Using UML State Diagrams for Modelling the Performance of Parallel Programs

Uso de Diagramas de Estado UML para la Modelación del Desempeño de Programas Paralelos

Jorge Ortega Arjona
Departamento de Matemáticas, Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Article received on June 28, 2007; accepted on October 24,2007

Abstract
There are many possibilities to design a parallel program in order to obtain the best performance possible. The selection of a program structure, as an organisation of processes, impacts on the performance to be achieved, and depends on the problem to be solved. Now, in order to select a program structure as the best in terms of performance, the software designer requires performance modelling techniques to evaluate different alternatives. If the structure of the parallel program can be modelled as a set of interacting processes, described in terms of UML State Diagrams, this paper presents a performance modelling to estimate the average execution time of a parallel program. Performance modelling is achieved by calculating the average execution time of a parallel program, described as a set of processes which run with deterministically and exponentially distributed execution times.
Keywords: .Performance modelling, parallel program, UML State Diagram

Resumen
Hay muchas posibilidades para diseñar un programa paralelo a fin de obtener el mejor desempeño posible. La selección de una estructura del programa, así como una organización de procesos, impacta sobre el desempeño a lograrse, y depende del problema a resolver. Ahora bien, para seleccionar una estructura del programa como la mejor en términos de desempeño, el diseñador de software requiere de técnicas de modelación para evaluar diferentes opciones. Si la estructura de un programa paralelo puede modelarse como un conjunto de procesos interactivos, descritos en términos de Diagramas de Estado de UML, este artículo presenta una modelación para estimar el tiempo de ejecución promedio de un programa paralelo, descrito como un conjunto de procesos que corren en tiempos de ejecución con distribuiciones determinística y exponencial.
Palabras clave: Modelación de desempeño, programa paralelo, Diagrama de Estado de UML

1 Introduction

During the last few years, parallel computing has been proposed as a potential solution for the increasingly complex problems in several research and development areas like quantum chemistry, fluid mechanics, weather forecasting, and others. Designing and programming parallel programs requires an extraordinary effort of the software designer, who has to balance between the complexity of the parallel implementation and the performance expectations. At the initial stage of parallel software development, the software designer counts only with the information of the problem to solve, the available parallel hardware platform, and the programming language to use. Based solely on this information and on the software designer experience, a parallel program is commonly designed and implemented. But, as parallel programming represents a high cost in terms of development effort and time, it would be an advantage to count with quantifiable information before further steps are taken during design and implementation. Hence, the software designer could be able to select a program structure or another, regarding the parallelism contained in the problem at hand. In general, a software designer does not know in advance which of the various parallel structures, described as a set of interacting processes, would have the desired execution time on a given parallel platform. Thus, the software designer faces two alternatives:

1. The software designer can implement the various parallel structures. The parallel hardware platform is available, so the implementations are possible. Nevertheless, this approach requires a lot of effort and time to test every possible solution, and therefore, it tends to be very expensive in terms of both, time and effort.

2. Instead, the software designer can model the various parallel structures, and try to find the best one by evaluating the models, using performance simulation models.

This paper presents an approach, based on the second alternative, to obtaining an *average runtime* of a parallel program. The basic assumption is that the whole parallel program consists of processes whose states are combined to obtain an overall state of the parallel program. The runtimes of the states are modelled by a random variable and its distribution function. Moreover, the model is based on the dependency between the states of the processes. This is described using UML State Diagrams (Booch, *et al.*, 1998; Fowler & Scott, 1997). Figure 1 shows the UML State Diagram of a simple parallel program consisting of two processes, A and B. Process A has two states a_1 and a_2, whereas process B has two states b_1 and b_2. The diagram indicates that processes A and B execute simultaneously in states a_1 and b_1. Process A can get to state a_2 only after finishing state a_1. However, process B can only get to state b_2 when both processes A and B have respectively finished states a_1 and b_1.

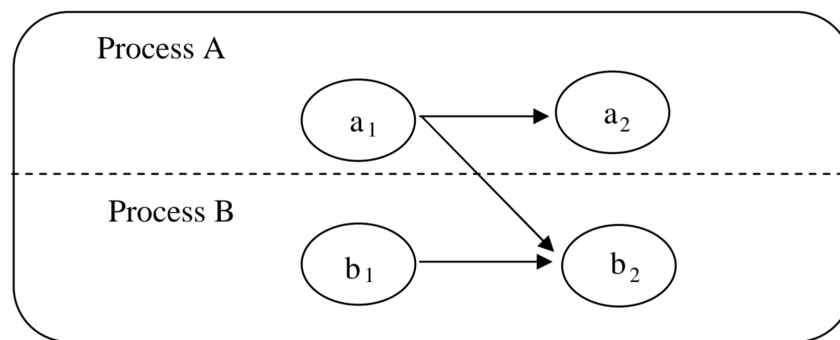

Fig. 1. UML State Diagram of a simple parallel program

The analysis of this kind of diagrams tends to be very complex when increasing the number of parallel processes and their states. However, if it could be found an equivalent state diagram which considers the states of the parallel program as a single entity based on the various possible state combinations of its processes, and also, it could be measured the runtime distributions of all processes during such states, then it would be possible to compute the distribution of the overall parallel program runtime. Moreover, in order to obtain more realistic models, it is necessary to model the behaviour within a state using distribution functions that approximate to measured empirical distribution functions.

The objective of this paper is to present an analysis method which can be applied to compute the distribution of the overall parallel program runtime based on an equivalent state diagram of the parallel program and measured data about the runtime distributions of the parallel processes. Section 2 presents some related work in the areas of Reliability Engineering, Performance Engineering, and Parallel Programming. Section 3 explains how to compute the average runtime of a program which consists of processes with exponentially distributed runtime variables (Kleinrock, 1975). Section 4 presents the analysis method that allows to approximate the overall parallel program runtime by modelling the processes' runtimes using exponentially and deterministically distributed random variables. Finally, Section 5 presents the execution of simulation models that solve the Heat Equation problem, as a case study to validate the method.

2 Related Work

Several other similar approaches have been developed for modelling the performance and reliability of software systems, whether these make use of reduction of state diagrams for Reliability Engineering (Billington & Allan, 1992), make use of UML diagrams for Performance Engineering (Pooley & King, 1999), or are used for basic parallel programming (Lui *et al.*, 1998).

Billington and Allan (1992) make use of space state diagrams and network modelling techniques for evaluating the reliability of a system, a model and/or a component. Mostly, these diagrams are used to represent failure-repair processes. The essence is to derive a set of equations suitable for series-parallel systems: *(a)* in series system, all components must operate for system success, and *(b)* for a parallel system, one component need to work for system success. So, these equations are used to deduce the probability of the system to be in down state or up state, by reducing the different probabilities of residing in each of the system states, deriving the approximate state probabilities for each model in a series-parallel system.

In a similar way, here we make use of UML state diagrams as space state diagrams for depicting the parallel components' states. However, the way in which the state probabilities are reduced is different: states are not reduced using equations of the series or parallel probability of residence of each component in each state, but the state of the system is globally considered by considering the precedence of states, and hence, the system state is modelled as a single entity, obtaining an equivalent UML diagram which takes into consideration only precedence of states to model the system's performance.

Pooley and King (1999) present a thorough revision of UML, and its potential to be used in Performance Engineering. They provide a brief but description of each UML diagram as a potential modelling tool for performance. Nevertheless, they only shallowly describe how to exploit use case diagrams, implementation diagrams, sequence diagrams, collaboration diagrams, activity diagrams, and state diagrams. They complement these UML diagrams with queuing models in order to derive performance models. Unfortunately, they do not deeper into further describing any of the UML diagrams for modelling performance. In the particular case of UML state diagrams, they ultimately mention that this approach "*requires a lot of work*", providing no further information about it.

In the present paper, we exclusively focus on UML state diagrams to reduce the states of a parallel system, obtaining an equivalent state diagram, which is actually used for performance modelling. In fact, the treatment given here goes beyond simply considering the state of the parallel components, deeper into an analysis of the states within the diagram and their reduction into a single equivalent UML diagram. The equivalent states of this UML diagram are modelled by deterministically and exponentially distributed variables.

Lui *et al.* (1998) perhaps provide the closest approach to the one presented here. They also make use of space state diagrams and series-parallel reduction, as well as exponentially distributed variables, for deriving performance models for simple fork-join parallel programs executing on a multiprocessor environment. Nevertheless, they do not take into consideration another or more realistic parallel program structure. Fork-join programs are common, but they tend to neglect the communication between parallel components. Hence, these programs do not cover other different types of parallel systems, such as Communicating Sequential Elements (Ortega-Arjona, 2000), which highly depends on the communication between parallel components.

In this paper, UML state diagrams are derived directly by the precedence relations of the parallel program structure. Thus, such state diagrams reflect the behaviour of the parallel program depending on both, computation (within components) and communication (which affects the precedence of computations). So, depending on the organization of parallel components, different state diagrams are obtained. These diagrams are reduced regarding the precedence of states into a simpler model of computation, which is actually used for performance modelling by taking into consideration variations of the time consumed by the states of the parallel program as a whole. The following sections explain how execution times are modelled using deterministically and exponentially distributed variables.

3 Using Exponentially Distributed Variables for Modelling Execution Times

In order to analyse a state diagram with only exponentially distributed runtimes, it can be used the state space method (Thomasian & Bay, 1986). In this method, every state s of the state space is characterised by the set of processes $P(s) = \{P_1, \cdots, P_n\}$ of the parallel program which execute simultaneously in state s. The runtime t_i of process P_i is an exponentially distributed random variable with parameter λ_i. The density and distribution functions of the runtime are respectively $f_i(t)$ and $F_i(t)$. The whole system changes from state s to state s'_i if process P_i is

the first to change state. The new set of processes $P(s'_i)$ running in the state s'_i results from $P(s) \setminus \{P_i\}$ joined with the set of state of processes $S_s(P_i)$ which can start if P_i is the first to change state in s (Figure 2):

$$P(s'_i) = (P(s) \setminus \{P_i\}) \cup S_s(P_i)$$

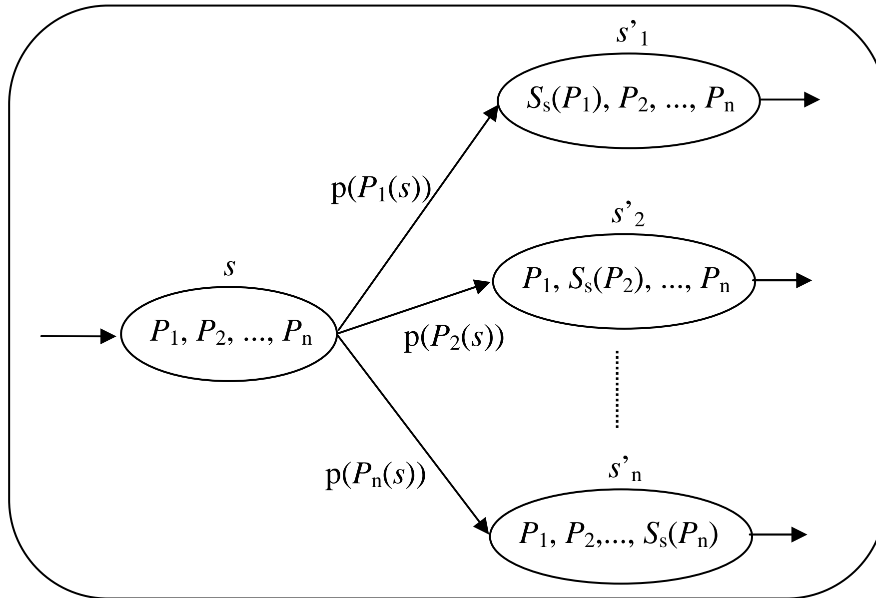

Fig. 2. UML State Diagram of a stage of the equivalent state diagram as part of the state space

Considering the example in Figure 1, the UML State Diagram for the equivalent state diagram is shown in Figure 3.

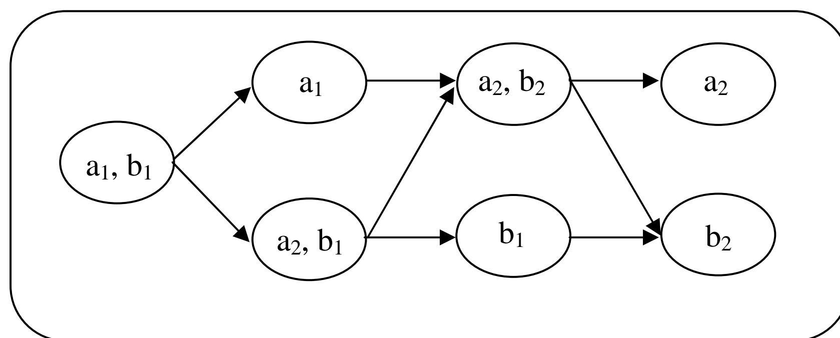

Fig.3. Equivalent UML State Diagram of the state space for the example in Figure 1

In order to calculate the average runtime of the whole parallel program, it is required to define the average service time in state s and the probability of P_i being the first process which changes state in s. Therefore, the average runtime of the whole parallel program ($E[S]$) can be recursively calculated using the following expression:

$$E[S] = \sum_{P_i \in P(S)} \mathrm{p}(P_i(S))(E[P_i(S)] + E[S'_i])$$

where:

$E[P_i(s)]$ is the average service time of process P_i in state s under the condition that P_i is the first process to change state;

$\mathrm{p}(P_i(s))$ is the probability that P_i is the first process to change state in s; and

S is the first state of the whole parallel program in the state space.

As the exponential distribution has the memoryless property (Kleinrock, 1975), the behaviour of the parallel program in state s is independent of its history. Using this property, it is obtained for $\mathrm{p}(P_i(s))$ that:

$$\begin{aligned}\mathrm{p}(P_i(s)) &= \mathrm{p}\left(\bigwedge_{j=1 \wedge j \neq i}^{n} t_i < t_j\right) \\ &= \int_0^\infty f_i(t) \prod_{j=1 \wedge j \neq i}^{n} (1 - F_j(t))dt \\ &= \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}\end{aligned}$$

and for $E[P_i(s)]$ that:

$$\begin{aligned}E[P_i(s)] &= \int_0^\infty t \frac{\partial}{\partial t} \mathrm{p}\left(t_i \leq t \middle| \bigwedge_{j=1 \wedge j \neq i}^{n} t_i < t_j\right)dt \\ &= \int_0^\infty t \frac{\partial}{\partial t} \frac{\int_0^t f_i(t) \prod_{j=1 \wedge j \neq i}^{n} (1 - F_j(t))dt}{\int_0^\infty f_i(t) \prod_{j=1 \wedge j \neq i}^{n} (1 - F_j(t))dt} dt \\ &= \frac{1}{\sum_{i=1}^{n} \lambda_i}\end{aligned}$$

It is noticeable that $E[P_i(s)]$ is equal to the first moment of the distribution of the minimum $\min(t_1,...,t_n)$, and it is independent of the process which change state first. Only the branching probabilities $\mathrm{p}(P_i(s))$ depends on i.

4 Using Deterministically and Exponentially Distributed Variables for Modelling Execution Times

Modelling the runtime of a process in a particular state using only a simple exponentially distribution is not a very realistic approach for the process' real behaviour. Hence, the use of other arbitrary distribution functions are proposed here as part of the models, and thus, it could be possible to analyse the models by the phase method. Moreover, the use of phase type distributions, like the Erlang distribution (Kleinrock, 1975), may simplify the analysis.

In general, there are parallel programs with processes whose states have runtime distributions with a small variance. Here, it is proposed the use of Erlang-k distributions E_k (Kleinrock, 1975), which requires a high number of phases (k) for its computation. Nevertheless, models that use Erlang-k distributions tend to become intractable because of the so-called *state space explosion* (Thomasian & Bay, 1986). To avoid this problem, the number of phases must be reduced and finite. This can be done by approximating the E_k-distribution (with its first moment E and variance V) by a state with a deterministic phase with parameter d and an exponential phase with parameter λ (Kleinrock, 1975). Therefore, the number of phases of one node is reduced from k to two (Figure 4).

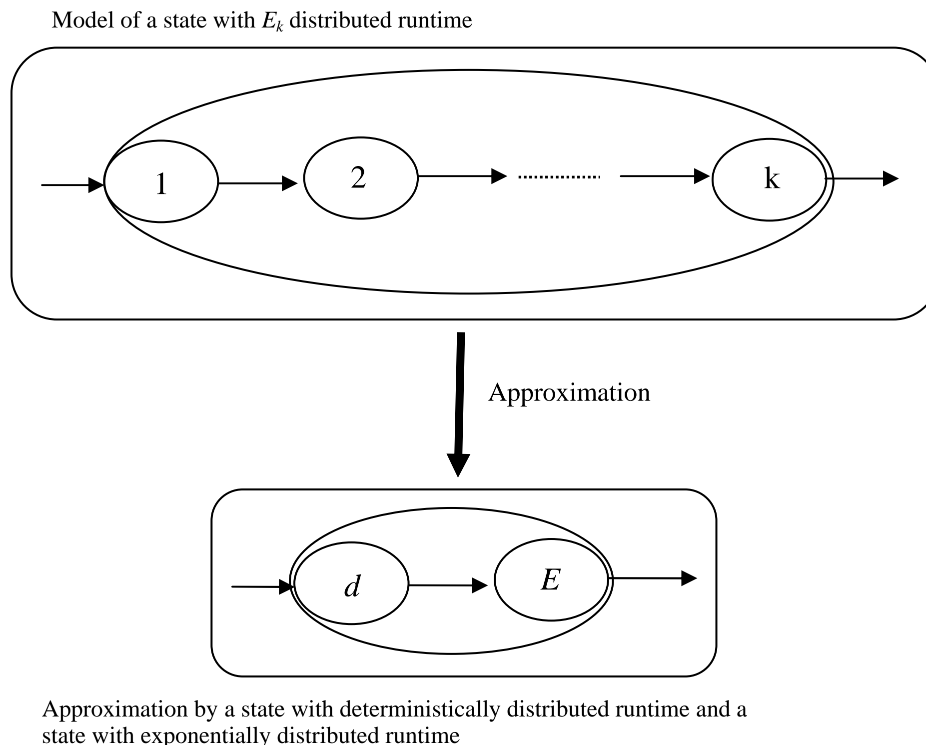

Fig. 4. Approximation of a state with E_k distributed runtime with two states with deterministically distributed runtime *d* and a state with exponentially distributed runtime *E*.

Approximating the runtime of the state of a process by a deterministic and an exponential phase implies that the modelled runtime always has the minimum of time d. This is a better model of the real behaviour than a simple exponential distribution function with a positive probability for all positive runtimes. Notice that for $\lambda = \sqrt{1/V}$ and

$d = E - \sqrt{V}$, the first two moments of the E_k-distribution and of the approximate distribution are the same. Moreover, a better approximation can be achieved by approximating with a deterministic distribution and an E_2-distribution with parameters $\lambda = \sqrt{2/V}$ and $d = E - \sqrt{2V}$. The second exponential phase causes a slower increase of the distribution function at time d. Thus, as the exponential phase increases, it tend to be close and closer to the original E_k-distribution.

Now, it is required to approximately analyse a state diagram consisting of states which model their runtime by deterministically and/or exponentially distributed random variables. Nevertheless, notice that the memoryless property of the exponentially distributed random variables is lost, since when introducing deterministically distributed variables, the state space method only allows an approximation to the exact value. Therefore, because the deterministically distributed variables do not accomplish the memoryless property, the behaviour in a particular state depends on all previous states, and since there are deterministically distributed phases running in s, they have been running from the start. Considering these dependencies, the modelling complexity increases to a point already untractable for small examples.

Hence, it is now required to approximately obtain the time in which the parallel program remains in state s'_i, considering the use of the approximation above. Let $P(s) = \{P_1, P_2, \cdots, P_k, P_{k+1}, \cdots, P_n\}$ be the set of processes which simultaneously execute in state s of the parallel program. Let us consider that the phases of a state of the process $P_j (1 \leq j \leq k)$ have deterministically distributed service times with parameter d_j. Now, $d = \min(d_1, d_2, \cdots, d_k)$ is the minimum of the deterministic phases and P_m the process have the shortest deterministic runtime in state s. Therefore, $d_m = d$.

For $k < j \leq n$ the service time is an exponentially distributed random variable with parameter λ_j. The running tasks in s'_i are obtained as $P(s'_i) = \{P'_1, P'_2, \cdots, P'_k, P_{k+1}, \cdots P_n\} \setminus \{P_i\} \bigcup S_s(P_i)$ with parameter $d'_j = d_j - E[P_i(s)]$ for $1 \leq j \leq k$ and λ_j for $j > k$.

It is noticeable that, for $j \leq k$, the random variable t'_j is approximated by a deterministically distributed variable with parameter d'_j. To obtain the branching probability and the expected remaining time in any state, let us make use of the Dirac function $\delta(t)$ (Kleinrock, 1975):

$$\delta(t) = \begin{cases} \infty, & if\ t = 0 \\ 0, & if\ t \neq 0 \end{cases}$$

$$\int_0^\infty \delta(t)dt = 1$$

Using this function and the previous definitions, three cases can be distinguished:

1. $1 \leq i \leq k \wedge i \neq m$. This means considering all deterministic phases except the shortest, and hence:

$$\mathrm{p}(P_i(s)) = 0$$

2. $i = m$. This means considering the shortest deterministic phase, and thus:

$$\begin{aligned}
\mathrm{p}(P_i(s)) &= \mathrm{p}\left(\bigwedge_{j=k+1}^{n} t_i < t_j \right) \\
&= \int_0^\infty f_i(t) \prod_{j=k+1}^{n} (1 - F_j(t)) dt \\
&= \int_0^\infty \delta(t-d)\, e^{-\sum_{j=k+1}^{n} \lambda_i t} dt \\
&= e^{-\sum_{j=k+1}^{n} \lambda_i t} \\
E[P_i(s)] &= d
\end{aligned}$$

3. $k < i \leq n$. This means considering the exponentially distributed phases, so:

$$\begin{aligned}
\mathrm{p}(P_i(s)) &= \mathrm{p}\left(t_i \leq t_m \wedge \bigwedge_{j=k+1 \wedge j \neq i}^{n} t_i \leq t_j \right) \\
&= \int_0^\infty f_i(t)(1 - F_m(t)) \prod_{j=k+1 \wedge j \neq i}^{n} (1 - F_j(t)) dt \\
&= \frac{\lambda_i}{\sum_{j=k+1}^{n} \lambda_j} \left(1 - e^{-\sum_{j=k+1}^{n} \lambda_j d} \right) \\
E[P_i(s)] &= \int_0^\infty t \frac{\partial}{\partial t} \mathrm{p}\left(t_i \leq t \middle| t_i < t_m \bigwedge_{j=k+1 \wedge j \neq i}^{n} t_i < t_j \right) dt \\
&= \frac{1 - \left(1 + \sum_{j=k+1}^{n} \lambda_i d \right) e^{-\sum_{j=k+1}^{n} \lambda_i d}}{\left(1 - e^{-\sum_{j=k+1}^{n} \lambda_i d} \right) \sum_{j=k+1}^{n} \lambda_i}
\end{aligned}$$

ISSN 1405-5546

5 Case Study – The Heat Equation Problem

The Heat Equation problem is to calculate the heat diffusion through a substrate, using a parallel program (Ortega-Arjona, 2000). Let us consider the simplest case, in which the Heat Equation is used to model the heat distribution on a one-dimensional body, a thin substrate, such as a wire. Different intervals expose a different temperature, determining a particular distribution at different times. The heat diffusion is obtained as data representing the way in which the temperature of each interval varies through time, tending to increase or decrease depending on the exchange of heat with other intervals.

A simple method developed for deriving a numerical solution to the Heat Equation is the method of finite differences (Geist *et al.*, 1994; Ortega-Arjona, 2000). Consider the discrete form for the one-dimensional heat equation:

$$A(i+1, j) = A(i, j) + \frac{\Delta t}{\Delta x^2}(A(i, j+1) - 2A(i, j) + A(i, j-1))$$

where i represents time steps and j indicates wire subintervals. The numerical solution is now computed simply by calculating the value for each interval at a given time frame, considering the temperature from both its previous and its next intervals (Ortega-Arjona, 2000).

Figure 5 shows a description of the Manager-Workers pattern (Ortega-Arjona, 2004) and the Communicating Sequential Elements pattern (Ortega-Arjona, 2000) as two Architectural Patterns for Parallel Programming (Ortega-Arjona & Roberts, 1998) whose runtimes are compared when solving a particular problem. These two architectural patterns are used to obtain two different solutions for the Heat Equation problem, represented as the equation above, on a cluster of 16 computers (Geist *et al.*, 1994). Notice that the UML State Diagram for each architectural pattern represents the data dependencies that such a pattern describes. For example, the data dependencies of the Communicating Sequential Elements pattern constrain that a process on stage i must wait until its own predecessor and the predecessor of its left and its right neighbour have changed state on stage $i-1$. On the other hand, the Manager-Workers pattern proposes that every process on stage $i-1$ must finish computing before changing state to level i. Notice that for the Manager-Workers pattern, the states marked with S are synchronisation states, which are considered to cause no delay (this means, they are deterministically distributed with parameter $d = 0$).

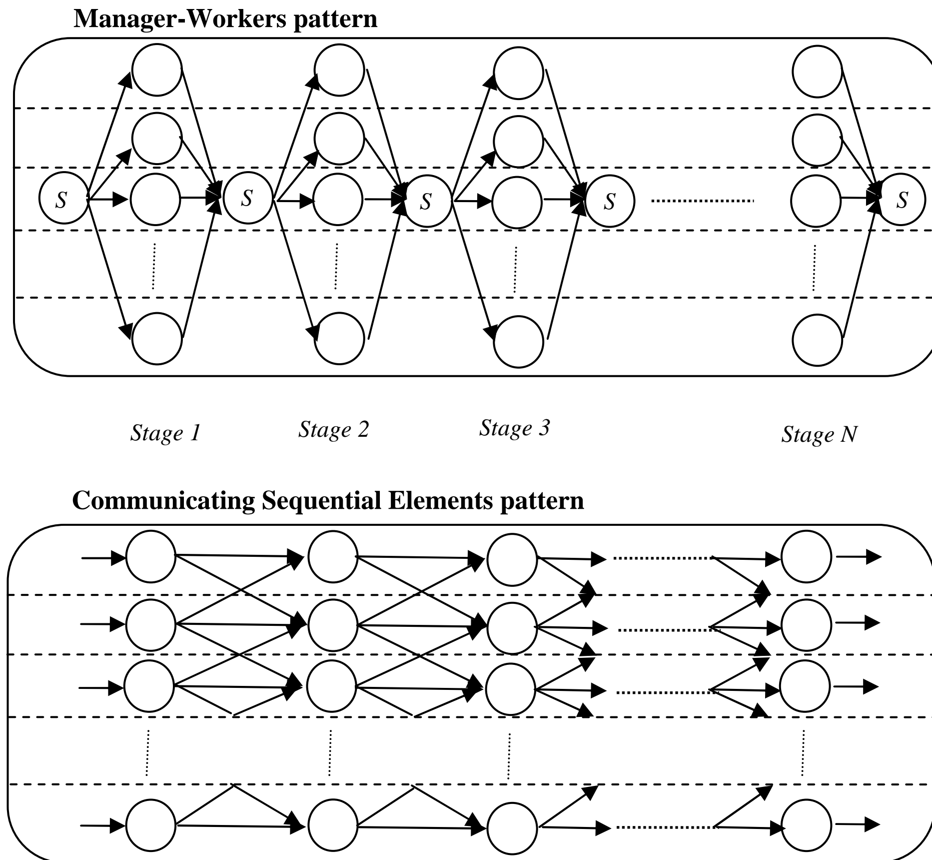

Fig. 5. UML State Diagrams for two Architectural Patterns for Parallel Programming

In order to compare the approximated runtimes with simulation results, ten simulations have been performed for each model and for both architectural patterns, considering the variations on *(a)* the number of processes, *(b)* the number of phases (k) for the Erlang-k distributions (E_k), and *(c)* variations of the parameters d and λ for the deterministic phase and the exponential phase, respectively. These variations represent different workloads for the parallel system. Table 1 shows only four of these variations, which are considered relevant for the present analysis, since they accomplish the *t*-test criteria for comparing the two sets of values (Weiss, 1999; Montgomery, 1991). The errors between approximated and simulation results lie between 0% and 1.5% of the greater result in these simulations. Notice that the present method should be theoretically exact if the simulation model consists of only deterministically or only exponentially distributed runtimes.

Some comments about the simulations and their results:

1. The comparisons between approximated results, exact values, and simulation results are obtained for the Communicating Sequential Elements pattern in Figure 5. It is supposed that all processes have identically distributed runtimes, as workload. The accuracy of the approximation is tested by workload distributions of type Erlang. In order to be able to compare results, λ and k have been chosen to get the first moment constant, for different variances.

Table 1. Comparison of exact, approximated, and simulation results

Workload	Exact		De-approximation	Simulation result
	States	Runtime		
$E_1(0.25)$	256	39.84	39.84	39.64±0.27
$E_2(0.5)$	5120	33.75	33.77	33.35±0.42
$E_4(1.0)$	**Too high computational costs for calculating it**		28.27	28.26±0.37
$E_8(2.0)$			25.43	25.67±0.53

The *de-approximation* column presents the computed runtime for approximated models. The $E_k(\lambda)$ distributed runtimes are approximated by a model with a deterministically distributed runtime and an exponentially distributed phase. An exact results is obtained only when the process runtimes are modelled by $E_1(\lambda)$ or $E_2(\lambda)$ distributed random variables. In the case of $E_4(\lambda)$ and $E_8(\lambda)$ distributed runtime, the computation costs of an exact Markovian analysis are too high in terms of the number of states in the state space. The approximation results and simulation results are compared to obtain information about the quality of the approximation method. The runtimes in the simulation models are also approximated by a model with a deterministically distributed runtime and an exponentially distributed phase. The approximated runtimes lie in the 0.99 confidence interval of the simulation results in Table 1.

2. The comparisons between approximated runtimes of a model structured with the Communicating Sequential Elements pattern and the approximated runtimes of a model structured with the Manager-Workers pattern are obtained for various workloads, represented by the parameters moment and variance, when solving the Heat Equation.

Table 2. Comparison of CSE and MW runtimes.

Workload	Runtime CSE (seconds)	Runtime MW (seconds)
Moment = 5 Variance=25	41.66±0.43	46.63±0.38
Moment = 5 Variance=5	30.27±0.26	32.46±0.22
Moment = 5 Variance=2.5	27.42±0.22	29.15±0.18
Moment = 5 Variance=0	21.0±0.17	20.97±0.21

It is noticeable from Table 2 that MW model presents always a higher execution runtime than the CSE model. The difference of the total expected runtime increases with the increasing variances of the process runtimes. In the case of constant process runtimes, the two models have very similar expected runtimes. This is, for the shortest runtime, the variance of the process runtimes is 0.

6 Conclusion

This paper presents a method to approximately compute the runtime of a parallel program. The method allows to evaluate models that are structurally more complex in terms of processes and their states. Using deterministically and exponentially distributed runtimes, more realistic models of the real behaviour can be obtained than using only phase type distributions. Thus, the method is composed of two approximations:

1. Runtimes are approximated by a deterministically distributed and an exponentially distributed runtime variable.
2. The overall runtime is obtained by an approximate evaluation of the model.

The experiments for solving the Heat Equation with CSE and MW models have shown that the approximation results differ less than 1.5 percent from exact Markovian results.

7 References

1. **Billinton, R., and Allan, R.N. (1992).** *Reliability Evaluation of Engineering Systems: Concepts and Techniques.* Springer.
2. **Booch, G., Rumbaugh, J., and Jacobson I. (1998).** *The Unified Modeling Language User Guide.* Addison-Wesley.
3. **Fowler, M., and Scott, K. (1997).** *UML Distilled.* Addison-Wesley Longman Inc., Reading MA.
4. **Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994).** *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press.
5. **Kleinrock, L. (1975).** *Queueing Systems. Volume 1: Theory.* John Wiley and Sons.
6. **Lui, J.C.S., Muntz, R.R., and Towsley, D. (1998).** *Computer performance bounds of fork-join parallel programs under a multiprocessor environment.* IEEE Transactions on Parallel and Distributed Systems, Vol. 9 No. 3.
7. **Montgomery, D.C. (1991).** *Design and Analysis of Experiments.* John Wiley & Sons, Inc.
8. **Ortega-Arjona, J., and Roberts, G. (1998).** *Architectural Patterns for Parallel Programming.* Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP 98).
9. **Ortega-Arjona, J. (2000).** *The Communicating Sequential Elements pattern. An Architectural Pattern for Domain Parallelism.* Proceedings of the 7th Conference on Pattern Languages of Programming (PLoP 2000).
10. **Ortega-Arjona, J. (2004).** *The Manager-Workers pattern. An Activity Parallelism Architectural Pattern for Parallel Programming.* Proceedings of the 9th European Conference on Pattern Languages of Programming and Computing (EuroPLoP 2004).
11. **Pooley, R., and King, P. (1999)** *The Unified Modeling Language and Performance Engineering.* IEE Proceedings – Software 146(2).
12. **Thomasian, A. and Bay, P. (1986).** *Analytic Queueing Network Models for Parallel Processing of Task Systems.* IEEE Transactions on Computers, December 1986.
13. **Weiss, B. (1999).** *Introductory Statistics.* Addison-Wesley.

Jorge L. Ortega Arjona *is a full-time lecturer of the Department of Mathematics, Faculty of Sciences, UNAM. He obtained a BSc. in Electronic Engineering, as well as a MSc in Computer Science, at UNAM, and a PhD from the University College London (UCL), U.K. His research interests include Software Architecture and Design, Software Patterns, and Parallel Processing.*