2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems

A Distributed Computing Framework for Parallelization of Coevolution in Multi-sided Conflicts

Rami Al-Motlak
Dept. of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287
rami.al-motlak@asu.edu

Jerzy W. Rozenblit, Faisal Momen
Electrical and Computer Engineering Department
The University of Arizona, Tucson, AZ 85721
{jr, momen}@ece.arizona.edu

Abstract

A large number of military simulation systems dealing with warfare and games with coordinated missions can be described as multi-sided conflicts. These systems usually involve groups sharing various alliances and relationships, each pursuing a range of different goals. A coevolutionary approach in modeling the dynamics of such complex systems allows all sides of the conflict to evolve their strategies or courses of action. The coevolutionary approach used in Sheherazade [9], a multi-sided simulation environment for Stability and Support Operations (SASO), allows each side to evolve strategies in turns against other strategies captured from the other sides. In such systems, the greater the number of plans that are evaluated, the better the final alternatives are likely to be. To improve the speed and efficiency in generating strategies, a distributed computing modular framework based on the coevolutionary approach used in Sheherazade was designed and implemented to provide the following salient features: provide a more natural model of how different sides interact in a conflict, parallelize the generation of strategies by the different sides, and improve performance by utilizing network computing capabilities.

1. Introduction

Older military war-game simulators have been designed for simulating conventional Major Theater of War operations, which typically involve two sides in conflict fighting towards simple goals such as gaining territory or maximizing enemy attrition. Their objective was to provide decision making support to the battle staff by rapidly prototyping scenarios and generating possible Courses of Action (COAs) for friendly and enemy forces. In FOX [8], an efficient plan evaluator, based on Genetic Algorithms (GAs), was developed to provide a trade-off between computational efficiency and accuracy, meeting challenges that usually arise for decision support in complex military domains. These types of simulators lacked the ability to represent an increasing number of warfare operations that involve two or more conflicting groups or forces having opposing or complementary goals. Stability and Support Operations (SASO) [11] was introduced by the US Army "to promote and sustain regional and global stability" and "to meet the immediate needs of designated groups, for a limited time, until civil authorities can accomplish these tasks without military assistance". SASO scenarios generally involve a number of groups or factions with varying interests, allegiances and capabilities, ranging from organized military forces to militia and terrorist organizations, media and refugees. Simulating such a complex environment with the aim of providing near real-time decision support .

A war-game simulator that models SASO was introduced in [9]. It brings in the use of multi-sided evolution or coevolution. The system is comprised of three parts: Sheherazade [7], a war gaming engine for SASO, ATACKS [6], a 3-Dimensional visualization platform, and a genetic algorithm that uses the Sheherazade wargamer to compute fitness values for courses of action. The model abstracts a four-part approach for representing complex military domains that involve multi-sided conflicts: setting up a SASO simulator scenario; the coevolution of courses of action by several agents; Sheherazade [7], the war-game simulator used to compute a fitness score for a course of action; and the analysis of coevolution results with scenario visualization using ATACKS. Figure 1 shows a general architecture for the four-part approach introduced in [9].

In Sheherazade [9], the war-game simulator is coupled with a genetic algorithm based coevolutionary environment. Agents are used as an abstraction in the simulation environment to represent the goals and objectives of each side of conflict. Agents are comprised of various SASO entities, which are the indivisible game pieces that carry out the COAs assigned to them. A COA for an entity consists of a movement schedule that directs the entity to move to a particular region at a specified time (within the bounds of the

978-0-7695-3602-6/09 $25.00 © 2009 IEEE
DOI 10.1109/ECBS.2009.37

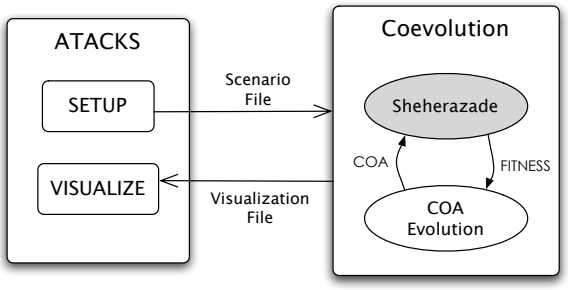

Figure 1. The four-part system architecture introduced in the coevolutionary approach

user-specified scenario time frame), and for combat-capable units, a list of the factions or other sides of the conflict to target for attack corresponding to each movement. The goal of the system is to search for good candidate strategies for each agent, where a strategy represents the set of COAs, one for each game piece owned by that agent, and present all the strategies to the battle staff for evaluation and to aid in decision support.

In the current approach, all sides in the conflict are allowed to evolve their own COAs against several static COAs captured from the other sides. This is done in a sequential manner in turns, where one agent evolves new COAs while the other agents wait. The highest scoring COAs from each agent are placed in a globally accessible hill with one slot per agent. An agent can update only its own slot on the hill with a higher scoring set of COAs and only when it is the currently evolving agent. Thus, the hill represents the current best COAs or strategies for each agent that the other agents evolve against. Given the large search space of even the abstract SASO domain used in Sheherazade, it is obvious that the longer the simulation is allowed to run (i.e. the number of agent COA-generation cycles) the larger the number of plans that are evaluated, and the better and possibly more varied the final alternatives are likely to be.

One of the major problems that exists in the current approach, however, is the sequential computational behaviors inherent in the architecture and the application's implementation. It does not take advantage of parallel hardware architectures or network capabilities currently available and with simulation times measured in hours, the system is believed to be slow in generating COAs and in evaluating strategies. Complex military domains provide many challenges to software developers due to the parallel computational natures inherent in such domains, therefore it is necessary to explore and evaluate new system architectures and techniques providing a trade-off between speed and the quality of generated plans. A speedup in generating strategies while keeping good quality is of most importance in these time-sensitive military applications. This paper presents a simulation framework that facilitates parallel implementations of such multi-sided conflict type of problems, along with a demonstrative implementation of a 'locale occupiers' game utilizing the framework.

2. Models for parallel COAs genertation

Improving the performance of COA generation for multi-sided conflicts is a crucial factor in the parallelization process. In the existing model [9], agents interact in a sequential manner by taking turns to put their best plans on a global hill. The strategy used by agents to decide when to update their best plan on the hill is another important element in the model. Currently, each agent uses a genetic algorithm to evolve plans for a predefined number of generations. The plan with the highest score is selected to challenge the hill, and if it results in a higher score, the corresponding COA on the hill is replaced by the challenging one. This paradigm has some disadvantages. Firstly, the order of agents' interaction and the number of generations the GA evolves before updating an agent's plan are parameters of the game setup. It may not accurately reflect how agents interact in the real world. Secondly, each agent must wait for its predecessors before it gets a chance to evolve and update its own plan.

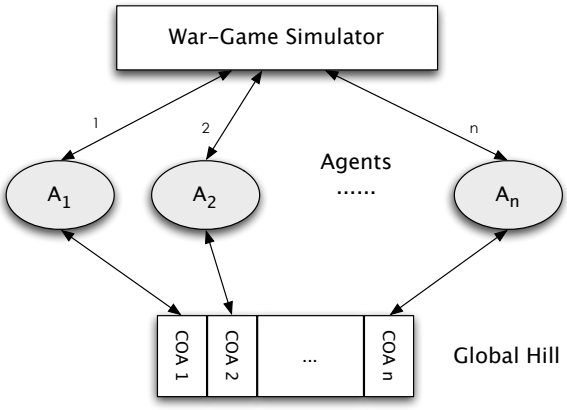

Figure 2. Agent interaction model in the sequential approach

Despite these issues however, the model can be considered to be fair. Each agent gets to evolve a predefined number of generations to come up with a new strategy. Furthermore, all agents wait uniformly for their next turn. Figure 2 shows a general overview of the architecture used in this approach [9]. It involves a war-game simulator attached to

a sequential coevolution process and shows how different agents interact with the global hill to achieve the COA updates.

For a game with n agents, the set of agents A and the Global Hill GH which is the set of best courses-of-action from each agent, can be represented as $A = \{A_1, A_2, \ldots A_n\}$ and $GH = \{COA_1, COA_2, \ldots COA_n\}$. Each agent is given a turn to evolve a predefined number of times. The best COA achieved is run against all COAs achieved so far by other agents on the global hill, and if a newly generated COA scores higher than that agent's dominant COA on the hill, the global hill will be updated.

By assuming that the game is repeated k times, where k is the specified number of turns needed to run the coevolution, an arithmetic formula can be derived to determine the total time needed to run the game. Here, T represents the total time to run the game, and T_j^i is the time required by agent j to iterate sequentially through evolution cycle i.

$$T = \sum_{i=0}^{k} T^i$$

$$T = \sum_{i=0}^{k} T_1^i + T_2^i + \ldots T_n^i$$

Figure 3 (top) shows the agent interaction model in the current sequential approach, and how the global hill is altered at the end of each agent's turn. Three agents are used in the illustration. The timing chart, shown in Figure 4, demonstrates the timing layout for 4 different agents during an iteration of the game. In the sequential scheme, the global hill update occurs at the end of each turn of an agent if and only if the strategy is found to score higher than the previous strategy.

2.1. Parallel coevolution with fixed update

In this approach, the constraint that an agent must wait in a predetermined sequence for its turn to update the hill is removed. All agents will start generating plans in parallel. Each agent runs concurrently on a separate processor or machine in a distributed system, making a local copy of the hill at a synchronization point, and using the GA to evolve plans for a predetermined number of generations. The update policy for each agent is to update its corresponding slot on the hill after evolving plans for a fixed predetermined number of generations.

The agent interaction model is basically a synchronization point, followed by concurrent evolution of new plans, followed by another synchronization point and so on until the specified number of evolution turns have elapsed. The COA update criteria in this model is basically the same as in the sequential model. At a synchronization point, each

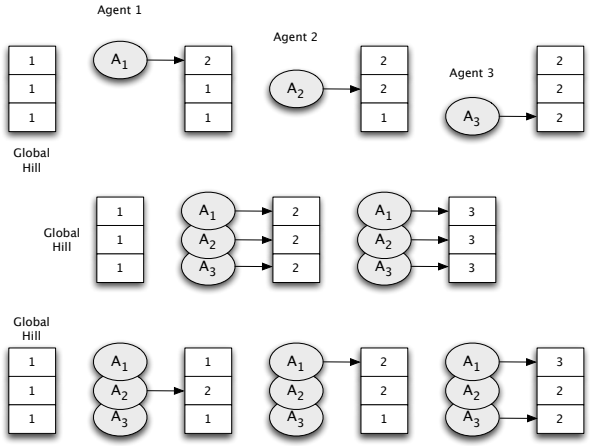

Figure 3. The Agent interaction model with global hill in the sequential approach (top), parallel coevolution with fixed update scheme (middle), and reactive update scheme (bottom)

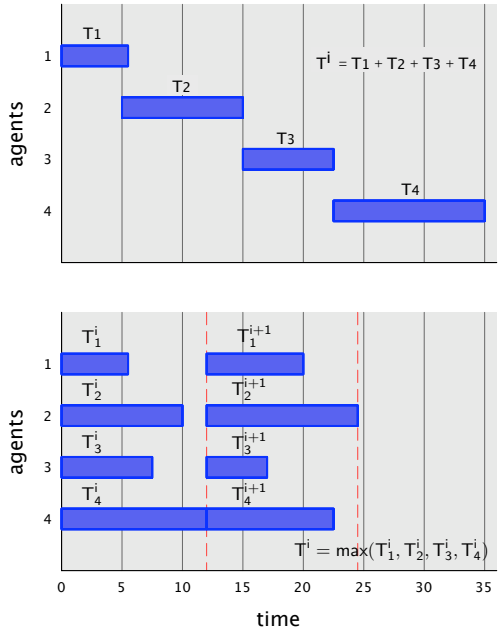

Figure 4. Timing chart for the sequental (top) and fixed update approach (bottom)

agent uses a copy of the current hill to evolve new strategies for a fixed predetermined number of generations. Once the best strategy is found, it is used to replace the current COA on the hill. Figure 3 (middle) shows the agent interaction model in this approach, and how each agent is challenging a copy of the global hill at the same time and updating its corresponding COA waiting for the next synchronization point to repeat the process.

By assuming that the game is run for k cycles, the total time T needed by the game to generate the strategies is given by:

$$T = \sum_{i=0}^{k} T^i$$

$$T = \sum_{i=0}^{k} Max(T_1^i, T_2^i, \ldots T_n^i)$$

The timing chart, shown in Figure 4, demonstrates the timing layout for 4 different agents running for 2 iterations using this scheme.

2.2. Parallel coevolution with reactive update

In this approach, the constraint that all agents must evolve for a fixed predetermined number of generations at each turn is removed. An agent does not have to wait for a certain number of generations before selecting a COA to challenge the hill. The synchronization points from the previous approach are removed. Each agent runs concurrently and maintains a local copy of the hill. Once an agent has generated a COA, it is scored against the local copy of the hill. If the generated COA scores higher than the current one on the local hill, it immediately attempts to update the central global hill by replacing the corresponding COA with the new one. At that point, a broadcast occurs to inform all other agents of the new state of the global hill. As a result, all running agents will have a copy of the newest COAs achieved so far.

Figure 3 shows the agent interaction model in the reactive update approach, and how each agent is challenging a copy of the global hill at the same time and updating its corresponding COA.

It may be observed that some agents might be slower than others to react and that the evolution process might take longer for some agents than others. Hence, the game might not be fair in the sense that each agent is not given an equal opportunity to update the global hill. However, the total time needed by the game to generate the strategies is guaranteed to be less than, or in the worst case equal to, the two previous schemes due to the fact that no overhead time is lost due to synchronization, and the removal of the constraint that all agents must evolve for a fixed predetermined number of generations. A COA update policy is of elevated importance in this model. Two different update polices can be considered in this approach.

2.2.1. Reactive update with multiple plan comparison. In this COA update policy, once the global hill is successfully updated by an agent and a broadcast occurs from the global hill to each local hill running under each agent, other agents not responsible for the change might be still working on scoring their own COA against the old local hill. This policy suggests that an agent keeps working on challenging the old known status of the hill while starting a new thread to process challenging its COAs against the latest hill status achieved so far. Since the global hill broadcast can occur at different times, an agent might be running multiple threads challenging its own COA against different instances of the hill. At the end, an agent will choose the COA with the highest score as its best COA and will update the global hill accordingly.

This policy increases the sensitivity of the system but at the expense of using more computing resources of the processor due to the creation of multiple threads for challenging different copies of the hill. Figure 5 (top) illustrates this update policy.

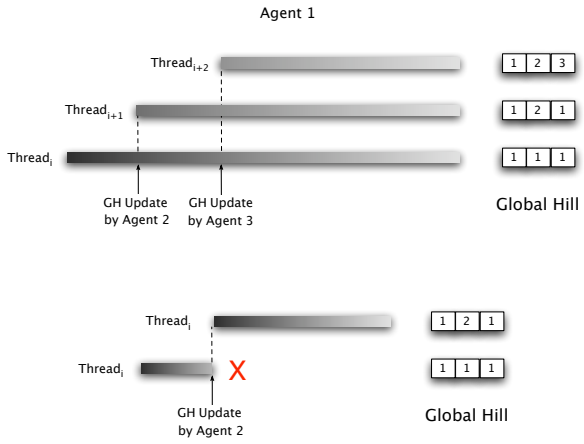

Figure 5. Update-policy with and without (bottom) multiple-plan-comparison

2.2.2. Reactive update without multiple plan comparison. In this COA update policy, once an agent updates the global hill and the new hill is copied to all other agents, any agent working on challenging its old local hill should abandon that process and start challenging the current state of the hill. This possibly can be unfair to agents that are slow to react, and can lead to certain agents always forcing their COAs on other agents by frequently updating the hill

causing the other agents to constantly reset their evolution processes, and therefore may only be useful if such an arrangement reflects the real-world nature of the multi-sided interaction. Figure 5 (bottom) shows a demonstration of this update policy.

2.3. A Hybrid Scheme for Parallel Coevolution

An more ideal approach for parallel plan generation could be derived by combining some of the features of the fixed update and the reactive update approaches. In this hybrid approach, the constraint that all agents must evolve for a predetermined number of generations is removed, and the necessity to update the global hill when an agent is done with challenging its local hill is removed as well. Instead, the global hill's update occurs at fixed time intervals, Δt, specified by the user as a reasonable value greater than the average time needed to challenge the hill and run the war-game simulator in the game. This would decrease the overhead that can occur due to many global hill updates and at the same time provides reasonable performance. The global hill update policy could be a simple "without multiple plan comparison" update policy but with no need to cancel the agent's process of challenging its old local hill before the update. The update can be reflected to its local hill after being done with that current COA evaluation process.

To achieve fairness in the game as in the old sequential model, a counter can be added to each agent to count the number of turns passed so far by each agent. The game will continue until all agents' counters indicate that all agents have reached the same number of iterations. The total time T needed by the game to generate the strategies is guaranteed to be less than the sequential method and the fixed update method because all agents are running continuously with no waiting overhead for synchronization. A timing chart, shown in Figure 6, demonstrates the timing layout using 3 iterations for this scheme.

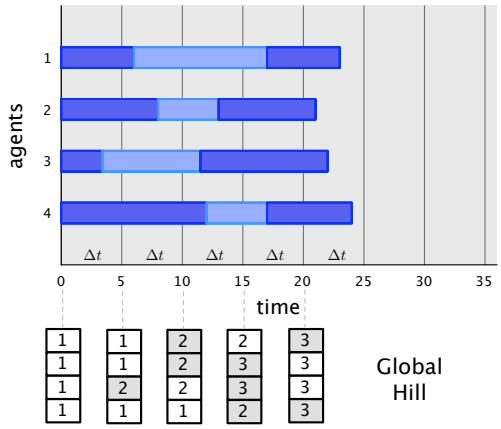

Figure 6. Timing chart of the hybrid method for parallel coevolution

3. Parallel GA Coevolution

Parallelization in our multi-sided conflict simulators can be achieved by changing the architecture, the agents' interaction model or the global hill update policy. Another level of parallelization can be accomplished at the functional and data level. The functional model splits the functionality into modules whereas the data model splits the data into smaller data chunks. GAs are used as the core algorithm for the coevolution process among agents. Hence, applying the functional and/or data models of parallelization on a GA plays an important role in speeding up the process of generating strategies.

Coevolution using parallel GAs must attempt to keep the quality of results and the way the GA functions the same, while providing a speedup factor. Both the single-population fine-grained GA and multiple-population coarse-grained GAs change the way the GA works [1, 3, 5]. The global single-population master-slave GA seems to be a reasonable choice for GA parallelization because of the similarity it shares with our current simulation environment. The use of this algorithm for coevolution can be accomplished by replicating the functionality of the war-game fitness simulator on multiple nodes, and by splitting the individuals in the population among sub-populations that can be evaluated separately on each simulator slave. Figure 7 shows a general overview of the new architecture which involves multiple war-game simulator slaves attached to a coevolution process breaking down the coevolution process to show more details of how different agents interact with the global hill to achieve the COA update.

The data layout is important in this model because the computation of relative fitness values for the selection process requires global communication among subpopulations. Synchronization is needed as well since the population is divided among several fitness simulator processes. Replication of fitness simulator is required for parallelization using this scheme. Individuals in a population can be divided equally among the slaves for fairness. A uniform distribution of individuals is not a must but is preferable.

4. Application Framework

Many design details, use case scenarios and system requirements in coevolutionary environments can be provided by domain experts. During the design process, many approaches can be used to model the system. Software architects and developers employ the knowledge gained from

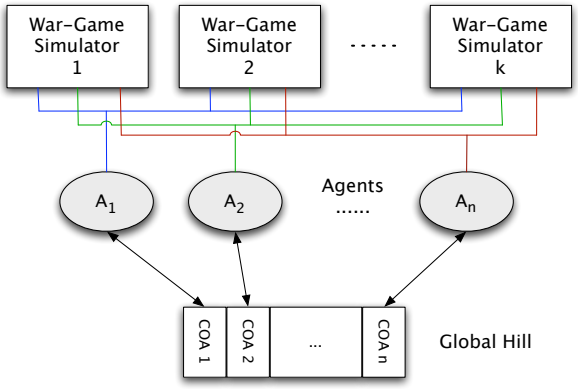

Figure 7. A detailed architecture of multi-sided conflicts using the global single-population master-slave GA

domain experts to develop software applications for simulating these environments. On the other hand, these simulation applications should help domain experts to gain a better view of how different components in the system interact and to study possible scenarios that will help in the decision making process. These simulation applications usually lack the ability to support any new approaches to model these systems, fail to adapt with new domain specifications that can arise from domain experts and most importantly may lack the ability to support different computing platforms and/or operating systems in the long run. As a result, an application framework for such complex environments is of utmost importance.

An application framework is a set of structures, templates and modules that provide a context to help a programmer in implementing custom applications for a specific domain [2, 4]. It defines a skeleton for the software application and forces the programmer to look at the big picture. The skeleton can be defined as a set of classes, abstract classes and predefined interactions among classes in a framework. Developers can then develop their applications on top of the framework, taking advantage of modularity, code reuse and design structure provided by the framework. A general application framework, MULTISIDED_CONFLICTS, is introduced as an application framework to address the parallelization of plan generation in multi-sided conflicts. It employs both the sequential coevolution and the parallel coevolution with fixed update approaches. It adds the utilization of parallel GA for the coevolution process as well.

Implementation utilizes a low-cost Java-based parallel computing platform for measurement and data processing purposes. The increase of high speed, low latency networking has provided a useful and inexpensive alternative to supercomputers and other parallel processing hardware. Tasks are distributed on a number of machines loosely connected by a network using the standard IP stack instead of running the entire task on a centralized high performance machine. The programming environment utilized is Java 6.0, from Sun Microsystems. Java [10] has definite advantages such as platform independence and suitability for network executions.

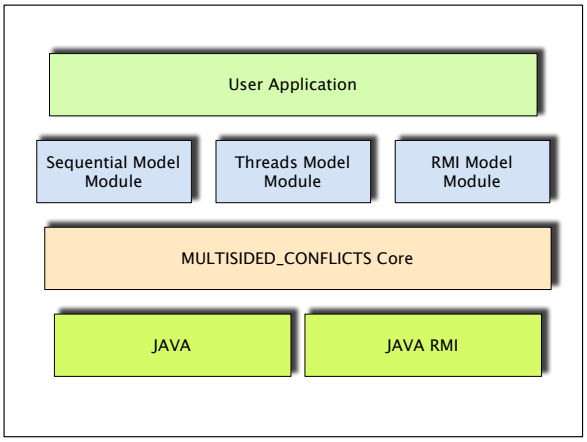

Figure 8. : Layered architecture for MULTISIDED-CONFLICTS

The MULTISIDED_CONFLICTS framework is a layered architecture consisting of four self-contained modules. The modules are built on top of the core JAVA and JAVA RMI framework, which provides the flexibility of the framework being supported by many vendors' operating systems and different hardware components by merely having a JVM installed on the system. Figure 8 shows the different software modules supported by the framework. Each of the modules that comprise the MULTISIDED_CONFLICTS framework can stand on its own or be implemented jointly with one or more of the others. The functionality of each component (or module) is as follows:

1. The core component: The core module provides the essential functionality of the framework. It provides the tools required to read initialization parameters from files, print results to files and measure critical time processing parameters. Furthermore, it provides the functionality needed to use the genetic algorithm for the coevolution of agents. It also provides the necessary abstract software components in a coevolutionary environment such as agents, the global hill and COAs.

2. The sequential model component: This module provides the essential functionality to construct a coevolution model of agents running based on the sequential

scheme where each agent is given a turn to evolve and access the global hill if needed. A sequential controller is the core of this module.

3. The threads model component: This module provides the essential functionality to construct a parallel coevolution with fixed update model of agents using threads running on the same machine. It also allows the evolving of all agents simultaneously while they access the global hill, if needed, as shared memory. A population of chromosomes, for an agent, can be divided into sub-populations while a proportional number of threads running fitness simulators are created for that specific agent. A threads controller is the core of this module.

4. The RMI model component: This module provides the essential functionality to construct a distributed parallel coevolution with fixed update model of agents using threads and RMI server/client architecture. Grid computing is exploited in this module and used for the coevolution process. It provides the server/client implementation for Agents, fitness simulators and the global hill. A controller is also provided as a component of this module.

Figure 9 shows an example of the layout of threads using the threads model where 3 agents are used, 2 fitness simulators for each agent and a model controller. The initialization and creation of threads is done by the controller which holds the global hill as shared memory. The controller will allow the model to coevolve based on the number of turns read during initialization.

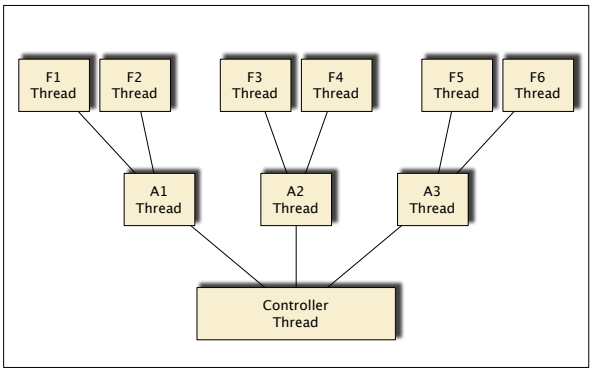

Figure 9. An example of a threads model layout

Alternatively, Figure 10 shows an example of the layout of components in an RMI model where 3 agents are used, a single fitness simulator for each agent and a global hill. All these nodes are servers that can be running on the same or separate JVMs which in turn can exist on a single or on several machines. Running the servers is done using batch files. The IP addresses and port numbers of the machines, where servers are to be run, must be known and should be stored in an initialization file before running the controller.

The MULTISIDED_CONFLICTS framework consists of several Java packages that provide complementary functionality for building applications using this framework. All customization and game run-time parameters are stored in initialization files. File Input/Output handling is done via a Tools package in the core module. The input files also provide for some custom parameters that are game dependent and need to be changed accordingly.

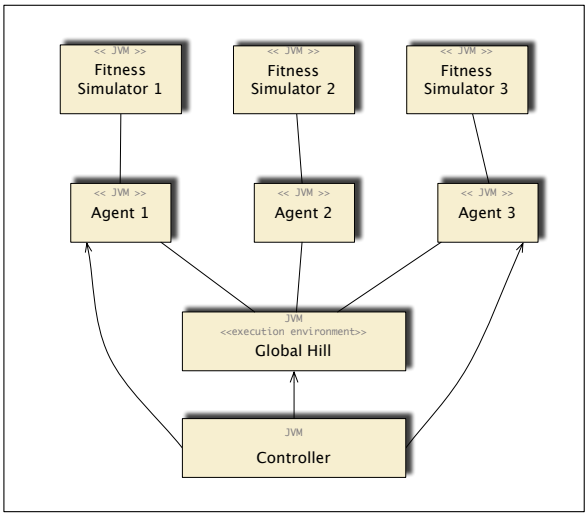

Figure 10. An example of an RMI model layout

5 Experimental Evaluation

A "Locale occupiers" game was implemented using MULTISIDED_CONFLICTS to evaluate the framework. The game consists of multiple players, each represented by one color, randomly placed on an n x n grid. The objective of each player is to move one element up, down, left or right at a time through the grid claiming as many remaining uncolored grid elements until no further movement is possible. The fitness is computed based on the percentage of the board occupied by each player at the end of the game. The usage of the framework made it an easy process to develop the game. Three different versions of the game were implemented using the three different models supported by the framework, namely, the sequential model, the concurrent model using threads and the concurrent model using JAVA RMI and threads. The parameters $n, x[], y[]$ were added to the initialization files as custom parameters required to overload a version of the method *RunSimulation*. The pa-

rameter n represents the size of the board, whereas both $x[]$ and $y[]$ represent the initial position of each player on the board. The values used for these custom parameters as well as the *SetofGenes* default parameter which is dependent on the game were: n = 40, $X[]$ = {5,6,7,2,1,6,4,8,2,3}, $Y[]$ = {8,9,1,2,5,6,4,2,3,6}, and *SetofGenes* = {L,R,U,D}. Other default values determined in running the experiments were: *PopulationSize* = 100, *CrossoverProbability* = 0.25, *MutationProbability* = 0.02, *ChromosomeLength* = 100, *NumberOfTurns* = 100, and *NumberOfIterationsPerCycle* = 10.

The parameters *NumberOfAgents* and *NumberOfSubGenerations* were varied in these experiments to analyze how a model will react regarding execution time, speedup and quality of results. The latter parameter is only important for the parallelized models and not used in the default sequential model. The system used to run the experiments was an AMD Athlon 3000+ 800MHz processor with 256 MB of RAM, and Microsoft XP Home edition Version 2002 with Service Pack 2.

Ten sub-generations were used in the parallelized threads model whereas five sub-generations were used in the parallelized RMI model. These two values for the parameter NumberOfSubGenerations were determined as optimized values for these two models by running several experiments on the hardware system used to conduct the experiments. These optimized values may vary if a different hardware system is used. The key is to utilize a value that will provide a high CPU utilization while not adding tremendous unjustified time latency due to the usage of virtual memory swapping. Conducting real-time experiments adds many time constraints that should be taken care of by optimizing the system to run efficiently on the available hardware.

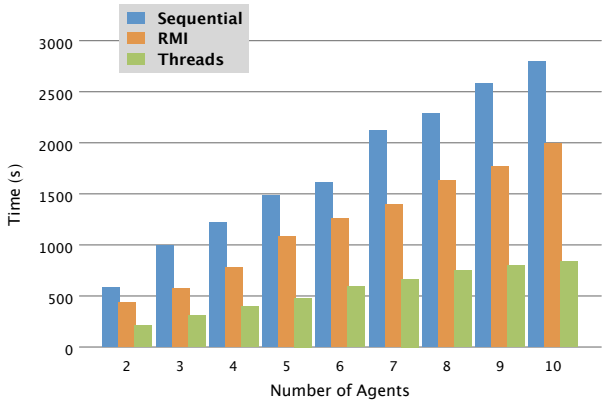

Figure 11. A histogram for the distribution of execution times among the three supported models

A histogram showing the distribution of execution times among the models with a variable number of agents is provided in Figure 11. A better understanding of measuring performance regarding execution time can be clearly observed by analyzing the line chart in Figure 12. The threads model has the tendency of converging towards a threshold limit of execution time as the number of agents increases, while the other models tend to linearly grow as the number of agents grows. However, the slope of the line for the default sequential model can be seen to be larger than that of the RMI model.

While running the experiments, it was observed that the sequential model had the worst CPU utilization. Increasing the load, by running the model with a larger number of agents, simply produced the effect of a waiting job queue but not an increasing CPU utilization. This explains the linear nature of the performance obtained in the experiment. On the other hand, the RMI model had the highest CPU utilization. The addition of agents created the effect of a waiting job queue as well since the maximum utilization is achieved so far due to running several servers on the machine all together. The threads model achieved a better CPU utilization than the sequential model and was adapting to the addition of new agents by increasing CPU utilization. Hence, it had the tendency to converge to a threshold execution time.

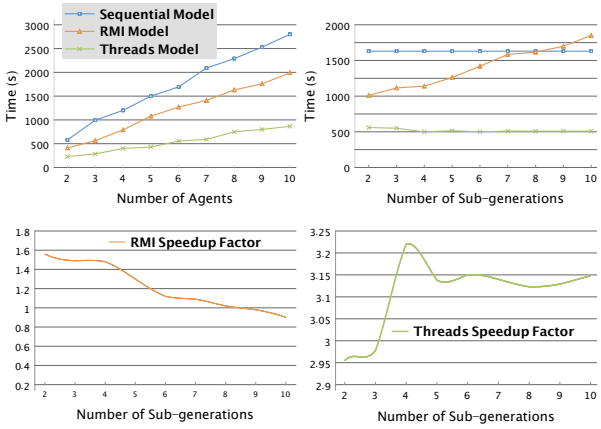

Figure 12. Execution times vs. number of agents (top left) and number of sub-generations (top right), and speedup factor of the RMI model (bottom left) and Threads Model (bottom right) over the sequential model

The speedup factor is calculated for the threads and RMI models with respect to the default sequential model. The mean value of the speedup factor for the RMI model was 1.445 whereas for the threads model the speedup was 3.165. A much better value can be achieved for the RMI model or even the threads model by providing more hardware re-

sources. In fact, the RMI model has high scalability allowing it to run on a fast LAN by sharing several separate CPUs. A fast LAN will guarantee that the overhead time lost for communication is minimal and comparable to that lost for communication among servers running on the same machine. This is achievable in today's technologies as 10/100 Mbps or faster network cards are available on almost all new LANs.

The second part of the conducted experiments was to change the number of sub-generations in the RMI and threads models while having a predefined number of agents. The median value of the number of agents between 2 and 10, which is 6, is chosen and fixed to be used later. The value 1626 seconds obtained from running the sequential model in the first part of the experiment is used for comparison. The number of sub-generations cannot be changed in the sequential model.

The execution times of the threads model were found to be the best. The model was able to accommodate further partitioning of its agents' generations by increasing CPU utilization. As a result, all execution times achieved by the threads model were comparable and within the same range. On the other hand, adding more partitions to the RMI model will be reflected in adding more fitness servers to the model, hence increasing CPU utilization until maximum is achieved. Subsequently performance decreases due to overhead generated by too much communication time latency. As a result, the RMI model performs well until the number of sub-generations becomes too high and it suffers after that from a decrease in performance. Figure 12 shows that execution times of the RMI model became even higher than that of the default sequential model after reaching a number of sub-generations equal to 7.

The speedup factor is calculated for the threads and RMI models with respect to the default sequential model. The mean value of the speedup factor for the RMI model was 1.193 compared to 3.11 for the threads model. The drop of the RMI speedup factor below 1.0 is observed after the number of sub-generations increases above 7, and it keeps dropping thereafter. Figure 12 also shows the speedup factor fluctuation by changing the number of sub-generations in both the RMI and threads models. The highest values seem to occur at small numbers of sub-generations in the RMI model whereas it seems to occur at the highest values of the number of sub-generations in the threads model.

The last part of the experiments was to study and analyze the quality of results obtained by running the three different models. Speedup was achieved by the threads and RMI models compared with the sequential model by using the same hardware resources. Furthermore, speed up can be enhanced further by using better hardware resources or a fast LAN compared to the sequential model which will suffer from enhancing performance even if better hardware resources are to be available. The last check point was to make sure that the quality of results achieved from running the RMI and threads models are comparable to those of the sequential model. The result set from the first part of the experiments was used for the analysis. The double values returned as scores for the fitness of COAs on the global hill are used at the end of the game to compare results and to observe the winner. A problem that may arise for winner comparison can be due to the random nature of the GA algorithm. As a result a game might end up with different winners in spite of the fact that the same model is used.

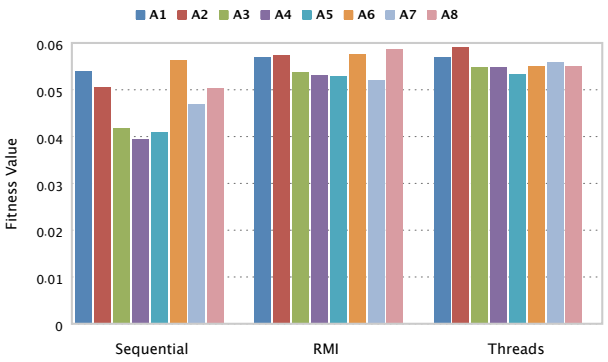

Figure 13. Quality of Results histogram for 8 agents

A closer look at the histogram in Figure 13 shows that fitness values in the RMI and threads model tend to fall within the same range with little variance while the values might be a little more distributed in the case of the sequential model. This is due to the fact that agents are running simultaneously and trying to compete and evolve strategies in real time unlike the sequential model where they are given turns in order. In fact, both the RMI and threads models create a more competitive environment among agents and tend to force agents to develop plans faster and in a more life-like style. However, the quality of results achieved in these two models is still comparable with respect to the results achieved from the sequential model.

6 Conclusion

The design, implementation and evaluation of a new application framework called MULTISIDED_CONFLICTS to tackle the problem of improving performance of plan generation in multi-sided conflicts was presented in this paper. It supports rapid development of RMI and thread based applications in this domain that can take advantage of a wide range of available concurrent computing resources. A reasonable and expected degree of speedup with an accompanying increase in CPU utilization was demonstrated through

the implementation of an example multi-sided game using the framework. Several avenues for future work remain. For instance, several new models based on other agent interaction approaches can be added to the framework. This will enable researchers to study and compare different models and approaches. More experiments especially in distributed environments can be conducted to study the maximum speedup that can be achieved in the RMI model in particular.

References

[1] P. Adamidis. Review of parallel genetic algorithms bibliography. Technical Report 1, Aristotle University of Thessaloniki, Thessaloniki, Greece, 1994.

[2] L. Deutsch. Design reuse and frameworks in the smalltalk-80 system. In T. Biggerstaff and A. Perlis, editors, *Software Reusability*, volume 2, pages 57–71. ACM Press, 1989.

[3] V. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183, San Mateo, CA, 1993. Morgan Kaufmann.

[4] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[5] S. Lin, W. Punch, and E. Goodman. Coarse-grain parallel genetic algorithms categorization and new approach. In *Sixth IEEE Symposium on Parallel and Distributed Processin*, Los Alamitos, CA, October 1994. IEEE Computer Society Press.

[6] J. W. Rozenblit, L. Suantak, and M. Barnes. Multi-agent approach to decision support in advanced tactical architecture for combat knowledge system (atacks). In *Advanced Simulation Technology Conference*, San Diego, April 2002.

[7] J. Schlabach and D. Hillis. Sheherazade: A research platform for decision support in military stability and support operations. Technical Report 01, BCBL-H, 2003.

[8] J. L. Schlabach, C. C. Hayes, and D. E. Goldberg. FOX-GA: a genetic algorithm for generating and analyzing battlefield courses of action. *Evolutionary Computation*, 7(1), Spring 1999.

[9] L. Suantak, D. Hillis, J. Schlabach, and J. W. Rozenblit. A coevolutionary approach to course of action generation and visualization in multi-sided conflicts. *IEEE International Conference on Systems, Man and Cybernetics*, 2(5-8):1973–1978, October 2003.

[10] Sun Microsystems Inc. The source for java developers. Online, http://java.sun.com, October 2007.

[11] U.S Army publication. *Field Manual (FM) 3-07 Stability and Support Operations*, February 2003.