A Parallel Shared Memory Simulator for Command and Control

Christophe *Jaillet*
Département de Mathématiques et Informatique,
Université de Reims Champagne-Ardenne
BP 1039, F-51687 Reims Cedex 2, France
christophe.jaillet@univ-reims.fr

Michaël *Krajecki*
LERI - EA2618,
Université de Reims Champagne-Ardenne
BP 1039, F-51687 Reims Cedex 2, France
michael.krajecki@univ-reims.fr

Jean *Fugère*
Collège Militaire Royal
CP 17000, Succursale Forces
Kingston, Ontario, Canada, K7K 7B4
fugere-j@rmc.ca

Abstract

This paper introduces a military application in the command and control field. The main feature of this study is the parallelization of the simulator. The simulator is object-oriented and written in C++. It uses the OpenMP standard for the parallel version. To produce an efficient parallel simulator, we have to deal with the dynamic load balancing problem.

Keywords parallelism, simulation, command and control, dynamic load balancing, OpenMP

1. Introduction

In the literature, a large number of models of attrition for classical warfare have been designed. In 1914, Lanchester Equations were introduced as a set of coupled ordinary differential equations. But this kind of model is not well suited to represent modern battlefield. Recently, new models like Einstein have been developed [7]. The main feature of these new models is the representation of each soldier by a small independent identity which acts autonomously using some local sensors.

The *RMCSim* (Royal Military College Simulator) is based on a multiagent-based simulation which is, with respect to some aspects, more general than Einstein. The theoretical model on which RMCSim is based on is the mobile cellular automata formalism.

Simulations are very time consuming for large battlefields (with numerous entities involved) and make numerous independent computations to determine the position of all these soldiers and the different combat results.

This is the main reason why we proposed to parallelize RMCSim. Two main approaches can be chosen to achieve this goal : developing a message-passing based simulator (with MPI or PVM [9, 5]) or designing a Parallel Shared Memory Implementation of RMCSim (using OpenMP [10]). A message passing based parallel version of ModSaf (Modular Semi-Automated Forces is an entity level simulation) has already been proposed [1]. In this paper, we will explain why we are actually developing the second solution using OpenMP and how to address the load balancing challenge.

In the next section, we will first introduce the RMCSim Project. In section 3, we will discuss how to parallelize the simulator and compare the message-passing and shared memory approaches. Section 4 is dedicated to the current development of the shared memory version of RMCSim using OpenMP. Finally, we will conclude by giving some perspectives for future work.

2. The design of RMCSim

RMCSim is a combat simulation project which can be compared in some aspect to the Einstein project [7]. RMCSim goes further by adding some new features (for example, the number of teams is only bounded by the computer capacity, the behavior vector is fully adjustable,...).

The key feature is the RMCSim's ability to conduct parallel simulations.

This project has been designed using UML and developed in C++, in order to be easily extensible to more complex models of agents, including terrain, weapon systems,

Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)
0-7695-1626-2/02 $17.00 © 2002 IEEE

etc.. The simulation is based on mobile cellular automata. RMCSim realizes a discreet simulation compare to continuous simulations which are generally based on the Lanchester Equations. In 1999, the authors had explored the benefits of a parallel Lanchester equations based simulator [4]. Each soldier (assimilated to an agent) is considered as a mobile cell, i.e. the neighborhood of each cell can be different at each simulation iteration.

2.1. Mobile cellular automata

Cellular automata involves the interactions of an ensemble of cells, each subject to specific rules of behavior (which usually depend on other cells in the immediate neighborhood) and whose behavior from an ensemble point of view leads to a global evolution of the cellular ensemble. Cells live on a grid, will possess various states of information such as its location on the grid, its neighbors location, etc... and, based on a set of evolutionary rules will move to a next state. This is done either simultaneously or one at a time in a stochastic fashion.

The classic example of a cellular automaton is the game of Life invented by John Horton Conway in the early 1970's. This cellular automaton, which has been the subject of considerable research, describes the behavior of an ensemble of cells where cells may be born, die and reborn, depending on the status of the eight nearest neighbors. This simple game demonstrated clearly that complex global behavior can result from simple rules governing the individual cell's behavior.

In the context of war gaming, a cell is generalized to a software agent, which can model different elements of the battlefield such as soldiers and tanks. Agents are also assigned a color side to reflect the competing armies (like "red" or "blue" for two teams but not limited to). The rules of evolution are also more complex and are designed to model the behavior of real entities on the battlefield. A war game simulation using this approach involves establishing the initial configuration of the agent entities, their capabilities, rules of engagement and then cycling through all the agents. A cycle where all of the agents have been processed is called a generation or iteration and the game continue until one side wins or after a finite number of generations have occurred. Clearly this type of model is ideal for parallel processing since we could look at each agent as a separate process on a separate processor.

2.2. Modeling an agent

Agents possess different characteristics, which enable them to make decisions concerning their movement, and generally the way they will carry themselves on the battlefield. In particular, agents are described by the following parameters.

- Zones of action, centered on the agent. This includes a displacement zone, a firing zone, a communication zone, etc.
- A state of health from alive and well to wounded, to dead.
- A personality vector which for the moment is modeled after the EINSTEIN model and include six components. It governs an agent attraction to move towards friends or foes, alive and wounded as well as the tendency to move towards its own flag or the enemy flag. Taking the enemy flag is often the criteria for a victory and the end of the simulation.
- Agents can have the same or distinct personality vectors.
- Agents can belong to distinct squadrons with different weapon systems.
- Agents can be made to move towards way points, thereby allowing for tactical planning.
- Agents have a hit probability, a radius of fire and a fratricide probability.
- The movement of agents can be overridden by *meta rules* such as *if the best move appears to be at (x,y) but at (x,y) there are more than z enemy soldiers, you will not go there*.
- Agents are affected by terrain conditions ranging from difficult terrain on which an agent can manoeuvre to outright obstacle such as buildings which must be contoured.
- The state of health of an agent goes from alive to wounded to dead. The above represents a brief description of an agents characteristics and part of our research interest is in defining meaningful evolution of an agent's personality (learning) to render the simulation more realistic.

2.3. Simulation

The combats are supposed to be done simultaneously and the health state of each agent is update on the completion of each iteration. If the health state of a particular agent is equal to null, then this agent is reputed to be dead and is no more taken into account until the end of the simulation process.

At each iteration, each agent updates the attraction vector of his zone of actions (using his behavior vector and his

sensor range). Then he chooses to move to the most attractive (and accessible) position on the battlefield. The reader may notice the agent has to make sure that the targeted position is not already occupied by an other agent.

To ensure fairness, at each iteration the order of processing the agents is determined using a nearly random distribution.

3. The different parallelization approaches

Thanks to the agent autonomy during the simulation, we can exhibit a large set of independent calculations. Moreover, this application is very time consuming because : in fact, the evaluation of the attractiveness of the action zone (to be repeated at each iteration for each agent) is expensive and does not need to be executed in a dedicated order. For all these reasons, RMCSim is a good candidate for parallelization.

We can plan to parallelize both parts of an iteration : the combat step and the movement step.

We can choose to parallelize the application by distributing the agents or the battlefield. Here, we discuss about the definition of the load unit used to evaluate and share the work to realize and not about the choice between the message-passing and the shared memory model. This second point will be discuss later in the paper.

In this work, we will only focus on the parallelization of the movement step because of preponderant part in the execution time. The combat component will be added subsequently, since these are less costly in time.

3.1. Battlefield based repartition

In its present form, the battlefield is viewed as a planar grid (a 2D-mesh terain surface). A more complex GIS (*Geographic Information System*) is actually under consideration and should be available before the end of 2002.

Parallelizing RMCSim by sharing the battlefield consists in splitting the grid into different geographical areas.

By splitting the grid, we are designing the load unit : each processor will be responsible for all the computations attached to one or more areas. Each area is considered with the agents actually moving on.

To determine the agent movements, the processors use the neighborhood and realize all the associated calculations.

3.1.1 Comparison between message passing and shared memory parallelization

By exploiting a message passing approach, the different geographical areas will be stored in the local memories associated to the processors managing each of these areas.

If the preferred solution exploits a shared memory model, the grid (and all the geographical areas) will be placed in the global memory to assure an access from all the processors.

When a processor evaluates an agent on the boundary of the area, it starts by determining the attraction of the various positions in an autonomous way. When the memory is shared, there is no problem since the processors need only read access to the information.

If the information is distributed, some messages will be essential to obtain the required information, but this operation is not really difficult to achieve.

The key obstacle to overcome is to ensure that the position is free before moving an agent on it. In this case, we can meet a concurrent write access problem in the particular case where the movement elected for the agent is placed under the responsibility of a different processor.

Therefore in a distributed approach, when Agent A leaves a geographical area placed under the responsibility of processor P_i and moves to a new area managed by processor P_j $(i \neq j)$, an exchanged between the two processors is required : P_i will address a request to P_j and if agreed, will send A to it.

If the grid is stored in a shared memory, the write access (to update agent positions) is a critical section which needs the use of locks to guarantee the exclusive write.

For both cases, the move of an agent leaving from one processor to another one induces an overhead which is visible as a message exchange or a critical section access.

3.1.2 Dynamic load balancing

The geographical areas assigned to the processors are not all equivalent with respect to the load because this one is mainly related to the number of agents to analyze.

This is the main reason to provide a dynamic load balancing scheme to balance, not only the geographical area, but the number of agents between the processors. So, processor P_i can manage a large number of nearly empty area (containing a very little number of agents) and, at the opposite P_2 can be responsible for only a crowded area. The reader may refer to [8] to have a more complete presentation of the dynamic load balancing problem.

Furthermore, when the memory is distributed, another kind of imbalance can arise related to the network congestion and depending on the density of communication which requires the processing of the various areas (the external areas of the grid have less neighboring areas and should require less communication exchanges).

This imbalance factor can also affect the behavior of the simulator if the memory is virtually shared (CC-NUMA system : Cache-Coherent Non Uniform Memory Access) by disturbing the cache manager.

3.2. Agent based repartition

Each agent is tagged with a unique identifier which is also independent of its team.

To parallelize the application while choosing to distribute the agents consists in splitting the whole agent set into various groups, disjoined and independent of the involved teams.

This operation amounts considering the various groups of agents as being independent, and carrying out calculations for each one of their elements in any order. Each processor is responsible for one or more groups of agents.

A processor analyzes the behavior of an agent who falls on it. It must evaluate the attractions of the various positions of the geographical agent neighborhood, then carry out its possible move by respecting the concurrent write access.

3.2.1 Comparison between message passing and shared memory parallelization

By using a shared memory, the grid storage is shared by all the processors even if the agents can only be managed by their associated processor. The evaluation of the neighborhood attractiveness is an easy task regardless of the research of informations; it is also the case for the movement of an agent if the concurrent write problem is well solved. For these two particular aspects, there is no difference (for the solutions using a shared memory) between the strategy of distribution of the battlefield and the one based on the agents.

The key advantage provided by this approach is the easier management of the load balancing problems. This point will be discussed in the next section.

At the opposite, when using a message passing model, and because of the agent based distribution, the grid management is more difficult. We have to develop a mechanism to manage the grid ; in other words, a scheme responsible for the distribution of the grid in the different local memories. Indeed, nothing indicates which processor must deal with this or that part of the grid. So the grid may be not distributed using the geographical information, thus any processor may try to access any part of it and exchanges data with any other processor. This point may be critical because it forbids to take into account the local character of a message according to the network of communication.

Even if this solution seems to be more difficult to address with a message passing system, it allows the distinction between the load balancing problem and the data location.

3.2.2 Dynamic load balancing

To achieve an efficient parallel simulation, it is necessary to provide a dynamic load balancing scheme since it is very difficult to predict the load evolution.

Initially, each processor is responsible for a group of equal size and it seems to be fair. The imbalance appearing during the simulation is mainly due to the neutralization of agents which are not going to be taken into account for the rest of the simulation.

It is thus our responsibility to correct this imbalance during the execution time. The frequency of this correction is a factor to be addressed since it can influence the global performance of the parallel execution.

At each iteration, we can choose to only evaluate the number of active agents. During a load balancing step, it is possible to try to globally correct the imbalance (which can be very cost effective) or to make a local correction if the underlying communication network is known.

It is well established that dynamic load balancing induces an overhead for both architectures (message passing or CC-NUMA) in message exchange or in cache management. It is thus difficult to answer the whole set of questions in general (local or global load balancing, frequency,...).

In this part of the paper, we have clearly showed that an agent based parallelization is preferable, especially when the splitting between the data management and the calculations is desired.

This is the reason why, in the last part, we will focus on an implementation in shared memory using an agent based approach for a CC-NUMA architecture.

4. Developing an efficient shared memory solution

Good experience in the use of OpenMP has already been gained by some members of the research team in the area of irregular applications [6, 3].

In order to accomplish this goal, we will use the OpenMP library which offers many advantages, including relative simplicity in its use, good support at both the industrial and research levels [2].

4.1. OpenMP overview

OpenMP gives the opportunity to write a parallel program in two ways:

- *Parallel loops*: this solution is the easiest. In fact, this approach is very popular when the programmer wishes to extend an existing sequential code to a parallel code. If a parallel section is defined, OpenMP will create a group of threads to execute the associated iterations in parallel.

- *Parallel sections*: this solution requires from the programmer a good expertise in parallel programming. The underlying model applied is near to the MIMD

one (Multiple Instruction Multiple Data). When the programmer defines a parallel section, OpenMP create a thread to execute the associated code. So a parallel program is a collection of explicit parallel sections.

OpenMP also provides some mechanisms to manage critical section stored in its library. Mainly, a new type (`omp_lock_t`) is defined and two functions are provided : `omp_set_lock` to set the lock, and `omp_unset_lock` to free the lock. Before threads can use it, lock must be initialized by a call to the specific function `omp_init_lock`.

At this point, the features of OpenMP which have been described are sufficient to parallelize RMCSim.

4.2. Parallel RMCSim

Actually, the parallel RMCSim is developed using the parallel loop approach. The main reason of this choice is the benefit of the reuse of existing sequential C++ code.

As already mentioned, the combat component will be added subsequently, since these are less costly in time.

The battlefield (represented by a grid) will be placed in shared memory and the agent's displacement will be computed independently and in parallel.

The reader may remember we have to deal with two difficulties concerning the management of the concurrent access write on the grid and the management of the load.

To solve the first difficulty, we can use the OpenMP locks. The programmer has to choose how many locks he wants to define for the parallel application. If only one lock is defined, it will be easy for OpenMP to manage it but the performance will be very poor because of a nearly sequential execution of the loop. At the opposite, if we define as many locks as the number of cells making up the grid, OpenMP may have some difficulties to manage all of them in an efficient manner. The solution can be found between this two extreme propositions : for example, we can define a lock for each row of the grid. It should be a good compromise between the lock cost overhead and the parallel gain.

For shortness, we will assume already defined C++ classes related to the management of the grid (referred in the following as *Grids*), of the agents (*Agents*) and of the agent position on the grid (*Positions*).

Here is a C++ pseudo code using OpenMP to give an idea how the work is being done.

The member function **computeAttractiveness** from the Agents class is the computation expensive one : it has to compute the attractiveness of each neighboring cells for the agent under consideration and to order them after that. The result of the member function **bestPosition** is the best move for the current agent according to the attractiveness of his neighborhood. The result of an other call to it will be the following best cell according to this criteria. In this way the processor will try to move the agent on the best free cell using locks to assure the correctness of the writing process.

```
parallelMoveSimulation(Grids grid, Agents agentTab,
int nbAgent ) {
 int iteration, a;
 Positions position ;
 bool endOfMove ;
 for(iteration = 0; iteration < n; iteration + +) {
  randomOrder(agentTab) ;
  #pragma omp for private(position) shared(grid, agentTab)
  for(a = 0;a < nbAgent;a + +) { /* parallel loop */
   AgentTab[a].computeAttractiveness();
   endOfMove = false ;
   while(!endOfMove) {
    position = agentTab[a].bestPosition() ;
    omp_set_lock(position.lock()) ; /* critical section */
    if(grid.free(position)) {
     agentTab[a].move(position) ;
     endOfMove=true ;
    } /* end if */
    omp_unset_lock(position.lock()) ; /* end of critical section */
   } /* end while */
  } /* end parallel for */
 } /* end for */
}
```

The load balance will be done automatically by OpenMP. The programmer can help it by providing a schedule clause on the parallel loop. The acceptable values for this clause are : *static* or *dynamic* (there exist two other derived values which are not giving to be discussed here).

If the value used is *static*, the iterations are divided into chunks and assigned at the compilation time to the threads. This solution is not suitable for RMCSim since we have already shown how an irregular application it is.

If the schedule clause is set to *dynamic*, the iterations are assigned at the execution time and it seems to be more accurate for our application because OpenMP will in this way correct the imbalance factor. But, we should be careful about the overhead being introduced this way.

Unfortunately at this time, we cannot provide any experimental results since parallel RMCSim is actually under development. The target parallel machines are Sun Fire 6800[1] (24 processors) and SGI Origin'3800[2] (256 processors) using the KAI[3] and SGI OpenMP compilers. The first results should be available within the next six months. Interesting readers should contact directly the authors.

[1] `http://www.sun.com/servers/midrange/sunfire6800/`

[2] `http://www.sgi.com/origin/3000/3800.html`

[3] `http://developer.intel.com/software/products/trans/kai/`

Concerning the parallel section approach, it will be investigated when the parallel loop experiments will be completed. This solution seems to be very interesting since the programmer can explicitly manage the load: therefore it is possible to develop application specific dynamic load balancing scheme. The authors plan to take advantage of the schemes they have already develop in an other context [8].

5. Concluding remarks and perspectives

To conclude this paper, we would like to outline the benefits of the shared memory solution for RMCSim:

- the shared memory paradigm is easier to program than the message passing one;
- the computation power of parallel machine allows us to analyze more complex and larger battlefield.

Even a simple model for the agent's personality vector and simple meta-rules yields interesting results. However given the need for a more realistic behavior of the agents evolving in complex battlefield conditions will require a more sophisticated model. Thanks to the parallel RMCSim, we plan to investigate the applicability of various domains to the creation of more interesting and powerful agent even if they are much more time consumer.

Amongst these we will look at fuzzy logic in computing the next state of agents as well as for the Meta-rules, and :

- the use of Constraint Logic Programming;
- optimization algorithms for personality evolution.

Acknowledgments

The authors wish to thank Dr. P. Allard, Dr. Y. Liang and M. Lemaire. Drs. P. Allard and Y. Liang are both involved in the RMCSim Project. M. Lemaire has been involved in the development of the C++ Simulator during a train course in 2001.

References

[1] S. Brunett and T. Gottschalk. An architecture for large modsaf simulations using scalable parallel processors. In *Spring Simulation Interoperability Workshop*, 1998.

[2] R. Chandra, L. Dagum, D. Khor, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, Californie (USA), 2001.

[3] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. Parallel implementation of an ant colony optimization methaheuristic with openmp. In *proceedings of the third european workshop on OpenMP (EWOMP'01)*, Barcelona, Spain, Sept. 2001.

[4] J. Fugère and Y. Liang. A parallel analytical solution of stochastic combat. In *High Performance Computing Systems and Applications, HPCS'99*. Kluwer Academic Publishers, Kingston, Ontario, Canada, June 1999.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing*. The MIT Press, 1994.

[6] Z. Habbas, M. Krajecki, and D. Singer. Shared memory implementation of csp resolution. *HLPP2001: International workshop on High-level parallel programming and applications in Parallel Processing Letters*, to appear, 2001.

[7] A. Ilachinski. Einstein: An artificial-life laboratory for exploiting self-organized, emergent behavior in land combat. Technical report, Center for Naval Analyses, Alexandria, Virginia (USA), sep 2000. `http://www.cna.org/isaac`.

[8] M. Krajecki. An object oriented environment to manage the parallelism of the FIIT applications. In V. Malyshkin, editor, *Parallel Computing Technologies, 5th International Conference, PaCT-99*, volume 1662 of *Lecture Notes in Computer Science*, pages 229–234. Springer-Verlag, St. Petersburg, Russia, Sept. 1999.

[9] MPI Forum. Mpi-2: Extensions to the message-passing interface. Technical report, University of Tennessee, 1997. `http://www.mpi-forum.org/docs`.

[10] OpenMP ARB. Openmp c and c++ application program interface. Technical report, OpenMP Architecture Review Board, 1998. `http://www.openmp.org/specs/`.