PERFORMANCE OF A MULTI-AGENT SIMULATION ON A DISTRIBUTED BLACKBOARD SYSTEM

KUM WAH CHOY, ADRIAN A. HOPGOOD, LARS NOLLE, BRIAN C. O'NEILL

Nottingham Trent University
School of Computing and Informatics,
Burton Street,
Nottingham NG1 4BU
United Kingdom
E-Mail: {kw.choy, adrian.hopgood, lars.nolle, brian.oneill}@ntu.ac.uk

Abstract: Research into multi-agent systems has increased over several years, leading to demand for multi-agent system simulators to investigate the interaction between different agents and their environment. The more complex and intelligent the agents become, the more difficult it is to simulate them. Distributing the simulation across different processors would improve its performance. This paper describes the implementation and the performance of a multi-agent test-bed called TileWorld on a distributed blackboard system, DARBS. The performance of the TileWorld simulation on a distributed blackboard system running on single processor and on multi-processors is also investigated.

Keywords: TileWorld, multi-agent simulator, distributed blackboard system, DARBS, distributed processing network

1. INTRODUCTION

Multi-agent systems research is currently on the increase. Investigating the behaviour of the agents' interaction with each other and with the environment is inevitably part of this research. As the behaviours of agents are complex and difficult to predictable, there is a need for a test-bed to simulate the agents in an artificial environment that can be both dynamic and static. TileWorld [Pollack and Ringuette, 1990] is a well-established test-bed for agent systems and the multi-agent version of it is called MA-TileWorld [Ephrati et al., 1995].

Unfortunately, this type of simulation becomes processor-intensive as the number of agents increases. An ideal solution to this would be to distribute the agents across different processors in a distributed processing network. To do this, a suitable distributed architecture is required. A distributed blackboard system such as DARBS (Distributed Algorithmic and Rule-based Blackboard System) [Nolle et al., 2001] would be a suitable architecture. This would increase the performance of the simulation and this paper will investigate the performance characteristics of a TileWorld simulation in a distributed blackboard system.

1.1 TileWorld Test-bed

From here onwards, MA-TileWorld will simply be referred to as TileWorld. In this test-bed, the world is set-up as a two dimensional grid. There are agents, tiles, holes, and obstacles in the TileWorld. The objective of the agents is to score as many points as possible. They score points by moving around the TileWorld to find and pick up tiles which they put into holes. The agents have a limited view of the TileWorld. The viewing radius is a variable that can be set by the designer. For example, in the DARBS TileWorld test-bed, the agents have a viewing radius of five cells. Each cell can only be occupied by one agent at a time. Agents cannot move to a cell with an obstacle in it. An example of a 10 × 10 TileWorld is shown in Figure 1.

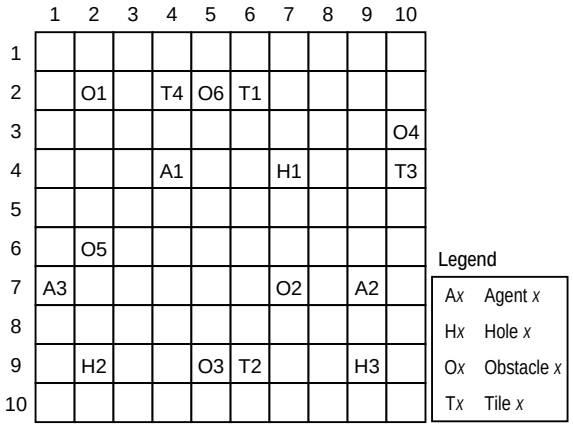


Figure 1: A 10 × 10 TileWorld

The holes, obstacles, and tiles in the TileWorld can change dynamically, i.e. they can appear and disappear in different locations in the TileWorld. The rate of change is set by a variable, and this can be used to reflect the dynamically changing real-world environment. The TileWorld test-bed has been widely used to test the behaviour and interaction of multiple agents in a dynamic environment [Lees et al., 2003; Kinny et al., 1992]. There is also a variant of the original TileWorld test-bed that includes "gas station" objects to top up the resources of the agents [Uhrmacher

and Schattenberg, 1998]. In this variant, the agent's resource-management skill is investigated by making each move consume fuel. Carrying a tile would cause the agent to consume more fuel. Therefore the agent would need to balance the consumption of resources with scoring points.

1.2 DARBS

A blackboard system is an artificial intelligence (AI) technique that is analogous to a team of experts who communicate their ideas by writing them on a blackboard [Engelmore and Morgan, 1988]. The experts are represented by sets of rules, conventional procedures, neural networks, or other program modules. These modules are termed knowledge sources (KS). The blackboard is an area of global memory containing evolving information. The system's current state of understanding of a problem is stored here as it develops from a set of data towards a conclusion. DARBS is a distributed blackboard system developed at the Open University and Nottingham Trent University [Nolle et al., 2001]. A general structure of DARBS can be seen in Figure 2. The original non-distributed blackboard system was called ARBS and was run on a single processor machine [Hopgood et al., 1998].

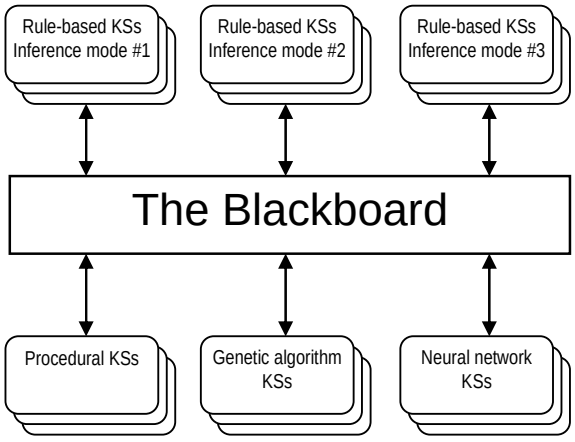


Figure 2: General DARBS structure

DARBS was selected for this work as it is a research-based distributed blackboard system that does not have a central control module. Commercially available distributed blackboard systems were not considered owing to their cost and the limitations on access to their source code.

In DARBS, the blackboard system is modelled based on the client/server model where the blackboard (BB) is the server and the KSs are the clients. Therefore, DARBS can be seen as a distributed blackboard system with the blackboard server running on one PC and other KS clients running on different PCs. This allows different KSs to run in parallel and thus to be truly opportunistic [Engelmore and Morgan, 1988]. For this reason, DARBS may also be considered as a multi-agent collaborative system [Jennings and Wooldridge, 1998]. As different KS clients can be run on different PCs, the agents in the TileWorld test-bed can run independently by running each agent as a KS client.

Information on the BB is organised into different levels/partitions. Each KS can then work on a set of these partitions to solve the overall problem on the BB. DARBS does not have a control module and as such uses broadcast messages to inform other KSs that a change in a particular partition of the blackboard has occurred. It is then up to the KS to decide what to do. It may either stop what it is doing and check exactly what has changed or it may finish what it is doing and then check.

1.3 Design criteria

One of the benefits of the blackboard architecture is that there is central storage of information about the current problem. All KSs store their working memory on the blackboard, visible to other KSs or users. New KSs can then be developed that can make use of this information or the information can be use for debugging purposes. A balance is needed, as too little information on the blackboard defeats the purpose of central storage and too much information would create high communication overhead. Partitioning the information on the blackboard can help, but again there is a balance. Too much information on a single partition would slow-down the blackboard server in searching for the information, and too many partitions would cause the KS clients to monitor more partitions for changes.

The format of the information stored on the blackboard is also important. The more intelligible the information, the greater the amount of data that is used to store it and the more data the communication channel has to send. Also putting the information in a format that is difficult to search or query would slow-down the overall system. With all these criteria in mind, TileWorld was designed and implemented on DARBS.

2. DESIGNING TILEWORLD ON DARBS

The natural way the TileWorld test-bed fits a blackboard system made it ideal for DARBS. All the TileWorld agents and objects interact with each other through a single TileWorld environment, hence, the agents can be directly implemented as KS clients and the world itself can reside on the blackboard. This gives all the agents equal access to view and change the world, thus allowing true parallel agent simulations. The setup of the TileWorld on DARBS is as shown in Figure 3.

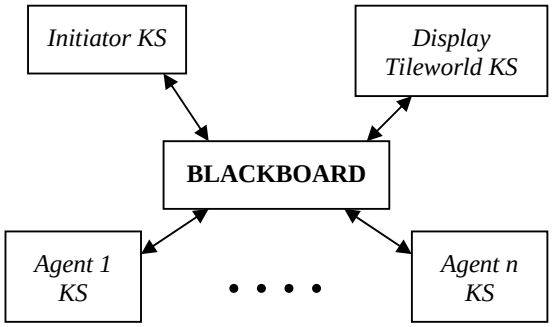


Figure 3: TileWorld on DARBS

The *Initiator KS* is the KS that sets up the TileWorld with its parameters and generates the world. The *Display TileWorld KS* displays the TileWorld and its contents in a graphical format so that users can follow the simulation easily. This will also help the users understand the simulation better. The *Display TileWorld KS* also needs to keep track of the changes in the world and make sure that the graphical representation is as up-to-date as possible. Each *Agent KS* controls their respective agent on the TileWorld and makes changes to the world according to the behaviours that are coded in them.

To simplify the design, the objects in the TileWorld (i.e. tiles, holes, and obstacles) cannot move themselves. However, it is possible to make the objects in the TileWorld dynamic, i.e. appear and disappear with time by simply adding another KS that uses a probability rate to change the number and position of tiles, holes, and obstacles in the TileWorld. One of the benefits of the blackboard architecture is that new KSs can easily be added to provide more agents or to change the way the TileWorld is being simulated.

Careful organisation of the data on the blackboard is required to make sure that the agents and all other KSs can interact with each other properly through the blackboard. The organisation of the data also needs to minimise the number of partitions with which a particular KS works. This is to reduce the number of times the KS needs to restart, since a KS restarts whenever a partition that is it working with has changed. As mentioned earlier, the format of the data on the blackboard also needs to be carefully constructed so that queries from the KS clients are as easy, efficient and legible as possible. With these criteria in mind, the blackboard was partitioned as shown in Figure 4.

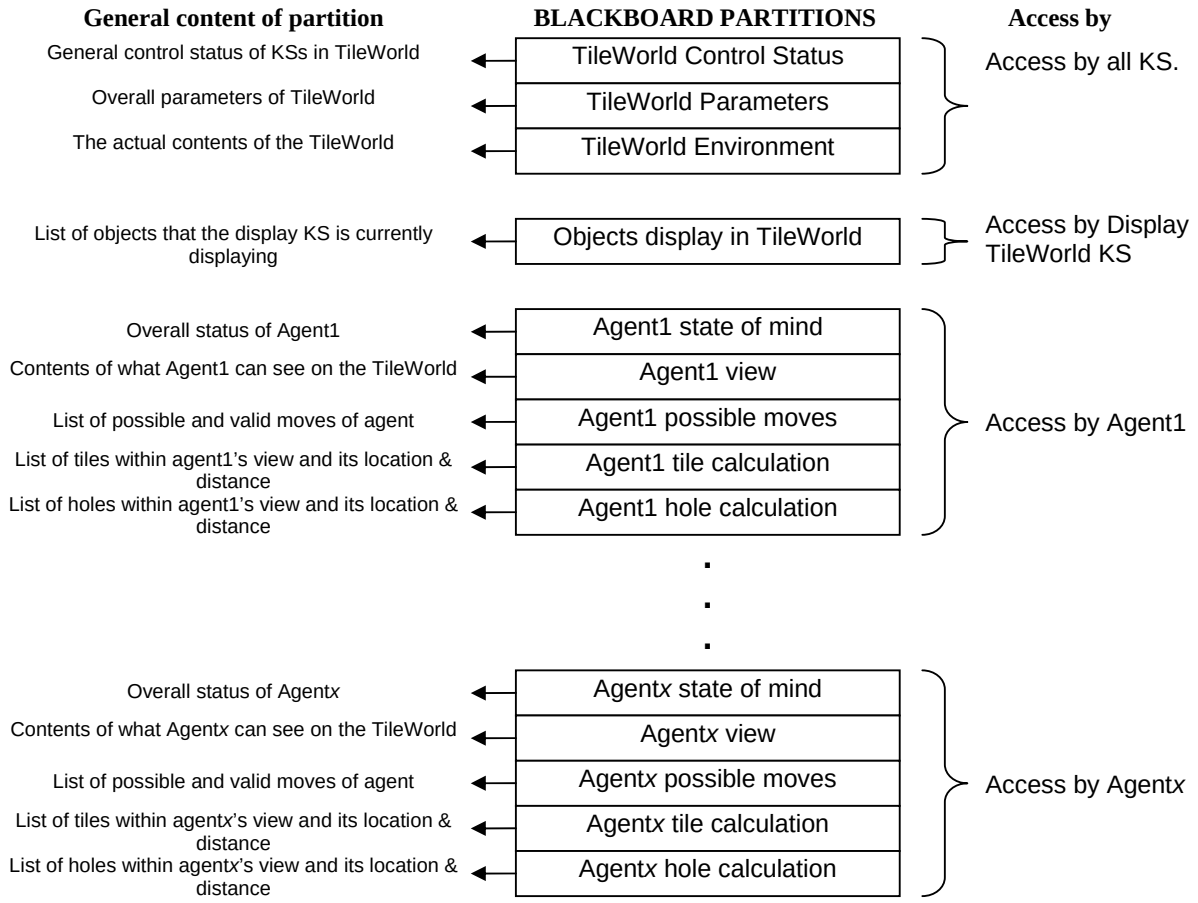


Figure 4: Partitions on the Blackboard

As can be seen in Figure 4, all Agent KSs can access the TileWorld Environment partition and, in theory, can see the whole TileWorld. This is because the actual contents of the TileWorld environment are stored in the TileWorld Environment partition and in order for Agent KSs to be able to change or see the TileWorld, access to the TileWorld Environment partition is needed. This could also be argued to be an incorrect implementation of the TileWorld test-bed, but it is assumed that all agents are benevolent and will only access those areas of the TileWorld Environment partition that they are intended to. The first thing each Agent KS will do is to update its own internal representation of the TileWorld and store it in their Agent View partition. This is done based on the agent's current position in the TileWorld and the viewing range of the agent (set as a parameter in the Agent KS's rule).

There are too many data strings on the blackboard to explain each in detail but the important data string format is that on the TileWorld Environment partition. The data string format on the TileWorld Environment partition is as follows:

[Location x , y contains NO AGENT , NO HOLE , NO OBSTACLE , NO TILE]
[*Agent 1* *Hole 1* *Obstacle 1* *Tile 1*]

The order of objects is important as this will allow easy query by the KSs. The order is Agent→Hole→Obstacle→Tile. If the location contains an agent then *Agent x* where *x* is the agent number will replace *NO AGENT* and the same goes for holes, obstacles and tiles. Examples of these are as follows:

[Location 1 , 4 contains Agent 3 , NO HOLE , NO OBSTACLE , NO TILE]
[Location 1 , 5 contains NO AGENT , Hole 21 , NO OBSTACLE , NO TILE]
[Location 1 , 6 contains NO AGENT , NO HOLE , Obstacle 18 , NO TILE]
[Location 2 , 4 contains NO AGENT , NO HOLE , NO OBSTACLE , Tile 2]
[Location 10 , 2 contains Agent 5 , NO HOLE , NO OBSTACLE , Tile 4]

3. IMPLEMENTING THE TILEWORLD

Three types of KSs were implemented in DARBS for this TileWorld test-bed: *Initiator KS*, *Display TileWorld KS*, and *Agent KS*. A screen capture of the TileWorld running on DARBS is shown in Figure 5. Each KS is implemented as a rule-based KS, described below.

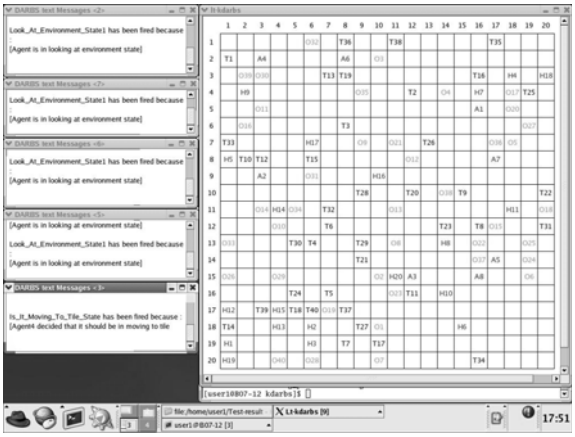


Figure 5: Screen capture of TileWorld on DARBS

3.1 Initiator KS

The main purposes of the *Initiator KS* are to set-up the parameters of the TileWorld and to generate the actual TileWorld from these parameters. These parameters are set in the rules of the KS and they include: the size of the TileWorld and the number of agents, holes, obstacles and tiles in the TileWorld. These parameters are stored on the TileWorld Parameters partition. Once these parameters are set-up on the BB, the *Initiator KS* will create the TileWorld and randomly place the agents, holes, obstacles and tiles in the TileWorld. This is done by using the standard C++'s rand() function with a user define random seed number. Once the TileWorld with all its objects is created and stored on the BB, the *Initiator KS* will terminate.

The two main rules of this KS and their functions are:

- Init_TileWorld
  - Set the size of the TileWorld
  - Set the number of agents in the TileWorld
  - Set the number of holes in the TileWorld
  - Set the number of obstacles in the TileWorld
  - Set the number of tiles in the TileWorld
- Create_TileWorld
  - Generate a random position within the TileWorld for each of the agents, holes, obstacles, and tiles.
  - Store this information on the blackboard.

3.2 Display TileWorld KS

The main purpose of the *Display TileWorld KS* is to display the content of the TileWorld in a graphical form. This is done with the help of the Qt library from Trolltech [Dalheimer 2002]. The Qt library provides the graphical function calls for drawing and updating the graphical TileWorld with its agents, holes, obstacles and tiles.

The *Display TileWorld KS* starts as soon as the TileWorld is created by the *Initiator KS*. The first thing that the *Display TileWorld KS* does is to draw a TileWorld of the size specified in the TileWorld Parameters partition. This TileWorld is drawn in a new GUI (Graphical User Interface) window. It then looks for the position of all the agents, holes, obstacles and tiles in the TileWorld Environment partition and updates the GUI window accordingly. Finally, it checks and deletes tiles that are on the TileWorld GUI but no longer on the TileWorld Environment partition. This is because tiles can be picked up by agents. This final check can also include holes and obstacles if they were dynamic objects. The *Display TileWorld KS* then remains dormant until a change in the TileWorld Environment partition has occurred. This is when the BB broadcast a change in TileWorld Environment partition. The *Display TileWorld KS* will be interrupted whenever the BB broadcast a change in TileWorld Environment partition.

The rules in this KS and their functions are as follows:

- Display_Initial_Screen
  - Draw the TileWorld grid and label it
- Update_Agent_Display
  - Find the location of all the agents in the TileWorld from the blackboard and display them accordingly.
- Update_Hole_Display
  - Find the location of all the holes in the TileWorld from the blackboard and display them accordingly.
- Update_Obstacle_Display
  - Find the location of all the obstacles in the TileWorld from the blackboard and display them accordingly.
- Update_Tile_Display
  - Find the location of all the tiles in the TileWorld from the blackboard and display them accordingly.
- Update_Total_Objects_Display
  - Find the objects that are currently being display in the TileWorld.
- Update_Deleted_Tile
  - Delete the objects that are no longer on the blackboard from the displayed TileWorld.

3.3 Agent KS

The *Agent KS*s control the agents in the TileWorld. The function of each agent is the same as they each have the same objective (i.e. to score points by putting tiles into holes). The general algorithm of an *Agent KS* is as follows:

Initialise and reset *Agent KS's* working memory
View the surrounding area of agent based on the viewing range limit defined in the agent's working memory.
Check to see what state the agent is in based on the following conditions:
If agent is not currently carrying a tile and there is no tile in vicinity OR if agent is carrying a tile and there is no hole is in vicinity, then agent is in Exploring State.
If agent is not currently carrying a tile and there is a tile in the vicinity AND there is no tile in the cell that agent is currently occupying, then agent is in Moving To Tile State.
If agent is currently carrying a tile and is occupying a cell with a hole, then agent is in Hole Filling State.
If agent is currently carrying a tile and there is a hole in vicinity AND there is no hole in the cell that agent is currently occupying, then agent is in Moving To Hole State.
If agent is not currently carrying a tile and the cell that it is occupying has a tile, then agent is in Picking Up Tile State.
Perform the following functions based on the agent's state :
If agent is in Exploring State, then generate and make a random move.
If agent is in Moving To Tile State, then make a move towards the closest tile from the current position.
If agent is in Moving To Hole State, then make a move towards the closest hole from the current position.
If agent is in Picking Up Tile State, then pick tile up.
If agent is in Hole Filling State, then drop tile in hole and calculate score.
Go back to step 2.

This algorithm is implemented as a set of 31 rules on each agent in the TileWorld. The *Agent KS* can be interrupted at anytime to restart the algorithm by a "partition TileWorld Environment changed!" broadcast message from the BB. The following is a list of all the rules with an explanation for selected rules.

- Initialise_Agent
  - Setup the viewing range the agent can see.
  - Initialise the internal state of mind of the agent.
  - Start the agent in Generate SearchSpace State.
- Update_Internal_Status
  - Reset back the agent's internal state of mind to the start state. This is required when the agent is interrupted and restarts itself.
  - Start the agent in Generate SearchSpace State.
- Generate_SearchSpace_State
  - Find out the coordinates that the agent can see based on the viewing range parameter that has been setup.
  - Find out the last coordinates that can be seen.
  - Set the agent to Look At Environment state.
- Look_At_Environment_State1
  - Look at the contents of the environment that the agent can view and place that information into the agent's view partition.
- Look_At_Environment_State2

  - Change the agent state to Thinking state as soon as the last coordinates that can be view by the agent is viewed.
- Is_It_Exploring_State
  - Change the agent state to Exploring state if the agent is in Thinking state and the agent is currently carrying no tile and there is no tile within the viewing range OR
  - Change the agent state to Exploring state if the agent is in Thinking state and the agent is currently carrying a tile and there is no hole within the viewing range.
- Is_It_Moving_To_Tile_State
  - Change the agent state to Moving To Tile state if the agent is in Thinking state and the agent is not carrying a tile and there is a tile within the viewing range and the tile is not on the same grid as the agent.
- Is_It_Hole_Filling_State
  - Change the agent state to Hole Filling state if the agent is in Thinking state and the agent is carrying a tile and the agent is standing on a grid with a hole.
- Is_It_Moving_To_Hole_State
  - Change the agent state to Moving To Hole state if the agent is in Thinking state and the agent is carrying a tile and there is a hole within the viewing range and the hole is not on the same grid as the agent.
- Is_It_Picking_Up_Tile_State
  - Change the agent state to Picking Up Tile state if the agent is in Thinking state and the agent is not carrying a tile and the agent is standing on a grid with a tile.
- Generate_Possible_Moves
  - Generate the 4 or less possible moves that the agent can make and store those moves in the agent's possible moves partition provided that the agent is in Exploring state or Moving To Tile state or Moving To Hole state. Finally added Check North Move Validity flag to the possible moves partition.
- Is_North_Move_Valid
  - If Check North Move Validity flag is set and the north move is valid then set the result on the possible moves partition. Change the Check North Move Validity flag to Check East Move Validity flag.
- Is_North_Move_NotValid
  - If Check North Move Validity flag is set and the north move is not valid then set the result on the possible moves partition. Change the Check North Move Validity flag to Check East Move Validity flag.
- Is_East_Move_Valid
  - If Check East Move Validity flag is set and the east move is valid then set the result on the possible moves partition. Change the Check East Move Validity flag to Check South Move Validity flag.
- Is_East_Move_NotValid
  - If Check East Move Validity flag is set and the east move is not valid then set the result on the possible moves partition. Change the Check East Move Validity flag to Check South Move Validity flag.
- Is_South_Move_Valid
  - If Check South Move Validity flag is set and the south move is valid then set the result on the possible moves partition. Change the Check South Move Validity flag to Check West Move Validity flag.
- Is_South_Move_NotValid
  - If Check South Move Validity flag is set and the south move is not valid then set the result on the possible moves partition. Change the Check South Move Validity flag to Check West Move Validity flag.
- Is_West_Move_Valid
  - If Check West Move Validity flag is set and the west move is valid then set the result on the possible moves partition. Change the Check West Move Validity flag to Finish Checking Moves flag.
- Is_West_Move_NotValid
  - If Check West Move Validity flag is set and the west move is not valid then set the result on the possible moves partition. Change the Check West Move Validity flag to Finish Checking Moves flag.
- Exploring_State
  - Generate a random step based on the possible moves and the last made move. Store the random generated step on the agent's state of mind partition. Change agent's Exploring state to Making A Move state.
- Moving_To_Tile_State
  - If agent is in Moving To Tile state, clear the current closest tile information from the tile calculation partition. Change the agent's state to Get Tile Distance state.
- Moving_To_Hole_State
  - If agent is in Moving To Hole state, clear the current closest hole information from the hole calculation partition. Change the agent's state to Get Hole Distance state.
- Get_Tile_Distance_State
  - If agent is in Get Tile Distance state, calculate the distance of all the tiles within the agent's viewing range and store that information onto the agent's tile calculation partition. Change the agent's state to Finding Closest Tile state.
- Get_Hole_Distance_State
  - If agent is in Get Hole Distance state, calculate the distance of all the holes within the agent's viewing range and store that information onto the agent's hole calculation

partition. Change the agent's state to Finding Closest Hole state.

- Find_Closest_Tile_State
  - If agent is in Finding Closest Tile state, pick the closest tile in the tile calculation partition and store that in the agent's state of mind partition. Change agent's state to Generate Step Closer To Tile state.
- Find_Closest_Hole_State
  - If agent is in Finding Closest Hole state, pick the closest hole in the hole calculation partition and store that in the agent's state of mind partition. Change agent's state to Generate Step Closer To Hole state.
- Generate_Step_Closer_To_Tile_State
  - If agent is in Generate Step Closer To Tile state, generate a random step closer to the closest tile based on the possible moves, last move made, and the coordinates of the closest tile. Store the random generated step on the agent's state of mind partition. Change the agent's state to Making A Move state.
- Generate_Step_Closer_To_Hole_State
  - If agent is in Generate Step Closer To Hole state, generate a random step closer to the closest hole based on the possible moves, last move made, and the coordinates of the closest hole. Store the random generated step on the agent's state of mind partition. Change the agent's state to Making A Move state.
- Pick_Up_Tile_State
  - If agent is in Picking Up Tile state, pick up the tile from the same grid and change state back to Generate Searchspace state.
- Hole_Filling_State
  - Drop the tile it is carrying into the hole it is standing on and recalculate the agent's score. Change agent's state back to Generate Searchspace state.
- Making_Move_State
  - If agent is in Making A Move state, record the current position of the agent as the last move and move the agent to the new location as stored in the agent's state of mind. Change agent's state back to Generate Searchspace state.

4. EXPERIMENT AND RESULTS

TileWorld implementation on DARBS will be referred to as TileWorld-DARBS from here onwards. The experiment carried out investigates the performance of the TileWorld simulation running on a distributed blackboard systems in a non-distributed set-up and in a distributed set-up. To run in a non-distributed set-up, TileWorld-DARBS is run on a single processor. To run in a distributed set-up, TileWorld-DARBS is run on multi-processors (i.e. one processor for the blackboard and one processor for each KS). The distributed set-up will be run in an ideal case, i.e. one processor per process. One of the aims of this experiment is to investigate the affects of adding more KSs to the performance of the TileWorld simulation for both single and multi-processors set-up.

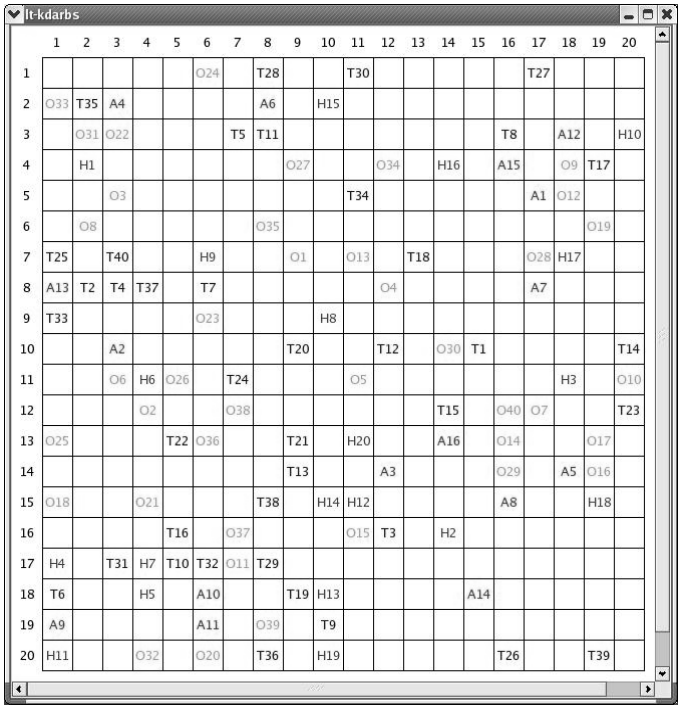


Figure 6: A 20 × 20 TileWorld generated with random seed 8

For this experiment, an Ethernet 100Mbps switch network of personal computers (PCs) is used. All the PCs used are AMD Athlon 1.67GHz processors with 224 megabytes (MB) of random access memory (RAM) running Red Hat 9 operating system with Linux kernel 2.4. A 20 × 20 TileWorld was created with 40 tiles, 20 holes and 40 obstacles. The position of the tiles, holes, obstacles and the initial positions of the agents were all randomly generated using C++ standard random number generator function, rand() with a seed of 8. The number of active agents in the TileWorld varied from one to sixteen depending on the set-up. Figure 6 shows the initial layout of the TileWorld used for the experiment.

This experiment is run in two set-ups. In the first set-up, TileWorld-DARBS is run with one to sixteen agents on a single processor. In the second set-up, TileWorld-DARBS is run with one to sixteen agents on multi-processors (i.e. a new processor is added to the network for every new agent KS). Figure 7 illustrates the single processor set-up with different number of agents.

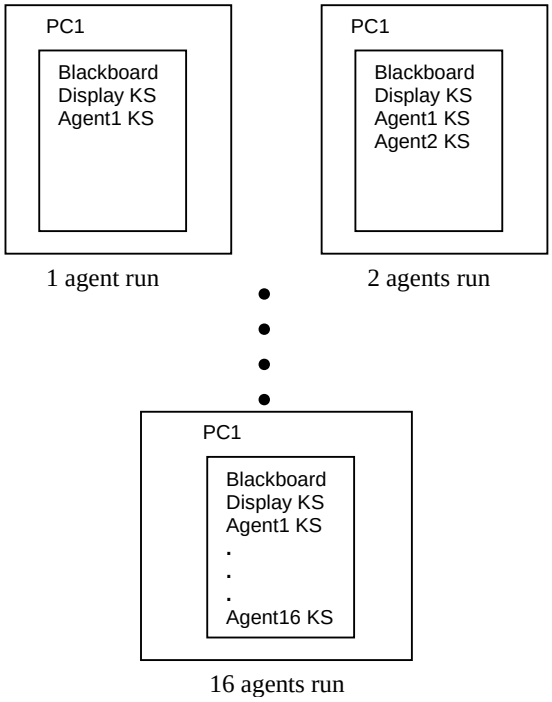
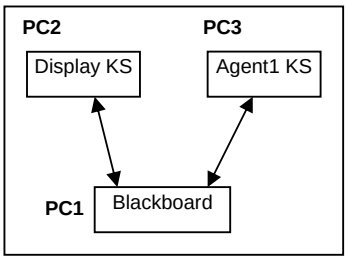
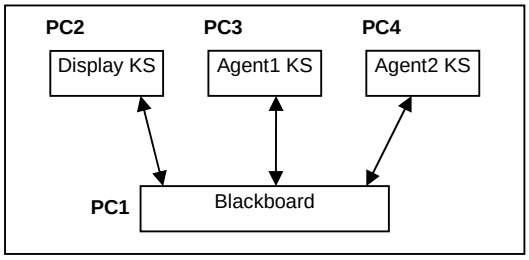


Figure 7: Single processor set-up for different number of agents

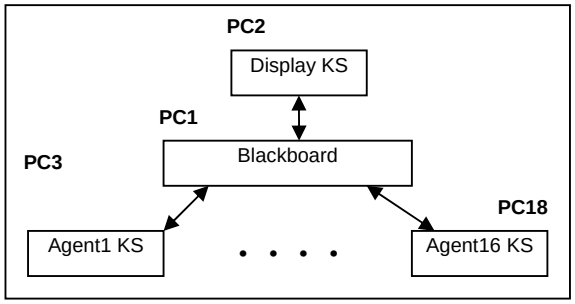
Figure 8 shows the multi-processors set-up with different number of agents. For every run of the experiment, each agent's average time per move is calculated over 50 moves. The overall average time per move for each run is then calculated as the average of all the agent's average time per move. The time per move is calculated by subtracting the time of an agent's move from the time of its subsequent move. An agent is considered to have made a move when it has changed the TileWorld environment (i.e. moved to another cell, picked up a tile, or dropped a tile into a hole). Restarts due to other agents changing the TileWorld are not considered as moves.



1 agent run



2 agents run



16 agents run

Figure 8: Multi-processors set-up for different number of agents

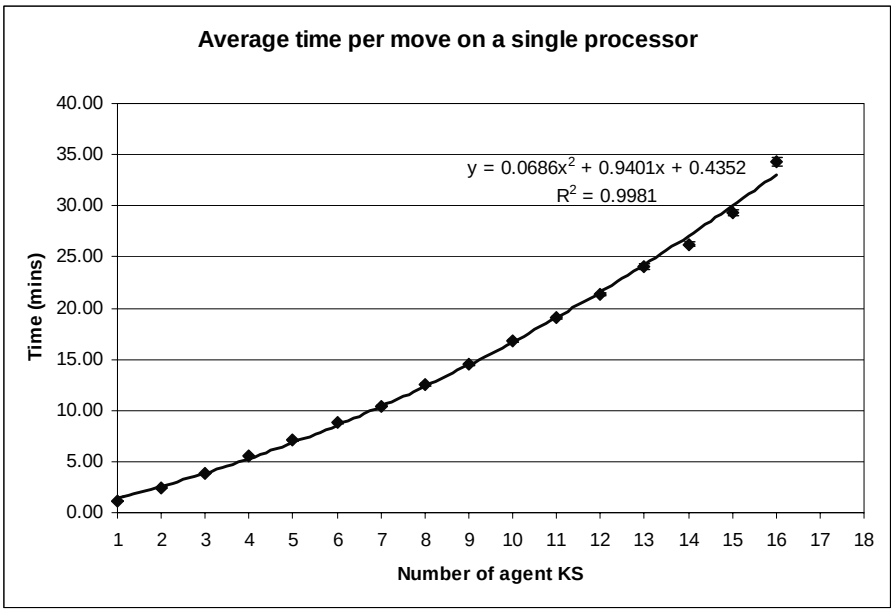


Figure 9: Average time per move for different number of agent KSs on a single processor

Figure 9 shows the results of the single processor set-up of the experiment. A 2nd order polynomial function is fitted onto the results to show the general trend of the results. The error bars show the standard error of mean for each run.

From Figure 9, it can be seen that the average time per move increases slightly more than linearly as the number of agent KSs increases. This is as expected, as the number of agent KSs increases, the single processor needs to time-slice [Tanenbaum, 1995] between more processes. For each time-slice, the processor needs to save the current process's context before switching to the next process's context. This is called context switching [Humphrey et al., 1995] and this takes up processing time.

The standard error of mean [Harper, 1991] is calculated as:

$$\sigma_M = \frac{\sigma}{\sqrt{N}}$$

where σ_M is the standard error of mean
σ is the standard deviation
N is the sample size (in this case it is 50)

The standard error of mean also increases as the number of agent KSs increases although this is not clear in Figure 9 due to the scale of the graph. The increase in standard error of mean is due to the increase in the standard deviation. This increase was because of the design of the reactive agents and how they interact with each another. As the number of agent KSs increases, the agent KSs start to compete among themselves. Take Figure 10 for example, where Agent1 and Agent2 are competing to get Tile1.

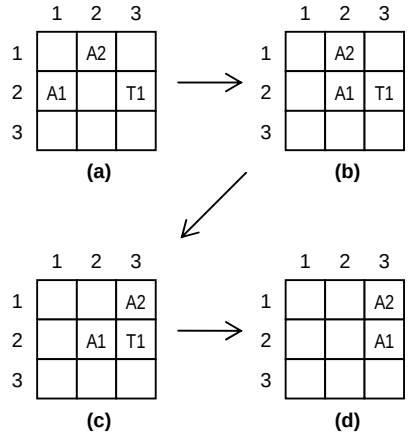


Figure 10: Example of competing agent KSs

Agent1 takes a step closer to Tile1 by moving to cell(2,2). Agent2 has two possible moves (cell(2,2) or cell(3,1)) that will bring it closer to Tile1 and has decided randomly to move to cell(2,2) only to discover that Agent1 is already in that cell (Figure 10 (b)). Agent2 now has to restart and think again where to move. Agent2 then decides to move to cell(3,1) (Figure 10 (c)). Agent1 now moves to cell(3,2) to pick up Tile1. Agent2 tries to move to cell(3,2) to pick up Tile1 but discovers Agent1 is already in that cell. Agent2 now has to restart and think again. As restarts do not count as moves, Agent2 has taken a long time to make its move (i.e. from cell(2,1) to cell(3,1)). This competition only occurs occasionally when the agent KSs are close to each other. Most of the time they are apart as the

TileWorld is relatively large. This restart strategy is not very efficient and was only implemented due to its simplicity. Although this restart strategy can be further optimised to only restart when the changes in the partitions are relevant to the agent's current situation, a better way would be to change the behaviour of the agents to cooperative agents instead of being reactive agents.

Figure 11 shows the results of the multi-processors set-up of the experiment. A linear function is fitted onto the results to simplify and show the general trend of the results. This is because the 2nd order polynomial function trend line produces an equation and R^2 value ($y = 0.0009x^2 + 0.0961x + 0.2988$; $R^2 = 0.9943$) that is very similar to the linear function. The error bars show the standard error of mean for each run. The results on multi-processors show a more linear trend compared to the single processor. This is because on multi-processors there is no time-slicing between the processes. The increase in average time per move is mainly due to the communication time between the processors. The standard error of mean also increases as the number of agent KSs increases and this is due to the same reason as that on the single processor.

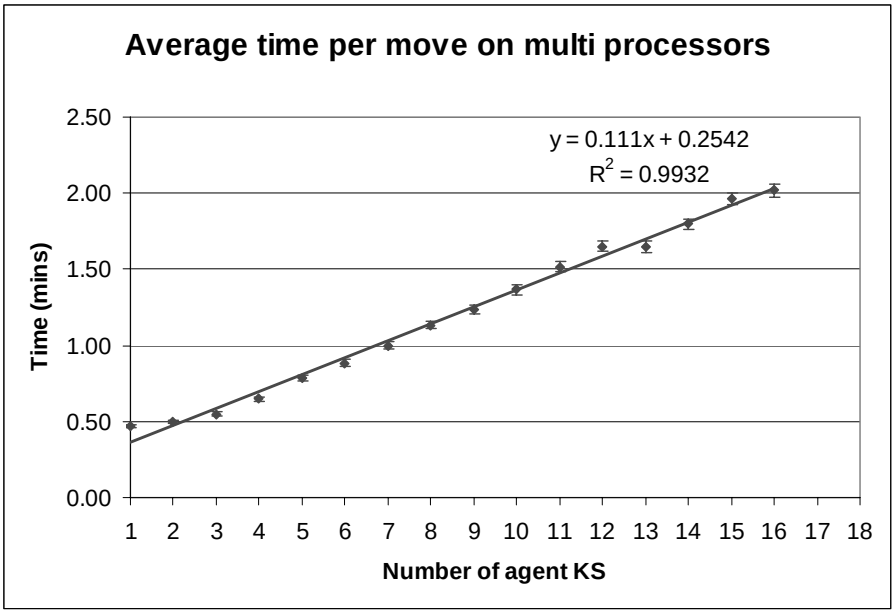


Figure 11: Average time per move for different number of agent KSs on multi-processors

For both single and multi-processors, the average time per move for each run is normalised to the average time per move for one agent KS. This will show the affects of adding new agent KSs to the system. The normalised value is calculated as:

$$t_N = \frac{NAgentTime}{OneAgentTime}$$

where *NAgentTime* is the average time per move for N agent KSs

OneAgentTime is the average time per move for 1 agent KS

t_N is the normalised time

Figure 12 shows the normalised time with increasing number of agent KSs on a single processor. A second order polynomial function is fitted onto the normalised values. The affects of adding more agent KSs on a single processor can clearly be seen in Figure 12. The slow-down as the number of agent KSs increases is approximately linear until four agent KSs. By adding four agent KSs in the TileWorld, the average time per move is approximately five times slower than one agent KS in the TileWorld. This increase in time is not linear and at sixteen agent KSs, the slow-down is close to thirty. The gradient of the polynomial function can be calculated by differentiating it as follows:

$$y = 0.058x^2 + 0.7944x + 0.3678$$

$$\frac{dy}{dx} = 0.116x + 0.7944$$

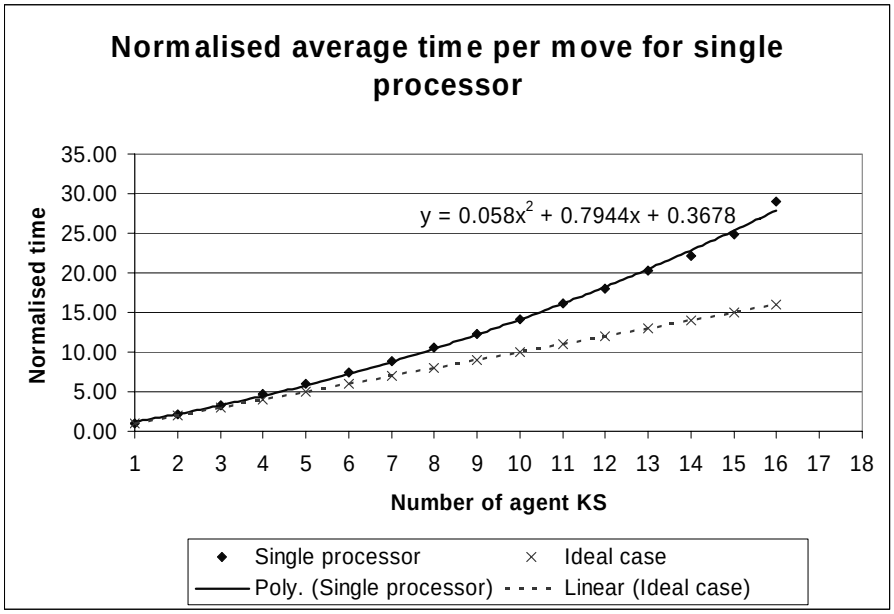


Figure 12: Normalised time for single processor

The derivative here shows approximately the rate of slow-down as the number of agent KSs increases up to sixteen agent KSs. In the ideal case this slow-down should be linear as the single processor would spend equal amount of processing time for every agent KS in the TileWorld. However, because of the large context switching overhead and an inefficient scheduling algorithm of the kernel, this slow-down is a polynomial function. The Linux kernel's scheduling algorithm dynamically recalculates each process's priority every epoch and as the number of processes increases, this recalculation becomes a burden [Bovet and Cesati, 2002].

Figure 13 shows the normalised time for the multi-processors set-up with a linear function fitted onto the normalised values. An ideal case is also plotted as a straight line with a normalised time value of one. In the ideal case, there should not be any slow-down as a new processor is assigned to every new agent KS in the TileWorld. From the experiment, the first two values are relatively close to the ideal case. However, from three agent KSs onwards, the slow-down is approximately linear with a gradient of about 0.238. This slow-down is mainly due to the communication overhead and the serial access to the blackboard. Comparing with single processor up to sixteen agent KSs, the rate of slow-down on multi-processors is a lot less. This is shown by the gradient of the two trend lines ($\frac{dy}{dx} = 0.116x + 0.7944$ and 0.238 respectively).

At sixteen agent KSs, the difference between the measured time and the ideal time is;
For multi-processors:
4.3214 – 1.0000 = **3.3214**
For single processor:
28.9718 – 16.0000 = **12.9718**
This shows that at sixteen agent KSs, the results on the single processor is a lot further from the ideal case than that of the multi-processors.

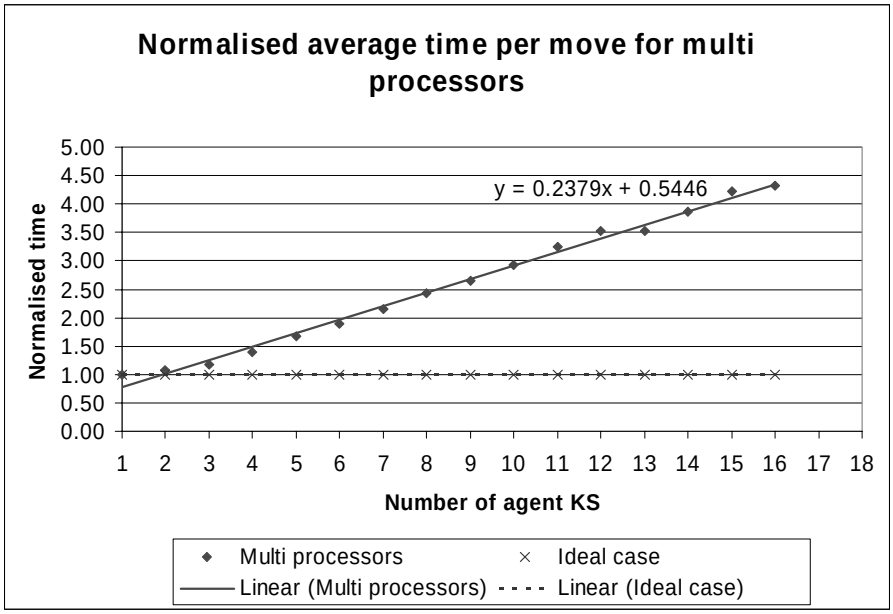


Figure 13: Normalised time for multi-processors

Figure 14 compares the average time per move on a single and multi-processors. A polynomial function and a linear function are fitted to the results of the single processor and multi-processors respectively. It can clearly be seen that as the number of agent KSs increases, the difference in time between single and multi-processors increases. For example, at two agent KSs, the difference is:

2.467 – 0.500 = **1.967 minutes**;

and at sixteen agent KSs, the difference is:

34.283 – 2.017 = **32.266 minutes**

This shows that the overhead from context switching on the single processor is more than the communication overhead on the multi-processors as the number of agent KSs increases. For two agent KSs, the ratio between single and multi-processors is:

$$\frac{2.467}{0.5} = 4.934$$

and for sixteen agent KSs, the ratio is:

$$\frac{34.283}{2.017} = 16.997$$

This means that with two agent KSs, the multi-processors set-up is about five times faster than the single processor set-up and this continues to increase up to approximately seventeen times faster with sixteen agent KSs (as can be seen in Figure 14).

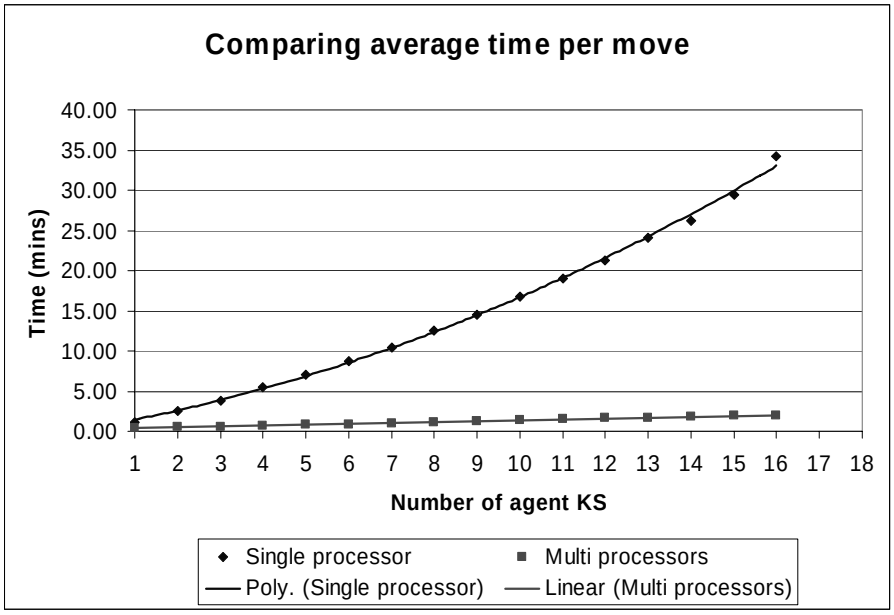


Figure 14: Single and multi-processors average time per move

5. CONCLUSIONS AND FUTURE WORK

The TileWorld test-bed has been successfully implemented on DARBS running on a single and multi-PCs using the Linux operating system. From the experiments carried out, it is evident that the performance of the TileWorld test-bed in a distributed set-up is better than on a non-distributed set-up. The performance of both distributed and non-distributed set-ups are far from their ideal case. For the distributed set-up, this is because of the communication overhead. For non-distributed set-up, this is because of the context switching overhead and the kernel's inefficient scheduling algorithm. Although the scheduling algorithm can be improved, the performance of the non-distributed set-up would still be far from the ideal case as context switching between processes takes up relatively large amount of processing time. Comparing the difference between actual performance results and the ideal case, the distributed set-up has a smaller difference compared to the non-distributed set-up. This is true for up to sixteen agent KSs. However, there will be a point when the blackboard saturates. This will be when the number of requests from KSs is more than what the blackboard can handle. The performance at this point will be difficult to predict without further experiments.

The standard deviation of the results increases as the number of agent KSs increases. This was observed to be due to the increase in number of restarts. The restarts were caused by the competitive interaction between agent KSs. This means that as the number of agent KSs increases, the accuracy of the readings becomes less. The restart algorithm can be optimised but this will only speed up the restart algorithm. A better way to overcome this is to change the behaviour of the agent KSs to cooperative and this is the subject of agent behaviour research.

The affects of adding agent KSs to the TileWorld in the non-distributed set-up is a polynomial slow-down function. In the distributed set-up on the other hand, the slow-down is approximately a linear function. The rate of slow-down up to sixteen agent KSs for the non-distributed set-up is approximately:

$$\frac{dy}{dx} = 0.116x + 0.7944$$

and for the distributed set-up, it is approximately 0.238. This means that by adding more agent KSs to the non-distributed set-up, the rate of slow-down increases but for the distributed set-up, the rate of slow-down remains constant at 0.238. This means that for large number of agent KSs, it is better to run in the distributed set-up. For small number of agent KSs, running on non-distributed set-up is still acceptable. However, the down side of the distributed set-up is that there is an assumption of unlimited processor resource. In practical application cases, there is a limited processor resource. Therefore, the KS processes need to be shared among the available processor resource. Future experiments will be to investigate the speedup and efficiency of sharing the agent KS processes among varying number of processors.

One cause of slowness in the execution of the agent KS is that the KS tries to fire all the rules even when certain rules are known beforehand to be dependent on other rules. A rule dependency table [Hopgood 1994] can be created before runtime to prevent the KS from trying to

fire any rule until the rules upon which it depends have fired.

An improvement for this TileWorld test-bed would be to have another KS that relays what the agent sees from the TileWorld Environment partition onto the Agent View partition. This way, the agent would have no direct access to the TileWorld Environment partition other than making a move on the TileWorld. This would make it easier to implement intelligent restarts as the Agent KS would only need to restart on changes to the Agent View partition and could ignore changes to the TileWorld Environment partition.

Another consideration is the way the KS checks whether a rule's condition is met. The current implementation checks every sub-condition before evaluating whether the composite condition is true. This can be improved on by evaluating the composite condition as it checks each sub-condition. Consider the following example:

```
IF
[
        condition1
        AND
        condition2
        AND
        condition3
]
THEN
[
        actions
]
```

The KS can stop checking *condition2* and *condition3* if *condition1* is false because the composite condition will be false. This type of on-the-fly evaluating would reduce the number of messages sent to the blackboard as each condition-check requires the KS to send a message to the blackboard. The use of on-the-fly evaluation can ultimately reduce the communication traffic, thus increasing the performance of the TileWorld test-bed in the distributed set-up.

REFERENCES

Bovet, D. P.; Cesati, M., 2002. "Understanding the Linux Kernel", 2nd Edition, USA: *O'Reilly*.

Dalheimer, M.K. 2002. "Programming with Qt". Germany: *O'Reilly*.

Engelmore, R.; Morgan, T.; ed. 1988. "Blackboard Systems". Great Britain: *Addison Wesley*, pp. 2-15.

Ephrati, E.; Pollack, M. E.; Ur, S. 1995. "Deriving multi-agent coordination through filtering strategies" *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 679-685.

Harper, W. M., 1991. "Statistics", 6th Edition, *UK:Financial Times Prentice Hall*.

Hopgood, A.A. 1994. "Rule-based control of a telecommunications network using the blackboard model", *Artificial Intelligence in Engineering*, 9, pp 29-38.

Hopgood, A.A.; Phillips, H.J.; Picton, P.D.; Braithwaite, N.St.J. 1998. "Fuzzy logic in a blackboard system for controlling plasma deposition processes" *Artificial Intelligence in Engineering*, 12, pp. 253-260.

Humphrey, M.; Wallace, G.; Stankovic, J., 1995. "Kernel-Level Threads for Dynamic, Hard Real-Time Environment", *16th IEEE Real Time Systems Symposium*, pp. 38-48.

Jennings, N. R.; Wooldridge, M. J.; ed. 1998. "Agent Technology: Foundations, Applications, and Markets". Germany: Springer, 1998. pp. 32-34.

Kinny, D.; Georgeff, M.; Hendler, J. 1992. "Experiments in Optimal Sensing for Situated Agents", *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, PRICAI'92, pp. 1176-1182.

Lees, M.; Logan, B.; Theodoropoulos, G. 2003. "Adaptive Optimistic Synchronisation For Muti-Agent Distributed Simulation", *Proceedings 17th European Simulation Multiconference*, pp. 77-82.

Nolle, L.; Wong, K. C. P.; Hopgood, A. A. 2001. "DARBS: A Distributed Blackboard System", *Research and Development in Intelligent Systems XVIII*, Bramer, Coenen and Preece (eds.), Springer, pp 161–170.

Pollack, M. E., and Ringuette, M. 1990. "Introducing the Tileworld: Experimentally Evaluating Agent Architectures", *Proceedings of the Eighth National Conference on Artificial Intelligence*, AAAI Press, pp. 183-189.

Tanenbaum, A. S., 1995. "A comparison of three microkernels", *Journal of Supercomputing*, Vol. 9, No. 1/2, pp. 7-22.

Uhrmacher, A.M.; Schattenberg, B. 1998. "Agents in Discrete Event Simulation", In: Andre Bargiela and Eugene Kerckhoffs (eds.) *Proceedings of the 10TH European Simulation Symposium ``Simulation in Industry – Simulation Technology: Science and Art'' (ESS'98)*, SCS Publications, Ghent, pp. 129-136.

AUTHOR BIOGRAPHIES

KUM WAH CHOY is a PhD student at the Nottingham Trent University. He obtained his BEng (Hons) in Electronics & Computing in 2001. He has spent a placement year with Xerox Ltd as a Software/Test Engineer. His current research is in the field of distributed embedded systems, agent system, blackboard systems, and artificial intelligence.

ADRIAN HOPGOOD is professor of computing and head of the School of Computing and Technology at the Nottingham Trent University, UK. He is also a visiting professor at the Open University. His main research interests are in intelligent systems and their practical applications. He graduated with a BSc (Hons) in physics from the University of Bristol in 1981 and obtained a PhD from the University of Oxford in 1984. He is a member of the British Computer Society and a committee member for its specialist group on artificial intelligence.

LARS NOLLE graduated from the University of Applied Science and Arts in Hanover in 1995 with a degree in Computer Science and Electronics. After receiving his PhD in Applied Computational Intelligence from The Open University, he worked as a System Engineer for EDS. He returned to The Open University as a Research Fellow in 2000. He joined The Nottingham Trent University as a Senior Lecturer in Computing in February 2002. His research interests include: applied computational intelligence, distributed systems, expert systems, optimisation and control of technical processes.

BRIAN O'NEILL graduated in 1968 from Trinity College, University of Dublin and obtained his PhD from Liverpool University in 1972. Before taking up the lecturing post at Nottingham Trent in 1977, he held research posts at Glasgow and Southampton Universities. His research activities originated from work on electronic instrumentation for powder flow control. This work required the development of fast real-time control and processing, using parallel processing, for its implementation. From this start he has widened his area of research to the design of hardware packet routing switches for inter-processor communication.