Diseño Inteligente de Redes Inalámbricas de Comunicación
Conserjería de Innovación, Ciencia y Empresa
P07-TIC-03044
http://diricom.lcc.uma.es

Málaga, 12 de Diciembre de 2009

Informe Ejecutivo

TÍTULO: SOFT-3.0-2009: Model Checking en Entornos Grid

RESUMEN: En este documento hacemos una propuesta para realizar *model checking* en entornos de computación grid. Los algoritmos para hacer *model checking* requieren una gran cantidad de memoria. Por ese motivo, existen propuestas de ejecución distribuida en las que la memoria disponible para la búsqueda de errores es la suma de la memoria de cada máquina. No obstante, este enfoque no es extensible a los entornos grid, que están principalmente pensados para descomponer un problema con grandes exigencias computacionales en tareas independientes. Nuestra propuesta parte de la idea de tranformar los altos requisitos de memoria de los algoritmos de búsqueda en altos requisitos computacionales. Realizamos un modelado teórico de nuestro algoritmo que permite estimar el tiempo medio de ejecución y mostramos algunos resultados preliminares.

OBJETIVOS:

1. Proponer un algoritmo para realizar *model checking* en entornos de computación grid.
2. Estudiar matemáticamente la ejecución para optimizar los parámetros y obtener estimaciones del tiempo de ejecución requerido por el algoritmo.

CONCLUSIONES:

1. Los resultados preliminares muestran que la generación de números aleatorias puede ser relevante en nuestra propuesta.
2. El modelo matemático nos permite determinar el valor óptimo para el único parámetro del algoritmo.
3. La probabilidad de que el error no sea encontrado decrece exponencialmente con el tiempo.

RELACIÓN CON ENTREGABLES: PRE: SOFTW-1.0-2008 (lectura necesaria)

DIRICOM

Diseño Inteligente de Redes Inalámbricas de Comunicación
Conserjería de Innovación, Ciencia y Empresa
P07-TIC-03044
http://diricom.lcc.uma.es

Málaga, December 12th, 2009

Executive Summary

TITLE: SOFT-3.0-2009: Model Checking in Grid Computing Environments

ABSTRACT: In this deliverable we propose a model checking algorithm for grid computing environments. Model checking algorithms require a large amount of memory. For this reason, some distributed algorithms exist in which the available memory is the sum of the memory of all the machines in the cluster of computers. However, this approach cannot be extended to grid computing environments, which are specialized in solving computationally costly problems by decomposing them into independent tasks. Our proposal is based on the idea of transforming the high memory requirements of model checking algorithms into high computing requirements. We model our model checking algorithm to estimate the average execution time and we show some preliminary results.

GOALS:

1. Propose an algorithm for model checking in grid computing environments.
2. Study in a mathematical way the execution of the algorithm to optimize the parameters and obtain some estimations of the execution time required.

CONCLUSIONS:

1. The preliminary results show that the random number generator can play an important role in our proposal.
2. The mathematical model allows us to determine the optimal value for the only parameter of the algorithm.
3. The probability of an error to keep uncover decreases in an exponential way as the time progresses.

RELATION WITH DELIVERABLES: PRE: SOFTW-1.0-2008 (mandatory reading)

Model Checking in Grid Computing Environments

DIRICOM

December 12^{th}, 2009

1 Introduction

Model checking [4] is a well-known and fully automatic formal method that can check properties of software systems. In model checking, all the possible program states are analyzed (in an explicit or implicit way) in order to prove (or refute) that the program satisfies a given property such as absence of deadlocks or starvation. Some other more general properties can be specified using a temporal logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). One of the best known explicit model checkers is SPIN [7], which takes a software model codified in Promela and a property specified in LTL as inputs. SPIN transforms the model and the negation of the LTL formula into Büchi automata in order to perform the synchronous product of them. The resulting product automaton is explored to search for a cycle of states containing an accepting state reachable from the initial state. If such a cycle is found, then there exists at least one execution of the system not fulfilling the LTL property (see [6] for more details). If such kind of cycle does not exist then the system fulfills the property and the verification ends with success. Java PathFinder (JPF) is another model checker that works on Java bytecodes. Out of the box, JPF can only check deadlocks and unhandled exceptions. However, it is possible to extend the model checker by specifying user-defined properties.

The amount of states to explore in model checking is very high even in the case of small systems, and it increases exponentially with the size of the model. This fact is known as *the state explosion problem* and limits the size of the model that a model checker can verify. This limit is reached when it is not able to explore more states due to the absence of free memory. Several techniques exist to alleviate this problem. They reduce the amount of memory required for the search by following different approaches. On one hand, there are techniques which reduce the number of states to explore, such as partial order reduction [10] and symmetry reduction [9]. On the other hand, we find techniques that reduce the memory required for storing one state, such as state compression, minimal automaton representation of reachable states, and bitstate hashing [6]. Symbolic model checking [3] is another very popular alternative to the explicit state model checking that can reduce the amount of memory required for the verification by means of a compact representation for set of states. However, although it is possible to check larger models with symbolic model checking, this technique also suffers from state explosion.

Instead of reducing the amount of memory required by model checking, other approach tries to gather a large amount of memory by combining different machines. This leads to the Distributed Model Checking [2]. In distributed model checking, the state space is divided into several machines which are connected by a network. Each machine explores one part of the search space and the total memory available for the search is the sum of the memory of the machines. The main problem of this approach is the communication time. In effect, the machines have to communicate to each other in order to coordinate the search in some way. If the communication is too frequent, the time required for the search could significantly increase. For this reason, distributed model checking is usually performed in specialized clusters of machines with high speed networks (fiber optics) [1].

Nowadays, computer grids are gaining importance in the computer science community. They are composed of hundreds or thousands of machines connected by a network [5]. The base idea in grid computing is to use the computational power of all the machines to solve a problem. Computer grids have been used to solve combinatorial optimization problems [11] and are the focus of several research projects around the world. Computer grids have several advantages with respect to specialized clusters of machines. First, they are cheap. Grids include non-dedicated machines, that is, machines that are used for other tasks during some periods of time. For example, a researcher could cease his/her machine to a grid when s/he does not need it (nights, weekends, holidays, etc.). Second, they

do not require a specialized hardware to work. The only hardware requirement is a network that connects all the machines. Even in this case, the network could not be available all the time. Third, it is possible to join a large number of machines.

However, grid computing also poses some challenges that have to be solved. For example, we have to deal with the heterogeneity of the machines. Different machines could have a different amount of memory, a different processor speed or even a different architecture. The machines enter and go out of the grid in a dynamic way and this means that a grid is dynamic and we have to deal to this feature.

In this deliverable we propose a way of using grid computing environments for model checking. We theoretically analyze the performance of the algorithm and make some experiments as a proof of concept.

2 The Proposal

As a first step in the design of a proposal for using grid computing in model checking, it is natural to review the distributed model checking approach. However, distributed model checking, as used today is not practical in grid computing environments for several reasons. First, in distributed model checking if a machine goes out of the cluster, the search is interrupted, since part of the explored state space disappears. Second, the high communication overhead among the machines in the cluster reduces the performance of the search when it is accomplished in a grid computing environment.

Grid computing is specially efficient when the problem is divided into independent runs, since each of these runs is sent to one machine to be executed. When the tasks are finished the results are sent back to the user. That is, grid computing environments are good to solve computationally costly problems. But, model checking is a memory costly problem. Thus, we search for a way of decomposing model checking into several independent tasks with less memory requirements than the original problem. This is not easy, since the known model checking algorithms perform a search in a graph and this graph is built on the fly as the search progresses. For this reason, it is not easy to define different regions in the graph to explore independently, since the graph in unknown at the beginning.

Our proposal is based on a random exploration of the search space. We assume that, in general, we do not know the states graph. Under this assumption we cannot determine regions in which the different tasks should focus the search. Instead, each task searches the graph in a different and random way. Assuming that there is at least an error state (accepting state) in the model and that the error trail to reach the accepting state fits into the memory of a machine, it should exist a random exploration of the search space such that the error can be found with the memory of a single machine. Thus, if the independent tasks explore the graph in a random way the error should be found sooner or later. In the following we mathematically model the process to find an estimation of the time required to find the error in the grid computing environment.

We use a master/slave approach in which the slaves perform the tasks and the master just gather the results and send more tasks to the grid environment while an error is not found in the model. Our model is based in the following assumptions:

- Each slave performs k independent random searches in the state space. Each search is executed until a predefined memory limit m is reached (usually, the memory of the machine). We assume that there is a given (usually unknown) probability for each search to find an error. This probability depends on the memory m and is denoted by $p(m)$.
- We have a fixed number of machines n in the grid.
- During the execution of one task in one machine an unexpected event could destroy the work of the task.
- The execution time of a search is $t_c(m)$ which depends on the memory available.
- The communication time is t_{com}, which is the time required to send the data to the slave before the execution plus the time required to send the results back to the master process.

3 Mathematical Model

Our goal in this section is to obtain a formula for the probability of finding a solution in a time τ or less. Let us first start by computing the number of searches performed in the time τ. Each task runs k independent random searches, so it requires a time $kt_c(m) + t_{com}$. In a time τ the number of completed tasks in the absence of unexpected events n_c should be

$$n_c = \frac{\tau}{kt_c(m) + t_{com}} \tag{1}$$

However, the task could be aborted by means of an unexpected event, so we have to model the impact of unexpected events in the tasks performance. The probability of a given task to run completely with the presence of unexpected events follows an exponential law in the duration of the task. That is, if the events are randomly distributed, the probability that no event occurs in t_1+t_2, $p(t_1+t_2)$ is $p(t_1)p(t_2)$, the probability that no event occurs in the first t_1 period multiplied by the probability that no event occurs in the second t_2 period. Thus, $p(t) = e^{-\lambda t}$ where $1/\lambda$ is the average time between two unexpected events. We assume that the λ parameter is dependent on the grid and does not change with time or load. Since one task requires $kt_c(m) + t_{com}$ time, the probability of surviving under the unexpected events is $e^{-\lambda(kt_c(m)+t_{com})}$ and the number of completed tasks at time τ from the beginning is

$$n'_c = \frac{\tau}{kt_c(m) + t_{com}} e^{-\lambda(kt_c(m)+t_{com})} \tag{2}$$

The total number of random searches is $n_s = knn'_c$ and the probability of finding an error trail in time τ or less is $1 - (1 - p(m))^{n_s}$. This probability can also be written as:

$$Prob(\tau) = 1 - (1 - p(m))^{\frac{kn\tau e^{-\lambda(kt_c(m)+t_{com})}}{kt_c(m)+t_{com}}} \tag{3}$$

We can compute the expected time of the algorithm in the following way:

$$\overline{\tau} = \int_0^\infty \tau \frac{d\ Prob}{d\tau} d\tau = [\tau Prob(\tau)]_0^\infty - \int_0^\infty Prob(\tau) d\tau = \frac{-(kt_c(m) + t_{com})}{kne^{-\lambda(kt_c(m)+t_{com})} ln(1 - p(m))} \tag{4}$$

If we have machines with different memory we can divide the total number of machines n into groups. Let us call n_m the number of machines with memory m. Then the new probability can be computed as:

$$Prob(\tau) = 1 - \prod_m (1 - p(m))^{\frac{kn_m\tau e^{-\lambda(kt_c(m)+t_{com})}}{kt_c(m)+t_{com}}} \tag{5}$$

In our master/slave algorithm we can only change the number of searches performed by one slave, k. It is natural to ask what is the best value for this parameter. We want to maximize the probability of finding an error. In this case we have just to derive $Prob(\tau)$ with respect to k and solve the equation $\partial Prob(\tau)/\partial k = 0$. The result for k (assuming that all the machines have the same memory) is:

$$k = \frac{\sqrt{t_{com}(\frac{4}{\lambda} + t_{com})} - t_{com}}{2t_c(m)} \tag{6}$$

Thus, if we know all the details of the grid and the model checking searches, we can set the optimal value for k. The shape of $Prob(\tau)$ is the one shown in Figure 1.

4 Experiments

In a first proof of concept we run the master/slave approach using a randomized Nested Depth First Search algorithm in the slaves. The algorithm was implemented based on Spin 5.4.3. The model used for the experiments was a variant of the 32-bit generator model found in [8]. We added an integer array to the model just to artificially increase the size required by one state of the model in memory, thus reducing the number of states that fit in the memory of

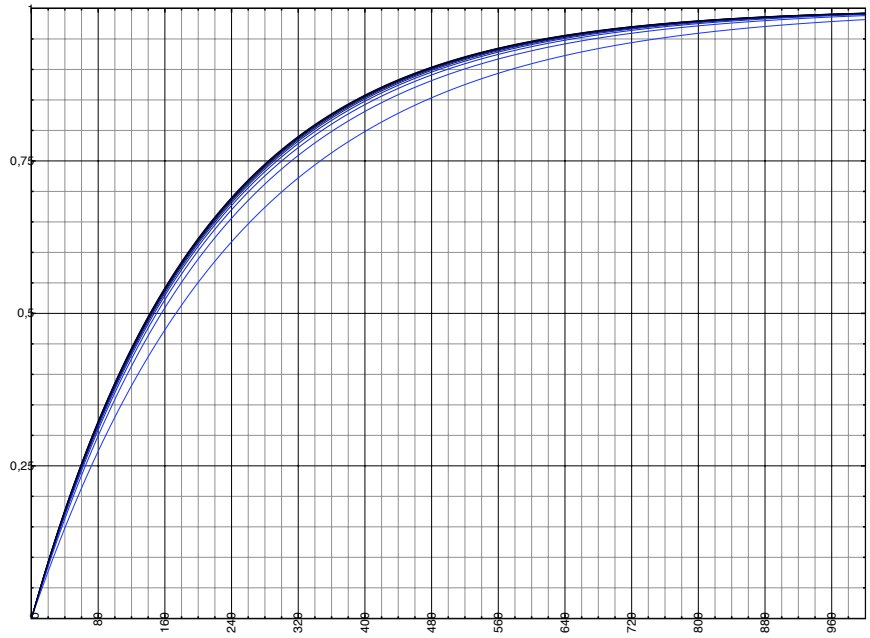

Figure 1: Probability of finding an error trail against time for different values of k.

one machine. We denote by N the size of the array. We also change the number of bits of the flag to B. A simple computation shows that the probability of finding the target state in one single run of the randomized NDFS is

$$p(m) = \frac{m}{(2^{B+1})(aN+b)} \tag{7}$$

where a and b are constants. Then, the average number of runs R required to find the target state is

$$E[R] = \frac{1}{p} \propto 2^B(aN+b) \tag{8}$$

We run the master/slave algorithm in a grid computing environment (around 300 machines) using different values for N and B. First, we fixed $B = 32$ and change N from 31 to 217 in steps of 31. Then, we fixed $N = 248$ and change B from 26 to 32 in steps of 2. For each combination of parameters we performed 100 independent runs. In Table 1 we show the average values.

N	R	B	R
31	3702.36	26	370.86
62	7109.29	28	1622.82
93	8800.61	30	8157.02
124	13417.70	32	26890.05
155	18186.55		
186	19144.38		
217	23618.79		
248	26890.05		

Table 1: Results for the master/slave approach in grid computing environments.

In Figure 2 we show the same results and we can observe a linear trend when we plot R against N and an exponential trend when we plot R against B, as expected. However, when $B = 32$ the empirical dependence of R

with N significantly deviate from the straight line (recall that we are plotting the average over 100 independent runs). We think that the reason for that can be found in the random number generator. We are using seeds with 32 bits, but the number of ways in which the search can be performed is higher than 2^{32}. Then, not all the possible ways of searching the state space are explored and the results could be biased. Thus, the random number generator plays an important role in this algorithm.

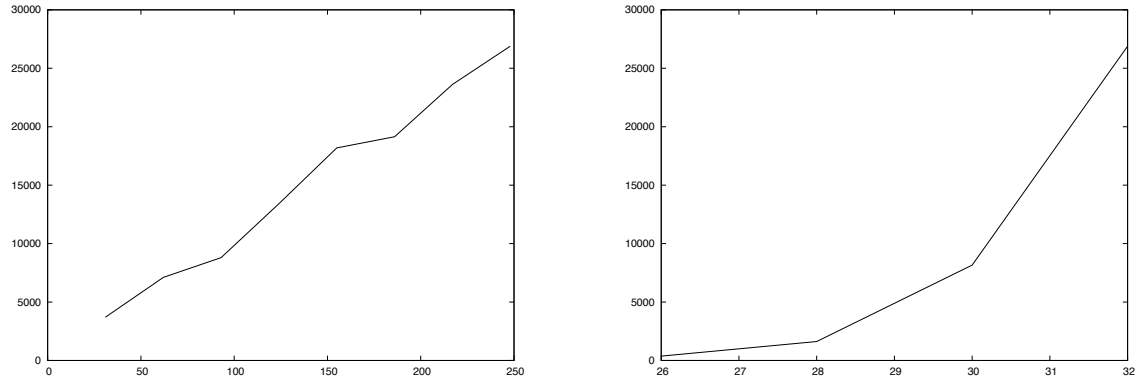

Figure 2: The results obtained for the master/slave approach. We can observe a linear trend in the left picture and an exponential one in the right picture.

5 Conclusions

In this deliverable we propose a way of using grid computing environments for model checking. In addition to the proposal we have modeled the performance of the approach in a mathematical way. We have shown some preliminary results using a random NDFS algorithm on a toy model. As future work we plan to empirically evaluate the approach using realistic models and we plan to change the random NDFS by metaheuristic algorithms.

References

[1] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *Formal Methods for Components and Objects*, number 4111 in LNCS, pages 259–279, November 2005.

[2] Jiri Barnat, Lubos Brim, and Jitka St&rcirc;íbrná. Distributed ltl model-checking in spin. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 200–216, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[3] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.

[4] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1999.

[6] Gerald J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[7] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[8] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 134–143, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Alberto Lluch Lafuente. Symmetry Reduction and Heuristic Search for Error Detection in Model Checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[10] Alberto Lluch-Lafuente, Stefan Leue, and Stefan Edelkamp. Partial Order Reduction in Directed Model Checking. In *9th International SPIN Workshop on Model Checking Software*, Grenoble, April 2002. Springer.

[11] Lakhdar Loukil, Malika Mehdi, Nouredine Melab, El-Ghazali Talbi, and Pascal Bouvry. A parallel hybrid genetic algorithm-simulated annealing for solving q3ap on computational grid. In *IPDPS*, pages 1–8, 2009.