Optimising X-Agent Models in Computational Biology

Dr. D. J. Worth and Prof. C. Greenough

Software Engineering Group Note SEG-N-001

Abstract

This note describes the work done on optimising an x-agent model that is used as a basis for calculations in computational biology. Optimisation of the computational implementation is a serious task as agent based modelling of biological systems seeks to use many millions of agents and will be run on parallel machines. The optimisation reported here is for the serial code only (a necessary first step) and timings are given to illustrate the speed increases obtained. The original code had a parallelisation structure in mind and this note also includes thoughts on parallelisation of the code.

1 Background

An introduction to the topics of x-agents (x-machines) and computational systems biology is provided by [1] which can be summarised as follows:

- X-agents use their own memory data and functions to map input data to output data.
- X-agents can communicate with each other via messages.
- Biolgical systems comprise biological entities.
- Computational biological systems comprise x-agents, one per entity in the natural system.
- Individual agents have properties analogous to the biological entity.

Researchers at the University of Sheffield have developed a framework for building x-agent models of biological systems based on the X-Machine Agent Markup Language (XMML) [2]. XMML files define the types of agents, their memory data and functions involved in a model and can be parsed to create a C program that will run a model for a number of iterations given input of a set of x-agents. The C program can be either a serial or a parallel program, the parallel program using MPI to pass messages between agents on different nodes.

2 The Optimisation Study

The aim of this work was to take existing source code and seek to "improve" it in the following ways:

- Make the source code more readable
- Make the algorithm easier to understand
- Increase the speed of execution
- Make it more flexible

We also had in mind the parallel implementation of the algorithm and how any improvements made to the serial code could be transferred to the parallel version.

3 Original Code

The original code was sent by Simon Coakley of DCS Sheffield. It is a serial version of the x-agent code generated by XMML2C from an xml file describing the models (using XMML, a markup language

for x-machines) for agent interactions. It is written in C and has 1049 lines of code. The source file `main.c` was put under version control using Subversion.

In the following discussion the term *partition* is used to refer to a part of the problem domain. In the code this is refered to as a *node*.

3.1 Data Structures

There are three main data structures in the code

xmachine_memory The x-agent. Stores its id, location x and y coordinates and movement forces.

message_location A message sent by an x-agent. Stores its id and location x and y coordinates.

node_information A partition of the domain. Stores its id, x, y and z extents, list of agents in the partition, list of messages for its agents.

All of these data structures are kept in linked lists so in addition to the data mentioned above they also store a pointer to the next object in the list. The pointer is `NULL` if this is the last object. When an object is added to a list it is always added at the end which involves traversing the list from the beginning each time.

3.2 Algorithm

The inputs to the code are the agents (read from an xml file), the number of partitions (called nodes in the code) and the number of iterations for which the model should be run. The partition information is hard-coded in the program, allowing 1, 2, 3 or 4 partitions with fixed coordinates in the problem domain (also fixed). Each partition has a message board that agents send messages to and read messages from in order to interact. As the agents coordinates are read in, each agent is assigned to one of the partitions and once all agents are read the iterations can begin.

An iteration comprises the following steps:

```
Get the first partition in the partition list
    Randomise the list of agents
    Get the first agent in the current partition
        Agent outputs data by posting a message to the message
        board of its containing partition
    Get the next agent
Get the next partition
Get the first partition in the partition list
    Randomise the list of agents
    Get the first agent in the current partition
        Agent retrieves messages from the message board of its con-
        taining partition and acts on any it needs to
    Get the next agent
Get the next partition
Get the first partition in the partition list
    Randomise the list of agents
    Get the first agent in the current partition
        Agent updates its position
    Get the next agent
Get the next partition
```

It is obvious that there are three parts to the algorithm given above which we describe in the following sections.

3.3 Step 1 - Output Data

Each agent in turn is asked to output its data which in this simple model is its id and location. The agent sends a message with this data in to its containing partition. When the message is received by the partition it is tested to check whether it originated in the halo region of the partition, if it did a new message containing the same data is sent to all the other partitions.

3.4 Step 2 - Input Data

Each agent in turn is asked to act upon messages stored by its containing partition. The agent ignores messages sent by itself but for all other messages it calculates the separation distance d_s defined by

$$d_s = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - 2r_c$$

where x_1, y_1 are the coordinates of the agent, x_2, y_2 are the coordinates of the message and r_c is the cell radius (set to 2 units here).

All messages with $d_s \leq r_c$ of the agent contribute to the action on the agent using the following equations for action in x and y directions respectively.

$$f_x = kd_s(x_2 - x_1)/\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
$$f_y = kd_s(y_2 - y_1)/\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

where $k = 0$ for $0 < d_s \leq r_c$ and $k = 0.1$ for $d_s \leq 0$.

The total action on the agent in the x and y directions is the sum of the f_xs and f_ys.

3.5 Step 3 - Move

Each agent is asked to move by adding the total action on the agent in the x direction to its x coordinate and adding the total action on the agent in the y direction to its y coordinate.

4 A Bug to Fix!

A bug was discovered in the code where the global variable `p_location_messages` set in `main()` to the message list for the current node was overwritten in `location_message_processing` when messages were sent to other nodes for agents in the halo region. The fix was to add a local variable in `location_message_processing` to store the message list before the loop over the other partitions and the write the stored value back to `p_location_messages` when the loop was complete.

This version of the source file is in **revision 2** of the Subversion repository.

This bug had already been identified and fixed by Simon Coakley in his version of the code.

5 Improvements

A number of improvements have been made to the code in order to reduce the execution time. It was clear from profiling results (using gprof and oprofile) that the majority of the execution time was spent in the input data phase (step 2). In particular calculating the distance between the agent and the messages was an $O(n^2)$ operation for n agents. One of the main thrusts wass therefore to reduce the number of times this operation is carried out, which is in fact the motivation for partitioning the agents. Partitioning should in theory reduce the number of distance calculations by a factor of n_p where n_p is the number of partitions as the work per partition is $(n/n_p)^2$.

5.1 Global Variables

The code has too many global variables which leads to confusing code, functions with hidden side effects and in particular to the bug described in Section 4. Therefore the following global variables

have been removed:

- `p_xmemory` - List of x-agents for current partition.
- `xmemory` - Current x-agent.
- `p_location_messages` - List of messages for current partition.
- `location_message` - Current message.
- `current_node` - Current partition.

The data they represent can be passed as function arguments making the objects a function works with more apparent and hopefully avoiding the confusion that existed before. One global variable remains - `p_node_info` - the list of partitions. It is required by a function at the end of a call chain in which the intervening functions have no need of it and so was left as a global variable.

This version of the source file is in **revision 6** of the Subversion repository.

5.2 Use of `SPINF`

The global value `SPINF` is used to designate the end of the domain, either as `-SPINF` or `SPINF`. Sometimes both the minimum and maximum coordinates of a partition use the positive value so in checks for the edge of the domain (e.g. in `location_message_processing`) we should use `fabs()`.

This version of the source file is in **revision 8** of the Subversion repository.

5.3 Split Partitioning From Reading in Agents

At present the partitioning is fixed and known before the agents are read in. This is OK for uniform agent distributions but not when the agents are clumped into one part of the domain. Also the number of partitions is hard coded as 1, 2, 3 or 4 and we should have something more flexible. The algorithm should be

1. Read all agents into a list
2. Decide on partitions based on agents in list
3. Distribute agents to the partitions.

Step 1 is done by `readinitialstates` so remove the distribution of agents from this function. Step 2 requires a new function `partition_data` that, given the number of partitions creates them. Use the hard coded partitions for now. Step 3 is part of `partition_data`.

This version of the source file is in **revision 9** of the Subversion repository.

5.4 Neighbours and Halo Regions

At present an agent looks at all messages that its partition that aren't from itself. This can be improved by noting that agents not in the halo region do not need to consider messages from other nodes. This involves having a flag on the agent to say whether it is in the halo region or not and a flag on the message to say whether it comes from a halo region or not. The implementation is

`xmachine_memory.in_halo` = 1 if in halo, 0 if not

`message_location.from_halo` = 1 if from halo, 0 if not.

This works because only those messages sent to other partitions are marked as coming from a halo region, messages within a partition are not marked even if they do come from a halo region.

Then in `inputdata()` an agent can use the flags to ignore messages from other partitions if it isn't in the halo region.

This version of the source file is in **revision 10** of the Subversion repository.

Using the coverage tool `gcov` we can check execution count of the line calculating the distance between agents and messages for 100 agents doing 150 iterations, see Table 1.

	Revision 9	Revision 10
1 partition	149850000	149850000
2 partitions	91234531	78702190
4 partitions	81256694	55698788

Table 1: Execution count of distance calculations between agents and messages

In a similar vein the code as it stands has an agent in a halo region sending messages to all other partitions, not just the ones with a neighbouring halo region to where the agent is found. It would be more efficient to restrict the messages sent to just the neighbouring partitions but that involves more data them we have at present on the partitioning. This data may be available from an improved partitioning algorithm.

5.5 Randomisation

The randomisation of agents in each node is not as efficient as it could be. There are multiple loops through the linked list of agents which is unnecessary. A better algorithm would be

1. Create an empty array of pointers to agents
2. Loop through the agent list and fill the array with the pointers to the agents
3. Generate random numbers 0 to $n - 1$ to pick agents from the array and add them to a new

linked list. Take care to ensure that when the same random number is chosen it is ignored and the process keeps going until all the agents are in the new list.

4. Set the new list on the node.

All this is done with pointers so no new memory is created.

The profiling tool `oprofile` shows that the time spent in randomising the agents is down from 10-20% to approximately 4% of the total time.

This version of the source file is in **revision 12** of the Subversion repository.

5.6 Partitioning

Initially we have implemented a repeated bisection algorithm that creates 2^n partitions for $n \geq 0$. The x and y extents of the domain are supplied and then the domain is bisected alternately in the x and y directions for n cycles. At present the whole domain is partitioned once when all the agents have been read in, but it could be extended to partition only the region containing the cloud of agents. The partitioning could be repeated at the end of each iteration to ensure the work load for each partition remains balanced.

6 Parallelisation

The parallelisation strategy is clear from the algorithm. Divide the region into a number of partitions and have a separate process work with the agents in a single partition. Messages from agents in the halo regions are sent to other partitions by some message passing protocol. Some points to bear in mind for the parallel code are:

- Ensure that the work load for each process is balanced by getting the same number of agents in each partition.
- The messages contain very little data. It may be very costly to start the mesage passing protocol compared with sending a single message so consider buffering them until the amount of data to be sent makes sending it worthwhile.

References

[1] http://www.dcs.shef.ac.uk/~rod/Integrative_Systems_Biology.html

[2] http://www.dcs.shef.ac.uk/~stc/x-agents/index.shtml