Efficient Parallelization of the Method of Moments for Queueing Networks Using Multi-Modular Algebra

Michail Makaronidis, Giuliano Casale*
Department of Computing
Imperial College London
{mm1909, g.casale}@imperial.ac.uk

ABSTRACT

The solution of service performance models based on closed queueing networks often relies on the Mean Value Analysis (MVA) algorithm, which is unable to solve exactly large models with multiple service classes and hundreds or thousands of users. The Method of Moments (MoM) algorithm has been introduced and addressed this problem, by relying on the exact solution of large linear systems with rational coefficients. In this paper, we focus on the design, analysis and implementation of a parallel solver for MoM linear systems. Parallelization is introduced using residue number systems and recombining the results by the Chinese Remainder Theorem. A comprehensive test set representative of modern applications is used for experimental evaluation. The overall result proves the enhanced performance of both the MoM algorithm over established ones, namely Convolution and RECAL, and of the parallel solver over the serial one.

Keywords

Performance Model, Queueing Network, Parallel Solution

1. INTRODUCTION

Queueing network models have proved to be very helpful in describing and analyzing the performance of resource sharing systems, see [3] for an overview. Typically, in order to evaluate a queueing network, one studies the behavior of several performance indices such as throughput, mean queue length, utilization and response times under increasing loads. In this paper, we consider product form queueing network models, which benefit from the availability of closed-form formula for the joint equilibrium probability distribution of the model. These formulas have allowed the development of computationally tractable algorithms to solve the analytical model [6].

*This work was supported by the Imperial College Junior Research Fellowship.

We consider in particular closed queueing networks, *i.e.*, models of systems where a constant number of jobs or users is served by a network of servers with heterogeneous service rates. Closed networks have been extensively investigated in the past [3] and have also been used for capacity planning and performance evaluation of web services running on multi-tier service-oriented architectures. In such models, requests of different types are usually modeled as different workload classes. For example, in order to evaluate the performance of a moden web application under low to heavy load, it is suggested to consider many hundreds of users and several workload classes. Despite that many exact analysis tools and algorithms are available [6], [24], [11], [12], the fact that they may contain several users and workload classes has made exact evaluation of such models infeasible in practice. This infeasibility is the result of high time and space complexity of the existing algorithms, such as the Mean Value Analysis (MVA) [24], which is widely used for evaluating closed queueing networks and only scales for models containing up to two or three classes. In particular, MVA complexity expressions grow combinatorially fast as the product of class populations, resulting in high memory requirements. Even though approximations have been researched in the past, such as local iterative methods [10], it is useful to further investigate and improve exact algorithms scalable on models with large numbers of users because approximate models are unable to provide estimates of probabilistic metrics.

The Method of Moments (MoM) algorithm, proposed in [9], is an efficient algorithm which is able to provide exact values of the performance indices of large multi-class queueing network models. MoM evaluates queueing networks by recursively computing a set of higher-order moments of the queue length distribution of each station. This set of moments uniquely determines the model performance at equilibrium. The resulting recursion is performed by removing a single job at a time from the network, while the number of computed higher-order moments remains constant during each step. Therefore, the number of steps performed by MoM grows linearly with the population size and provides an important computational improvement compared to the established algorithms for multiclass systems.

The main computational cost of the MoM algorithm is the solution of a large linear system with rational coefficients during each step, used to evaluate the next set of higher order moments. As these linear systems are numerically difficult, exact linear algebra is usually needed to compute them. While this creates overheads, we argue that using finite field arithmetic to address the problem enables efficient parallelization of the algorithm, which is useful to fully leverage on the new generation of multi-core processors. Our contribution is therefore to introduce a novel tool that integrates in the MoM algorithm a parallel solver for linear systems with rational coefficients over finite fields (modular arithmetic). This is a novel approach in the field of performance evaluation; similar approaches have been used in the field of cryptography, where linear systems exhibiting the same characteristics are observed. None of the existing solvers available for solution of linear systems using modular arithmetic are able to cope with the arbitrary large numbers encountered during the evaluation of queueing networks with large population sizes. We show using extended experimentation that the parallel implementation achieves substantial performance gains compared to serial implementations of established methods such as Convolution [6] and RECAL [12].

The paper is organized as follows. In Section 2 we give background concepts. Section 3 defines the parallel modular solver used in our tool and adaptations introduced for integration within MoM, whereas in Section 4 the implementation is described from an architectural perspective. The experimental results are examined in Section 5. Finally, Section 6 gives conclusions and outlines future work.

2. BACKGROUND

2.1 Introduction

We consider closed product-form queueing networks with M queues and R classes. Jobs, after being routed probabilistically through the queues and serviced, re-enter the network after a delay Z_r for each class $r = 1, \dots, R$. The service demand $D_{k,r}$ is defined as the product between the average service time and the average number of visits of class r jobs at queue k. The population of class r jobs is the integer N_r, whereas $\vec{N} = (N_1, N_2, \dots, N_R)$ is the population vector and $N = N_1 + N_2 + \cdots + N_R$ is the total number of jobs circulating in the network.

Let $n_k = \sum_{r=1}^{R} n_{k,r}$ be the total number of jobs in queue k in a state of the system and define $G(\vec{N})$ as the normalizing constant which ensures that the equilibrium probabilities sum to one. It is known that

$$G(\vec{N}) = \sum_{\vec{S} \in S(\vec{N})} \left(\prod_{r=1}^{R} \frac{Z_r^{n_{0,r}}}{n_{0,r}!} \right) \left[\prod_{k=1}^{M} \left(n_k! \prod_{r=1}^{R} \frac{D_{k,r}^{n_{k,r}}}{n_{k,r}!} \right) \right] \quad (1)$$

where $\vec{S} = \{n_{k,r} \,|\, k = 1, \dots, M, r = 1, \dots, R\}$ is a state describing the number of jobs across queues and $\vec{S}$ is the state space.

The normalizing constant $G(\Delta\vec{m}, \vec{N})$ refers to a model which differs from the original queueing network by having included $\Delta m_k \geq 0$ additional "replicas" of queue k, $\forall k = 1, 2, ..., M$, *i.e.*, stations with identical service demands. Note that, from a numerical standpoint, we can obtain an upper bound for the maximum number of digits $n_{\max}$ of the normalizing constant of any model by replacing all service demands D_{kr} with $D_{\max} = \max_{k,r}\{D_{kr}, Z_r\}$. The resulting queueing network model has balanced demands, thus [9]

$$n_{\max} = \log(G_{\max}) = \lceil N \log D_{\max}(N + M + R) \rceil, \quad (2)$$

where $\lceil \cdot \rceil$ rounds up to the closest integer.

2.2 Established Algorithms

Computation of the value of the normalizing constant G of a queueing network model is usually of low importance on its own, unless one is interested in evaluating state probabilities. The goal of most algorithms is the computation of several performance indices, such as the mean throughput of a class X_r, the mean queue length $Q_{k,r}$, or higher-order moments. Computating such indices is straightforward if ones knows the corresponding normalizing constants. For example [20]

$$X_r(\vec{m}, \vec{N}) = \frac{G(\vec{m}, \vec{N} - 1_r)}{G(\vec{m}, \vec{N})}, \quad (3)$$

while

$$Q_{k,r}(\vec{m}, \vec{N}) = D_{k,r} \frac{G(\vec{m} + 1_k, \vec{N} - 1_r)}{G(\vec{m}, \vec{N})}. \quad (4)$$

Other important performance indices can be derived using these two [20], such as the mean response time $R_r(\vec{N}) = N_r / X_r(\vec{N}) - Z_r$, the utilization $U_{k,r}(\vec{N}) = D_{k,r} X_r(\vec{N})$, and the mean residence times $R_{k,r} = Q_{k,r}(\vec{N}) / X_r(\vec{N})$.

2.2.a Convolution Algorithm

The Convolution algorithm, first presented in [6], aims at evaluating the normalizing constant by using the equation:

$$G(\Delta\vec{m}, \vec{N}) = G(\Delta\vec{m} - 1_k, \vec{N}) + \sum_{r=1}^{R} D_{k,r} G(\Delta\vec{m}, \vec{N} - 1_r) \quad (5)$$

where $\Delta\vec{m} - 1_k$ denotes the removal of queue k from the network and similarly $\vec{N} - 1_r$ is a population vector with a job less of class r. The above equation is valid for any choice of queue k, thus allowing recursive solution to evaluate $G(\Delta\vec{m}, \vec{N})$. A variant of (5) where queues are added and not removed is used by the LBANC algorithm [11] and is called the Convolution Expression:

$$G(\Delta\vec{m} + 1_k, \vec{N}) = G(\Delta\vec{m}, \vec{N}) + \sum_{r=1}^{R} D_{k,r} G(\Delta\vec{m} + 1_k, \vec{N} - 1_r) \quad (6)$$

Convolution and LBANC have $O(N^R)$ complexity in both time and space with respect to a growth of the total population. This makes it unsuitable for models containing more than two or three classes and more than a few tens of jobs. Notice that LBANC is just an unnormalized version of the Mean-Value Analysis (MVA) algorithm, presented in [24], thus also MVA suffers from the same computational issues.

2.2.b Recursion by Chain Algorithm (RECAL)

In RECAL, first presented in [12], each queue is completely associated with a specific class, the self-looping class, composed by jobs looping through the service station forever. The RECAL algorithm uses the following Population Constraint as a fundamental recurrence equation

$$N_r G(\Delta\vec{m}, \vec{N}) = Z_r G(\Delta\vec{m}, \vec{N} - \vec{1}_r) + \sum_{k=1}^{M} [(m_k + \Delta m_k) D_{k,r} G(\Delta\vec{m} + \vec{1}_k, \vec{N} - \vec{1}_r)] \quad (7)$$

which can be solved recursively for $G(\Delta\vec{m}, \vec{N})$. Time and space complexity grows as $O(N^M)$ which makes RECAL suitable only for networks with small numbers of queues.

2.3 Method of Moments

The Method of Moments (MoM) is a queueing network performance evaluation algorithm introduced in [7,9]. MoM simultaneously uses (6) and (7) into a system of linear equations to efficiently compute the normalizing constant. For a model with R classes, this proceeds as follows. We first define a basis of higher-order moments as the set

$$V(\vec{N}) = v(\vec{N}) \cup v(\vec{N} - 1_1) \cup ... \cup v(\vec{N} - 1_{R-1}),$$

where

$$v(\vec{n}) = \{G(\Delta\vec{m}, \vec{n}) | \textstyle\sum_{k=1}^{M} \Delta m_k \leq R, \Delta m_k \geq 0\}$$

is the set of normalizing constants of all possible models with population $\vec{n}$ and $l = \sum_k \Delta m_k$ added queues for all $0 \leq l \leq R$. It has been proven in [9] that there always exist two square matrices $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$, defined by the coefficients of the Population Constraint (7) and the Convolution Expression (6), that relate using linear expressions all and only the normalizing constants in the bases $V(\vec{N})$ and $V(\vec{N} - \vec{1}_R)$. That is, $V(\vec{N})$ satisfies the matrix difference equation

$$\boldsymbol{A}(\vec{N})V(\vec{N}) = \boldsymbol{B}(\vec{N})V(\vec{N} - \vec{I}_R), \quad (8)$$

where both $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$ have order independent of the total population N. If $\boldsymbol{A}(\vec{N})$ is not singular, then the basis $V(\vec{N})$ can be computed recursively from $V(\vec{N} - \vec{1}_R)$ as:

$$V(\vec{N}) = \boldsymbol{A}^{-1}(\vec{N})\boldsymbol{B}(\vec{N})V(\vec{N} - \vec{I}_R) \quad (9)$$

If $N_R = 0$, we can apply a similar recursion to the class $R-1$ that terminates when $V(\vec{0})$ is reached, since it contains only normalizing constants $G(\Delta\vec{m}, \vec{0}) = 1$ or $G(\Delta\vec{m}, \vec{0} - \vec{1}_R) = 0$ for any class r. In the special case of $\boldsymbol{A}(\vec{N})$ being singular, other approaches need to be used [7,9]. The main benefit of the above algorithm is that MoM's time complexity is log-quadratic in the total computation size, whereas its size complexity is log-linear, thus providing a sharp complexity gain compared to established algorithms for closed queueing networks. Knowledge of $V(\vec{N})$ and $V(\vec{N} - \vec{1}_R)$ implies the knowledge of $G(\vec{N})$, $G(\vec{N} - 1_r)$, $1 \leq r \leq R$ and $G(\vec{1_k}, \vec{N} - 1_r)$, $1 \leq k \leq M$, $1 \leq r \leq R$. These normalizing constants are sufficient to calculate performance indices and the remaining can be used to estimate variances and covariances of the performance metrics [9].
The general steps in the MoM iterations can be summarized in the following algorithm [9]:

Method of Moments (MoM)

```
1: Compute $V(N_1, 0, \dots, 0)$ using an efficient single class
   method, such as Convolution [6] or LBANC [11].
2: for $r = 2$ to $R$ do
3:     Initialize elements of $V(N_1, \dots, N_{r-1}, 0, \dots, 0)$
       based on the previous class results and the
       termination conditions $G(\Delta\vec{m}, \vec{0}) = 1$ and
       $G(\Delta\vec{m}, \vec{0} - \vec{1}_r) = 0$.
4:     for $n_r = 1$ to $N_r$ do
5:         Setup and evaluate (9) obtaining
           $V(N_1, \dots, N_{r-1}, n_r, 0, \dots, 0)$
6:     end for
7: end for
8: Return mean performance indices using and the
   normalizing constants in $V(N_1, \dots, N_R)$ and
   $V(N_1, \dots, N_R - 1)$.
```

In practice, MoM can achieve better performance than all existing methods for sufficiently large networks, e.g. a few tens of jobs and more than two or three classes [9].

3. PARALLEL SOLVER DESIGN

Because the computational costs of MoM mainly depend on the time needed to solve the linear systems, the proposed parallel tool can greatly enhance the scalability of the method. Observe first that, since the normalizing constant is the result of sums and products of service demands, if the latter are rational numbers it is always possible to express $G(\vec{N})$ as a rational number itself. Moreover, if the demands are scaled by $N_1!N_2! \cdots N_R!$, it can be easily shown that $G(\vec{N})$ always reduces to an integer value provided that all demands are integers as well. Thus, by a change of scale we can always represent the MoM linear system as a linear system with integer coefficients.

Stemming from the integer or rational representations, our parallel solver converts the MoM linear system into a *set* of linear system having coefficients that are defined over different moduli. Such linear systems are independent of each other and can then be solved in parallel. Finally, the results are combined using the classic Chinese Remainder Theorem [18]. The current paper focused on this congruence technique, which can be combined with sparse matrix optimizations as those proposed in the original paper. Below we summarize the main operations and definitions needed to carry out the above parallelization steps.

The Chinese Remainder Theorem states that, if we have k positive integers $n_1, n_2, \dots, n_k$ ($n_i \in \mathbb{N}^*, \forall i = 1, 2, ..., k$), $\mathbb{N}^* = \mathbb{N} - \{0\}$, which are pairwise coprime and which represent the moduli used to define the independent linear systems, then, for any given integers $a_1, a_2, \dots, a_k$, there exists an integer x solving the system of simultaneous congruences

$$x \equiv a_1\ (mod\ n_1),\ x \equiv a_2\ (mod\ n_2),\ \dots,\ x \equiv a_k\ (mod\ n_k)$$

Furthermore, it is proved that all solutions x to this system of equations are congruent modulo the product $N = n_1 n_2 \dots n_k$, i.e.,

$$x \equiv y (mod\ n_i) \Rightarrow x \equiv y (mod N)$$

This guarantees that the solutions $a_1, a_2, \dots,$ of independent modulo linear systems can be combined to reconstruct the original solution x of the non-modular linear system, provided that the entries of the latter are smaller than N.

The extension of the above approach to negative numbers is simple and we take the following sign convention

$$x = n_i a_i + r,\ sgn(r) = sgn(x),\ |r| \leq |n_i|$$

Other conventions are possible, but the above is often preferable for computational efficiency.

3.1 Solution Method

In this section we present the procedure followed to solve the linear system $\boldsymbol{A}\vec{x} = \vec{b}$ using modular arithmetic in detail and explain adaptations specific to MoM. Adaptations are needed as the parallelization of the solution is not straightforward: several properties are required to hold for the Chinese Remainder Theorem to be applied as we explain below.

3.1.a Determining a lower bound for the product of moduli

As observed in the previous section, the product of the moduli used in the finite field linear systems determines the maximum value of the normalizing constants that we can represent in the original linear system. Thus, the choice on how many linear system consider in parallel throughout the MoM recursion and the values of their moduli has to be done carefully in order to avoid situations where results cannot be reconstructed correctly. Since the value of the normalizing constant values grows throughout the recursion, it is non-trivial to decide the product of the moduli. Below we discuss the technique implemented in our tool.

To determine a lower bound for the product of moduli we need to find a number $M_{thres} \in \mathbb{N}$ such that [22]:

$$M_{thres} = 2\max(|d|, n(n-1)^{(n-1)/2}\max(\boldsymbol{A})^{n-1}\max(\vec{b})), \tag{10}$$

where n the number of equations and $d = det(\boldsymbol{A})$. Also, for a matrix $\boldsymbol{B} = [b_{ij}]$, we denote by $\max(\boldsymbol{B})$ the maximum absolute element $\max(\boldsymbol{B}) = \max_{i,j}|b_{ij}|$. A similar definition exists for the maximum absolute element of vector $\vec{b}$. The issue with the above formula is that it requires the value of $d = det(\boldsymbol{A})$ which is not easy to compute on the MoM linear system which may have thousands of equations and variables. Furthermore, no explicit bounds for the absolute value of the determinant of matrix $\boldsymbol{A}$ have been developed so far, thus one can only bound this value using inequalities. For this purpose, we here use Hadamard's Inequality [22], which for real valued matrices consisting of n column vectors v_j states that

$$|det(\boldsymbol{A})| \leq \prod_{j=1}^{n} \|v_j\|, \tag{11}$$

where $\|v_j\|$ the Euclidean norm. Furthermore, if entries on the $n \times n$ matrix are bounded by the maximum element $\max(\boldsymbol{A})$, then $|det(\boldsymbol{A})| \leq n^{n/2}\max(\boldsymbol{A})^n$. Therefore, (10) becomes

$$\begin{aligned} M_{thres} = 2\max(&n^{n/2}\max(\boldsymbol{A})^n, \\ &n(n-1)^{(n-1)/2}\max(\boldsymbol{A})^{n-1}\max(\vec{b})) \end{aligned} \tag{12}$$

This equation can be computed in $O(n^2)$ operations.

If we had to solve a single linear system, this is the point where the search for a lower bound for M_{thres} would end. However, as the next step, the determination of the moduli, comes at a high computational cost, we propose to perform it only once, at the start of the execution. Consequently, we have to adapt (12) to provide a threshold value big enough to accommodate for all possible MoM linear systems that will arise throughout the recursion.

Let us first observe that, from (6)-(7), that the maximum absolute value of an element encountered in matrices $\boldsymbol{A}(\vec{N})$ or $\boldsymbol{B}(\vec{N})$ is

$$\max(\boldsymbol{A}(\vec{N})) = \max(\boldsymbol{B}(\vec{N})) = D_{\max}(\max(\vec{m}) + R), \tag{13}$$

where $\max(\vec{m})$ denotes the maximum queue multiplicity value in the model. Throughout the rest of the paper we always assume that $\max(\cdot)$ returns the maximum of the *absolute* values of its matrix or vector argument. Moreover, a maximum limit on the value of normalizing constant G can be obviously derived from (2) as

$$G_{\max} = base^{N\lceil \log_{base}(D_{\max}(N+M+R))\rceil}, \tag{14}$$

where N is the total population of the queueing network model and *base* the base of the numerical system used. Furthermore, an adequate bound for the maximum element of vector $\vec{b}$ has to be constructed, where $\vec{b}$ is derived from the right hand side of (8) as $\vec{b} = \boldsymbol{B}(\vec{N})V(\vec{N} - \vec{I}_R)$. $V(\vec{N} - \vec{I}_R)$ is actually the collection of normalizing constants for a network involving one population less. It is known that the values of normalizing constants are at least weakly increasing as the total population increases. As the true maximum element of matrix $\boldsymbol{B}$ remains below $\max(\boldsymbol{B})$, which is constant throughout the execution, one can then bound the maximum element of all $\vec{b}$ vectors as the maximum element of vector $\vec{b}_{\max} \approx \boldsymbol{B}(\vec{N})V(\vec{N})$. This vector would be the vector $\vec{b}$ we would have to evaluate if we had to process a queueing network with one population more in the last class. Therefore, we can claim that $\max(\vec{b}_{\max}) \leq n\max(\boldsymbol{B})G_{\max}$, where n the linear system order. As $\max(\vec{b}) \leq \max(\vec{b}_{\max})$, we can consider in the general case that:

$$\max(\vec{b}) = n\max(\boldsymbol{B})G_{\max} \tag{15}$$

We are now able to present a final formula for M_{thres}, guaranteed to be able to be computed at the beginning of MoM's execution and be adequate to be used throughout all iterations. This final formula can be constructed, in the case of binary representation where $base = 2$, by starting from (12) and substituting (13), (14) and (15), leading to:

$$\begin{aligned} M_{thres} = 2\max\{&n^{n/2}K^n, \\ &n^2(n-1)^{(n-1)/2}K^n 2^{N\lceil \log_2(D_{\max}(N+M+R))\rceil}\}, \end{aligned} \tag{16}$$

where $K = D_{\max}(\max(\vec{m}) + R)$.

3.1.b Determining the moduli

After determining M_{thres}, we need to choose a set of moduli $m_1, m_2, \ldots, m_s \in \mathbb{N}$, subject to $M = m_1 m_2 \cdots m_s \geq M_{thres}$, which define the set of linear systems to be solved in parallel. Such moduli must be coprime to apply the Chinese Remainder Theorem [18] yielding the extra condition

$$gcd(m_i, m_j) = 1 \; \forall i \neq j$$

Lastly, the following property must hold: $gcd(M, d) = 1$. Satisfying and verifying the two last conditions can be time consuming. To ease this procedure, we select as few primes as possible, thus opting for the largest possible moduli [18], [22], [19]. Such primes satisfy the coprimality requirement by definition and greatly reduce the probability of encountering a case where $gcd(M, d) = 1$ [23], [4], [25].

In general, there are many alternative strategies one can use in order to select the moduli that result in the minimum total runtime. In the case of a shared-memory multiprocessing system, experiments given in Section 5 show that the most efficient strategy is to select at least as many moduli as the number of processing cores; this approach minimizes the number of linear systems that need to be solved in parallel and maximizes the percentage of runtime in which the processors are doing useful work. In theory, this strategy scales well with the number of cores if we disregard the time needed for primality testing of each candidate prime modulo. In reality, the time needed for such tests, e.g. when using Java's arbitrary large prime generator, does not scale well sufficient for increasing bit length of the primes. Therefore, we introduce an adaptation of this strategy when dealing with very large models. In the following pseudocode, b_i denotes modulo length and n_i the number of moduli. Intuitively, we start from an initial estimation of b_i and halve it as many times as needed to satisfy $b_i \leq b_{thres}$. At each halving the number of residual linear system doubles, as the product of moduli must remain of constant length.

Modulo Size Selection

```
1: $\langle b_0, n_0 \rangle := \langle \lceil \frac{\log_2(M_{thres})}{n} \rceil, n \rangle$, where $n$ the number of
   initially desired moduli (number of processors)
2: while $b_i > b_{thres}$
3:     $\langle b_{i+1}, n_{i+1} \rangle := \langle \lceil \frac{b_i}{2} \rceil, 2n \rangle$
4:     $i := i + 1$
5: end while
```

In our implementation, $b_{thres} = 2,500$. The total number of moduli selected will be $s = n_k$. This approach minimizes the amount of time needed to select the moduli but produces, at the same time, more in number yet simpler residual linear systems. From experimental results it is verified that this approach and the bit length threshold value of 2,500 may produce better results in practice. Selecting smaller moduli does not improve performance, as the number of residual linear systems that has to be solved in parallel in such cases becomes excessively large.

3.1.c Formulating residual systems of equations

In this step we produce s new linear systems, one for each selected modulo $m_1, \ldots, m_s$, named as residual linear systems. More specifically, for each modulo m_k a new residual matrix $\boldsymbol{A}_k$ is calculated, where every element $a_k(i,j)$ of it is evaluated as $a_k(i,j) = a(i,j)\ mod\ m_k$, where $a(i,j)$ is the respective element of the matrix $\boldsymbol{A}$. A similar definition is used for the residual vector $\vec{b_k}$: $b_k(i) = b(i)\ mod\ m_k$, where $b(i)$ is the i^{th} element of the vector $\vec{b}$. From a design point of view, there are two important factors that influence the efficiency of the algorithm in practice. Firstly, the specific behavior of the modulo operation used when evaluating negative elements: common case for MoM is for the matrix $\boldsymbol{A}$ to contain small negative numbers; if the modulo operation does not allow for negative results, then these small numbers are "mapped" to much greater ones. This is inefficient, as the time needed for arithmetic operations on such numbers is influenced by the numbers' length. On the other hand, this is not the case if one allows for negative modulo results which is the option chosen in our tool.

3.1.d Solving the residual systems

At this point, each one of the k linear systems must be solved. In general one could use several different exact solution techniques, such as Gaussian Elimination or LU Decomposition. For this work we used Gaussian Elimination, however the same solution technique can be applied using a different method depending on the problem requirements, the hardware architecture and the impact of possible optimizations. Two operations must be performed. Firstly, the value of each unknown of the residual solutions vector $\vec{x}_k$, where $\boldsymbol{A}_k \vec{x}_k = \vec{b}_k$, should be calculated. This can be accomplished by applying the Gaussian Elimination algorithm on a not square matrix. Secondly, the value of the residual determinant d_k of $\boldsymbol{A}_k$ must be obtained. One can easily evaluate the determinant of a square triangular matrix using $O(n)$ operations, as the product of the diagonal elements; however, in the general case $\boldsymbol{A}_k$ at this stage may not be square, as there may be need to perform row and column deletion to accommodate for indeterminable elements; deletion is preferred to multiplication with the transposed matrix, as for large matrices this technique may be costly. One should note that it is not that the matrix $\boldsymbol{A}_k$ needs inherently to be non-square. This matrix is actually just a simple formalism used to represent a collection of equations and can be transformed to an equivalent square matrix that defines the same linear system. Therefore, we can define a canonical form of the linear system: $\boldsymbol{A}_C \cdot \vec{x} = \vec{b}_C$. This canonical system is defined by a square matrix $\boldsymbol{A}_C$ which has full rank (*i.e.* all rows and all columns are linearly independent) and defines a right-handed coordinate system in the corresponding Euclidean space with dimension equal to the rank. In such a case, its determinant is always positive. This way one can disentangle the exact computation of the residual determinant d_k from the actual residual matrix $\boldsymbol{A}_k$. Both linear systems, *i.e.* the one defined by $\boldsymbol{A}_k$ and the corresponding canonical one, would yield the same results if solved; therefore, we can suppose that the system is actually in the canonical form and expect a positive determinant. If it is negative we just negate it. If it is zero, then the initial linear system cannot be solved. It is important to clarify that the actual canonical system is never used or calculated and is merely a theoretical tool, verifying the correctness of the solution approach and limiting the simplifications one can introduce. To conclude, after Gaussian Elimination is performed on the $m \times l$ matrix $\boldsymbol{A}_k$, we calculate determinant as $det(\boldsymbol{A}_k) = |\prod_{i=1}^{l} a_{ii}|$. This value is the same as the one we would have acquired from the canonical full-rank linear system, as we have transformed the non-square matrix into row echelon form.

3.1.e Recombining the results

By this step, a residual determinant d_k and a residual vector of solutions $\vec{x}_k$ has been determined for each of the k residual systems. In order to arrive to the final solution vector $\vec{x}$, one must perform the following operations ([18], [19] and [22]):

1. Multiply each element of each solution vector $\vec{x}_k$ by the respective residual determinant d_k. This results in a new vector $\vec{y}_k$.

2. Recombine the s residual determinants d_k to obtain the final determinant d, using any conversion algorithm. If $d = 0$ then the linear system cannot be solved.

3. Recombine the elements that are in the same positions of the s vectors $\vec{y}_k$, *i.e.* correspond to the same unknown, to obtain one vector $\vec{y}$.

4. Obtain the final vector of solutions $\vec{x}$ as $\vec{x} = \frac{\vec{y}}{d}$, accounting for uncomputable elements as well.

The Chinese Remainder Theorem guarantees that we can obtain a final number f from a set of residue numbers f_i; the actual algorithm we use to derive the exact number that satisfies the theorem and the congruences is the single-radix conversion algorithm [19]. Let the modular multiplicative inverse of a number $a \in \mathbb{Z}$ modulo a number $n \in \mathbb{N}$ be a number $x \in \mathbb{Z}$ such that $ax\ mod\ m = 1$, or equivalently using the congruence relation: $ax \equiv 1 (mod\, n)$. The modular multiplicative inverse can be calculated using the Extended Euclidean algorithm [13]. Stemming from this definition the conversion algorithm is as follows [19].

```
Single-Radix Conversion Algorithm
1: Calculate $c_i = \frac{M}{m_i}\ mod_{-1}\ m_i$ for $i = 1, 2, \dots, s$, where
   $mod_{-1}$ the modular multiplicative inverse operator.
2: Calculate the final number $f \sim \{f_1, f_2, \dots, f_s\}$ as
   $f = \left[\sum_{i=1}^{s} \left(\frac{M}{m_i} c_i f_i\right)\right]\ mod\ M$.
```

3.1.f Pseudocode

The pseudocode of the parallel implementation when used in conjunction with MoM is the following:

```
Parallel Multi-modular Solver
1: Compute lower bound for the product of moduli,
   $M_{thres}$, using 16
2: Determine moduli $m_1, \dots, m_s$
3: for each MoM iteration
4:     Formulate residual systems of equations
5:     Solve using Gaussian Elimination
6:     Combine residual solutions using the Single-Radix
       algorithm
7: end for
```

3.2 Further Results

3.2.a Maximum Number of Moduli

It is not beneficial to select infinitely large moduli. In practice, we limit the maximum length at $b_{thres} = 2,500$. Suppose now that the initial desired length of each modulo is just above that threshold level, for example it is $b_{thres} + 1$. This would mean that our algorithm will double the number of selected moduli and halve their length, which would become $\lceil \frac{b_{thres}+1}{2} \rceil = 1,251$. One could claim that not eough primes exist with such bit length; therefore their product may not be larger than M_{thres}, thus causing the procedure to fail in this worst case.

We will investigate the maximum M_{thres} value guaranteed to be feasible by the current algorithm. To achieve this we can use the prime bounding function $\pi(x)$, which gives the number of primes less than or equal to x. Several approximate formulas exist for this function; $\pi(x)$ has been bounded in [16] as:

$$L(x) = \frac{x}{\ln(x)}(1 + \frac{1}{\ln(x)}) < \pi(x) < \\ < \frac{x}{\ln(x)}(1 + \frac{1}{\ln(x)} + \frac{2.51}{\ln^2(x)}) = U(x) \quad (17)$$

The above bound holds for $x \geq 355,991$, which is clearly more than sufficient for our case. We can claim that when selecting primes according to their bit length b, at least n_b primes, where $n_b = L(2^{b+1} - 1) - U(2^b)$ must exist with this bit length. The above formula is true for $b \geq 19$. As 2^{b+1} is a composite number, we can simplify the above formula for n_b as $n_b = L(2^{b+1}) - U(2^b)$, which can be written as:

$$n_b = \frac{2^{b+1}[\ln(2^{b+1}) + 1]}{\ln^2(2^{b+1})} - \frac{2^b[\ln^2(2^b) + \ln(2^b) + 2.51]}{\ln^3(2^b)}, \quad (18)$$

which yields $n_{1,251} \simeq 2.17 \cdot 10^{749}$. It is clear that this number of moduli (*i.e.* number of necessary residual systems) is far above what computer systems can handle; the limit exists only in theory and does not impose any practical limitations.

3.2.b Growth Rate of M_{thres}

The rate of increase of M_{thres} directly influences the scalability of the algorithm. If we use arbitrarily large moduli, it defines how fast the bit length of each modulo grows, whereas if we limit the maximum length of the selected moduli, *i.e.* if we may select more moduli than the number of available processors then it defines the rate at which we should increase the number of processing cores in order to maintain the same efficiency as the problem size increases. Regarding the modular linear system solver, complexity stems from the order of the linear system. M_{thres} is given by (16), with the length of its binary representation with regard to n being $l_{thres} = \lceil \log_2(M_{thres}) \rceil = O(n \log(n))$.

3.2.c Complexity

The asymptotic computational cost, in terms of number of operations, of the linear system solution algorithm using modular arithmetic remains the same as using the established approach. We will ignore the cost of primality testing, as modern algorithms are of polynomial complexity (*i.e.* the AKS Primality Test [1], which runs in $O(\ln^{6+\varepsilon}(p))$ [21], or even $O(\ln^{4+\varepsilon}(p))$ [14], where p the exact number to be tested). This means that their complexity may be able to be reduced further in the future. The most complex operation that needs to be performed by the Linear System solver presented in this paper is Gaussian Elimination, which can be performed using $O(n^3)$ operations, same as the simple serial one; however, constants may vary. Space requirements of the parallel linear system solver are linearly related to the number of residual systems. For example, if s residual systems are formed, the total space requirement will be $(s + 1)O(n^2) = O(n^2)$.

4. TOOL IMPLEMENTATION

4.1 Main Features

Among the main features of the implemented program are the ability to process queueing network model files adhering to a common format and the evaluation of the queueing network model using Convolution, RECAL of Method of Moments. Fast non-recursive implementation of RECAL is available, as well as the ability to select which solver, *i.e.* the single-threaded or the multi-threaded modular one, is required to be used in conjunction with the MoM algorithm. However, the application can automatically select the best one to solve a particular queueing network based on the produced linear system size and experimental calibration. The needed number of threads can be specified by the user. Apart from normalizing constants, evaluation of performance indices, such as mean throughput of a class r (X_r), and mean class r queue length for queues of type k (Q_{kr}) is possible. At the end of the evaluation, time and memory usage statistics are provided.

4.2 Abstract Architecture

Each one of the above top-level operations of the tool are performed by one or more subsystems. Several more subsystems exist as well, to support non component-specific operations used widely. In general, each subsystem depends only on subsystems of lower levels, thus eliminating circular dependencies, *i.e.* cases where a system depends on its subsystem from an operation and the subsystem depends on the upper level system for another operation. It is best to avoid writing code that contains such dependencies, as they reduce the code maintainability and the ease at which improvements and modifications can be applied.

The *Control* subsystem is responsible for recognizing the user's input and calling and initializing the other subsystems in the correct order and using the correct arguments. It is also fundamental for the correct communication with the user, which can be a human or an external application.

The *Queueing Network Model* component is responsible for parsing a model description from an input file and for storing the queueing model representation in memory. Furthermore, it provides stable interfaces for the other components to read and set model properties and values. Lastly, it can perform some basic tasks on the model, such as calculation of a maximum limit $G_{\max}$ for its normalizing constant. The role of this component is very important, as it abstracts the actual model representation in memory and the actual input file syntax from the rest of the implementation. Thus, if one wanted to provide support for different input files, only this part of the implementation needs to be modified and tested.

The *Abstract Solver* component is responsible for computing the normalizing constant of a queueing network model and for computing its performance indices, *i.e.*, the mean throughputs and the mean queue lengths. The abstract solver can use many different algorithms to evaluate a model and can use other components.

The *Exceptions* component supports the *Abstract Solver* and the *Queueing Network Model* components by providing *Exception* objects that are contains meaningful messages for the user, shall an error occur during any of the above operations.

4.3 Code Design

The final implementation consists of more than 5,000 lines written in Java 6. The code is split among 29 Java classes. We tried to keep a simple design and maximize code re-use and abstractions, hoping to assist further contributions, and produce an even more complete tool. On the topmost level, the source code is organized in 6 Java packages.

The `Control` package contains the Main class and is responsible to interpret the user's input and calling the other subsystems appropriately. It implements the Control abstract component. This package also contains the code that auto-selects the most applicable MoM version depending on the model details, as well as an a control class which can be used as an API by an external program.

The `DataStructures` package contains classes defining objects that are used to represent several of the data structures used throughout the program. Examples of such objects are the queueing network models, a class defining arbitrarily long rational numbers etc. Its classes support the operations of both the Abstract Solver and the Queueing Network Model abstract components.

The `Exceptions` package, defines classes used to enhance the exception handling of the code and to provide meaningful error messages to the user, should an error occur. It corresponds to the Exceptions abstract subsystems.

The `LinearSystem` package, which contains classes that implement the various different solvers of linear systems $\boldsymbol{A}\vec{x} = \vec{b}$. The objects of this package are only needed when the user selects the Method of Moments algorithm to evaluate the system. Tasks that are part of a linear system solver but can be implemented in parallel are contained in this package as distinct special objects. Architecturally, this package supports the operations of the Abstract Solver component.

The `QueueingNet` package contains classes implementing three distinct algorithms to evaluate a queueing network: Convolution, RECAL and MoM. For the case of RECAL two implementations exist: a simpler implementation using recursive calls and a more efficient for larger populations non-recursive one. These classes correspond to the Abstract Solver component. All different solvers implement are designed in such a way so as to maximize code-sharing. Furthermore, they all implement a common interface, in order to be handled by the `Control` package in the same way.

The `Utilities` component contains the implementation of functions that are used throughout the implementation and do not correspond to queueing networks or linear systems in particular. For example, such methods may include code that performs operations on or between matrices, code to discover the memory usage and timing routines. This package supports all other components.

5. EXPERIMENTAL EVALUATION

5.1 Methodology

To quantify the impact of any proposed modification and improvement and to investigate how theoretically better solutions would measure in the real-life scenarios, a set of different queueing network models were used for experimental evaluation These models are of comparable complexity to the ones pertaining to full-scale J2EE applications [9], defining several workload profiles. Parameters range from $R = M = 2$ to $R = M = 5$ (16 cases); each case can come at three load levels: total population of 100, 300 or 900 jobs, making a total of 48 cases. Populations are split equally among classes, with rounding to the nearest integer. This configuration was chosen as it is the worst case for the MVA algorithm. Population 100 corresponds to the low load scenario, population 300 to the medium load and population 900 to the high load one. Delay times are set to zero; this may benefit the runtime of recursive algorithms.

The test machine had 2 Intel Xeon E5540 Processors (8M Cache, 2.53 GHz, 4 physical cores each plus Hyper-Threading) and 32GB RAM. Ubuntu 9.04 with GNU/Linux kernel 2.6.28-15-generic(x86_64) and Oracle's JVM, Version 6 Update 16 was used, with maximum runtime limit was set to 1 hour. In order to eliminate the effects of page swapping on the runtime, the maximum memory available for the JVM was limited to 25 GB. In order to evaluate the performance of the parallel implementation of MoM, instances using 2, 4 and 8 threads were examined. Unless declared otherwise, the time to evaluate a single model corresponds to the time interval starting from the moment the queueing model description input file is loaded in memory to the moment all performance indices, *i.e.* mean throughputs and mean queue lengths, are returned.

5.2 Convolution

As shown in Table 1, the Convolution algorithm can only be applied on the most simple models. Its performance quickly degrades as the number of classes or queues increases; the same degradation is noticed as the total population increases. This can be explained by its high time

Table 1: Runtime and memory needed to evaluate a queueing network with $R = 4$, $M = 3$ and $N_{tot} = 100$. This case shows how faster all MoM implementations are in comparison with Convolution and RECAL. Even though RECAL achieves a good result in this case, it cannot scale efficiently as the number of jobs increases. Low space complexity of all MoM versions are highlighted.

Algorithm	Runtime (sec)	Memory Usage
Convolution	131.6	3.4 GB
RECAL	18.3	382 MB
MoM (serial)	7.4	0.9 MB
MoM (2 threads)	3.4	2 MB
MoM (4 threads)	3.7	3 MB
MoM (8 threads)	5.0	5 MB

and space asymptotic complexity. A key result highlighting the algorithm's prohibitive complexity is the fact that from all the test-case models, only the ones with 2 classes were able to be evaluated within the time and space constraints for all three load levels. In total, 42% of the tested models were feasible.

5.3 RECAL

The performance of the RECAL algorithm is hampered by its high asymptotic complexity for both space and time. Even though RECAL in general it performs better than Convolution, as 46% of the tested models were feasible, it still does not scale sufficiently well. The runtime is very sensitive to the number of queues M. RECAL is only able to evaluate within the time and space constraints all three load cases in only the simplest test-case model, which contains 2 classes and queues ($R = M = 2$). Convolution is more sensitive to increments in the number of classes, and thus is able to solve less systems with high number of classes and lower number of queues; in such cases RECAL is better, at least for small populations (low and medium load cases).

5.4 Method of Moments

5.4.a Memory Usage

As observed in Table 1, the primary computational advantage of MoM is its low space complexity. Even when the parallel modular linear system solver is used, which is characterized by higher space requirements that the serial version, the memory usage is proportional to the number of selected moduli and thus does not increase substantially.

5.4.b Runtime Results and parallelization

As it is evident from Table 1, the MoM algorithm is in almost all cases the fastest way to evaluate the tested queueing network models, enabling evaluation of up to 90% of them within the experiment constraints. There are several important factors that influence the performance of both the serial and the parallel versions of MoM that need to be clarified.

The primary computationally intensive part of the MoM algorithm, *i.e.* the part that most of the runtime is spent, is the linear system solver. A breakdown of the runtime for several execution instances is given in Table 2, which proves that the runtime is dominated by the time needed to solve the linear system. Therefore, the complexity of a queueing network model can be quantified, as far as MoM is concerned, by the order of the corresponding linear system it produces: this order is $\binom{M+R}{R} R$.

Table 2: The runtime percentage breakdown on different sub-operations, namely the time needed by the MoM solver itself (MoM), by the moduli selector (Moduli), by the linear system solver (LS) and the matrix-vector multiplication on the right part of (8) (Multiplier). The data pertains to the model with $R = M = 4$; however, qualitatively similar results are obtained from any MoM execution.

Load - Threads	MoM	Multiplier	LS	Moduli
Low - 1	0,27%	0,24%	99,49%	0,00%
Low - 2	0,75%	0,77%	90,66%	7,81%
Low - 4	0,74%	0,92%	97,05%	1,30%
Low - 8	0,70%	0,69%	97,47%	1,14%
Medium - 1	0,10%	0,41%	99,49%	0,00%
Medium - 2	0,14%	0,65%	96,92%	2,28%
Medium - 4	0,20%	1,14%	94,25%	4,40%
Medium - 8	0,21%	1,06%	98,16%	0,57%
High - 1	0,04%	0,83%	99,14%	0,00%
High - 2	0,03%	0,60%	97,37%	2,01%
High - 4	0,05%	1,07%	98,01%	0,87%
High - 8	0,07%	1,62%	95,55%	2,76%

The total runtime of the serial MoM is almost proportionally related to the number of jobs circulating the network. This is because the MoM algorithm must solve one linear system per job and all such linear systems have the same size. The only difference between them is that the magnitude of the contained values grows monotonically, thus making each subsequent solution harder.

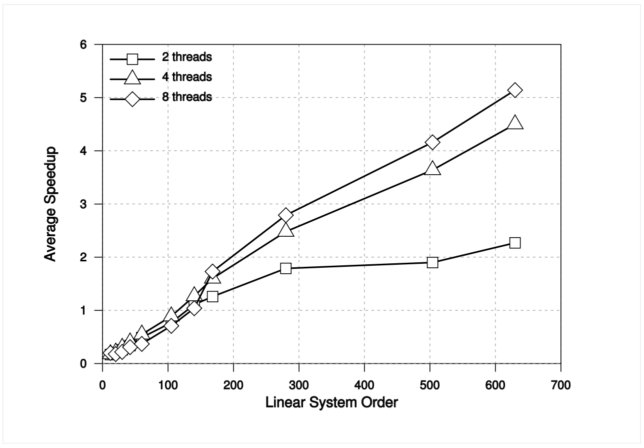

Figure 1: Average achievable speedup for each matrix size when evaluating all models using the parallel algorithm with 2, 4 and 8 threads.

Which algorithm is better clearly depends on the matrix size that corresponds to the model used. For models sufficiently small to have linear system order less than approximately 120, the parallel may result in a slowdown instead of a speedup in the average case. The converse is true for larger linear systems. This is attributed to parallelization

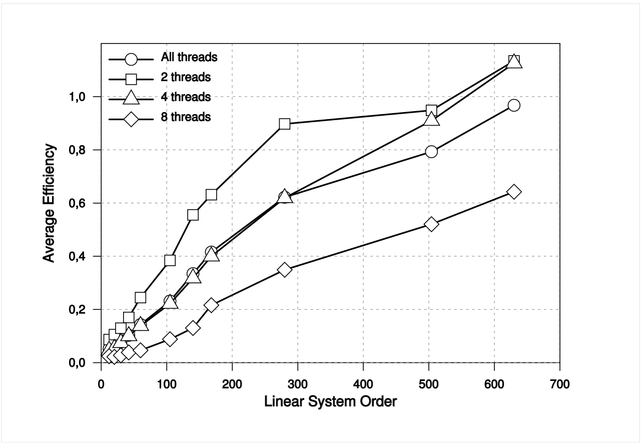

Figure 2: Average efficiency achieved for each matrix size when evaluating all models using the parallel algorithm with 2, 4 and 8 threads.

overhead, including modulo selection, primality tests and context switches, and to the modulo selection strategy that may limit the maximum length of each modulo, resulting in the selection of more moduli than available processing cores in some cases. The relation between the average speedup attainable for each matrix size when using the parallel algorithm with 2, 4 and 8 threads is presented in Fig. 1, whereas Fig. 2 presents the average efficiency. Overall, the performance scales well, both as an average for all threads and in the case of 8-threaded version. For medium to large complexity models, depending on the particular case and configuration, we achieve almost ideal efficiency. It is preferable to use as many threads as the maximum number of physical cores of the machine - 8 in our case. The 8-threaded version performs similarly to the best one for small matrix sizes and is the best for larger matrix sizes, *i.e.* with order of more than 140.

A speedup more than the ideal one can be observed at the largest matrix size. This is attributed to the fact that this last point is based on the runtime result of only one model, as the serial algorithm was able to evaluate only the corresponding low load case. Therefore, from a statistical perspective, this single result is not as representative as the others of the totality of the test cases and the different models. Furthermore, similar superlinearities, *i.e.* parallelization efficiencies greater than 1, have been reported in [18]. In our case, superlinearities stem from the relatively worse performance of the serial MoM: even though the number of steps of Gaussian Elimination are polynomially bounded, these steps can deal with exponentially long operands in the worst case [17]. This may not cause increased runtime when one uses fixed-size floating point arithmetic, but it can affect the speed of exact computations. It is clear that the serial MoM, which contains arbitrarily large values in the linear system, reaches the point where this phenomenon becomes measurable before the parallel MoM; in the case of the parallel MoM the values in each residual system are bounded by the respective modulo.

5.4.c Limitations of MoM implementation

All implementations of MoM that failed to evaluate a test model did so as a result of this time constraint and not due to their memory usage. However, as it is important to know the maximum model that the MoM implementations can evaluate, various larger models were tried independently of the main testing procedure. These models had equal number of classes and queues ($R = M$), as this is the hardest configuration for MoM. The larger model that did not fail on the test machine when imposing the maximum memory usage limit of 25GB was the one with $R = M = 7$, far above the limitations of the other established algorithms. This involved creating and solving a linear system with order equal to $24,024$. The next linear system with $R = M = 8$ was infeasible, as it required solving a linear system of order $102,960$. Memory was inadequate to store the approximately 10^{10} arbitrarily long rational numbers.

5.4.d Experimental Growth Rate of G

The normalizing constant G grows, theoretically, exponentially fast, leading to a linear growth rate for its length [9]. We can verify this growth in the experimental results, using as a base the number of digits of the low load case throughout the test-sets and representing the length corresponding to medium and high load profiles with regard to this base. The average growth rates for all test-cases are presented in the following table, where one can verify the linear growth:

Table 3: Average Relative Length of G

Low Load	Medium Load	High Load
1	3.02	9.06

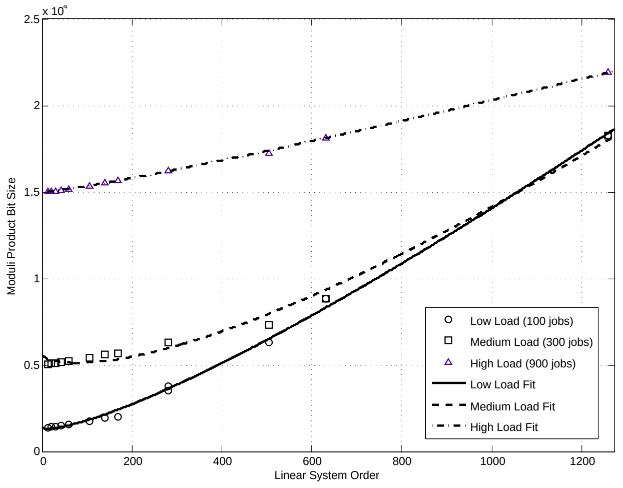

Figure 3: Relation between the linear system matrix size and the bit size of M_{thres}. Fitting the data using an $n \log n$ model verifies the theoretical result of $O(n \log n)$ rate of growth.

5.4.e Experimental Growth Rate of M_{thres}

Fig. 3 presents the relation between the bit size $b = \lceil \log_2(M_{thres}) \rceil$ of M_{thres} and the linear system order . The theoretical growth rate has been proven to be $O(n \log n)$ and is confirmed by the experimental results. However, the growth curve is sufficiently close to being linear, proving the scalability of the parallel algorithm. Even if the optimal maximum bit length per moduli remains constant in the future, one would have to add more processors at an $O(n \log n)$ rate in order to maintain the same parallelization efficiency as complexity increases.

6. CONCLUSION

6.1 Summary of Current Approach

Overall, the parallel solver exhibits very good results, both regarding theoretical-algorithmic qualities and performance. First and foremost, it exhibits a very good speedup and efficiency when compared to the serial linear system solver. Secondly, it exhibits good scalability properties: we need to add $O(n \log n)$ new moduli in order to maintain the same runtime for a harder problem. Usage of more processors decreases the parallelization overhead related to primality testing, as it enables selection of smaller moduli.

On the other hand, when evaluating complex models the M_{thres} value can become large sufficient to require solution of multiple residual linear systems per processing core due to the maximum modulo size limit. However, it may still be beneficial to solve more smaller residual systems in parallel than to use the serial algorithm, as the cost of solving a more complex, in terms of number length, linear system grows faster. This behavior is a direct result of the fine-tuning of the parallel solver and the modulo selection strategy.

6.2 Future Work

In this paper we have proposed the integration of a parallel linear system solver using exact residual arithmetic in the Method of Moments algorithm [9]. Our method proves the satisfactory improvements introduced by this parallelization in the performance of the MoM algorithm and the exact large linear system solution in general.

To further improve performance, one can integrate the existing parallel solver with an optimized version of MoM [7], [8]. The main computational cost of queueing network evaluation using MoM stems from the linear system solution; therefore, any decrease in linear system order is expected to produce a cubic ($O(n^3)$) decrease in algorithm runtime. Furthermore, the usage of sparse matrices should be investigated, aiming to reduce memory requirements. In this case, matrix re-ordering using the Reverse Cuthill-McKee algorithm [15] or the Approximate Minimum Degree algorithm [2] could be required to reduce the number of operations and "fill-ins".

Parallelization overhead can be further reduced by using a precomputed prime moduli list, possibly in conjunction with an increase in the maximum modulo size. Faster primality tests should be investigated, such as an adaptation of the Agrawal-Kayal-Saxena (AKS) primality test [14] that features $O(\ln^{4+\varepsilon}(p))$ time complexity. The possibility of producing a stricter bound for the M_{thres} value should be studied; it may involve producing a stricter bound for the maximum normalizing constant of the queueing network model than the current one or better bounding the value of the maximum possible determinant of the linear system.

7. REFERENCES

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] S. Balsamo. Product form queueing networks. *Lecture Notes in Computer Science*, pages 377–402, 2000.

[4] E. H. Bareiss. Computational solutions of matrix problems over an integral domain. *Journal of the Institute of Mathematics and its Applications*, 10:68–104, 1972.

[5] M. Bertoli, G. Casale, and G. Serazzi. Java modelling tools: an open source suite for queueing network modelling and workload analysis. In *Proc. of QEST*, pages 119–120, 2006.

[6] J. P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Comm. of the ACM*, 16(9):527–531, 1973.

[7] G. Casale. An efficient algorithm for the exact analysis of multiclass queueing networks with large population sizes. In *Proc. of joint ACM SIGMETRICS/IFIP Performance*, pages 169–180. ACM Press, 2006.

[8] G. Casale. CoMoM: efficient class-oriented evaluation of multiclass performance models. *IEEE Transactions on Software Engineering*, 35(2):162–177, 2009.

[9] G. Casale. Exact analysis of performance models by the method of moments. *Under submission*, 2009.

[10] K. Chandy and D. Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Comm. of the ACM*, 25(2):126–134, 1982.

[11] K. Chandy and C. Sauer. Computational algorithms for product form queueing networks. *Comm. of the ACM*, 23(10):583, 1980.

[12] A. Conway and N. Georganas. RECAL—a new efficient algorithm for the exact analysis of multiple-chain closed queuing networks. *Journal of the ACM (JACM)*, 33(4):768–791, 1986.

[13] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms*. p. 859–861, MIT press, 2001.

[14] R. E. Crandall and J. S. Papadopoulos. On the implementation of AKS-class primality tests. *Unpublished (http://images.apple.com/ca/acg/pdf/aks3.pdf)*, 2003.

[15] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69: Proc. of the 1969 24th national conference*, pages 157–172, 1969.

[16] P. Dusart. Autour de la fonction qui compte le nombre de nombres premiers. *PhD. Thesis, Univ. de Limoges*, 1998.

[17] X. G. Fang and G. Havas. On the worst-case complexity of integer gaussian elimination. In *Proc. of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 28–31, 1997.

[18] H. González and E. Martinez. A parallel code for solving linear system equations with multimodular algebra. *Revista Investigacion Operacional*, 23(2):175–184, 2002.

[19] C. K. Koc. A parallel algorithm for exact solution of linear equations via congruence technique. *Computers & Mathematics with Applications*, 23(12):13–24, 1992.

[20] S. Lam. A simple derivation of the MVA and LBANC algorithms from the convolution algorithm. *Computers, IEEE Transactions on*, 100(11):1062–1064, 2006.

[21] H. W. Lenstra and C. Pomerance. Primality testing with gaussian periods. *FST TCS 2002: Found. of Software Techn. and Theoretical Computer Science*, pages 1–1, 2002.

[22] M. Morhác. Error-free algorithms to solve special and general discrete systems of linear equations. *Applied Mathematics and Computation*, 202(1):1–23, 2008.

[23] M. Newman. Solving equations exactly. *Journal of Research of the National Bureau of Standards, 71B*, 4:171–179, 1967.

[24] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM (JACM)*, 27(2):313–322, 1980.

[25] D. M. Young and R. T. Gregory. *A survey of numerical mathematics*. Dover Pubns, 1988.