THE COMPLEXITY OF PERFORMANCE ANALYSIS IN PARALLEL ALGORITHMS AND SYSTEMS

Giuseppe Iazeolla
University of Rome II Via O. Raimondo, 8
00173 Roma, Italy

Abstract: The paper deals with the computational complexity of the performance analysis of parallel algorithms and concurrent systems. The type and origin of this complexity is discussed and the way to deal with it is investigated. Basing on a symbolic notation of the large markovian state-space, an approach is proposed which takes advantage of computer algebra and symbolic-manipulation languages to collapse the state-space, restructure it and introduce efficient methods for global balance equation solution, based on either closed-form or on low-cost gaussian elimination techniques. The position of the approach relative to more conventional methods for the solution large markovian systems (such as the decomposition and aggregation method and the stochastic Petri net method), is also discussed.

1. Introduction

Concurrent processing systems, including conventional supercomputer architectures and innovative supercomputing systems based on MIMD or SIMD, can be conveniently analyzed using F/J models, that is queueing models with Fork and Join service centers operated by multiple parallel servers with synchronization constraints on arrivals and departures [12, 18, 29, 22, 25, 26].

The same can be said of parallel algorithms, as applied to game playing, theorem proving, parallel query execution, heuristic optimization, VLSI layout, motion planning etc [17].

On the other hand, the Markov stochastic process underlying such models quickly leads to such an enormous state-space that the numerical solution of balance equations is unfeasible in time-complexity terms. Published results generally deal with "open" F/J queuing models [2, 3, 8, 12, 15, 17, 23, 24, 25, 26], while for "closed" models only approximate methods are available [18, 21].

There exists a general lack of exact and efficient solution methods for "closed" case. In this paper, closed F/J models are dealt with.

The paper is organized as follows. In Section 2, the solution complexity of the conventional (sequential) model is briefly recalled. In Section 3, the parallel and concurrent model is discussed, and its explosive time-complexity is investigated. In Section 4, the way to deal with it is treated and an approach is outlined, based on a symbolic notation of the large markovian state-space. The approach is also discussed and its position relative to more conventional methods for the solution large markovian systems, such as the decomposition and aggregation method and the stochastic Petri net method, is treated.

2. Solution Complexity in Sequential Models

The conventional system-state notation, used for queueing models of sequential systems or algorithms, is generally based on information on the customer number at various delay centers. Indeed, for a model S consisting of K centers, the instantaneous state is denoted by the vector

$$\mathbf{s} = (n_1, n_2, ..., n_K)$$

(1)

where $n_i$ denotes the number of jobs at center i ($i = 1, ..., K$), and $\Sigma_i n_i = N$, with N the number of modeled jobs.

The set of all possible states as (1), defines the sequential model state-space, denoted $F_S$, whose cardinality $C_S$ is easily shown [22] to consist of number

$$C_S = \left( \binom{N+K-1}{K-1} \right)$$

states. Under some reasonable assumptions [16], the state-to-state transition process defines an homogeneous continuous-time Markov chain, whose solution provides the steady-state probabilities $\pi(\sigma)$ for all states $\sigma$ in $F_S$.

$$Q_S$$

denote the Markov transition-rate matrix, of size $C_S$. The steady-state probability distribution $\pi = \{ \pi(\sigma), \sigma \in F_S \}$ can be derived from the solution of the linear system of the process' balance equations $\pi = \pi Q_S$, subject to the normalizing condition $\sum_\sigma \pi(\sigma) = 1$ for all $\sigma$ in $F_S$.

Matrix $Q_S$ is generally sparse and its solution time-complexity is $O(C_S^3)$ by gaussian elimination methods. By appropriately ordering states in $F_S$, however, matrix $Q_S$ can be arranged into block-diagonal form [6] and the problem complexity lowered down to $O(C_S^2)$ or else $O(C_S)$ operations, depending on cases. According to (2), however, quantity $C_S$ grows quickly with N, the number of jobs, and so it does the problem time-complexity.

From probabilities $\pi(\sigma)$, $\forall \sigma \in F_S$, the performance indices based on queue-length distributions, or those based on center-to-center average delay can be derived. By accompanying notation in (1) with information on the instantaneous position of a marked job, the center-to-center distribution delays can also be obtained [11]. In this case, however, a further markovian process (the marked-system process) has to be introduced and its transient solution derived. This furtherly increases the model solution cost versus N.

3. Solution Complexity in Parallel Models

Section above has briefly outlined the growth with N, the job number, of the performance-analysis costs when sequential system and sequential algorithm models are dealt with. In this Section, the problem is treated of the way in which such costs become prohibitively expensive when the system or algorithm includes parallel and concurrency issues.

3.1. The model

The considered model is depicted in Figure 1. It is denoted model S and consists of two sections, $S_1$ and $S_2$, connected by a pair of Fork-and-Join centers (denoted F and J). As in Section 2, model S includes K centers and N jobs. Centers are denoted by P and are eventually completed by a Fork-and-Join pair. Upon passage through the Fork node F, each job splits itself into its siblings tasks, which then enter center P. Upon completion of service in P, tasks may directly themselves into the Join center J.

Center J can be either an AND-join center or an OR-join center.

In the AND-join case, the tasks entering center J are kept waiting for the completion of all their siblings. When all the siblings from a given job are in J, the job is recomposed and finally routed out from center J.

In the OR-join case, instead, as soon as any one of the tasks from a given job terminates service in P, all of its siblings are removed from center P or elsewhere and the job is soon recomposed and routed out from center J.

In Figure 2, center P is depicted in more detail. It consists of p independent servers ( $P_i$ through $P_p$ ), and of a FIFO queue (any other service-time independent discipline can be used). Tasks stemming out from the...
Fork center, enter center-P's servers, if any is available, or enqueue themselves into center-P's queue. Upon exit from P, tasks may either direct themselves into the Join node, or enter the P-center queue again, or enter a new P center, by following the "a-path" exit.

If p > 1 and the related Fork-and-Join pair is also present (in other words it is also h > 1), then center P is called a concurrent center. If, instead, the F-and-J pair is absent, then P is called a sequential center, even if p=1.

In a concurrent center, quantity p gives the center parallelism level, and quantity h the center splitting factor. The pair (h,p) gives the center concurrency level. In a sequential center it is always (h,p) = (1,p). In a concurrent one, the pair (h,p) may take any values and (h, p) = (2, 2) denotes the smallest conceivable concurrency level.

Servers P1 through Pn in a concurrent center, may, in turn, be themselves concurrent centers, and so on, recursively.

The model section S2 is supposed to include only task-handling centers (in other words, either concurrent centers, or sequential centers fed by still unjoined tasks, coming from a P-center "a-path"), while section S1 only includes job-handling centers (in other words, only sequential centers). Therefore, jobs in S2 stemming out from a J-center, soon enter a new F-center and so on, each time again splitting themselves into tasks. On the contrary, jobs moving to center in S1 remain always unsplit. Section S1 is thus called the job-section, while section S2 the task-section, of model S.

When S is viewed as the model of a concurrent system, the system itself can be seen as a closed network of processing and I/O centers with N jobs and K service centers, partly sequential ones (those in S1), and partly concurrent ones, or "a-path" fed sequential ones, (those in S2).

When S is seen as the model of a parallel algorithm, then the algorithm can be thought of as having the following cyclic structure: there is some preparatory sequential computation, processed by the algorithm sections in S1, followed by a parallel one, processed by sections in S2. Then again a sequential stage in S2 (possibly using the results of the previous parallel stage), followed by a parallel one in S2, and so on. When OR-join type centers are used in S2, the model can be seen to represent applications involving parallel search that consists of explorations of multiple paths. Examples are game playing, theorem proving, query execution in a distributed environment, etc.

3.2. State space analysis

In the discussion below the assumption holds that the J-centers are of the AND-join type. This does not introduce a loss of generality in the state-space complexity analysis. Indeed, it is easy to see that when an OR-join center is introduced, a just smaller state space is obtained, instead, and the analysis can proceed as for the AND-join case, except for the lesser time-complexity.

According to Figure 2, a task exiting servers P1 through Pn, may either enter center J or may direct himself back into P's queue (let us neglect the "a-path" for the moment). In the former case, the task may face two possible situations:

(i) its remaining h-1 siblings are waiting for him in J or, else,
(ii) some of them are still in center P's servers or queue.

In case (i), the job is recomposed and is routed out from center J. In case (ii), the considered task remains in J, waiting for the completion of its siblings.

In order to deal with such situations, a new notion σ for state-space has to be introduced, different from (1) and featuring what follows:

1) it must give information on how many tasks of the considered job are in J and how many are still in P (or elsewhere, in case the "a-path" was followed);
2) it must give information on the later task's position in P's queue (or elsewhere) and,
3) it must indicate such information for each of the N jobs and each of the K centers.

This is much more than it can be expressed by the numerical vector notation in (1), and requires a new definition of state-space: a symbolic one, as discussed soon. Two separate notions are to be, to this purpose, introduced: the job-state and the system-state notion.

3.2.1. The job-state: Let A be a job, and assume it resides in a given center P, where it is split into h sibling tasks. The state of job A (called the "job-state") can be expressed by a string $s_A = (b_1, b_2, \ldots, b_H)$ of h symbols taken from the alphabet $V = \{P, P_1, \ldots, P_k, P\}$, $k \leq N - P$. For example by string:

$$s_A = P_1 P_2 P_3 P_5 P_9 \ldots P_{33}$$

which denotes that number two A's tasks are in J, three occupy as many P's servers, and the remaining ones hold the 5th, the 9th etc, and the 33rd position in P's queue, respectively.

It is obvious that not every string on alphabet V denotes a job-state physically consistent with the system topology. State (3), for example, denotes that the A's tasks hold non-contiguous positions in P's queue. This implies that the tasks from some other job (say, job B and/or C etc.) hold the queue intermediate positions. One can easily convinced that this is physically possible only if the system topology admits looping paths on the queue's center (i.e. the Fig.2 case). In the following, it is assumed that consistent job-states are dealt with.

By considering that job A may reside in any of the P-centers in S2, one may say that the job-state definition requires an alphabet with a distinct symbol of type P and a distinct symbol of type J for each distinct center P, and each J, in S2. For a center i in S1, instead, a conventional numerical notation $n_i$ of the type used in (1) remains sufficient.

3.2.2. The system-state: Let A, B, \ldots, W be the N modeled jobs, each splitting itself into h tasks, i.e. $A = (a_1, a_2, \ldots, a_H)$ times, $B = (b_1, b_2, \ldots, b_H)$ times, and so on. Symbols A, B etc may be used to eventually denote jobs of a distinct type, class etc.

Let $s_A, s_B, \ldots, s_W$ be the respective (consistent) job-states at a given time. The system-state at this time, in a parallel and concurrent model, is the string obtained by ordinarly concatenating the instantaneous job-states. In other words, the instantaneous system state is denoted by string:

$$\sigma = (s_A, s_B, \ldots, s_W)$$

(4)

It is obvious that not any concatenation of consistent job-states yields a system-state consistent with the system topology. If string $s_A$ in (4), for example, is the one given by (3), then state $\sigma$ is a consistent system-state if and only if some other job-string (say, $s_B$ or $s_C$, etc.) denotes occupation, by the corresponding tasks, of the queue positions left by job A's tasks. In the following, it is assumed that consistent system-states are dealt with.

Example 1: Assume $S_1 = \{T\}$ and $S_2 = \{P, D\}$ where P is a concurrent center with $h=p=2$, and D is a sequential center fed through an
"u-path" from P. Tasks served from D return into P's queue. Let us have N=3 jobs (say A, B, C). Then strings $\sigma_1 = (T \ T \ D D)$, $\sigma_2 = (T \ P \ T \ D D)$, $\sigma_3 = (T \ D D \ D D)$ give examples of system states. In state $\sigma_1$, jobs A and B are in center T, while job C is split into h=2 tasks, one in service in D and its sibling first-in-line in D's queue. In state $\sigma_2$, job A is also split into two tasks, one in J and its sibling in one available processor in center P. In state $\sigma_3$, job B is split into its tasks which are second and third in line in D's queue. In state $\sigma_4$, all system tasks are enqueued at center D. In state $\sigma_4$ both jobs A and B have one task in P and its sibling in J.

Remark: In comparison with (1), string $\sigma$ now introduced by (4), includes detailed information on the status of job splitting at various delay centers and related queues. It can thus be used for performance studies ranging from the most simple to the most complex. It can be recalculated by the discovery of regularities into its internal structure. Indeed, cardinality $C_C$ can be expressed (DeMar89) by the formula:

$$C_C = \sum \mathcal{R}(n_1, n_2, \ldots, n_h)$$

where $n_i$ ranges from 0 through N for all centers i in $S_I$ and from 0 through $h!$ for all centers in $S_J$, while symbol $\mathcal{R}(n_1, n_2, \ldots, n_h)$ denotes an expression consisting of combinatorial terms in the variables $n_1, n_2, \ldots, n_h$, each term multiplying a coefficient, whose value is bound by quantity $(h!n_1n_2\cdots)^N$.

In the latter quantity, the multinomial coefficient can be recognized, which gives the number of distinguishable sequences of N objects (the tasks) of N types (the jobs) in each concurrent center P.

Formulas (5) and (7) show that the performance evaluation of parallel algorithms and concurrent systems is an NP-hard problem asymptotically, in the number of counters N and the parallelism level h.

Table 1 gives a documentary comparison of the growth of $C_C$ (the concurrent state-space cardinality in (5), in comparison to the sequential one in (2), for a system with $K=4$ centers. In the concurrent case, system $S$ was as described in Example 1 above. In other words, the smallest concurrency level ($h=2$) was applied.

### Table 1. Explosive growth for minimal concurrent state-spaces

<table>
<thead>
<tr>
<th>N</th>
<th>$C_C$</th>
<th>$C_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>89</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>65x10^3</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>27x10^3</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>39x10^4</td>
<td>165</td>
</tr>
<tr>
<td>10</td>
<td>14x10^5</td>
<td>286</td>
</tr>
<tr>
<td>12</td>
<td>10x10^5</td>
<td>455</td>
</tr>
<tr>
<td>14</td>
<td>15x10^6</td>
<td>640</td>
</tr>
<tr>
<td>16</td>
<td>14x10^7</td>
<td>1,330</td>
</tr>
<tr>
<td>20</td>
<td>87x10^7</td>
<td>1,771</td>
</tr>
</tbody>
</table>

Let $Q_C$ now denote the Markov transition-rate matrix, of size $C_C$. The growth of $C_C$ with N in the smallest concurrency case ($h=2$) as well, proves the prohibitive costs to be faced to obtain $\pi = \pi(C, \dot{Q} + FC)$ by the solution of system $\pi = \pi Q_C$ by numerical methods. The numerical solution methods, in this case, quickly show their limits.

The symbolic nature of state definition, however, naturally suggests the use of a symbolic approach as an alternative to the numerical one, as discussed next. The approach effect is to either decrease the problem hardness, or turn it into tractability by the discovery of regularities into its internal structure. In Section 4 the features of the symbolic approach are discussed.

4. The Symbolic Approach

The basic motivation for the new approach is the fact that solution techniques based on the numerical solution of the Markovian global balance equations $\pi = \pi Q_C$ quickly leads to the loss of useful information on the process structure. Indeed, once the system parameter symbols have been replaced by their numerical values, the possibility of algebraically comparing subsystem processes for the scope of analyzing the process structure is lost.

It would instead be more suitable to replace symbols with their numerical values at the latest useful time and, in so doing, to use computer algebra in an unconventional way. While computer algebra is usually invoked at the conclusion of the modeling process, in order to simplify formulas, it might be advantageous to use it from the start, as a tool for the computer-aided investigation of the large markovian process and for the suggestion of possible solution tracks.

From this consideration, an approach has been introduced in [20] and [9] based on computer algebra and algebraic manipulation languages, consisting of two stages. In the first stage, the enormous state space is conveniently collapsed by exact lumping [19], and an interesting state space restructuring technique is introduced. In the second stage, the reduced process transition-rate matrix $Q_C$ is recursively block-banded and scanned in the search for structure regularities. This leads to the determination either of closed-form solutions, as in [9], at least for the simplest algorithm topologies, or of low-cost gaussian elimination ones for the system $\pi = \pi Q_C$, as in [20].

In the gaussian elimination case, it can be proved [20] that the state space lumping lessens the state space by a factor of $N!$ and that the process restructuring and matrix block-banding reduces the gaussian elimination time-and-space cost within the limits of the most efficient sparse-matrix methods. It is also proved that no relevant costs are incurred for the space state lumping and restructuring, nor for the matrix block banding.

In the closed-form solution case, the solution is derived from a computer-algebra driven inspection of system $\pi = \pi Q_C$. In [9], it is proved that some algorithm or system topologies introduce regularities, in the global balance equations $\pi = \pi Q_C$. In this case, a small set of recurrence equations can be extracted which, when solved, gives vector $\pi = [\pi(C), V + FC]$ with no need for gaussian elimination in the global system. For the class of models where such regularities are discovered, low polynomial time-complexity results, in performance evaluating systems and algorithm models.

The LISP-based "muSIMP" and "muMATH" algebra systems [28], [30] have been used for the experimental aims of the method, and a solution package has been developed which runs on a PC environment and is being moved to a larger system (a hypercube concurrent multiprocessor) for larger state space analysis. Similar results can be obtained by the use of other commercially available algebraic manipulation tools such as Macsyma, Reduce etc., running on small, medium or large computer systems.

4.1 The approach versus existing literature

Two main approaches, among others, have recently been emphasized in literature to deal with large markovian systems: the decomposition and aggregation approach [6, 7, 21, 29] and the stochastic Petri net one [11]. The decomposition and aggregation method is a numerical method that splits the large matrix into smaller submatrices and, in so doing, introduces time-complexity savings at the expense of obtaining only approximate results. The approximation acceptability depends on the existence of the "decomposability" condition, which may be hard to meet in general models. The symbolic approach this paper introduces, instead, is an exact-solution approach which may lead to closed-form solutions, in so doing, the symbolic approach obtains comparable and even larger savings in time-complexity terms, without sacrificing precision of results and without being restricted to world scenarios which satisfy decomposability conditions.

The Petri net approach provides an ad-hoc language (the net) to describe the system, or algorithm, under study and a tool to translate that description into a markovian process. The produced state-space matrix, however, remains large and is solved by conventional iterative methods. The symbolic method, instead, is a method to collapse a large state space into a smaller one, to restructure it, and to scan it in the search for regularities. In comparison to stochastic Petri nets, then the symbolic approach has two principal, advantageous features:

A) It is based on a language (LISP) which is appropriate for at least three reasons: (i) it is a standard and has wide distribution; (ii) it provides
numerous useful features, including automatic storage allocation and garbage collection, arbitrary precision integers, and tools for lexical scoping of language; and (iii) it is the host language for several algebraic manipulation systems whose subroutines may be of some interest for programmers implementing alternative semantics [14].

Any complex markovian-process solution tool generally includes four main parts: 1) an user-interface to allow the user to specify the system, or algorithm, under study; 2) a generator of the underlying markovian process; 3) a manipulator of the process itself for restructuring purposes and; 4) a calculator of the process’ invariant measure. In accordance with recent proposals (such as in [1]), for the use, in part (1), of more powerful specification methodologies than Petri nets, the symbolic approach introduces a generative specification of the model by production rules and demonstrates that, by this means, parts (2) and (3) can be efficiently implemented to yield a substantial state-space reduction and give the process rate matrix a convenient structure. This substantially reduces the time-and-space complexity of part (4). Hence, in comparison to even more recently proposed methods, like Colored Stochastic Petri Nets [13], which also introduce some state space reduction, the symbolic method maintains the advantage of conveniently exploiting the structure of the transition matrix and of not being restricted to world scenarios such as are implied by the so-called “regular nets” [5].

Conclusions
The performance evaluation of parallel algorithms and concurrent systems has been shown to be an NP-hard problem, in general terms, because of the exponential growth of the markovian state space, asymptotically, with the number of customers and the parallelism level. A new approach to performance evaluation has been outlined, based of computer algebra and symbol manipulation languages. The position of the approach relative to more conventional methods, found in literature for the solution large markovian systems (such as the decomposition and aggregation method and the stochastic Petri net method), has been also discussed.

References


