Agent-based Technologies in EURACE

EURACE Winter School
Genoa, 18-21 Nov 2009

Shawn Chin, Chris Greenough, David Worth

Software Engineering Group
Computational Science & Engineering Department
Rutherford Appleton Laboratory

shawn.chin@stfc.ac.uk
christopher.greenough@stfc.ac.uk
david.worth@stfc.ac.uk

Overview

- (a very brief) Introduction to agent-based modelling
- FLAME Framework + Simple demo
- Introduction to parallelism
- Parallelism in FLAME
- Notes on designing efficient models

Agent-based modelling

- A bottom-up approach: simple behavioural rules of agents at micro level generate complex behaviour at macro level
- Benefits:
  - Involves natural description of a system
  - Flexible and extensible
  - Can reproduce emergent phenomena

[Experiment] *Start with 10-15 people in random position. Each person does the following:*

• Select two others at random, remember them as friend A and friend B

• RULE 1: Move freely while trying to keep A between yourself and B (imagine that A is protecting you from B)

• RULE 2: Move freely while trying to keep yourself between A and B (imagine that you are protecting A from B)

FLAME
Flexible Large-scale Agent Modelling Environment

- An agent-based modelling framework
- Initially developed by Simon Coakley (University of Sheffield). Extended in collaboration with STFC.
- Originally targeted at biological systems
- Developed further under the EURACE project:
  - Now support larger class of models (e.g. economic models)
  - Extension of the X-Machine Markup Language (XMML)
  - Optimised performance (serial and parallel)
  - Ported to various HPC machines (supercomputers) and Operating Systems

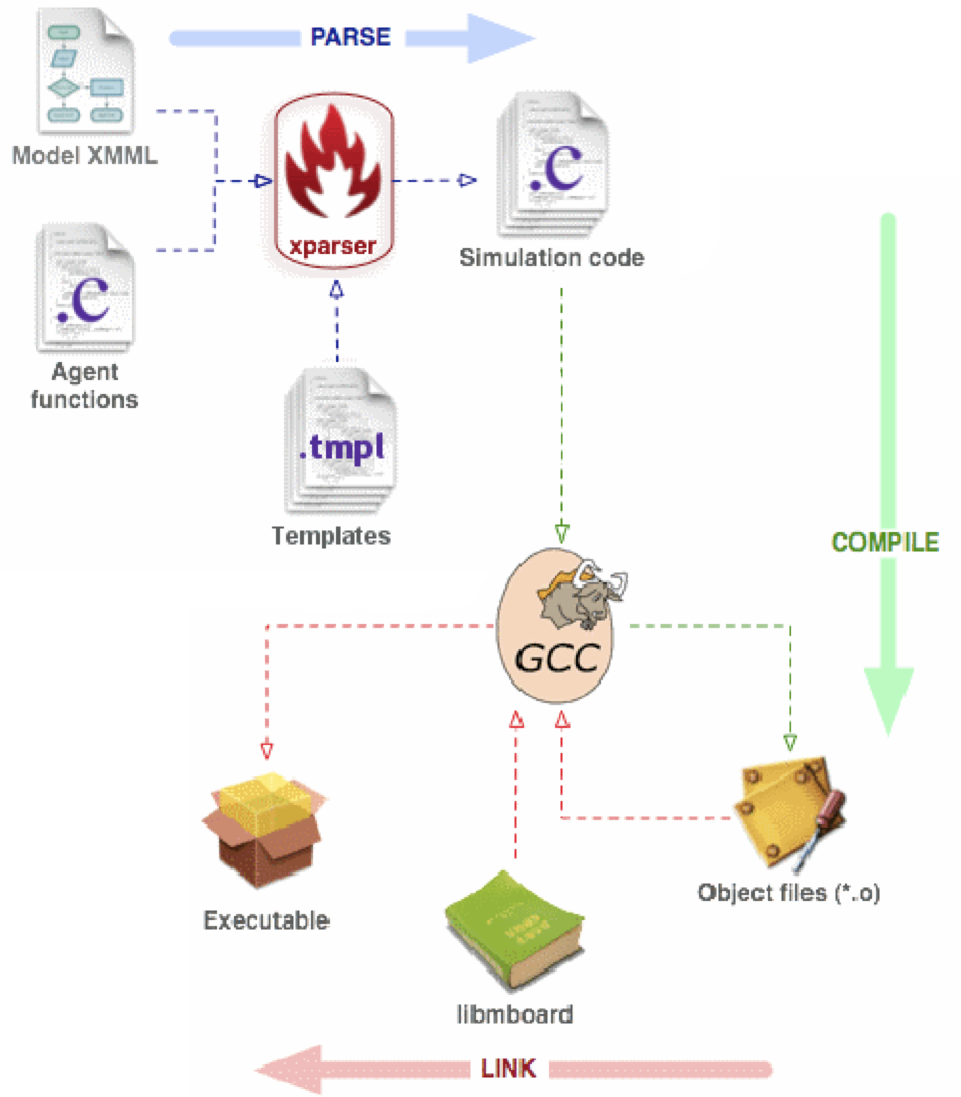
FLAME consists of two components

- **Xparser**
  - Tool that generates application based on defined model
  - Can generate both serial and parallel simulations
  - Generates state diagrams
  - Generates Makefile (automate compilation)

- **Message Board Library**
  - Supporting library that handles data management
  - Enables agents to interact efficiently with environment
  - Allows the simulation to be run in parallel

Using FLAME

1. Describe the model
2. Code behaviour of each agent
3. Generate simulation code using the Xparser
4. Build the executable using “make”
   - Compiles code to object files
   - Links with necessary libraries, include Message Board library
5. Run the executable on an initial population
6. Observe results



Science & Technology Facilities Council

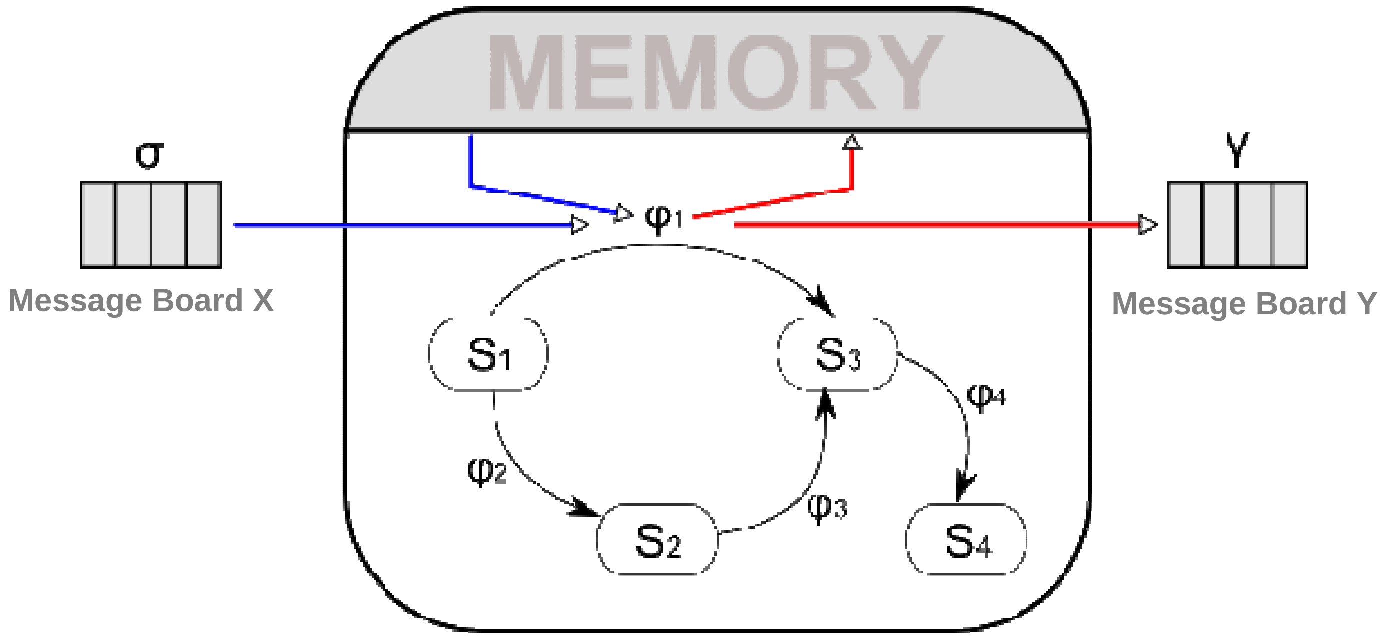
Creating a model

What do you need to define?

- **Agents**
  - Memory
  - Behaviour
- **Messages** (information flow between agents)
- **Optional extras**
  - Environment constants
  - Custom data types
  - Custom time units

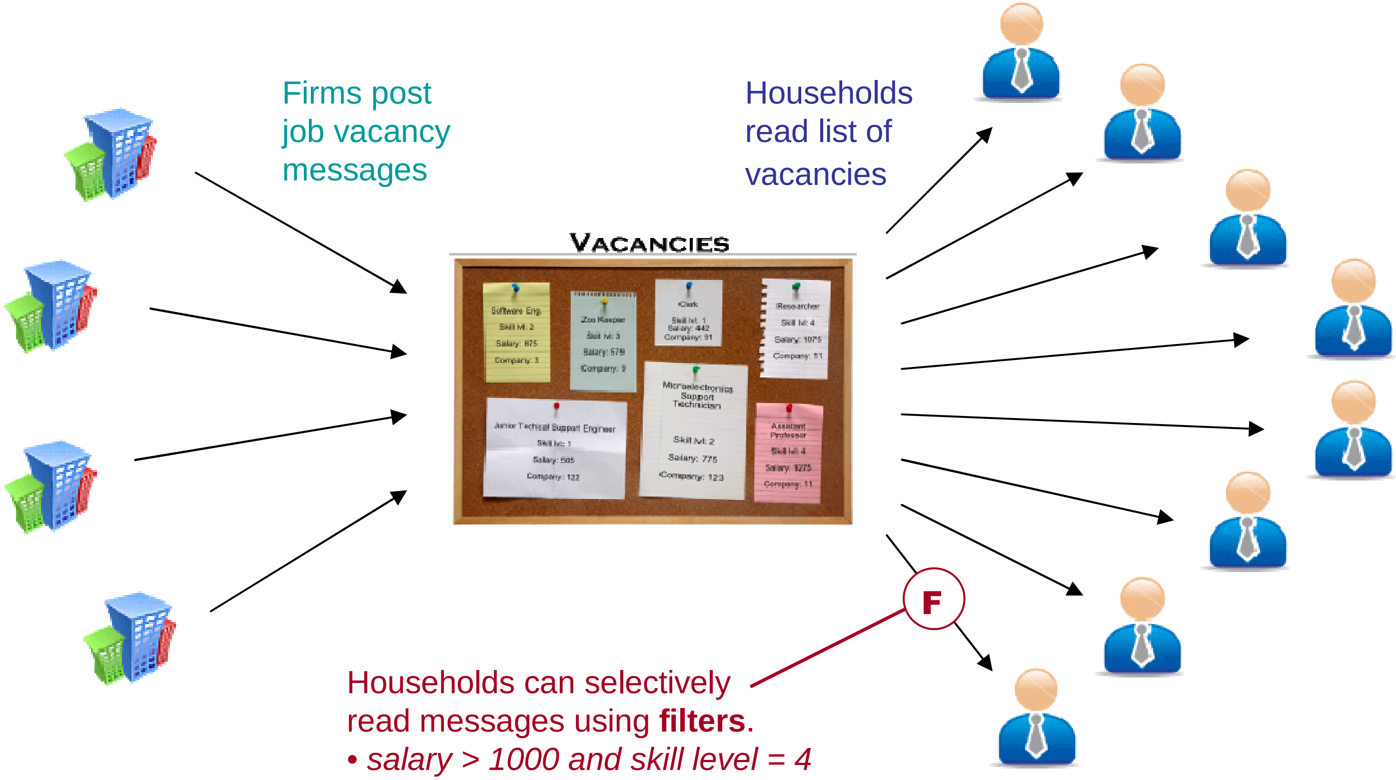
Agents

Agents defined based on formal concept of Communicating Stream X-Machines (CXSM)



Message Boards

Agents communicate through message boards.



For **efficiency** and **consistency**, there are some constraints:

- No direct agent-to-agent communication
  - Use filters to achieve same effect
- Agents cannot remove messages from boards
  - Boards are automatically cleared after each iteration
- Agents cannot both write and read the same board within a function
  - Use separate functions. First write, then read.

Science & Technology Facilities Council

Demo

Modelling the ABM experiment

Agent behaviour:

- Start at random position, facing random direction, and having two random friends
- Move one step in that direction
- Adjust direction based on friends' location (RULE 1 or 2)

In this demo, we will:

- Watch how to create a simple model from scratch
- Run the simulation and view the results
- Examine the generate state graph
- Examine the generate process flow graph
- Explore how FLAME generates code based on model definition

Parallelism

Why run in parallel?

- Large simulation - Cannot fit in memory of one processor
- Long runtime - Time to solution prohibitive

Why is it difficult?

- Efficient parallelism non-trivial and time consuming
- HPC architecture complex and always changing!
- Optimisation of a parallel simulation often architecture dependent

With FLAME, parallel code can be generated automatically

- Easy and portable
- Future-proof – no need to rewrite all models for new architecture, just use updated FLAME

Parallel performance

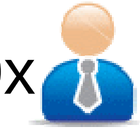
- The performance of a parallel application is often judge by its scalability – how many processors can it make efficient use of.

Q: “If I use N processors, will my simulation run N-times faster?”

Short answer: In most cases, **no**.

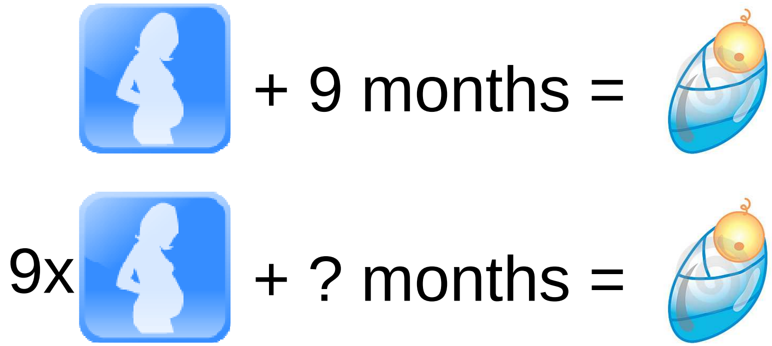
Science & Technology Facilities Council

 + 10 hour = 100 origami birds 

10x  + ? hour = 100 origami birds 

- Some algorithms scale very well – individual units of work are independent of each other
- Often referred to as “embarrassingly parallel” tasks

Science & Technology Facilities Council



- Some algorithms cannot be parallelised – each work unit depends on the outcome of the previous

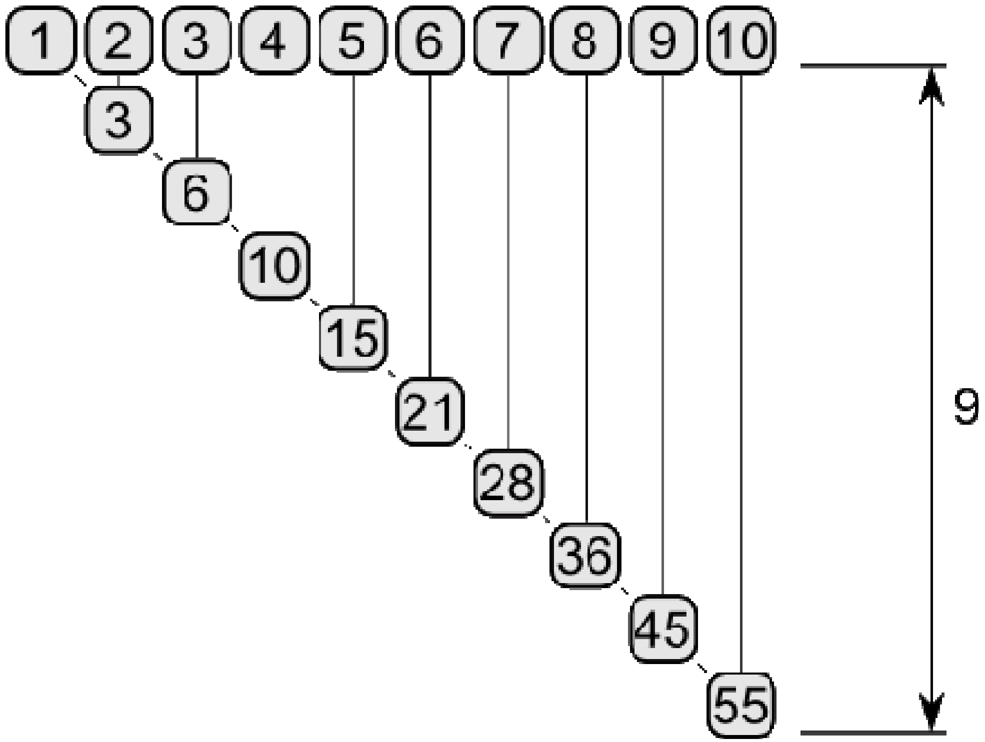
 + 10 hour = a $30m^3$ hole 

10x  + ? hour = a $30m^3$ hole 

1000x  + ? hour = a $30m^3$ hole 

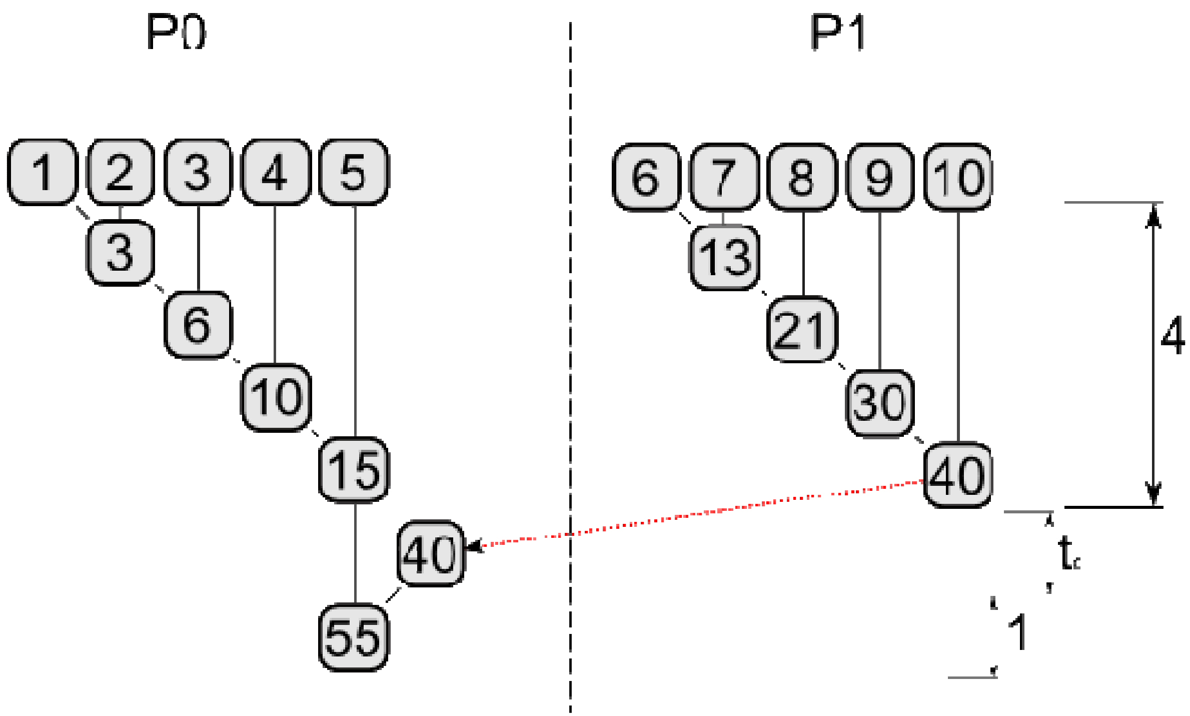
- In general, parallelised algorithms will scale reasonably well up to a certain level, after which you get diminishing returns

One person adding 10 numbers: $T_{1p} = 9$



Two people adding 10 numbers: $T_{2p} = 5 + t_c$

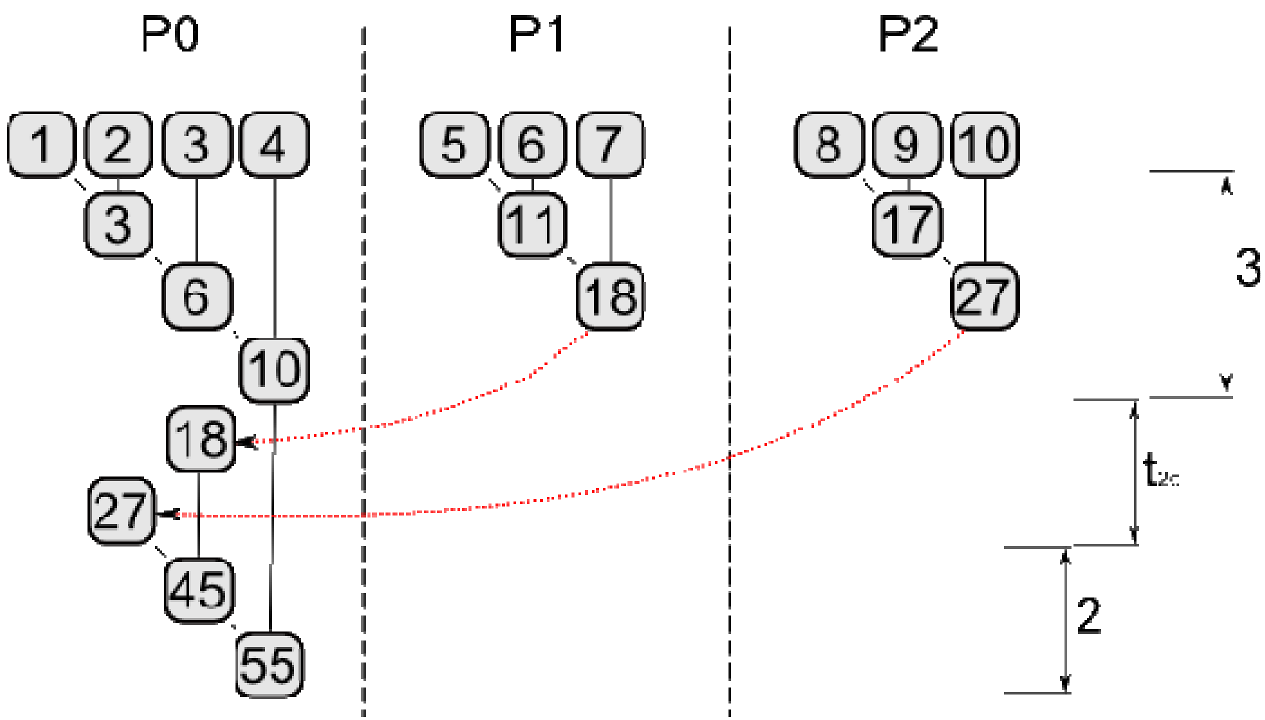
Where t_c is the time taken by P0 to gather sum from P1



Three people adding 10 numbers: $T_{3p} = 5 + t_{2c}$

Where t_{2c} is the time taken by P0 to gather sums from P1 and P2

- Would take longer if load is not balanced



Issues that affects scalability:

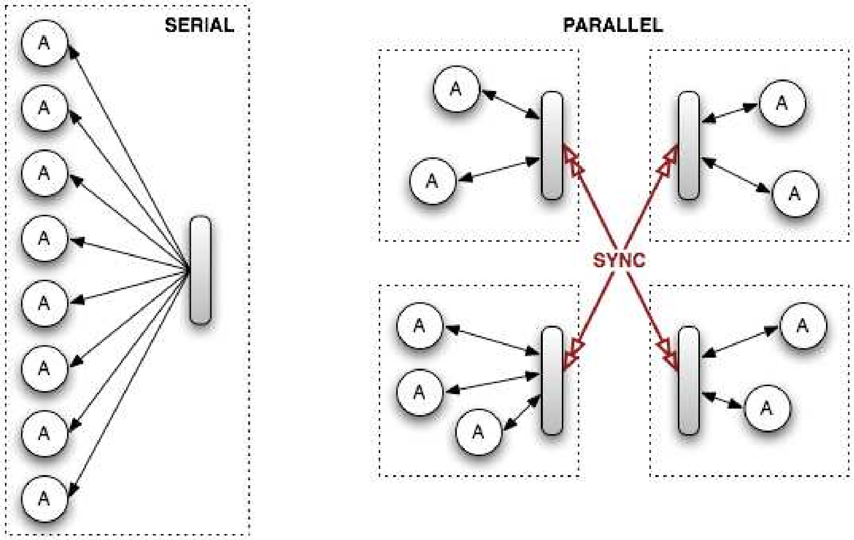
- Serial bottlenecks
  - Directly influence the maximum number of processors an application can scale to (see *Amdahl's Law*)
- Communication overheads
  - May increase as more processors are used
  - May increase with problem size
- Load balance
  - When there are dependencies between processors, total time to solution will be dictated by slowest processor

Parallelism in FLAME

“Agents interact with other agents only through messages”

Parallelism achieved by:

- Distributing agents across multiple processors
- Ensuring that agents have equal access to messages



"Distributing agents across multiple processors"

For efficiency, we ideally want:

- Balanced computational load
- Minimal communication overheads
- Optimal use of memory

In practice, can only choose **one**!

So… compromise!

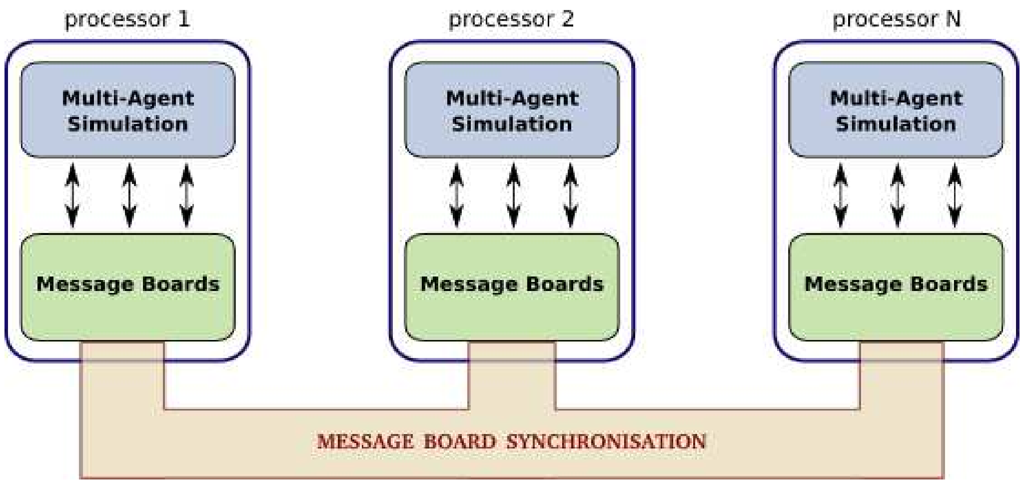
To achieve good compromise:

- Context aware population partitioning
  - Geometric partitioning for agents with spatial information
  - Consideration of weighted interaction graphs
  - Consideration of weighted agent function dependency graph
- Dynamic agent migration
  - Computation load and communication pattern change during simulation; adapt accordingly

(need further research)

“Ensuring that agents have equal access to messages”

- Message Boards library handles synchronisation of boards across all processors



For efficiency, we cannot simply replicate messages

- High latency of inter-process communication
- High memory usage

What we do:

- Data synchronisation performed in background thread to overlap with computation (agent functions)
- Simultaneous synchronisation of multiple boards
- Support for multiple architecture-dependent sync algorithms
- Selective replication of messages based on agent requirements
- Caching of agent constants used as filter parameters

Requires extra information from modellers

- Message filters, constants
- Communication networks (under development)

Science & Technology Facilities Council

Designing efficient models

- Minimise serial bottlenecks
- Write messages as early as possible, read as late as possible
- Use message filters whenever possible
  - Agent locality
  - Recipient-centric message definitions
- Declare agent memory as constant if they never change
- Use of framework functions whenever possible (sort, random)

Questions?