Architectural Performance Models

Estimating the Contribution of Software Structure to the Performance of a Parallel Software Architecture

Jorge L. Ortega-Arjona and Graham Roberts
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, U.K.
{J.Ortega-Arjona, G.Roberts}@cs.ucl.ac.uk

Abstract

Parallel System programming requires sophisticated and cost-effective performance estimation techniques for successful development. Architectural Performance Models, based on Architectural patterns, a component simulator and a performance analysis, are presented here as an approach to estimating the performance of parallel applications, by obtaining the contribution to performance from their Software Structure. This paper presents a brief introduction to the Architectural Performance Models, their development and use, and an experimental evaluation using a case study in order to validate the accuracy of their estimations.

Keywords

Parallel Software Architecture, Architectural Patterns, Software Structure, Component Simulator, Scale-model, Performance Analysis and Estimation.

1. Introduction

Parallel Systems Programming allows the exploitation of a number of computing resources performing simultaneous activities to achieve a common objective. By its nature, Parallel Systems Programming requires extra effort from the software developer, because of the increased complexity that comes from programming several resources executing simultaneously. Furthermore, as Parallel Systems Programming is considered a means for improving performance — viewed here as the reduction of execution time— the software design has to take into account sophisticated and cost-effective practices and techniques for performance measurement and analysis. In particular, it is of great practical interest to obtain performance data during the design stages and before implementation, since this enables the software developer to carefully choose the order and organization of computations and communications between components.

Such considerations are among the premises of the Architectural Performance Model. An Architectural Performance Model is based on Architectural Patterns as descriptions of the Software Structure of parallel programs, a component simulator, and a performance analysis of parallel applications, executing on the Parallel Virtual Machine (PVM) standard (Geist *et al.*, 1994).

The definition of Software Structure comes from the idea that the Software Architecture of a program can be considered in terms of groups of components, which have a specific responsibility and different rate of change during the program's lifetime. This idea was inspired by the concept of the *"Layers of Change"* (Brand, 1994). In a previous paper, we have defined the groups of components as Software Site, Software Structure, Software Skin, Software Services, Software Space Plan and Software Stuff (Ortega & Roberts, 1999b). For the purposes of the present work, let us consider only the definitions for Software Site and Software Structure: *Software Site* is defined as the set of software components associated with hardware and software environment around the program. *Software Structure* is the set of components that represent the fundamental structural organization schema for execution and communication. Finally, let us consider as non-structural components all other groups that, complementary to Software Site and Software Structure, define the Software Architecture of a program.

In our opinion, the role that simulation techniques can play in parallel software development has not yet been fully recognised. Simulation based methods and tools can help manage complexity of software development for parallel systems. In this context, the fundamental advantage of simulation is flexibility. Simulation methods and tools make it possible to compare the behaviour of different Software Structures on the same hardware platform, to assess the effect of different organizations, changes on components, or even to study the performance of a parallel program on several existing, required, or hypothetical computing situations. Simulation methods and tools require neither the oversimplifications which are commonly used to deal with complex hardware/software systems through analytical models, nor the availability of fully developed software program. However, it should be noted that accurate simulations are in general computationally expensive (Zelkowitz & Wallace, 1998).

2. Architectural Performance Model

Architectural Performance Modelling is characterised by a low simulation overhead, thanks to the adoption of models of program components which are at a higher level of abstraction. The parallel software programs simulated using the Architectural Performance Model can range from a complete parallel program to a partially implemented program design. The simulation of the whole system, using the available information about hardware and software, makes it possible to obtain estimates of the parallel system performance.

The Architectural Performance Model combines components representing Software Structure along with component simulator objects representing the internal tasks performed by each component in the program. Basically, a complete parallel software architecture consists of a network of interconnected components. Each component is simulated using a component simulator object, which has been implemented using a Markovian generator kernel, giving estimates of the component execution (Ortega & Roberts, 1999a). The component simulator objects are connected following the Architectural Pattern used to define the Software Structure, setting up a kind of "scale-model" representative of the complete parallel software application. The effect of Software Structure then can be taken into account by means of predefined components sharing the same physical medium, which generate a given statistical contribution of the Software Structure to the performance of the parallel Software Architecture.

2.1. Simulation

A typical Architectural Performance Model simulation consists of the following phases:

1. The Software Structure modelling code is built with information from the Architectural pattern. Code can be supplied either as a complete parallel program or as a skeleton of code representing the structural relations among components. Component computations have been replaced by a component simulator that takes into account the time parameters spent or required in the actual code, setting up a concurrent scale-model version of the parallel program.

2. The simulation is performed by executing the scale-model program produced. A suitable number of executions to record the events occurring in a specific point of the scale-model are obtained.

3. The information on system performance produced during the simulation is examined. As component simulators produce a trace of program time execution in the form of processing, communicating and idling time, it is also possible to collect more in-depth information about the parallel program behaviour and system performance off-line. If the obtained performance is not satisfactory, changes to the Architectural Performance Model can be applied, ranging from adding or substracting components, test for other architectural patterns or configurations, or considering different hardware allocation strategies. This entails repeating all the previous steps several times.

A point that is worth discussing in more detail is how a parallel program is modelled by Architectural Performance Models. The parallel program behaviour is modelled at task level by a set of component simulators, which model the time of tasks executing on every component. For the sake of simulation accuracy, these simulators should reproduce as closely as possible the distribution of sequence and timing of the run-time requests expected from the actual program. A way to obtain this is to measure the execution time between communications, and the time of communications themselves. The time parameters are then estimates found by direct measurement under suitable test conditions or through statistical and/or analytical models (Sötz, 1990; Smith & Williams, 1993; Foster, 1994). Another possible approach relies on the examination of the source code, analysing the performance behaviour of each task on the particular sequential or parallel computer where it will actually be executed (Mohr, 1990; Smith & Williams, 1998).

2.2. The Basic Mathematical Model

A good performance model is able to explain available observations and predict future circumstances, while abstracting unimportant details. However, conventional computer system modelling techniques, which typically involve detailed simulations of individual hardware components, introduce too many details to be of practical use to parallel program designers. In this section, we introduce a performance modelling technique that provides an intermediate level of detail. This technique is certainly not appropriate for all purposes: it is specialized for obtaining the performance contribution of Software Structure. However, it can be proven useful for a wide range of parallel design problems.

Let us define performance of a parallel program as the time that elapses from when the first component starts executing on the problem to when the last component completes execution. Performance refers to the responsiveness of the parallel system — the time required to respond to events, or the number of events processed in some time interval (Smith & Williams, 1993; Foster, 1994; Bass *et al.*, 1998). This definition is not entirely adequate for all parallel computers, but it is sufficient for our actual purposes.

Traditionally, performance models considered for parallel programming specify a metric such as execution time as a function of problem size, number of processors, number of tasks, and other algorithm and hardware characteristics (Foster, 1994; Culler *et al.*, 1997). However, for software modelling purposes, this performance models can be simplified, assuming that during execution, each software component is processing, communicating, or idling, as illustrated in Figure 1.

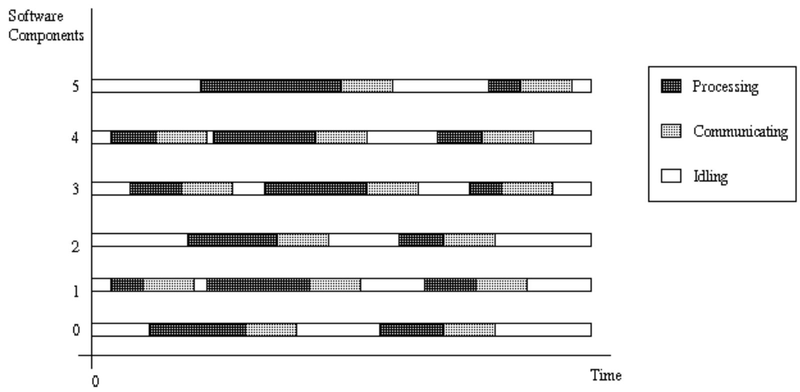


Figure 1. Space-Time View of a PVM parallel program of 6 software components. Each component spends its time processing, communicating, or idling.

Let T^i_{proc}, T^i_{comm}, and T^i_{idle} be the time spent processing, communicating, and idling, respectively, on the *ith* component. Hence, average total execution time *T* can be defined as the sum of computation, communication, and idle times on an arbitrary component *j* (Foster, 1994; Culler *et al.*, 1997),

$$T = T_{proc}{}^{j} + T_{comm}{}^{j} + T_{idle}{}^{j}$$

or as the sum of these times over all components divided by the number of components *N*,

$$T = \frac{1}{N}(T_{proc} + T_{comm} + T_{idle})$$

$$T = \frac{1}{N}\left(\sum_{i=0}^{N-1} T_{proc}{}^{i} + \sum_{i=0}^{N-1} T_{comm}{}^{i} + \sum_{i=0}^{N-1} T_{idle}{}^{i}\right)$$

The last definition is often more useful, since it is typically easier to determine the total execution time of a parallel program in terms of the time spent computing, communicating, and idling of individual components.

As our goal is to obtain the Software Structure contribution to a parallel software performance, it is necessary to develop a mathematical expression that specifies execution time as function of Software Structure and other contributions. This models should be as simple as possible, while providing acceptable accuracy.

Let us consider that the total processing, communicating and idling times are the result of the contribution of all the components of the parallel software architecture, which spend their time processing, communicating or idling. Components are classified and grouped depending on their particular objective and their rate of change in the

Software Architecture as Software Site, Software Structure, and the rest as non-structural software components (Ortega & Roberts, 1999b). The expression for total execution time can be presented in terms of these groups of components as:

$$T = \frac{1}{N}[(T_{Site} + T_{Str} + T_{NonStr})_{proc} + (T_{Site} + T_{Str} + T_{NonStr})_{comm} + (T_{Site} + T_{Str} + T_{NonStr})_{idle}]$$

Rearranging terms, the total execution time for any parallel program is obtained as function of the three contributions, as follows:

$$T = \frac{1}{N}[(T_{proc} + T_{comm} + T_{idle})_{Site} + (T_{proc} + T_{comm} + T_{idle})_{Str} + (T_{proc} + T_{comm} + T_{idle})_{NonStr}]$$

$$T = \frac{1}{N}(T_{Site} + T_{Str} + T_{NonStr})$$

Where T_{Site}, T_{Str}, and T_{NonStr} are respectively the contribution times of the Software Site, Software Structure, and the non-structural components. According to this expression, the total execution time of a parallel program contains contributions of each one of the groups of components. Our aim here is obtaining a close representation of the contribution of Software Structure and the non-structural components, and verify their use for a total estimation of performance by experimentation.

3. Experimentation

The Architectural Performance Models for performance estimation is based on the assumption that contributions to the total execution time of a parallel program due to Software Site, Software Structure and non-structural components are independent from each other from. Therefore, the average contribution of Software Structure depends exclusively on the structural organization schema for execution and communication, and can be obtained by experimentation. However, in order to develop the experiments, let us consider the following assumptions to reduce model complexity.

- The simulation model is based on the Parallel Virtual Machine (PVM) standard (Geist *et al.*, 1994). Therefore, low-level hardware details such as memory hierarchies and the topology of processor interconnection network are considered to be solved by the PVM environment.

- Scale analysis is used to identify insignificant effects that can be ignored in the analysis. For example, if a program consists of an initialization step followed by several thousand iterations of computations, and if initialization is very expensive, then we consider only the computation step in our analysis.

- Empirical studies are used to calibrate simple component models rather than developing more complex models from first principles.

Experimentation using execution-driven simulation seems to be the option to determine the effectiveness of our approach. Execution-driven simulation is the recommended option when program execution time has no simple dependence on input data, which seems to be the case of Software Structure. In execution-driven simulation, every parallel program task is modelled by a component simulator containing a parameters obtained from actual execution of the real parallel program. During the simulation, the parallel program is actually executed (i.e., it is not simulated). Nevertheless, the behaviour of the Software Structure — communications and interconnections organization — and the non-structural components are still simulated. Finally, the execution times measured from the real program and those obtained from the simulation are compared. In order to observe the contribution of Software Structure, changes adding or substracting components are made in both, real program and simulation, comparing results. To confirm the accuracy of our approach based on obtaining of Software Structure contribution in concurrence, other group of executions and simulations are performed, maintaining Software Structure unchanged, but modifying the hardware, aiming to verify the accuracy of this method for different parallel hardware configurations.

3.1. A Case Study: The Polygon Overlay Problem

In order to illustrate how experimentation has been carried out to verify the accuracy of the Architectural Performance Model approach, an example based on the Polygon Overlay Problem (Wilson & Lu, 1996) is used. This section briefly describes the problem and a solution proposed (Winder *et al.* 1996), based on the Manager and Workers Architectural pattern (Ortega & Roberts, 1998).

The Problem

Suppose we have two maps, A and B. Each map covers the same area, and is decomposed into a set of non-overlapping rectangular polygons (Figure 2). Our aim is to overlay the maps, that is, to create a new map consisting of the non-empty polygons in the geometric intersection of A and B. This is a common problem that frequently arises in geographical information systems, in which the first map might represent soil type and the second, vegetation. Their overlay shows how combinations of soil type and vegetation are distributed.

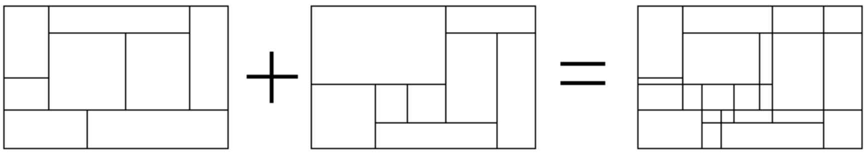


Figure 2. Example of Polygon Overlay.

In the general case, polygons in the original maps may be non-convex; their overlay may then contain polygons which consist of multiple disjoint patches. This is often dealt with by representing each non-convex polygon as the union of two or more convex polygons. A post-processing stage then re-labels disjoint patches which are to be treated as a single polygon. However, in order to simplify this problem, it is required that all polygons be non-empty rectangles, with vertices on a rectangular integer grid $[0...N]\times[0...M]$. It is also required that input maps have identical extents, that each be completely covered by its rectangular decomposition, and that the data structures representing the maps be small enough to fit into physical memory. It is not required that the output map to be sorted, although all of the input maps used in this example are usually sorted by lower-left corner (Wilson & Lu, 1996).

The Solution

A typical parallel programming approach for this sort of problem is to built a Manager-Workers system. We have a set of workers which do the actual polygon overlaying, a manager that, on request, gives pieces of work to the workers. Once processing is finished, the manager is sent the results by the workers (Winder *et al.*, 1996). The communication pattern among the active objects, using an architectural notation proposed by Bennett (1997), is shown in Figure 3. The manager and workers are all made active objects.

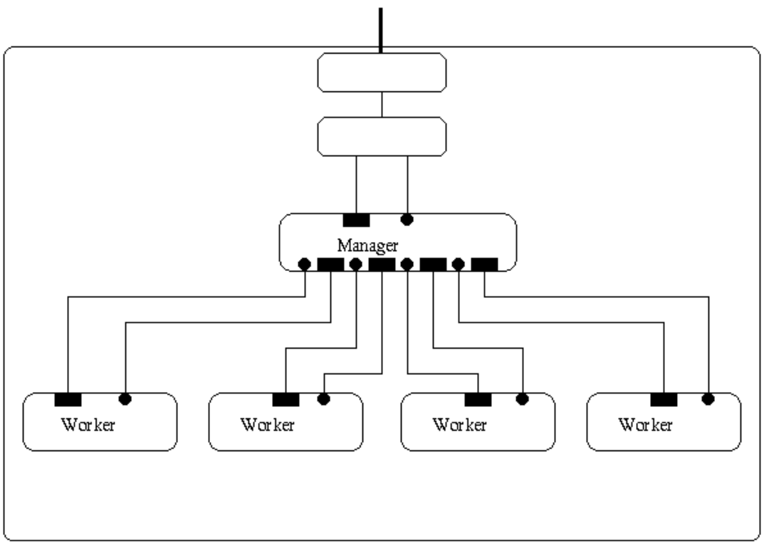


Figure 3. Architectural representation for the Manager and Workers pattern.

Let the two input maps be called M_1 and M_2. The solution goes through all the polygons belonging to M_1, and for each one of them, the goal is to find all the intersections with any polygon in M_2. The key to efficiency is to limit the part of M_2 that has to be looked at to find these overlaps (Winder *et al.*, 1996).

3.2. Experimentation Results

After carrying out the steps to obtain the Architectural Performance Model, both event-driven simulation and real program are executed concurrently on a single processor, and measured for different component configurations modifying the number of workers. The performance behaviour in terms of average total execution times measured in the real program and estimated by the simulation program, considering different number of worker components, are presented in Table 1.

Number of Workers	**Real Average Execution Time (seconds)**	**Simulation (seconds)**
1	25.317	17.387
2	16.183	8.499
3	13.260	5.661
4	12.147	4.443
5	12.597	3.529

Table 1: Real and simulated average total execution times in concurrent execution

Notice that simulated times reflect a similar trend to the measurements on the real execution. The tendency can be observed more clearly in Figure 4, which shows the average execution characteristic for both real and simulation programs.

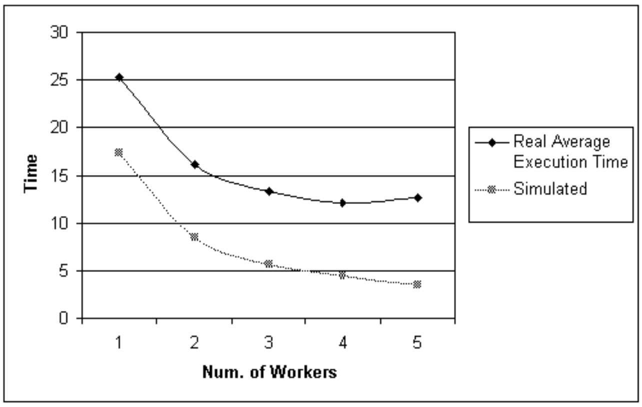


Figure 4. Average execution characteristic for real and simulation programs in concurrent execution

Now, observing the difference between the partial results, let us consider the following:

- The total times obtained from measuring the processing, communicating and idling times of the real program is composed of all three contributions from Software Site, Software Structure and non-structural components. Thus, the expression for the total execution time for the real program remains as:

$$T_{Real} = T_{Site} + T_{Str} + T_{NonStr}$$

- The total times obtained from the even-driven simulation are numerically representative only for the non-structural components contribution, whose distribution is affected by the Software Structure, but they do not account for its contribution. Thus, the expression for the total execution time considering the values from the event-driven simulation covers only the contributions from Software Site and non-structural components, as follows:

$$T_{Sim} = T_{Site} + T_{NonStr}$$

- Both, event-driven simulation and real program were executed using exactly the same hardware and environment resources, so we can assume that the time due to Software Site component contribution is the same for both executions. Substracting from the expression for total execution time in the real program the obtained event-driven simulation values, the following is obtained:

$$\begin{aligned} T_{Real} - T_{Sim} &= (T_{Site} + T_{Str} + T_{NonStr}) - (T_{Site} + T_{NonStr}) \\ &= (T_{Site} + T_{Str} + T_{NonStr} - T_{Site} - T_{NonStr}) \\ T_{Real} - T_{Sim} &= T_{Str} \end{aligned}$$

This expression means that it is possible to obtain the estimation of the contribution due to the Software Structure for each configuration in concurrent execution by directly substracting the simulation times from the real execution times. Table 2 shows the estimations of the Software Structure contribution in the concurrent case for different number of components.

Number of Workers	Software Structure contribution
1	7.929
2	7.684
3	7.599
4	7.705
5	9.068

Table 2: Estimated Software Structure contribution in concurrent execution

Now, using the Software Structure and non-structural contributions from the scale-model, it is possible to obtain estimations of total execution times for different parallel variations of the program. Software Site is changed for each case, while Software Structure and non-structural components are kept unchanged. In order to corroborate and evaluate the accuracy of the Architectural Performance Model, the estimations are compared with the actual average total execution times, obtained from executing and measuring several times the real program. For each case, the percentage of error is calculated, as shown in Tables 3, 4, 5, 6, and 7.

Number of Processors	Software Structure contribution (seconds)	Simulated Non-structural contribution (seconds)	Total Estimation (seconds)	Real Average Execution Time (seconds)	Percentage of Error
1	7.929	17.387	25.317	25.317	0%
2	3.965	8.293	12.258	15.471	20.77%

Table 3: Estimated and real execution times for a Software Structure with 1 manager and 1 worker

Number of Processors	Software Structure contribution (seconds)	Simulated Non-structural contribution (seconds)	Total Estimation (seconds)	Real Average Execution Time (seconds)	Percentage of Error
1	15.369	16.997	32.367	32.367	0%
2	7.685	7.986	15.670	17.180	8.79%
3	5.123	5.389	10.512	11.550	8.99%

Table 4: Estimated and real execution times for a Software Structure with 1 manager and 2 workers

Number of Processors	Software Structure contribution (seconds)	Simulated Non-structural contribution (seconds)	Total Estimation (seconds)	Real Average Execution Time (seconds)	Percentage of Error
1	22.796	16.984	39.780	39.780	0%
2	11.398	8.151	19.549	22.354	12.55%
3	7.599	5.458	13.057	14.362	9.09%
4	5.699	4.160	9.859	15.463	36.24%

Table 5: Estimated and real execution times for a Software Structure with 1 manager and 3 workers

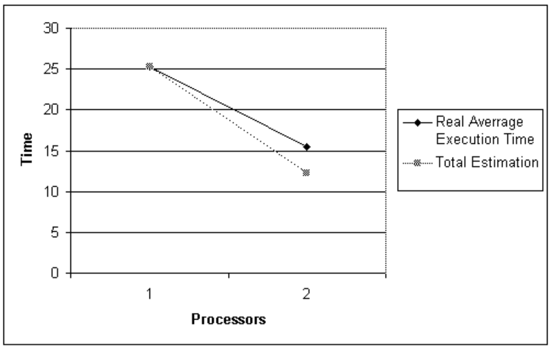
Number of Processors	Software Structure contribution (seconds)	Simulated Non-structural contribution (seconds)	Total Estimation (seconds)	Real Average Execution Time (seconds)	Percentage of Error
1	30.818	17.772	48.590	48.590	0%
2	15.409	8.309	23.718	26.603	10.84%
3	10.273	5.647	15.919	15.922	0.02%
4	7.705	4.159	11.863	12.004	1.17%
5	6.164	3.3605	9.524	14.582	34.69%

Table 6: Estimated and real execution times for a Software Structure with 1 manager and 4 workers

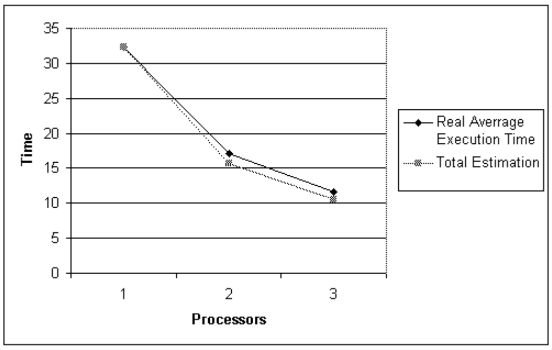
Number of Processors	Software Structure contribution (seconds)	Simulated Non-structural contribution (seconds)	Total Estimation (seconds)	Real Average Execution Time (seconds)	Percentage of Error
1	45.340	17.643	62.983	62.983	0%
2	22.670	8.912	31.582	36.73	14.01%
3	15.113	5.926	21.039	22.688	7.27%
4	11.335	4.475	15.810	22.572	29.96%
5	9.068	3.600	12.669	15.923	20.44%
6	7.556	2.998	10.554	15.869	33.49%

Table 7: Estimated and real execution times for a Software Structure with 1 manager and 5 workers

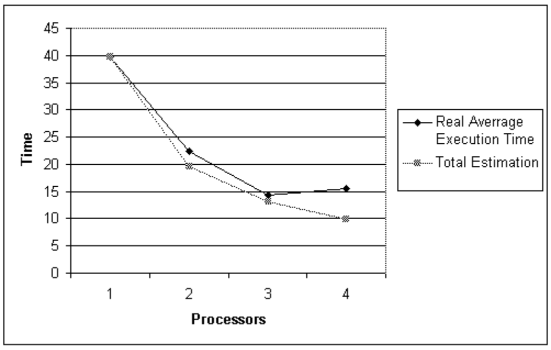
The comparisons between the total estimation (obtained from adding Software Structure estimated contribution and non-structural simulated contribution) and the real average execution times for case are shown in Figure 5.



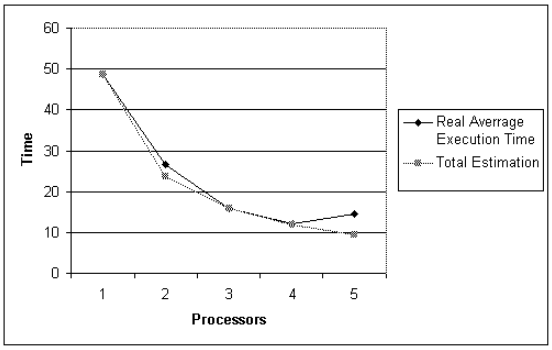
(a) One manager and one worker



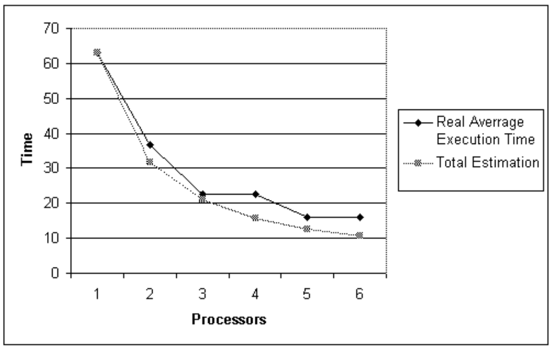
(b) One manager and two workers



(c) One manager and three workers



(d) One manager and four workers



(e) One manager and five workers

Figure 5. Comparisons between the total estimation and real average execution time for each different Software Structure case

4. Summary and Conclusions

In summary, the Architectural Performance Model approach makes it possible to simulate a parallel application program by combining initial information about its Software Structure and time parameters of its components. The Software Structure of a complete parallel software program consists of a network of components, connected in the form of an Architectural pattern. The component time behaviour is simulated by a component simulator object that has been implemented using a Markovian generator kernel, giving estimates of the execution of each component based on parameters of different distributions that represent its time behaviour. The effect of Software Structure then can be taken into account by means of predefined components, concurrently sharing the same physical medium, which indirectly generate a given statistical contribution of the Software Structure to the performance of the parallel software architecture. The Architectural Performance Model, then, can be seen as a kind of scale-model which represents the complete parallel software application time behaviour.

The estimation of the performance of a parallel program using Architectural Performance Models is based on the following basic assumptions:

- The Software Architecture of the program can be expressed in terms of Software Site components, Software Structure components and non-structural components. Each group of components provides an independent contribution to the total performance of the parallel program.

- The Software Structure contribution depends exclusively on the fundamental organisation of components and connections, expressed in the form of an Architectural pattern.

- Non-structural components behaviour is simulated using a component simulator that exhibits an average execution time for each component.

- The Architectural Performance Model uses information of both the Architectural pattern and simulation time parameters to estimate contributions of both structural and non-structural components respectively during concurrent execution. This information can be used in the performance analysis to obtain good estimations of the performance of a complete or partially developed parallel program for several different hardware and software variations.

The Architectural Performance Model is presented here as an approach to approximately estimate in an isolated form the contribution of Software Structure to the performance of a Parallel Software Architecture. This information may serve in the future as a method to evaluate performance between different Software Structures. For example, in the case study presented, the performance contribution of a Software Structure based on the Manager and Workers Architectural pattern is estimated for structural variations, reflecting that the contribution of Software Structure oscillated between 31.32% and 71.98% of the total average execution time. Even though there is a large difference between these values, this information can be used effectively as an approximation window to estimate the probable performance of systems sharing the same Software Structure, and compare with other probable program solutions which may use other Software Structures.

The results of the evaluation tests of all our performance estimations have been particularly encouraging. Architectural Performance Models led to figures that in the average case were more than 12.41% and, in the worst case, about 36.24% different from those obtained by running the actual program on the real computing environment. Furthermore, using the component simulator in a concurrent execution results efficient due to the scale-model simulation speed is too high to be compared to the execution rate of the actual program on a single processor.

Unfortunately, the ease of use of Architectural Performance Models is not equally satisfactory. The construction of the scale-model code is not easy and requires a sufficient degree of familiarity with the component simulator itself. Furthermore, setting up the component simulator parameters is a time-consuming task. These activities, perhaps, lend themselves to be automated by a software tool.

5. References

Bass, L., Clements, P., and Kazman, R. (1998), *Software Architecture in Practice.* Addison-Wesley, Reading Massachusets.

Bennett, D. (1997), *Designing Hard Software. The Essential Tasks*. Manning Publication Co., Greenwich, Connecticut.

Brand, S. (1994), *How Buildings Learn. What happens after they're built.* Phoenix Illustrated, Orion Books Ltd.

Culler, D., Singh, J. P., and Gupta, A. (1997), *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann Publishers.

Foster, I. (1994), *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Co. Reading, Massachusets.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V. (1994), *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press. Cambridge, Massachusets.

Mohr, B. (1990), *Performance Evaluation of Parallel Programs in Parallel and Distributed Systems*. Lecture

Notes in Computer Science, H. Burkhart (ed.). Springer-Verlag.
Ortega-Arjona, J. L., and Roberts, G. (1998), *Architectural Patterns for Parallel Programming*. 3rd European Conference on Pattern Languages of Programming, EuroPLoP'98. Kloster Irse, Germany. July 1998.
Ortega-Arjona, J. L., and Roberts, G. (1999a), *A Pattern-based Simulation. Simulating the Actor Model using the Active Object Behavioral Pattern*. Research Note RN/99/25. Department of Computer Science, University College London.
Ortega-Arjona, J. L., and Roberts, G. (1999b), *The Layers of Change in Software Architecture.* 1st Working IFIP Conference on Software Architecture, WICSA1. San Antonio, Texas, USA. February 1999.
Smith, C. U., and Williams, L. G. (1993), *Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives.* IEEE Transactions on Software Engineering, Vol. 19, No. 7, July 1993.
Smith, C. U., and Williams, L. G. (1998), *Software Performance Engineering for Object-Oriented Systems: A Use Case Approach*. Performance Engineering Services and Software Engineering Research. Santa Fe, New Mexico.
Sötz, F. (1990), *A Method for Performance Prediction of Parallel Programs.* Lecture Notes in Computer Science, H. Burkhart (ed.). Springer-Verlag.
Wilson, G. V., and Lu, P., eds. (1996), *Parallel Programming using C++*. Scientific and Engineering Computation Series. The MIT Press. Cambridge, Massachusets.
Winder, R., Roberts, G., McEwan, A., Poole, J., and Dzwig, P. (1996), *UC++*. In *Parallel Programming using C++*, Wilson, G. V., and Lu, P., eds.
Zelkowitz, M. V., and Wallace, D. R. (1998), *Experimental Models for Validating Technology*. IEEE Computer, May 1998.