ISSN 1512-1232

ON MULTI ROBOT SIMULATION: MULTI-THREADING APPROACH FOR IMPLEMENTATION

Issa Fadi[1], Obe Olumide[2], Ion Dumitrache[3]
[1, 2, 3] Politehnica University of Bucharest, Faculty of Automatic Control and Computer Science

Abstract

In this paper we discuss the use of multi-threading programming in mobile robots simulators. Using the additional processing power of multi-core processors can open new possibilities and areas for researchers to experiment with. We offer an approach for implementing multi-threaded simulators in order to take benefit of the current advancements in multi-core processors, and we examine which components of a simulator can benefit from the extra cores, and what challenges we have to overcome in order to successfully implement this programming technique.

Keywords: *Multi-threading, Simulator, Mobile Robots, Multi-Core processors*

1. Introduction

Mobile robot simulation developed from simulating a single mobile robot [1],[2], to simulating multiple mobile robots together [3]; whether they are groups of similar robots or different ones (heterogeneous robots). With these advancements in simulation, developing simulators became more complex, and has become an area for research by itself. Programmers will always try to find better software architectures for implementing simulators, taking benefit of the advancements in software engineering and also from the advancement of the computer hardware (processors, memory, graphics, communication, etc).

A recent simulator usually has the ability to simulate several robots together, making tests on robot groups and multi robot systems achievable. These new types of tests on groups of mobile robots require additional costs due to the nature of multiple robots systems [4], which makes the option of using simulation a more convenient solution to carry out these tests before being applied to real robots.

Although the final aim is real robots, it is often very useful to perform simulations prior to dealing with real robots [3]. This is because simulations are easier to setup and less expensive [5] (this can be especially important for laboratories in which access to physical mobile robots is a limited resource).

Mobile robot simulators developed in companionship with software development, and when the object oriented programming language paradigm came up, simulators development utilizes these new language paradigms [6]. The most important benefit of these new object oriented programming languages was the huge cut in time and complexity needed to implement the same functionality compared to the functional programming languages used before, besides, software developed using object oriented language such as Java will be easy to understand and easy to debug. These benefits allowed the researchers to increase the scope of simulation, since now they can spend more time on the simulation process rather than on writing the simulator itself. This shifted the focus towards new needs and requirements from simulators; meeting these needs will require better use of the available hardware and software techniques.

In this paper we first discuss some bottlenecks that we have to overcome if we want simulators to continue developing, then, we review some of the current approaches of implementing simulators and their limitations. Finally, we introduce our approach of using multi-threading for the implementation, apply some tests and discuss the results.

ISSN 1512-1232

2. Limitations and the need for parallel programming

With the increased requirements needed to simulate increased numbers of mobile robots in more complex environments [5], we need to improve the traditional approaches in order to keep pace with the increasing demands. Improving performance can be done either by using more powerful hardware, or more efficient software architectures that take benefit of the new hardware.

Efforts to improve system performance on single processor systems have traditionally focused on making the processor more capable; these approaches to processor design have focused on making it possible for the processor to execute more instructions per time unit by using higher clock speeds [7].

The increasing performance of processors in personal computers started to take a different path from what had been followed before. The requirement for higher performing processors is linked to the demand for more sophisticated software applications [8]. This lead to a new approach in processor design, an approach meant to improve personal computer's performance by adding extra processors (as in symmetric multiprocessing designs which have been popular in servers and workstations). This happened because increasing the speed of a processor was becoming more challenging, as chip-making technologies approached the physical limits of the technology [9].

The first dual core processor for personal computers, produced by AMD in 2005, marked the start of multi-core processors era. Multi-core processors represent a major evolution in computing technology which will help to overcome single-core performance. Besides, current operating systems such as MS Windows, Linux and Solaris are now capable of benefiting from multi-core processors [10] because their constructs allow a single process to execute more than one stream of instructions concurrently [11].

Multi-core processors will continue increasing the number of cores they have, and more cores are expected to come out (from 2 and 4 cores available now to 8 and 16 cores) in the near future [12], so the challenge will be: how could we take benefit of this additional processing power and incorporate it in mobile robots simulations.

Having multi-core processors does not guarantee that the performance of the software will increase automatically, because the software has to be designed to take advantage of available parallelism. If it has not, there will not be any speedup at all [13], and developers usually need to explicitly code the use of threads into their applications [11].

Recently, processors can have as many as eight cores, so writing the program in a one thread approach can only use one of these eight, leaving seven cores without any work to do. This indicates the big opportunity that exists if we take advantage of these extra cores. This advantage can come in the form of higher number of simulated robots concurrently, or in the form of more complex control strategies. Next we are going to discuss using multi-threading programming in order to take benefit of multi-core processors in mobile robots simulators, but first we will review current approaches used in implementing these simulators.

3. Current approaches

One of the most commonly used approaches for implementing the multi mobile robot simulator uses the client-server architecture [14]. The simulator and the virtual world that it provides is the server, while the clients are the robots that occupy the virtual world. The server provides the virtual hardware for those robots, it models their bodies and physical interactions with their simulated environment, it responds to their commands to change position or begin a movement and provides them with sensory feedback, but it does not control the robots; control comes from the clients [8].

The advantages of this model are: allowing robot control programs to be written in different languages, separate the simulation from the robot control programs, and distribute the computational load over several computers. The main disadvantage is the big delay introduced in the form of latency. In this model, the added complexities of communication protocols are to be

ISSN 1512-1232

taken into consideration when developing simulators.

Another approach is to use distributed computing for implementing the simulation by relying on several processes and multi-tasking among them to achieve the simulation. This approach also suffers from the same limits the previous approach has, like the overhead of latency, bandwidth, network communication, synchronization that would be required between nodes and coherence checking (needed to ensure that all processes have the same updated data). Besides, troubleshooting and diagnosing problems in a distributed system are more difficult, because the analysis may require connecting to remote nodes or inspecting communication between nodes [15].

4. Multi-threading approach for multi mobile robot simulator implementation

The use of multi-threading programming is the key to take advantage of the increasing number of processing cores in central processing units in each new generation of processors; it will be necessary if we want simulators to continue developing in features and performance, while supporting larger number of simultaneously simulated robots. Multithreaded software applications – programs that run multiple tasks (threads) at the same time to increase performance for heavy workload scenarios, are already positioned to take advantage of multi-core processors [10].

Our goal is to try to program the simulator in a way that takes benefit of current processors, while at the same time avoid implementing them in the same way as discussed in past paragraph in order to avoid their drawbacks. So the main idea of our approach is to use multi-threading programming as a component of the simulator architecture design, and not just in control method implementations where it is left to the programmer to decide whether to implement multithreading or not.

A thread of execution by definition is a "fork" of a computer program into two or more concurrently running tasks [16]. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share this data.

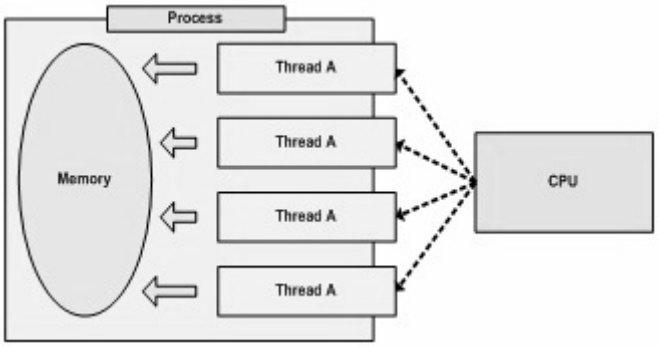

Figure 1: Four threads running on a single core processor sharing the same resources.

A thread usually does not work continuously; it pauses from time to time according to a given period that is called "sleep time". The sleeping behavior essentially interacts with the thread scheduler to put the current thread into a wait state for the required interval, then when the required sleep time has elapsed, it wakes up and continues to work from the point it stopped at [17]. During the sleeping period, other threads will have their chance to take advantage of the available resources. In our simulator case, the sleep time of each robot thread represents the latency of that robot, which is the time that a robot takes after finishing calculating its current action and before it starts to calculate for its next decision.

Threads are distinguished from traditional multitasking operating system processes in that:

- processes are typically independent, while threads exist as subsets of a process;
- processes carry considerable state information, where multiple threads within a process share state as well as memory and other resources;

ISSN 1512-1232

- processes have separate address spaces, where threads share their address space;
- processes interact only through system-provided inter-process communication mechanisms;
- context switching between threads in the same process is typically faster than context switching between processes [16].

These differences between threads and processes add another benefit which this time is related to the developer; programming using multithreading will relief programmers from taking care of communication between the processes of the simulation, and in return, rely on the operating system to do this task. This will allow the developer to concentrate more on functional tasks instead of dealing with the communication overhead; besides, inter-process communication is fast and easier for the operating system to handle.

Multi-threading is a popular programming and execution model that allows multiple threads to exist within the context of a single process while still being able to execute independently.

The threaded programming model provides an abstraction of concurrent execution, and when applied to a single process it enables parallel execution on a multi-core system [16].

This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines. This is because the threads of the program naturally lend themselves to truly concurrent execution [16].

An example that illustrates the above idea is shown in Figure 2

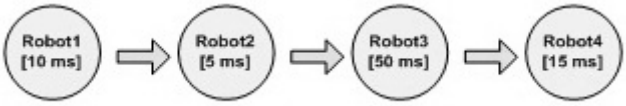

Figure 2: Multi mobile robot simulation sequence

In the non threaded approach when dealing with simulating multi robot systems, the robots are put in an array, and then sequentially the CPU resources are passed to each of them, in their turn, to calculate their actions, and when each of them finishes, then the available resources are passed to the next robot, and so on, until all robots in the array do their jobs. Then this cycle starts again, from the first robot to decide its next step, and so on. The performance of this approach is acceptable when all robots are of the same type, or if they need similar time to complete their decision making process. In the case of a scenario like the one shown in Figure 2, in which we are simulating four robots at the same time (each of them needing a different time to accomplish its control strategy calculations), we encounter the following situation: the Robot3 will need 50 milliseconds to finish the calculations in order to take a decision, and this will affect the simulated Robot4, since it cannot start its own calculations until Robot3 finishes. This situation causes a lag that the Robot4 is not responsible of. This is one shortcoming of not using threads in the programming. Threading will eliminate this problem even when we only have single core processor, because if we use multithreading and assign a separate thread to each simulated robot, then the slow robot (Robot3) will not slow down all the simulation and other robots till it finishes its calculations. Instead, each robot will only affect itself, and the robot with easy calculations will be simulated normally as it should be, while the complex robot will stay in its place until finishing its calculations without affecting the other robots on their simulation.

In Figure 3 we see how a multi-core (actually a quad core) processor divides its work; we notice that all the processing cores share the same memory, and they connect among themselves using a shared bus, so when we implement each simulated robot in a separate thread, then the task of control method of each one of the robots will be processed independently; this means that if we have four simulated robots, each one will calculate its own decision on a separate processing core, thus, not waiting for the others to finish before making their own calculations.

ISSN 1512-1232

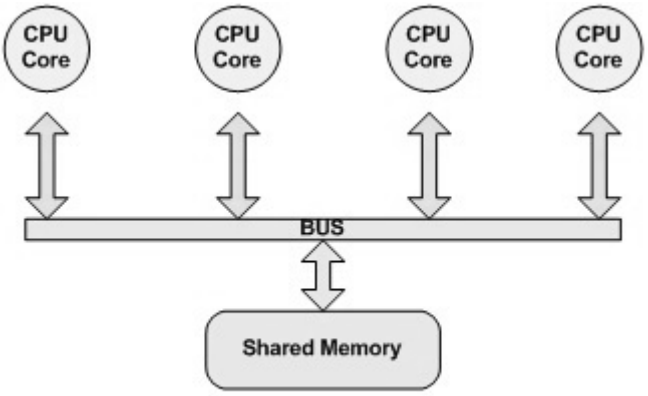

Figure 3: A Multi-Core Processor architecture

Multithreading programming needs support from the operating systems in order to work properly and today's operating systems, such as Microsoft Windows XP and Windows Vista, all support multithreading [7].

So having the hardware that supports multithreading, and having the operating system that supports it, we still need the programs to possess executable sections that can run in parallel. That is, rather than being developed as a long single sequence of instructions, programs are broken into logical operating sections. In this way, if the application performs operations that run independently of each other, those operations can be broken up into threads whose execution is scheduled and controlled by the operating system.

On single processor systems, these threads are executed sequentially, not concurrently. The processor switches between the threads quickly enough that both processes appear to occur simultaneously [7]. In multi-core processors threads can run more or less independently of each other without requiring thread switches to get at the resources of the processor.

The approach of making a separate thread for each mobile robot is most suitable when we simulate heterogeneous robots, because each of them needs a different amount of calculations. The operating system will switch between threads on a time basis. This way, it assures a fair distribution of processing resources among the threads, assuming we give all the robot threads the same priority. We can think of the robot threads as agents (Figure 4), and the set of robots is a multi-agent system, this is because each robot thread meets the requirement for being considered as an agent, since it is autonomous (or partly autonomous depending on the user needs), each has its own local view to the world, and decentralization is met, since there is no controlling agent, each agent acts separately, reading from its environment through its sensors, and then sends its actions to the actuators as commands according to the duty (task) that the agent has to perform.

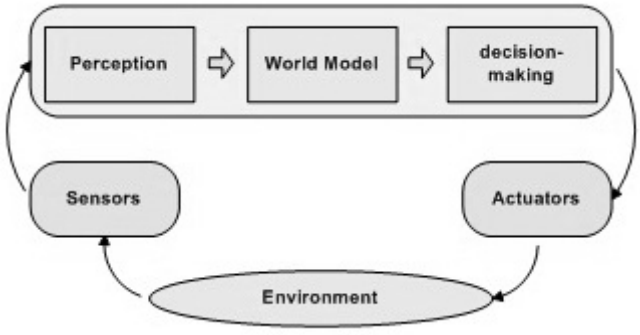

Figure 4: A mobile robot as an agent.

5. Implementation Details

In addition to our idea of assigning each simulated robot to a separated thread, and running

ISSN 1512-1232

them all asynchronously relying on the operating system to manage context switching among them, we will investigate if there are additional tasks than can be run independently so they will be a good candidate to be processed in their own threads. We must first know where we can apply multithreading programming, and how many threads are sufficient. It is also important to know when we should stop trying to divide the application to more threads (the number of threads depends on what they are doing, if they spend most of their time idle, waiting for a period of time to pass, or waiting for a result from another thread, then we can use a lot of these threads).

Threads that spend most of their time idle are not going to compete on the available CPU resources, and they can easily share the CPU. On the other hand, if the threads are CPU heavy (require a lot of calculations), then there is no point of further dividing the work to more threads [18].

In Figure 5 we see the whole mobile robot simulation system divided into two parts, a mandatory part that exists in all the simulations, and an optional part that may or may not exist; it depends on the needs of each simulation (e.g. whether it is for testing an abstract new control method or a control method for a real physical robot).

If we look at the optional part of the simulation system, we notice that we cannot apply multi-threading, neither on the robot interface nor on the physical robot (obviously). In the robot interface we cannot apply it because it is usually just an interpretation of the commands from the simulation to the machine language of the real robot (the commands that it understands). Besides, it is not a computation intensive task, so no real benefits could be achieved by splitting this work to several threads, but it can be better to handle all of the interpretations in a thread separated from the other ones of the mandatory part.

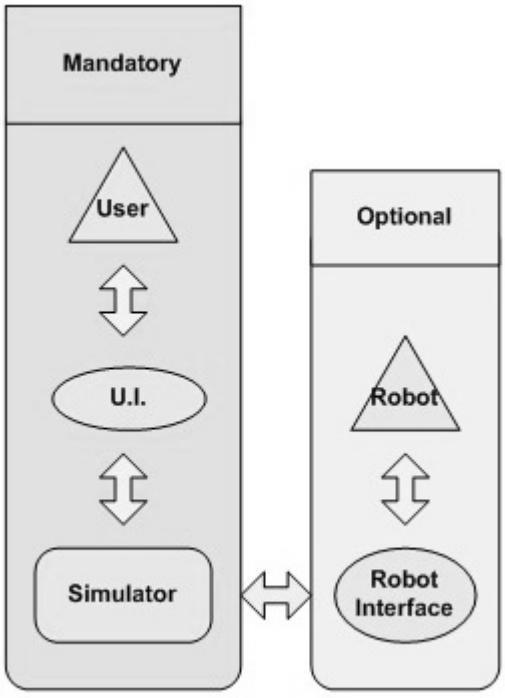

Figure 5: The whole Simulation System

So now we have to try to use multiple cores by implementing threading model in the mandatory part of the simulation. The area in which most of the multi-threading can be applied is in the simulator component from Fig.5, the robot control has by nature a distributed architecture, since all of its sensors, actuators, and even its "thinking planner" have their own control, and this makes it quite suitable to be adapted to parallel programming, which should be considered from the beginning stages of developing simulators.

We will investigate the possibility for implementing multithreading in each of the basic components (Figure 6) of simulator:

ISSN 1512-1232

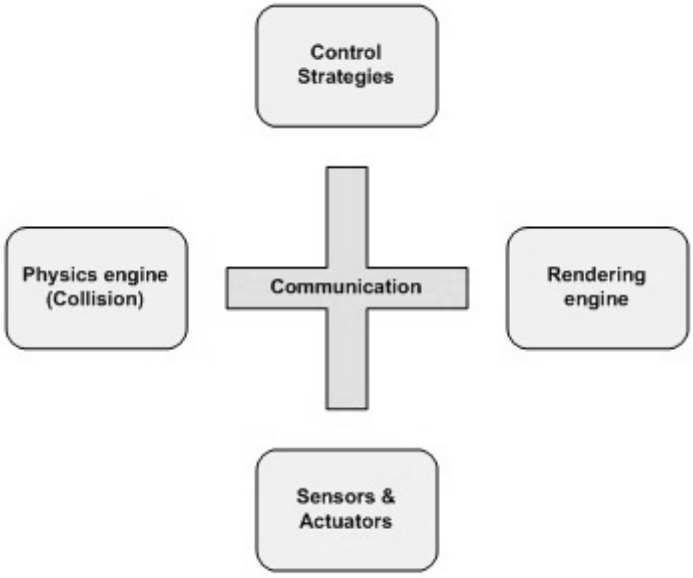

Figure 6: Communication between the software components in a simulator

We will investigate the possibility for implementing multithreading in each of the basic components (Fig.6) of simulator:

- Sensors and actuators

Theoretically, each sensor and actuator object can be implemented in its own thread, since a sensor does not depend in its work on other sensors readings. The same can be said about differential wheels for example. This does not mean that it is efficient to implement each sensor in a separate thread, because if we take into consideration the current number of cores in recent processors (currently a maximum of eight cores), they are still far from being efficient for a very small scale multi-threading (fine grained multithreaded) [19]. Besides, large number of threads will increase the overhead of context switching. Instead, it is better to keep the implementation of sensors and actuators within the same thread in which the whole robot will be represented; this will reduce the number of threads to a more reasonable one.

- Control strategies

Using threads in control strategies depend on the strategy itself (its ability to be divided into independent sections), and on the preferences of the programmer that implements that control strategy, since not every programmer will favour the multithreading implementation due to its additional difficulty. Therefore, we cannot ensure from the design stage of the simulator whether the control strategies are going to be implemented using multithreading or not; this is left to the programmer to decide according to his preferences.

- Rendering engine

Rendering (till now) is only possible to be done in one thread[20]. There is some research on implementing multithreaded rendering, but the most current version of Java still does not have this feature (we are implementing this simulator using Java), so we cannot speed up the rendering process no matter how many processing cores or processors we have, but we can at least separate the rendering from the rest of the simulation components.

If we look at the user interface component, with its basic purpose of presenting simulation results to the user, we can think of separating the rendering task from the actual simulation calculations, by making it in a separate thread that reads from an array of robots in a frequent and ordinary way to update its results; this way we are separating it from the calculations.

The rendering process will be like what is shown in Figure 7. In the normal way, the robots are put in a vector (1-dimension array), and then a buffered image is passed to them from the U.I.; each robot will render its representation on the buffered image after it finishes its own calculation, and then it passes the buffered image to the next robot and so on, until all the robots finish their

ISSN 1512-1232

rendering process. After that, the buffered image will be sent to the U.I to be shown to the user, and a new buffered image will be created and then passed to the first robot to start this routine again for the next frame.

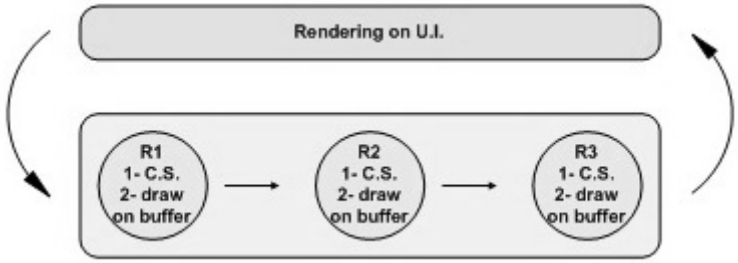

Figure 7: Rendering in a multi mobile robot simulator

We will make some modifications to the way in which current approaches of rendering are done, because using the renderer in its common used way makes us face a problem: if one robot is doing its own calculations, it cannot render itself on the buffered image until it finishes its calculations for the decision to be taken, therefore it will slow down the entire rendering process even if every robot is running in its own thread. But rendering cannot be multithreaded itself, so it has to be run in a sequential form. In order to overcome this shortcoming, we will introduce a new layer between the robots and the rendering; this layer will be an array of shadow objects.

The shadow class by definition is a class that has the same attributes of its main object, but it does not do any significant calculations; in our way it will be as shown in Figure 8.

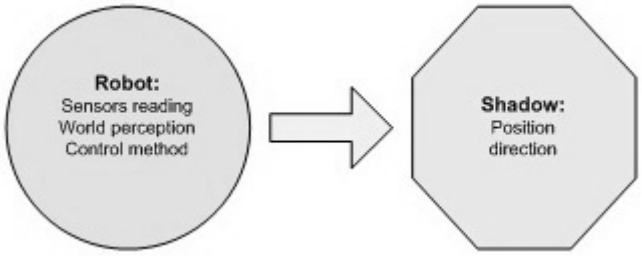

Figure 8: Main and Shadow classes of a robot

The shadow class of each robot will usually contain these information:

- the position of the robot;
- the robot direction;
- a method for drawing the robot: since the user interface will show the robot rendered based on its shadow information, this will leave the main robot class doing its control strategy calculations without interruptions;

The robot class has the following attributes and methods:

- read the information from the sensors (actually in the simulation it asks the sensors to calculate their values based on the environment).
- based on the information received from step1, and from other information received from communicating with other robots, the robot builds its own perception of the world, in preparation to pass this information to the control method; this information can be world representation, or just hints or information useful for the control method to decide what to do next.
- the actual control method calculation, here depending on the task assigned to the robot, and the control method used to accomplish that task. The calculations are made and a decision is taken to be applied for the next step of simulation; this step can be as complex as deciding the action from ground up, or as simple as relying on the last step, in both cases these calculations are separated from the rendering to prevent any slowdown of the whole rendering system.

The concept of dividing the robot class into two classes, one that makes the calculations, and a

ISSN 1512-1232

shadow class that contains representative information about the robot, this separation between the work and the representation gives us a big benefit which is eliminating the slowdown of the rendering process if one robot needs a lot of calculations to finish its current task, since the rendering object will only deal with the shadow objects without interfering at all with how the calculations are done. In case the robot is very busy doing its calculations to determine its next position, then its shadow object will keep the previous state, so the robot is always represented in the rendering, regardless if it finishes its calculations or not; the (main) robot object will update its corresponding shadow object whenever it has new information. This way, although we created one more layer that we have to deal with (which programmatically is more challenging), the results are worth this sacrifice in coding simplicity in order to obtain a more responsive user interface.

Since we separated the rendering task from the robot control simulation, the rendering task will get the needed information about the simulated robot from the shadow classes as shown in Figure 9.

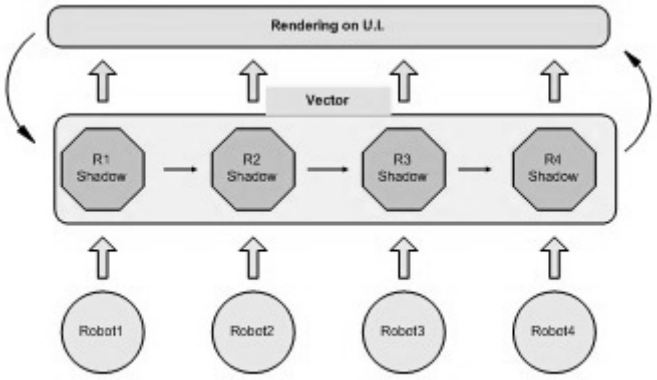

Figure 9: Shadow layer to improve rendering response.

After each robot changes its position or direction, it sends this new information to the shadow object that represents it.

Another benefit when dealing with shadow classes is that when rendering there is no need to wait for a robot object to finish its current calculations in order to get its position for rendering. Instead, it will get the most recent position from its corresponding shadow object, and thus, it is guaranteed that the problem we face when one slow robot holds down all the other simulated robots will not happen when using shadow objects.

Since the rendering process is also on a separate thread, then to set its sleeping time depends on how many frames per second we want the simulation to run at, usually 60 fps is the maximum that an LCD monitor can support. So setting it higher than that has no tangible effect. Besides, the human eye cannot detect the difference of any animation above that level.

The order in which the rendering happens is as follows:

1) the renderer connects to the environment class and gets information about any object in the environment, except the robots (obstacles, lights, balls, boxes, etc) and render these objects on the user interface.
2) scan the list of shadow robot objects and from them render the corresponding robot on the user interface.
3) after finishing scanning all the robot shadow objects, sleep for a specific time (related to the desired frame rate the user wants).
4) after waking up, repeat the above mentioned steps.

- Physics engine

The physics engine is used to check collisions among robots themselves and with obstacles; it is not the type that the sensors of robots detect and then try to avoid, instead it is when the collision happens and the robots are not aware of it. We found that it is better to put this task in a separate thread that frequently checks for any collision between these robots, and when it is detected it is

ISSN 1512-1232

considered as an error; this is why we need to run a separate routine to check collisions frequently to ensure that there is no possibility of two overlapping robots (which is physically impossible).

Robots threads will be running asynchronously, so making collision checking to take part in each robot object will interrupt its working on its own control method.

Another reason why it is better to put collision detection on a separate thread, rather than being as a method in the robot class, is that we are trying to simulate several types of robots. Each one has different latency, which means, different sleep time in implementation terms as a thread. For example, if a robot has a high latency (50 milliseconds), while another robot has a 10 milliseconds latency (it updates its state 100 times a second), and the collision detection routine is in the robot class, then, there exists a possibility that a fast robot will collide with the slow response one, and continue before the slow robot even ends its sleeping time (which is a logical error). This is why we believe that implementing the collision detection in a separate thread is better because it eliminates this problem.

When implementing the collision detecting thread we choose the smallest sleep time among all robot threads and make it the sleeping time of the collision detection thread. This way we will not encounter a situation when a robot with high latency (slow response) will collide with a fast response (low latency) robot without being detected. The steps in which this thread will work are:

1) ask the environment class to give the collision thread the most recent list of robots positions and their shape attributes (like diameter for a circle shaped robot, the center position of each robot, etc).
2) do the collision detection calculations.
3) report back any abnormal situation if encountered and send this information to the user interface to inform the user about this problem.
4) send signals to robots to stop doing any task and to stay in place (in our version of the simulator, when two robots collide, they simply stop moving and stop the tasks they were doing).
5) sleep for an amount of time equal to the lowest latency among the simulated robots.
6) start these steps again, once the sleep time passes.

So now we have several threads, each robot with its own thread, but they are still not completely independent. This shows up in the following two conditions:

a) the first condition takes place before the robot starts to calculate its own action, when it needs to collect information about its environment. And since the robot threads do not communicate directly among themselves (this is if we are trying to simulate robots that do not communication between themselves), then the robots have to rely on a separate class that offers them some environment information (to allow their sensors to get their supposed values).
b) the second condition is when testing collisions between robots themselves and robot-obstacle collisions. In this case, each robot thread needs to know about other robot's position and obstacles positions.

As a solution for both of these two cases, we will create a separate class called "Environment", and assign it to a separate thread. For the first condition it will help the robots by allowing their sensors to obtain their values according to what exists in the environment, while for the second condition it will contain the information about the robots position, shapes and obstacles, and give them to the collision test method to check for any possible collision. By presenting this new class that plays the role of a communicator among robots threads, the previous two conditions will be solved without affecting the performance of other tasks.

We notice that our approach does not put any limits on the number of simulated robots, the only limit is the performance of the processor used for the simulation.

ISSN 1512-1232

6. Results and comparison

We made some tests to analyze how our approach affects the performance in real simulations, so we built a multi mobile robot simulator based on this approach, the tests were done on an INTEL Q6600 quad core processor, each core running at 2.4 GHZ. In order to test results on one core system for comparison purpose we artificially forced the program to use only one core from the available four cores and ignore the other ones, thus it is approximated to the performance of running on a single core processor while maintaining all the other specification of the computer intact.

To test the scalability of our approach and see if there is a limit on the number of threads that can be run before the performance starts to degrade we built simple mobile robots with very little computations needed for their decisions, then we tried to simulate as much mobile robots as we could, so we started by simulating one robot, then we were increasing the number of simulated robots while monitoring the performance of the simulation. We set a target of 30 fps (frames per second) and we were monitoring if it drops while adding more robots.

The results were the following:

- first we used a single core processor, then we started to increase the number of simulated robots, until it reached 26 robots. The frame rate could not keep pace with the simulation, and it dropped to 28 fps.
- in the case of a dual core processor, we could simulate up to 80 robots simultaneously without any performance hit (it stayed at 30 fps), after that the performance started to decrease. We noticed that the increased number of simulated robots compared to the single core processor (an increase of about 55 robots) is more than double the number of simulated robots that could be simulated on a single core without speed sacrifice. This can be explained due to the fact that the number of simulated robots is not the only factor affecting the simulation performance, because there are other tasks in the simulation that require calculations from the processing unit no matter how much is the number of the simulated robots, like the rendering task, the user interface task, and some other small tasks.
- for a triple core processor, the number of simulated robots before the performance started to decrease was 120 robots, an increase of only 40 robots compared to an increase of 55 robots when going from single to dual cores. This can be explained by the increased overhead for the operating system of switching context between threads, since now we have more than 120 threads running together.
- for a quad core processor, the number of possible simulated robots before dropping in performance was around 148 robots, an increase of 28 robots compared to the triple core processor. We also noticed that the increase rate is decreasing; it was 55 in the step from single to dual core, and 40 in the step from dual to triple core, and now it is 28 in the step from triple to quad core processor, which shows that the increase in performance is not linearly related with the increase of available processing cores, but it was good compared to the single core processor.

We made another test to see how well the simulator performance scales when the robot models need a lot of computations to decide their decisions. We built an artificial control algorithm that needs four millions multiply operations for each action or step, and then we tested the simulator to see how many robots can be simulated according to the number of cores available to the software. Here are the results:

- running the simulation on single core processor: the program acted normally almost in real-time for a number of five simulated robots; above this number the simulation started to slow down and the results showed a clear lag in time between each step and the next one.
- when we tried the same previous context on a dual core processor, the speed became normal again, so we started to increase the number of robots by one each time while checking the simulation speed. We noticed that we could increase the number of robots from five to nine; after that it became slow again and with big lags between each two consequent steps.
- we tested the previous context on triple core processor, and then increased the number of robots

ISSN 1512-1232

again; it was running well until it reached thirteen robots.

- we repeated the test again on a quad core processor; It reached sixteen simulated robots, going above this number the performance started decreasing.

The increase in performance going from one core to four cores was close to linear increase.

Next, in Figure 10, there is a screenshot from the simulator when 20 simple mobile robots (each robot with just 3 distance infrared sensors) are running simultaneously while avoiding collision with each other and with the triangle obstacle in the middle using a simple obstacle avoidance algorithm. The simulation was running at 60 frames per second without any slowdown or drop in the frame rate thanks to the distributed work among several cores, while the same scenario when running using only single core caused the frame rate to drop to 48 frames per second.

This approach is not very suitable for very large number of swarm robots, since switching between threads is going to take away any additional benefit of increased threading due to the additional overhead. As a workaround solution, several threads of similar robots can be grouped together in one thread, but then, we will be increasing the complexity of programming since now we have also to deal with the task of arranging threads and the communication between them; in other words, shifting toward the problems that distributed computing deal with (massive parallelism type of computing).

The system overhead of the threads management is done by either the operating system or the application; with more processors running, the operating system has to coordinate more. As a result, each new processor adds incrementally to the system-management work of the operating system. This means that each new processor contributes less and less to the overall system performance[7].

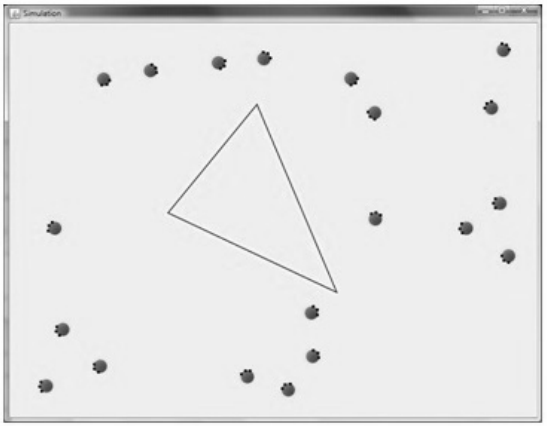

Figure 10: A simulator screenshot while running

After applying the mentioned concepts in the multi mobile robot simulators, we noticed a big improvement in the overall performance, and it shows that simulated robots in multithreaded software can easily take benefit of the additional cores of current processors. We also noticed that the more independent the calculations of the robot are, the more scalable the simulator will be.

We saw how taking benefit of several processing cores can help increase the scope of simulation, whether it is increasing the number of simulated robots, or using more complicated and demanding control methods without affecting the performance of other simulated robots.

7. Conclusions

As a conclusion, we can say that there are two main benefits coming from the approach of threading up the simulation work, the first one being the elimination of the possibility that one robot waits for another one to finish its turn before it gets the control, the second benefit will be the improvement in performance that can be achieved using multi core processing. We also noticed how using multithreading programming approach in dealing with multi mobile robot simulators could increase the scope and size of the simulation, which opens new ways of testing areas, and a new way of looking at simulation of larger numbers of robots simultaneously.

ISSN 1512-1232

8. References

1. Nilsson, Theodor, "KiKS is a Khepera Simulator", (2001-03-14), *Umea University.*
2. Mondada, Francesco, Franzi, Edoardo and Ienne, Paolo, (1993), "Mobile robot miniaturisation: A tool for investigation in control algorithms." *Proceedings of the 3rd International Symposium on Experimental Robotics*, Kyoto, Japan.
3. Michel, Olivier, (2004), "WebotsTM: Professional Mobile Robot Simulation", *International Journal of Advanced Robotic Systems*, Volume 1, Number 1, pp. 39-42.
4. Liu, Jiming, Wu, Jianbing, (2001), "Multi-Agent Robotic Systems", *CRC Press International Series on Computational Intelligence.*
5. Katsumi Kimoto, Shinichi Yuta, "Autonomous mobile robot simulator - a programming tool for sensor-based behavior Autonomous Robots", *Autonomous Robots,* Springer Netherlands, Volume 1, Number 2 / June, 1995, pages 131-148.
6. Fishwick, Paul, (1995), *"Simulation Model Design & Execution: Building Digital Worlds"*, Prentice Hall.
7. Intel®, (January 2003), "Hyper-Threading Technology Technical User's Guide".
8. The Rossum Project, Open-Source Robotics Software, Retrieved July 20th 2009 from http://rossum.sourceforge.net/
9. "Microprocessor": Retrieved July 20th, 2009, from: http://en.wikipedia.org/wiki/Microprocessor.
10. Advanced Micro Devices, Inc. "Multi-core White Paper": Retrieved July 20th, 2009, from: http://www.sun.com/emrkt/innercircle/newsletter/0505multicore_wp.pdf.
11. Microsoft MSDN http://msdn.microsoft.com
12. Halfhill, Tom R., (12/31/07), "The Insider's Guide to Microprocessor Hardware, The Future of Multicore Processors". *Reed Electronics Group*.www.MPRonline.com
13. Akhter, Shameem, Roberts, Jason, (2006), *"Multi-Core Programming Increasing Performance through Software Multi-threading"*, Intel Press; 1ST edition.
14. Gerkey, Brian P., Vaughan, Richard T., Howard, Andrew, (2003), "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". *Proceedings of the International Conference on Advanced Robotics (ICAR*) pages 317-323, Coimbra, Portugal.
15. Distributed computing: Retrieved July 20th, 2009, from: http://en.wikipedia.org/wiki/Distributed_computing
16. Lewis, Bill: (1995), *"Threads Primer: A Guide to Multithreaded Programming"*, Prentice Hall.
17. Thread. sleep: Retrieved July 20th, 2009, from: http://www.javamex.com/tutorials/threads/sleep.shtml
18. Guz, Zvika, Bolotin, Evgeny, Keidar, Idit, (2009), "Many Core vs. Many-Thread Machines: Stay Away From the Valley". *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25-28.
19. Gravinghoff, Andreas, Keller, Jorg, "How to Emulate Fine-Grained Multithreading", Fern University Hagen, Germany, retrieved on 20th July 2009 from: http://www.fernunihagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/forschung/berichte/bericht_227.pdf
20. Fedy Abi-Chahla (09/16/2008), "Multi-Threaded Rendering" Retrieved on 20th July 2009 from: http://www.tomshardware.com/reviews/opengl-directx,2019-6.html

Article received: 2010-08-04