory is dynamic. The expressions `(M:7,7)`, as in the case of the `Bank` agent indicates that this agent receives 7 message types and sends 7 message types. For the `Clearinghouse` agent 3 message types are received and two types are sent. Note that these numbers refer to the message types, not the number of messages as this will be population and model state dependent.

In Table 3 the arrows (`-->`) show the direction of the messages and the `F` indicates that the message is filtered. At the end of each line the entry `(daily,0)`, for example indicates the period and phase of a message.

The final example of the output from the analysis tool is the inter-agent communications table show in Table 4. The table displays the possible inter-agent communication based on the communications defined in the model definition. Although this has been weighted using the message size as a measure of a message's volume the table takes no account of

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

Table 3: : Message activity of Bank agent including timing and filtering

the numbers of particular agents in the population. Despite this it gives some indication of the likely communications traffic between agent types which can be used to inform any agent distribution process in a parallel implementation of the model.

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

Table 4: : Weighted Inter-agent Message Analysis

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

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

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

The Timer Package : The *Timer* package is used to measure elapsed CPU time for functions

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

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

5.2 FLAME Verification

Verification and validation of the FLAME implementation is again made difficult by its nature. We must verify and validate FLAME itself and we must also verify the applications generated by FLAME in both their serial and parallel forms. FLAME has two distinct parts: the `xparser` which generates the application from the model 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 the application with its interface to any parallel hardware being used through the MPI (Message Passing Interface).

libmboard was developed using an agile test driven methodology and consequently has is own set of unit tests and test programs. 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 gathered a number of test examples (model and their associated C code) which help verify the FLAME implementation. 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 is correct is by very careful walk throughs. In Appendix B we have described some of these very basic models that are used in verifying the FLAME generation process. The main characteristic of these models is that 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.

5.3 Model Validation

Validating the outputs of any simulation code generated by FLAME is a difficult process. This will require mining the outputs of the application and making comparisons with analytic or observed results. There are very few tools that can perform this task. A simple model validation tool, the `sim_validator` applies a set of *simulation rules* to an SQL database of the simulations results. The rules are defined in a file and a snippet from the rules file is given below:

```
::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 defined: `firm_payment_account` for example is a reference to a Firm's memory variable `payment_account`. Rules can then be defined using these variables and a standard programming operators.

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 process to devise these rules includes a great deal of input from the model developers.

5.4 The Verification and Validation Process

During the development of FLAME and the EURACE model each of the tools described above were used during the verification and validation process. Much of this process relied on having an expected set of values for the various outputs of these tools and ensuring that differences from these expected values were explained. This is far from an automatic and rigorous process but given the maturity of the tools available it was the best possible.

6 Performance of the FLAME Framework

As we have already stated 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, i.e. a set of agents.

The two fundamental design features of FLAME are that all communication between agents takes place through a specified message board 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. Clearly it is 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 and the most important task performed by the FLAME framework is message and message board management. Although applications make use the message board functions for writing and reading messages, the message board synchronisation process is potentially the most costly.

Table 5 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. 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 5. Runs were for a 16 region problem on 2 nodes for 40 iterations.

	No overlapping		Overlapping	
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 5: Percentage of total time spent synchronising top nine message boards

The graphs of Figure 5 show the time delay between `MB_SyncStart` - the start of a message board synchronisation - and `MB_SyncComplete` - the time of completion. The upper two graphs show these times with no overlapping. Note the difference in the scales between the graph for `Node 0` and that for `Node 1` - about a factor of 10 difference. At the start of a simulation it is clear that the `eurostat_send_macrodata` message board is a significant beneficiary of overlapping. Once overlapping has been introduced the effects of `eurostat_send_macrodata` disappear and the wait time for the message boards even out. However this analysis shows that the `order`, `order_status` and `bank_account_update` message boards are holding back the simulation.

On a closer examination of the EURACE model's state graph 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. So despite there being a potential serial bottleneck in the single `Eurostat` agent its communications overhead is hidden in the communications overlapping. Similarly the `order` message board benefits less since order messages are required very soon after they are sent.

7 Performance of EURACE Model

7.1 Assessment and Benchmarking for the EURACE Model

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 resources - 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 performs 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 speed-up of parallel implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelised.

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

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

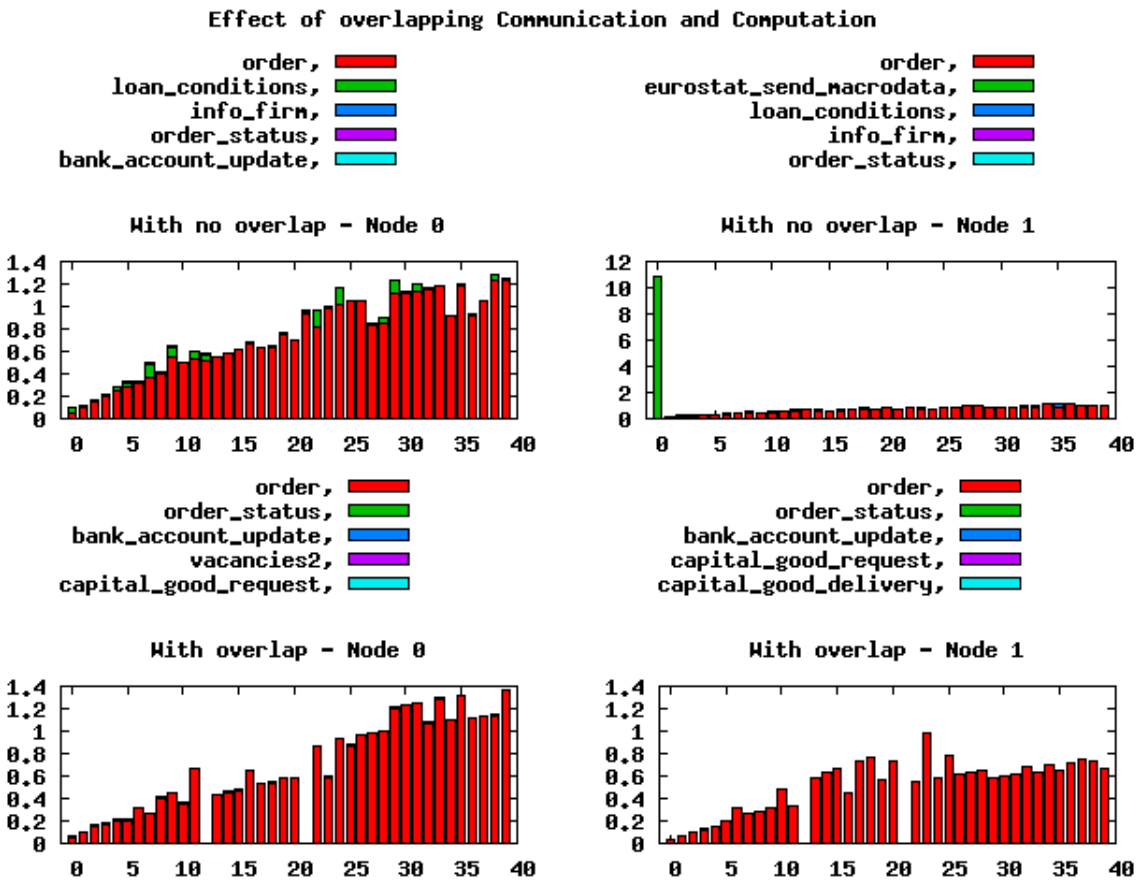

Figure 5: Elapsed times (s) for message board synchronisations with and without communication/computation overlapping

In the limit, as N tends to infinity, the maximum speed-up tends to $1/(1-P)$. In practice, performance 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 sped up by a maximum of a factor of 10, no matter how large the value of N used. For this reason, as 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 large part of the craft of parallel programming consists of attempting to reduce the component $(1-P)$ 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 parallelised and the speed-up that can be gained is:

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

where N is the number of processors, S is the speed-up, and α the non-parallelisable 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, it proposes 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 over N processors.

In both these formalisations of potential parallel speed-up the proportion of serial activity

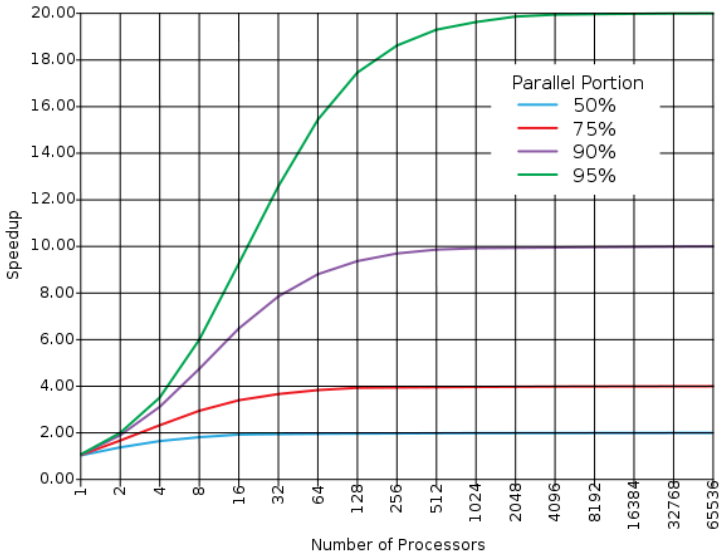

Figure 6: Amdahl's Law

has a significant effect. However Gustafson's Law gives the hope that for a *sufficiently large* problem parallel performance can be demonstrated.

Although neither of these models completely characterises the exact 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.

7.2 Serial Performance Analysis

This section comprises tables of agent function CPU times recorded with the timer package as the EURACE model has evolved in various revisions. Our objective is to demonstrate the approach to assessment and optimisation. In all cases the code was compiled with debugging and profiling turned on which will clearly have an effect on the absolute timings. However not on the proportion of time spent in each routine.

The agent population used for these initial assessments was the basic unit of population defined in Table 1 which contained 1688 agents and all nine agent types.

The starting point of this analysis was a elapsed time assessment of the model using the TP Library. Table 6 shows the results of this experiment. It is clear that one of the `clearinghouse` agent's functions has taken up over 72% of the total run time.

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

When this function was profiled with the GNU `gprof` tool (see Figure 7) 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. A number of approaches would potentially lead to a halving of the number of calls to `aggregateDemand` - a core routine of `newPrice`.

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 7: Initial top ten counts of message Reads and Writes (population: 1688 agents)

During a second phase the message counts table (Table 7) was generated. This shows the number of message of a particular type that have passed through the message board system. From the timings (Table 6) it is clear that the `clearinghouse` agent is a serious bottleneck in the simulation. This is re-enforced by Table 7 as the `order` message is the most used message type and this is the primary input message for the `clearinghouse` agent.

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

Table 8 show the message counts after these improvements had been implemented.

Writing			**Reading**	
Message Name	**Counts**		**Message Name**	**Counts**
order	11929		order	11929
job_application	3704		dividend_per_share	6400
job_application2	3408		quality_price_info_1	4480
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 8: Optimised top ten counts of message Reads and Writes (population: 1688 agents)

These changes significantly lowered the time taken by the `clearinghouse` and this was enough to bring the second placed function in Table 6 to the top in Table 9.

```
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 7: Call graph for `clearinghouse_receive_orders_and_run` showing work done in most expensive call path

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

Clearly a similar analysis should now performed on the `Household_stock_beliefs_formation` as it now takes up 61% of the processing. This type of iterative analysis and optimisation has been used to improve the performance the EURACE model.

7.3 Population Partitioning

In Section 3 we briefly described the essence of the parallel implementation within FLAME: the population of agents and the message boards are distributed over the processor array such that some form of load balance can be achieved. As explained in Section 3 the inter-agent communication tables of the `FLAME_analysis` tool give us some indication how to distribute the agents as does the relative size of each section of the population. There are various ways in which the partition can be achieve: round-robin, geometric, separator driven. For the EURACE model a separator driven approach is used. This partitioning seems appropriate as the large model

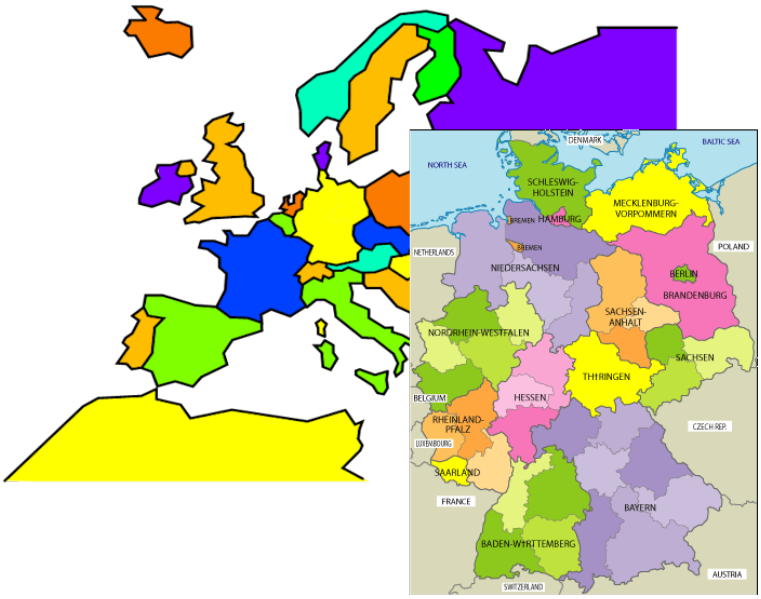

Figure 8: Partitioning based on a regional separator

populations are generated on a region by region basis using the standard unit of population (see Table 1). As part of the model definition each agent holds in its memory which region it is associated with. This value is used in the model to control the range of some of the inter-agent communications. For example, households general only deal with banks and firms in their region. Within the EURACE model these regional identifiers give some locality to agents processes. As shown in Figure 8 these could be country divisions or regional divisions within a country.

7.4 Phase I Parallel Performance Analysis

Having now performed a serial optimisation of the EURACE model we consider its parallel performance. In initial testing and benchmarking FLAME generated applications have shown reasonable parallel performance. However this is very dependent on the model - its size and complexity - and the population size; but the performance was encouraging. It was noticeable that as the number of messages used by a model increased some erratic behaviour was observed on some of the systems being used. The details of these benchmarking are in [3].

For the first parallel benchmarks the EURACE model and population is characterised in Table 10. Table 1 gives the initial breakdown of agents types within the population. It is most important to note that the number of message types is 62 - a great many more than any of the previous test models. 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

Model	Agents	Messages	Population
EURACE	9	62	30,000

Table 10: Details of the EURACE Model

of processes is increased for a fixed initial population. The profiling of agent functions running on a 2-node system is shown in Tables 11 and 12. It is clear that the `household` is dominant on Node 0 (14.2% of the cpu times) and the `clearinghouse` agent function for finding the correct market price for assets is dominant on Node 1 (35% of the processor time).

At this point we need to comment of the distribution of the agents over the compute nodes. From Table 1 we see that `household` and `firm` agents dominate the population. Table 4 shows us that `household` and `firm` agents are involved in much of the communications so it is reasonable to distribute these populations so that households are *near* their desired firms. This cannot be done from the initial data of the model as the household-firm associations are only generated during the course of the simulation. Also since the EURACE model in its current form only restricts households to communicating with firms in the same region we can use region information to group households and firms from the static data. However, for this small population the `household` and `firm` agents are distributed evenly between the two compute nodes. The other agents in the population are allocated to a single node.

We can now try to put these initial results in some form of context. From Tables 11 and 12 we can see that household functions dominate. This is not unreasonable as they are the majority agent on each node. 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 6, this will only have an effect if the function in question has potential parallel tasks. Unfortunately the `clearinghouse` particularly 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 `clearinghouse` is top of the list. All other agents - and processors - much wait until the `clearinghouse` 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: EURACE model 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: EURACE model 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 9. 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.

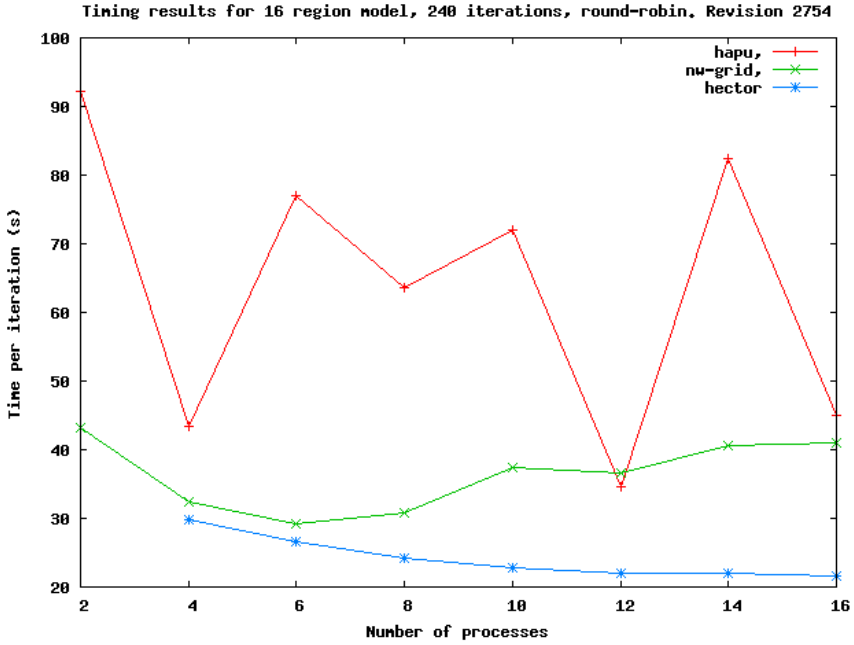

Figure 9: Average iteration time (s) for 240 iterations of 16 region EURACE model

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 deficiencies 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 than not becoming stuck in an infinite loop. The model is very complex and it is very difficult to debug. Programming conventions and testing procedures - as defined in the project - have helped eliminate many problems but as one might expect there will be residual bugs.

7.5 Phase II Benchmarking

As with most software developments the process is one of iterative improvement. The EURACE model is no different. The results in Figure 9 and Table 13 are from early in the development.

The next figures and tables show results after rather detailed refinements of the model have taken place.

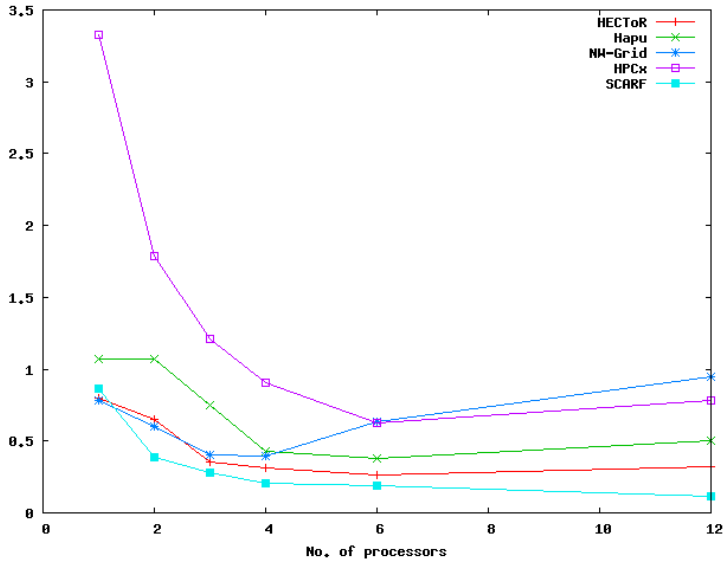

Figure 10: 20,212 agents, 12 regions

Procs	HECToR	NW-Grid	Hapu	HPCx
1	0.798000	0.783857	1.071690	3.330214
2	0.649357	0.600000	1.068810	1.789929
3	0.350190	0.405214	0.746429	1.212595
4	0.314262	0.392381	0.428500	0.903405
6	0.266381	0.630905	0.376381	0.626500
12	0.319143	0.948024	0.503643	0.783548

Table 14: Average iteration time (s) for 240 iterations of 12 region (20,212 agents) EURACE model

Both these sets of results show a significant improvement in parallel performance of the model though they are still not really adequate. We have illustrated the development and refinement process used within EURACE and have shown that the framework provided by FLAME does not impose a significant overhead to the application.

7.6 Dominance of Serial Components

Throughout the assessment critical algorithms have been reviewed 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 significantly more testing and detailed evaluation and assessment of the EURACE algorithm will lead to a robust simulator.

We have identified that the EURACE model contains some serial bottlenecks. In particular the `clearinghouse` agent. From Table 12 see that this agent can be responsible for about 35% of the run time of a simulation. As there is only one agent of this type and all `household` agents communicate (through the `order` and `order_status` messages) with this one agent on a daily basis it is very clear that this agent could create a serious problem.

The `clearinghouse` agent responsible for satisfying the financial orders being placed by each household. It essentially manages the supply and demand. Within this part of the model

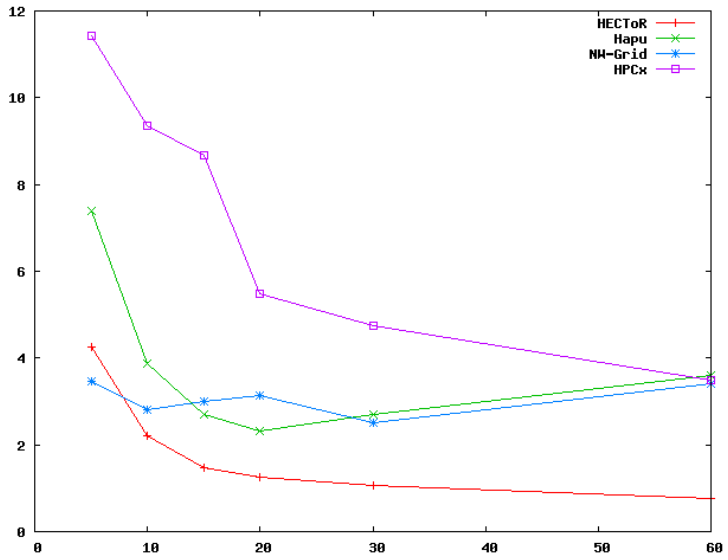

Figure 11: 101,044 agents, 60 regions

Procs	HECToR	NW-Grid	Hapu	HPCx
5	4.260786	3.454667	7.392095	11.422095
10	2.198643	2.821857	3.867690	9.367476
15	1.473929	3.000000	2.704500	8.683452
20	1.258405	3.133048	2.316595	5.470405
30	1.067667	2.514833	2.706690	4.737167
60	0.774833	3.410143	3.605976	3.50000

Table 15: Average iteration time (s) for 240 iterations of 60 region (101,044 agents) EURACE model

there are a number of controlling parameters. The `trading_activity` is one such parameter. It controls the probability that a particular household will place an order and hence it controls the total number of households making requests to the `clearinghouse`. So with the `trading_activity` set to zero no household will make a request but with increasing values more and more households will be making orders. So by performing simulations with varying values of `trading_activity` we can observe the effect on performance of this serial bottleneck.

The experiment performed a series of runs of the model varying both the population size and the value of the trading parameter. From these experiments we should be able to observe the effects of both Amdahl's and Gustafson's Laws. Table 16 and Figure 12 display these results. We see that varying this parameter has a dramatic effect on the performance of the EURACE model. With trading set to zero there is the increase in elapsed time with population one hopes for - linear in population size. However once trading is activated the performance becomes much more complex. For the large values this serial part appears to dominate the simulation. For low values of the trading parameter we see an initial increase in elapsed time - maybe following Amdahl's Law - but then a decrease, as the population size grows. This could be attributed to the Gustafson effect but the elapsed time returns to growth as the population continues to increase. Understanding the effect of varying this single parameter is clearly complex. However it does show how a small serial component of the model can have a dramatic effect on its parallel performance.

Regions/Proc	No. Agents	Trading Activity			
		T=0.02	T=0.01	T=0.005	T=0.0
1	16844	97.906	37.163	18.403	2.322
2	33684	617.120	339.146	96.711	6.822
3	50524	4533.217	1442.034	598.720	13.447
4	67364	8547.810	2547.859	1077.721	17.077
5	84204	-	5784.514	3054.110	26.057
6	101044	-	2620.614	1074.894	35.352
7	117884	-	1010.944	554.409	45.040
8	134724	-	761.719	305.876	57.390
9	151564	-	733.395	415.251	66.603
10	168404	-	-	967.845	81.746
20	336804	-	-	3619.854	257.164
30	504712	-	-	-	640.419

Table 16: Solution time (secs) per 10 iteration with varying parameter values

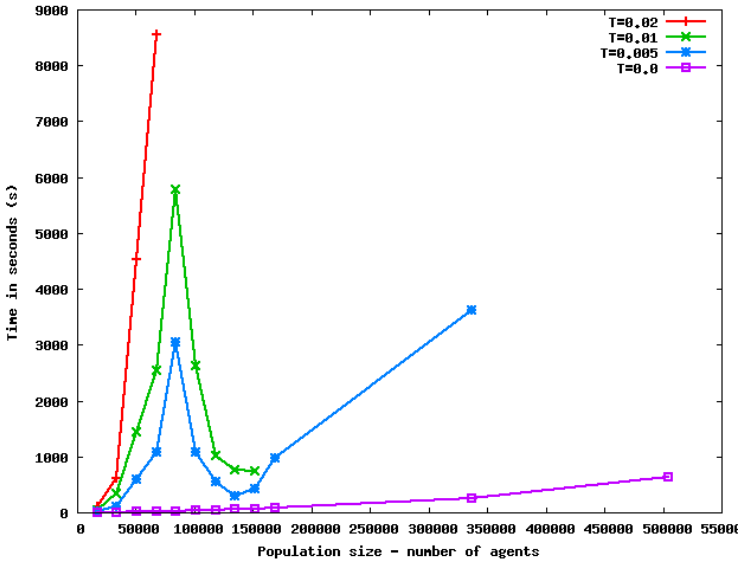

Figure 12: Elasped time with parameter variation

8 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 FLAME's use in a number of EURACE related simulations in addition to the complete EURACE model on populations ranging from a few hundreds of agents, through to tens and hundreds of thousands of agents. In some of these simulations the parallel implementation of FLAME has shown reasonable scalability and parallel efficiency but in others 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 achieved this to a degree: the model has been defined, important parameters have been given 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 performed. In the process a detailed assessment of the FLAME generated code, the serial and parallel implementations and the EURACE model itself have been performed. Message counts, function

times and synchronisation times are a few of the measures that have been used together with a detailed static analysis of the model to identify the performance deficiencies in both the FLAME framework and the EURACE model.

All this analysis has led to improvements in FLAME and the EURACE Model which in general have increased its computational performance. However the presence of substantial serial components in the model has resulted in very poor parallel scalability. It is well known that parallel speed-up 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 `clearinghouse` - have a significant impact on the parallel performance of the model. These types of potential problems were understood at the start of the project and the modellers took steps to avoid them. The `clearinghouse` 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 `clearinghouse` 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] C. Greenough, DJ Worth, LS Chin, M. Holcombe and S Coakley (2009), Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework, EURACE Project Report D1.4, 2009

[2] The EURACE Project Web - http://www.eurace.org

[3] L.S. Chin, D.J. Worth and C. Greenough (2010), An Approach to the Parallel Implementaton of Agent-Based Simulations, to be published as a Rutherford Appleton Laboratory Technical Report

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

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

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

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.

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

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.

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.

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 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 follow 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 Mode*l is one such application. The circle 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`.

outputdata : post agent's position on the message board.

inputdata : read message board information to find nearest neighbours.

move : computes to necessary change in position for this iteration of the model

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 `circle` agent. Each test has `model.xmml` file and a set of initial data (`0.xml`).

Test 1 : Model: single `circle` agent type; Initial population of no agents.

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

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

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

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

Test 6 : Model: Four `circle` 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 `circle` agent also provides a good mechanism to check the parallel implementation against the serial. Such is the nature of the model, that 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.