Imperial College London
Department of Computing

Efficient Analysis of IT Sizing Models

by
Michail A. Makaronidis

Submitted in partial fulfilment of the requirements for the
MSc Degree in Advanced Computing of Imperial College London

September 2010

1. Abstract

Capacity evaluation and planning usually relies on producing a closed queueing network model and predicting its performance indices. Until recently, analytical modelling of such networks was performed by using algorithms such as Convolution, RECAL or the Mean Value Analysis (MVA), prohibiting evaluation of systems offering multiple service classes to hundreds or thousands of users, a case commonly encountered in modern applications. Acknowledging this demand for performance evaluation, the Method of Moments (MoM) algorithm was introduced and addressed this problem. It was the first exact algorithm able to solve closed queueing networks with large population sizes. The MoM algorithm relies on the exact solution of large linear systems with integer coefficients of thousands of digits.

The primary focus of this project is the production of an optimised implementation of the MoM algorithm as well as the algorithmic design, analysis and implementation of an exact parallel solver for linear systems it defines. Parallelisation is introduced in both algorithmic and implementation level by performing the operations over residue number systems and recombining the results by application of the Chinese Remainder Theorem. Various techniques have been introduced at all stages of this parallel solver to achieve improved time complexity and practical performance. Moreover, the procedure features several methods to achieve high robustness during error propagation when encountering a series of ill-conditioned linear systems which may be defined by the MoM implementation.

Furthermore, a comprehensive test-set that corresponds to the requirements of modern application has been designed and is used to compare the performance of the different algorithms and configurations. Theoretical and experimental results regarding MoM and solver scalability are also presented. The overall result proved the improved performance of both the MoM algorithm over the established ones, namely Convolution and RECAL, and the parallel solver designed as part of this project over the serial one. The parallel MoM is the most efficient approach for evaluating models with several classes and queues and many hundreds of users.

Lastly, much attention was paid in the efficient architecture and implementation design from a software engineering perspective, as the current implementation will be a part of the Java Modelling Tools (JMT) set of applications and may be augmented and improved in the future.

Key words: Capacity Planning, Closed Queueing Networks, Exact Linear System Solution, Multiclass Networks, Multimodular Algebra, Parallel Linear System Solution, Performance Analysis, Residue Number System

“What is the wisest thing? Number.”

Pythagoras

This project would not have been accomplished without the comments, continuous advice and suggestions of my supervisor, Dr. Giuliano Casale. I would therefore like to wholeheartedly thank him for the countless hours he spent discussing the problems encountered and giving invaluable feedback for my work. I am deeply indebted for his support and personal effort.

Michail A. Makaronidis

2. Contents

3. Introduction

3.1. Performance Modelling

One of the principal factors that influences the entire life cycle of a computer system is **performance**; it influences the design, development, configuration and tuning of the system. Other factors that play a decisive role in a system's evaluation are cost, usability, available features, reliability and security [1]. Performance can be evaluated by three main methods:

- **Measurement**, which is possible only once a system is built and able to run.
- **Simulation Modelling**, which is a technique with a very wide range of applications. However, it is characterised by the possibility of high development and computational cost to achieve accurate results – i.e. a trade-off exists between accuracy and cost [2].
- **Analytical Modelling**, in which the model should satisfy a set of assumptions and have certain properties, which will make the subsequent formulation of mathematical equations characterising it behaviour feasible [2]. If this is the case, the system's evaluation can be tacked with typically less computational cost than the Simulation Methods [3].

In this project, we will only consider this last method of Analytical Modelling.

To describe and analyse the performance of a resource sharing system, queueing network models have proved to be very helpful, as they are characterised by adaptability and a wide range of capabilities. Their modelling power is not limited to computer systems; other established areas of application are production systems and communication networks [2]. All these real-life systems share a common attribute: they contain one or more resources (machinery, communication links, CPU time, I/O throughput, etc.) that are of finite capacity and therefore need to be shared between the jobs (product parts, people, telephone calls, programs) that use them. Queueing networks, their subcategories and significant results will be presented in more detail on the next chapter, "Introduction to Queueing Networks" (p. 11).

In this project, we consider a special subcategory of queueing networks named **product form queueing networks**. These networks have several specific properties, allowing a simple closed-form expressions that lead to the development of efficient algorithms to evaluate their performance. However, as we see during the following chapters and presentation of the algorithms, this efficiency does not lead to sufficiently small execution times for the majority of the models representing systems in real-life size. This is owed to the fact that most of these algorithms rely on a recursive computation of a significant quantity of the network model, called "**normalising constant",** which assures that the state probabilities sum to one in the Markov chain underlying the model. Other methods instead compute directly mean performance indices, such as the mean queue length - without need of probabilities. This recursive computation needs to be ran over an exponentially large state space and thus leads to prohibitive computational cost even for small network sizes.

This situation is often made worse by the inability of exactly evaluating the performance of systems serving customers belonging to different workload classes, i.e. customers that cause a different burden to the system depending on their type. Such systems are usually represented using special "**multi-class**" models and are of huge importance for the modelling and the evaluation of the performance of computer servers and multi-tier web applications, which are the most common IT architectures behind modern web sites; it is common in these systems that applications or user requests may have significantly different costs, depending on their type (HTTP request, FTP request, etc.) [4]. Evaluation of the performance characteristics in these models may easily become intractable for large population sizes [4], where hundreds or thousands of users belonging to different classes access the services of a system. This large population is however essential to estimate the performance of modern applications.

Acknowledging this problem and the demand for performance evaluation tools that exists from the designers of current applications, an algorithm called **Method of Moments** was proposed, first in [4] and further optimised in [3]. The algorithm focuses exactly in reducing the cost of computing normalising constants, and therefore performance metrics, for closed multiclass product form queueing network models with large population sizes, making such evaluation feasible. One of the goals of this project is the implementation of an optimised version of the MoM algorithm.

It is also important to highlight that all these modelling, estimation and analysis of system performance would not be so prevalent should advanced analytic and simulation engines not exist. There is a wide variety of software packages which allow the definition and exact or approximate solution of many different queueing networks, such as open, closed, mixed, single or multi-class and others. These engines, typically coupled with a GUI front-ends for increased usability, are in wide use by members of the academia, the industry and the sector's specialists. An example of such a complete GUI with an analytic engine is JMVA, which uses the JMT engine to perform Mean Value Analysis (MVA) of queueing network models [5].

3.2. Aim of this Project

The **main aims** of the project are:

1. to **produce an optimised Java implementation of the "Method of Moments" algorithm**, as presented in [4],
2. to **develop and integrate in the algorithm a parallel solver for linear systems with integer coefficients over finite fields** (modular arithmetic); this is needed as none of the existing solvers available for solution of linear systems using modular arithmetic are able to cope with the arbitrary large numbers that will be encountered during the evaluation of queueing networks with large population sizes,
3. to **compare** in terms of run-time, memory usage and other metrics **the MoM algorithm** developed **with the established ones** in the field of product form queueing networks' evaluation, such as Convolution and RECAL, as these are presented in [4] and
4. to **implement an interface enabling usage of the produced algorithm by JMVA**, which is an application of the Java Modelling Tools [5].

This will assist in reducing a single linear system with arbitrary large coefficients to many simpler ones. It can be seen that the process is inherently parallelisable, as each produced linear system can be solved independently of the others. This is of great importance, as no parallel prototypes currently exist and parallelisation is expected to improve the algorithm's performance significantly.

3.3. Significance of this Project

As it can be ascertained from this brief introduction, this project and the MoM algorithm in general are of primary importance, because they tackle the problem of the unavailability of a fast algorithm for evaluating the performance of large IT models. In particular, there are two main areas which this Project can influence:

- **Firstly**, this project **builds upon the most efficient family of algorithms**

currently available for evaluating closed multi-class product form queueing networks, namely MoM, by adding a new parallel solver with the aim to further increase speed and efficiency. This approach under no way trivial, as no such solver exists that is able to cope with the demands of MoM. Furthermore, it is necessary to produce such a version that concentrates on speed, as the currently established algorithms are of no practical use when it comes to solving networks with large population sizes.

- **Next**, because it **integrates this** state of the art **algorithm and implementation with an open-source software framework** (JMVA and JMT) widely used by practitioners, students and researchers for evaluating such networks. JMT is a very popular suite for capacity planning, and has been downloaded many thousands of times since its first release in 2006 [6]. Therefore, such an implementation and analysis has the potential to influence many users throughout the world.

3.3.a Technical Challenges

There are several important technical challenges that this project must overcome:

1. Abstract interfaces used to communicate with the JMVA GUI need to be defined. This is a complex procedure due to the size and the continuous development of the JMT project. The JMVA's code and requirements must first be understood. The produced code should be able to interface with the JMVA, without redundancy and prioritising ease of understanding by others and ease of adaptability to changes. This is actually a requirement for the code of the entire project, as it will be released in the public domain and many different users and developers should be able to easily understand, maintain and augment it.
2. The Java implementation of the currently established reference algorithms, namely RECAL and Convolution. These algorithms are in wide use today, together with LBANC and the MVA algorithm, however their significant computational and memory cost highlights the importance of an efficient implementation. Only an efficient implementation could produce results of sufficiently good quality to be able to be compared to the ones of the MoM algorithm and lead to reliable conclusions.
3. As it will be seen during the presentation of the algorithms in “Chapter 5”, the Java implementation of the MoM algorithm is significantly more complicated than that of the established algorithms. Apart from the general algorithm and the data structures needed for the implementation, a particular challenge stems from the need to solve exactly a linear system of equations dealing with extremely large numbers. It is estimated that the approach followed in the current project, i.e. reducing each complicated linear system to many easier ones, will be more efficient than the existing MoM approaches and prototypes presented in [4], as it can be more easily parallelised. This particular challenge is interwoven with one of the “Scientific Challenges” presented below, namely the theoretical and algorithmic formulation of such a solver.

3.3.b Scientific Challenges

The two main scientific challenges which need to be tackled for the successful completion of the project are the following:

- The complete comparison in terms of computational requirements (run time, memory usage and other metrics) of the implemented algorithm and the reference implementations of the established ones. Such a comparison must be done in a scientifically rigorous manner and every result worthy of remark should be explained. Detailed results should be provided for several sample queueing networks of varying complexity and population and discussed.

- The formulation and analysis of a parallel algorithm to solve large linear systems of equations with integer coefficients using modular arithmetic. As it has been already noted, no such solver exists that can cope with the great order of magnitude of the numbers arising during convolution equations, so a significant challenge arises in defining, efficiently implementing, testing and discussing this algorithm. The need for parallelisation further increases said challenge. Lastly, the restriction that the solver should be portable between different computer architectures and operating systems greatly reduces the number of different tools, libraries and techniques available.

  This approach of solving such a large linear system using modular arithmetic is completely novel in the field of queueing networks. The main area of applications for such solvers until now was in the fields of cryptography. It is speculated that the parallelisable nature of the solver algorithm can result in an even more optimised version of MoM.

3.4. Summary of chapters to follow

Chapter 4 contains a brief introduction to queueing network models for system performance evaluation. Definitions and formulas for the main performance indices in networks containing a single or multiple service centres are presented. The important class of product form queueing networks is examined and the Jackson, Gordon-Newell and BCMP Theorems are stated and discussed.

Chapter 5 first focuses in the in-depth introduction to the notation and the main definitions results needed for the formulation of the existing algorithms and MoM. Afterwards, each of these main algorithms is presented in detail.

Chapter 6 contains a detailed presentation the algorithm used to solve exactly linear systems in parallel using modular arithmetic as well as several important related results.

Chapter 7 discusses the implementation's structure, the challenges faced and various optimisations that were introduced.

Chapter 8 presents the experimental method and results that is used to compare MoM to the other established algorithms. Furthermore, the advantages of the parallel solver are discussed and other important properties are verified.

Chapter 9 contains the most important conclusions that summarise the entire project. Current limitations and areas of future investigation are highlighted.

Chapter 10 contains a presentation of the software tools already available to analyse such networks – including JMT –, as well as the program's User's Guide.

4. Introduction to Queueing Networks

Queueing network models are widely used to evaluate the performance of congestion systems, i.e. systems that provide a finite resource or capacity to several users. Such systems include, among others, computer systems, communication networks and production systems. System performance evaluation using such models consists of the accurate representation of the systems as a model (definition and parametrisation) and its subsequent use to calculate **performance indices**, such as resource utilisation, system throughout and response time [2].

4.1. Queueing Systems

A model using a general service centre is described using:

- The arrival process of incoming users or customers,
- The service process,
- The queue's size (buffer where waiting customers are being hold),
- The scheduling algorithm of said queue and
- The set of available parallel service centres.

Using **Kendall's notation**, a queueing system is described as A/B/c, where A symbolises the arrival process, B the service process and c the number of available service centres. In this case, we assume an infinite queue size and “First Come First Served” (FCFS) scheduling. Two well known examples are the M/M/1 system, which has Poisson (Markov) arrival process, exponential (Markov) service process and one service centre, as well as the M/G/1 system which is the same, except for a non-Markovian (general) service distribution [2].

Systems like the M/M/1 and M/M/m are associated with a Markov process with a special structure, which is called a **birth-death process**. Intuitively, this process has states $\{0, 1, ...\}$, where each state denotes the number of jobs in the system, i.e. the queue's length including the job being served, if it exists. A birth consists of an arrival of a new job and a death of a completion of a job and its subsequent departure from the server. This birth-death process leads to a simple closed-form solution of the stationary state probabilities, thus the initial system can be easily evaluated using analytical expressions.

4.1.a Important Results for an M/M/1 System

An M/M/1 system like the one presented in Fig. 2 is characterised by an exponential arrival rate λ and service rate μ . Therefore, the **birth rate** (new arrival) is equal to λ and the **death rate** is equal to μ (new departure). The system is considered to be in the **stable** case if $\lambda < \mu$, or equivalently its **utilisation**:

$$\rho = \frac{\lambda}{\mu} < 1 \tag{4.1.1}$$

In the stable case the **stationary probability** $P(k)$ of the system having k customers – equivalently, the associated birth-death process being in state k – can be computed as $P(k) = \rho^k (1-\rho)$ for $k \geq 0$. It is clear that the stationary probabilities must sum to 1: $\sum_{k=0}^{\infty} P(k) = 1$.

The **average queue length**, including any task serviced, is calculated as $L=\frac{\rho}{1-\rho}$. The **mean response time**, which is the sum of the residual time of the task in service and the service times of the jobs in the queue, is given by the formula $W=\frac{1}{\mu-\lambda}$.

For a more detailed discussion of single server models, the reader can refer to a general reference book such as [7].

4.1.b Important Results for an M/M/m System

An M/M/m system is associated with a birth-death process which has **birth rate** λ and **death rate** $min(k+1,m)\mu$. This results can also be explained intuitively, as an M/M/m systems essentially consists of m parallel servers. The aggregate arrival rate is λ and the rate of returning from state $k+1$ to k is $(k+1)\mu$, up to a maximum of $m\mu$ if all servers are busy.

Similarly to the case of M/M/1 systems, an M/M/m system is considered **stable** when the **utilisation** $\rho=\frac{\lambda}{m\mu}<1$. Then, the stationary queue length probability $P(k)$ of the system having k customers is given by the following formula:

$$P(k)=\begin{cases} \frac{P(0)(m\rho)^k}{k!}, & 1\le k\le m \\ \frac{P(0)(m^m\rho^k)}{m!}, & k>m \end{cases} \tag{4.1.2}$$

The value of $P(0)$ can be computed using the normalising equation $\sum_{k=0}^{\infty} P(k)=1$. The **average queue length** is given by $L=m\rho+\frac{P(m)\rho}{(1-\rho)^2}$ and the **mean response time** can be calculated as:

$$W=\frac{1}{\mu}+\frac{P(m)}{m\mu(1-\rho)^2} \tag{4.1.3}$$

4.1.c Summary of Notation

The notation followed in this Report is summarised in the following table:

Quantity	Notation
Arrival Rate	λ
Service Rate	μ
Utilisation	ρ

Quantity	Notation
Probability of having k customers	$P(k)$
Average Queue Length *(including task serviced)*	L
Mean Response Time	W

4.2. Queueing Networks

More complicated systems can be described as a network of resources, using a collection of interconnected single service centres as those examined previously. **A queueing network can be generally defined by defining three main parameters:**

- **Service centres**, thus designating for every service centre the service time, the maximum queue length, the scheduling method used and the number of servers it contains.
- **Customers**, which can be described by their quantity in closed networks, by their arrival process to each of service centre for the open networks and also by their type for multi-class networks.
- **Network Topology**, which describes in which way the multiple service centres of the network connect with each other and how the customers can move between them. The most significant network topologies are the tandem, cyclic and central server ones. The central server topology is presented in Fig. 1 (p. 13).

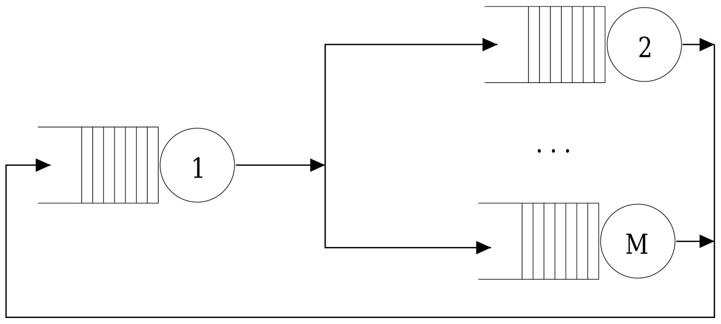


Fig. 1: *An example of the central server topology.*

4.2.a Single and Multi-class Networks

Depending on whether multiple different **classes** (types) of customers are defined, a network can be considered as multi-class. In such a queueing network, different behaviour and demands of each customer type can be modelled, as well as different external arrival processes, service demands and types of network routing.

Interwoven with the definition of classes for customers lies the notion of **chains**. A chain actually consists of a set of classes that represent different phases or steps in the processing of a system for a given customer. A particular customer, part of a chain, can move from one class to another after being processed by a particular service centre, according to certain rules. Chains can be used to model different customer routing behaviour dependent on the past history; for example, a customer representing a job in a computer system could be modelled as requiring two steps by using classes: one for program loading and one for execution [2]. After being processed by a particular service centre, customers can move between these two classes and possibly follow a different routing afterwards depending on their current class. Multi-class models aim in representing more precisely the initial system's behaviour and therefore leading to more accurate and detailed performance indices.

However, we do not consider chains and multi-chain networks in this project, as it has been proved in [8] that multi-chain networks can be reduced to multi-class ones by definition of some **equivalence classes**.

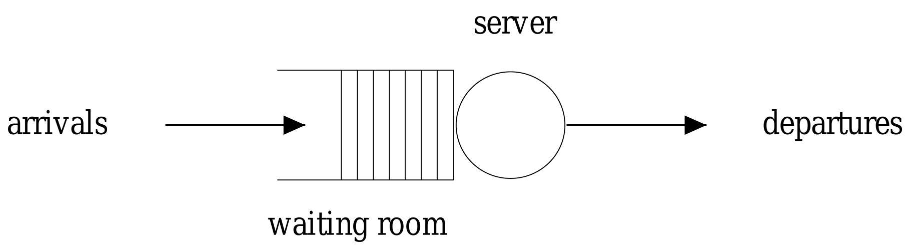


Fig. 2: *A single service centre queueing system.*

4.2.b Open and Closed Queueing Networks

Networks can be divided depending on whether external arrivals or departures are allowed. If external arrivals and departures are possible, then the resulting network is considered **open**, whereas if they are not possible it is considered as **closed**.

4.3. Fundamental Analysis of Queueing Network Models

The goal of analysing a system as a queueing network model is the evaluation of not only the performance of the system's components – modelled by different service centres –, but also its performance as a whole. Queueing network analysis is based on defining and analysing the stochastic process that governs the model and is usually a Discrete Space Continuous Time Homogeneous Markov Process [2].

The state of this stochastic process is typically defined as the number of customers in each queue. The network's behaviour can be evaluated by analysing the evolution and characteristics of this process. In a network containing M service centres, n_i denotes the number of customers in each centre. Then, the joint queue length can be represented as a vector $\vec{n}=(n_{1,} \ldots , n_M)$ and its stationary distribution as the row vector $\vec{P}(\vec{n})=\vec{P}(n_{1,} \ldots , n_M)$. By $\vec{P}$ we denote the stationary state probability vector of the Markov process and by $\boldsymbol{Q}$ its transition rate matrix. If the network is stable, i.e. if the utilisation of all service centres is below 1 , then the stationary state probability $\vec{\pi}$ can be, in principle, straightforwardly evaluated by solving the following linear system, known as **global balance equations:**

$$\vec{P}\boldsymbol{Q}=\vec{0} \tag{4.3.1}$$

and normalising the solution to sum to 1. $\vec{0}$ denotes the row vector, all elements of which are equal to 0 . Then, the needed performance metrics can be derived. Particularly, one must first determine the set of all states, then determine the transition rates between each pair of states and finally write this as an equation system for a steady state Markov chain $\vec{P}\boldsymbol{Q}=\vec{0}$ and solve for $\vec{P}$.

Even though the procedure may sound simple, it is only feasible only for small networks. Solving the produced linear system is characterised by high computational complexity. The cardinality of the process' state space, i.e. the number of different states and balance equations, can make the linear system's solution infeasible in practice. For example, the state space of an open queueing network is infinite; an exact solution can be obtained only in special cases, depending on the particular structure of the matrix $\boldsymbol{Q}$. Closed queueing networks are associated with a state space which grows exponentially with the number of network parameters (number of service centres, customers and classes) [2].

Being able to define multiple classes and large population size is not a luxury; it is necessary to accurately model and evaluate modern IT and other systems. Furthermore, a trade-off exists between the needed accuracy and the computational cost of the model's analysis. Queueing networks are however regarded as powerful tools and are widely used, as in several cases their solution can be obtained in a more simple and efficient manner. One such case is the case of the **operational analysis** of a queueing network, which only provides asymptotic bound on performance indices and therefore is appropriate only as an initial approach. Analysis of **product form queueing networks** however can provide efficiently more accurate results than these of the operational analysis method.

4.3.a Product Form Queueing Networks

Using product form queueing networks performance indices such as queue length distribution, average response time, resource utilisation and throughput can be evaluated for each component and for the complete system. The contribution of product form queueing networks lies in the fact that, if certain assumptions regarding the system's characteristics hold, then the stationary joint queue length probability can be defined using the underlying Markov process in a product form solution:

$$P(\vec{n})=\frac{1}{G}V(n)\prod_{i=1}^{M} g_i(n_i) \qquad (4.3.2)$$

As before, $\vec{n}=(n_{1,}\dots,n_M)$ is the row vector of queue lengths. By G we denote the normalising constant, n is the total network population, n_i denotes the number of customers in each centre and the functions V and g_i depend on the network's parameters and the type of service centre i, $i=1,2,\dots,M$ respectively. For open networks $G=1$, whereas for closed ones $V(n)=1$. The function g_i is the stationary queue length distribution of node i in isolation for the case of an open network.

For the case of networks with multiple classes of customers, R denotes the number of these classes and S the network state, which includes the customer population at each service centre. Then, for a multi-class product form network, the stationary state probability distribution $\boldsymbol{P}$ is given by the following formula:

$$P(n_{1,}\dots,n_M)=\frac{1}{G}\prod_{r=1}^{R} V_r(K_r)\prod_{i=1}^{M} g_i(n_i) \qquad (4.3.3)$$

In the previous formula K_r is the population in class r, $r=1,2,\dots,R$ and the function V_r depends on network parameters.

Product form queueing networks can be analysed using algorithms that exhibit polynomial time complexity in the number of network components [2]. This good balance between accuracy and computational cost is the main reason of their prevalence.

A product form solution only holds under the specific assumptions of quasi-reversibility and partial balance [2]. Quasi-reversibility means that for a given service centre, the current state, the past departures and the future arrivals are mutually independent. Partial balance is a prerequisite for quasi-reversibility and states that the probability flux, i.e. the time average transition rate, out of a state S due to arrivals of type r customers is equal to the probability flux in state S due to departures of type r customers. For more details the reader can refer to [9].

Two important results for product form queueing networks are **Jackson's Theorem** for open networks and the **Gordon-Newell Theorem** for the closed ones.

Jackson's Theorem

A very important result for open queueing networks was presented by Jackson in [10]. Jackson's Theorem specifies the conditions, under which a product form solution in open queueing networks exist. These are the following: [11]

- The network can have any (unlimited) number of customers.

- Every node in the network can have Poisson arrivals from outside the network.
- A customer can leave the system from any node.
- All service times are exponentially distributed.
- The queueing scheduling is FCFS.
- The service centre i consists of m_i identical servers, each with service rate μ_i

Then, if the network in stable, i.e. $\lambda_i < m_i \mu_i, \forall i = 1, 2, \ldots, M$ then the steady state probability of the network can be expressed as the product of the state probabilities of the individual nodes:

$$P(\vec{n}) = P(n_1, n_2, \ldots, n_M) = P(n_1) \cdot P(n_2) \cdot \ldots \cdot P(n_M) \tag{4.3.4}$$

If the service rates are constant, then $P(\vec{n}) = \prod_{i=1}^{M} (1 - \rho_i) \rho_i^{n_i}$.

Gordon-Newell Theorem

Gordon and Newell in [12] extended Jackson's Theorem for closed product form queueing networks. The same assumptions hold, with the added constraint that no customer can enter or leave the system, as is evident in closed networks. The Gordon-Newell Theorem then states that the steady state probability distribution exists and is given by:

$$P(\vec{n}) = P(n_1, n_2, \ldots, n_M) = \frac{1}{G} \prod_{i=1}^{M} \frac{D_i^{n_i}}{\prod_{j=1}^{n_i} a_i(j)} , \tag{4.3.5}$$

where $a_i(j)$ is the scaling factor of the exponential server i when its queue length is j and the D_i is the **service demand** at a queue i, i.e. the product between the average service time and the average number of visits of jobs at node i. The normalising constant G is defined by the following formula:

$$G = \sum_{\boldsymbol{n} \in S} \prod_{i=1}^{M} \frac{D_i^{n_i}}{\prod_{j=1}^{n_i} a_i(j)} \tag{4.3.6}$$

and is not easy to compute, because of the large state space. This is highlighted by the example at the end of this chapter (section 4.4).

BCMP Theorem

The BCMP theorem, named after the authors of the paper where it was first described, is an extension of Jackson's Theorem over a much larger class of networks. Furthermore, it defines the four types of the so-called BCMP queueing networks and expresses that the stationary state distribution is expressed as the product of the distributions of the single queues with necessary parameters and, in the case of closed networks, with the normalisation constant.

A network of M interconnected queues is known as a **BCMP network** if each of the queues is of one of the following four types ([2], [13]):

- Type I: Multi-class service centre with FCFS queueing discipline and exponential service time distribution, identical for all customer classes.
- Type II: Multi-class service centre with Processor Sharing (PS) scheduling and arbitrary phase type service time distribution, i.e. formed by a network of exponential stages.

- Type III: Multi-class service centre with infinite number of servers, i.e. IS scheduling, and arbitrary phase type service time distribution.
- Type IV: Multi-class service centre with Last Come First Served with pre-emptive resume (LCFS-PR) scheduling and arbitrary phase type service time distribution.

The requirement for a phase type service time distribution means that the service time must have rational Laplace transformations, namely of the form $L(s)=\frac{N(s)}{D(s)}$.

Furthermore, the following conditions must hold for the BCMP Theorem to be applied:

1. External arrivals to a node must form a Poisson process and
2. A customer leaving node i must either move to a new queue j with p_{ij} or leave the system with probability $1-\sum_{j=1}^{M} p_{ij}$, which is non-zero for some subset of the queues.

Then, the BCMP Theorem is states that for an open, closed or mixed queueing network in which each queue is of type I, II, III or IV, the steady state probability distribution is given by:

$$P(n_{1,}n_{2,}\ldots,n_M)=C\cdot P(n_1)\cdot P(n_2)\cdot\ldots\cdot P(n_M) \tag{4.3.7}$$

where C is a normalising constant. Extensions have been made for multi-class networks, as well as for state-dependent routing. For a multi-class network with R classes, the BCMP theorem is formulated as follows: [4]

$$P(\vec{S})=\frac{1}{G(\vec{N})}\left(\prod_{r=1}^{R}\frac{Z_r^{n_{0,r}}}{n_{0,r}!}\right)\left[\prod_{k=1}^{M}\left(n_k!\prod_{r=1}^{R}\frac{D_{k,r}^{n_{k,r}}}{n_{k,r}!}\right)\right] \tag{4.3.8}$$

Where $\vec{S}\in S(\vec{N})$, $n_k=\sum_{r=1}^{R} n_{k,r}$, Z_r the **mean delay** of a class r , i.e. the mean time before a job re-enters the network after its completion and $G(\vec{N})$ is the normalising constant which ensures that the above probabilities sum to one:

$$G(\vec{N})=\sum_{\vec{S}\in S(\vec{N})}\left(\prod_{r=1}^{R}\frac{Z_r^{n_{0,r}}}{n_{0,r}!}\right)\left[\prod_{k=1}^{M}\left(n_k!\prod_{r=1}^{R}\frac{D_{k,r}^{n_{k,r}}}{n_{k,r}!}\right)\right] \tag{4.3.9}$$

4.4. Example

In this example we will present the calculation of the normalising constant G for a very simple queueing network with $R=2$ classes, $M=2$ queues, $\vec{N}=[1,1]$ the population per class, $Z_r=0$ for all classes r and $D_{k,r}=1$ for all queues k and classes r .

To perform this calculation, we will use eq. (4.3.9) and enumerate all the state space. Substituting the values of Z_r and $D_{k,r}$ results in the much more simplified equation below:

$$G(\vec{N})=\sum_{\vec{S}\in S(\vec{N})} 1$$

Such a simplification is necessary, as this example aims at presenting the big state space of even this very simple model. However, this means that in our case, all different states are equiprobable. In particular, this model has 6 states, i.e. 6 different ways in which the different jobs can exist. The star (*) denotes the job which is at the first position of each queue.

State	**Queue 1**		**Queue 2**	
	Class 1 Jobs	*Class 2 Jobs*	*Class 1 Jobs*	*Class 2 Jobs*
1	1*	1	0	0
2	1	1*	0	0
3	1*	0	0	1*
4	0	1*	1*	0
5	0	0	1*	1
6	0	0	1	1*

Therefore, the final normalising constant G is equal to 6. If we did not know that the network's states were equiprobable, we would now use this normalising constant in conjunction with equation (4.3.8), in order to calculate the probability of encountering each state.

The number of states increases exponentially as the network becomes more complex. This is evident even in the simple example above, as we have to consider all different permutations of job orders for each queue. For example, addition of just one more job per class increases the total population from 2 to four jobs but increases the number of different states to 30.

5. Product Form Queueing Network Algorithms

5.1. Introduction

For the description of the following algorithms the notation used in [4] is followed. This notation is summarised in the following table:

Quantity	Notation
Number of classes	R
Number of jobs in class r	N_r
Network Population per class	$\vec{N}=(N_1,\ldots,N_R)$

Quantity	Notation
Total Number of jobs	N
Number of Queues	M
Service Demand of class r at queue k	$D_{k,r}$
Mean Delay of class r	Z_r

As **service demand** $D_{k,r}$ of a class r at a queue k we define the product between the average service time and the average number of visits of class r jobs at node k. **Mean delay** Z_r of a class r is the mean time before a job re-enters the network after its completion.

In this section, we will consider closed product form queueing networks consisting of load-independent queues servicing jobs according to a First Come First Served (FCFS), Processor Sharing (PS) or Last Come First Served with pre-emptive resume (LCFS-PR) basis. Essentially, these are the BCMP network types I, II and IV respectively, as defined in section "BCMP Theorem" (p. 16). In that section, we provided the equations (4.3.8) and (4.3.9) which describe the steady (equiv. equilibrium) state probability distribution of the model: [4]

$$P(\vec{S})=\frac{1}{G(\vec{N})}\left(\prod_{r=1}^{R}\frac{Z_r^{n_{0,r}}}{n_{0,r}!}\right)\left[\prod_{k=1}^{M}\left(n_k!\prod_{r=1}^{R}\frac{D_{k,r}^{n_{k,r}}}{n_{k,r}!}\right)\right], \tag{4.3.8}$$

where $\vec{S}\in S(\vec{N})$, $n_k=\sum_{r=1}^{R}n_{k,r}$ and $G(\vec{N})$ is the normalising constant which ensures that the above probabilities sum to one:

$$G(\vec{N})=\sum_{\vec{S}\in S(\vec{N})}\left(\prod_{r=1}^{R}\frac{Z_r^{n_{0,r}}}{n_{0,r}!}\right)\left[\prod_{k=1}^{M}\left(n_k!\prod_{r=1}^{R}\frac{D_{k,r}^{n_{k,r}}}{n_{k,r}!}\right)\right] \tag{4.3.9}$$

We can obtain an upper bound for the maximum number of digits of the normalising constant of any model. This can be done by replacing all service demands D_{kr} by $D_{max}=max_{k,r}\{D_{kr},Z_r\}$, therefore obtaining a queueing network model with balanced demands. For such models, a closed form expression for the maximum normalising constant G_{max} exists [3], therefore the maximum number of digits of G_{max} is equal to:

$$n_{max}=\log(G_{max})=N\log\left[D_{max}(N+M+R)\right] \tag{5.1.1}$$

There are two approaches regarding the analysis of product form queueing network models: it can either be performed using the Mean-Value approach, where one recursively computes mean queue lengths and mean throughputs, or by the normalising constant approach, which aims at the computation of $G(\vec{N})$. If this normalising constant is computed, the other performance indices such as utilisations and mean response times can be easily computed [14].

As $\Delta\vec{m}$ we denote a vector of non-negative integers. Then, the normalising constant $G(\Delta\vec{m},\vec{N})$ will refer to a model which differs from the original queueing network by having included $\Delta m_k \geq 0$ additional "replicas" of queue k, $\forall k=1,2,\ldots,M$.

5.2. Computing Performance Indices

Computation of the value of the normalising constant G of a queueing network model is usually of low importance on its own. What is more important is the computation of several performance indices, such as the mean throughput of a class or the mean queue length.

For example, the mean throughput of a class r can be computed as [14]:

$$X_r(\vec{m},\vec{N})=\frac{G(\vec{m},\vec{N}-1_r)}{G(\vec{m},\vec{N})} \tag{5.2.1}$$

Furthermore, the mean class r queue length for queues of type k can be given by [14]:

$$Q_{k,r}(\vec{m},\vec{N})=D_{k,r}\frac{G(\vec{m}+1_k,\vec{N}-1_r)}{G(\vec{m},\vec{N})} \tag{5.2.2}$$

Other important performance indices, such as mean response times or utilisations, can be obtained using the two above indices X_r and $Q_{k,r}$ [7]. However, even though computing these performance indices may have the same asymptotic complexity as computing the normalising constant G, in the other algorithms presented except Method of Moments the corresponding constant is much higher.

5.3. Established Algorithms

5.3.a Convolution

The Convolution algorithm, first presented in [15], aims at evaluating the normalising constant. This is accomplished using the following expression:

$$G(\Delta\vec{m},\vec{N})=G(\Delta\vec{m}-1_k,\vec{N})+\sum_{r=1}^{R}D_{k,r}G(\Delta\vec{m},\vec{N}-1_r) \tag{5.3.1}$$

where $\Delta\vec{m}-1_k$ denotes the removal of queue k from the network. The above equation is valid for any choice of queue k. Therefore, it can be solved recursively to evaluate $G(\Delta\vec{m},\vec{N})$. The computation uses the initial conditions $G(\vec{0},\vec{N})=\prod_{r=1}^{R}\frac{Z_r^{N_r}}{N_r!}$ and $G(\cdot,\vec{0})=1$.

It is evident from the formulation of this algorithm that it is characterised by high computational cost. Particularly, it has exponential complexity in the number of classes and polynomial in the total population – $O(N^R)$ in both time and space. In practice, this means that it is unsuitable for models containing more than two or three classes and more than a few tens of jobs.

Modern systems, which need models containing many different classes of users and several hundreds of jobs, cannot be evaluated using this algorithm.

5.3.b Recursion by Chain Algorithm (RECAL)

In this algorithm, first presented in [16], each queue is completely associated to a specific class, which is named the self-looping class. This class is composed by jobs looping through the service station forever. The RECAL algorithm uses as a recurrence equation a formula known as the **population constraint**:

$$N_r G(\Delta\vec{m},\vec{N}) = Z_r G(\Delta\vec{m},\vec{N}-\vec{1}_r) + \sum_{k=1}^{M}\left[(1+\Delta m_k) D_{k,r} G(\Delta\vec{m}+\vec{1}_k,\vec{N}-\vec{1}_r)\right] \quad (5.3.2)$$

The population control equation can be solved recursively to evaluate $G(\Delta\vec{m},\vec{N})$. The computation uses the initial conditions as Convolution: $G(\vec{0},\vec{N}) = \prod_{r=1}^{R}\frac{Z_r^{N_r}}{N_r!}$ and $G(\cdot,\vec{0})=1$. This algorithm has both time and space complexity of $O(N^M)$, which makes it suitable only for networks with small number of queues.

5.3.c Mean-Value Analysis Algorithm (MVA)

The Mean-Value Analysis algorithm, which was presented in [17], avoids the direct evaluation of the normalisation constant. It uses an iterative method to calculate mean queue sizes, mean waiting times and throughputs. As many loops as the number of the customers of any type who circulate in the network are performed. Traffic equations and Little's Law on a particular node and for a particular population size $k-1$ are connected, using the Arrival Theorem, with the same data in a network with population k. Its complexity grows exponentially with the number of classes.

5.3.d LBANC

The LBANC algorithm (Local Balance Algorithm for Normalizing Constants), which was first proposed in [18], is actually the un-normalised version of the MVA algorithm. They are therefore characterised with the same computational requirements and the only difference is that LBANC aims at computing normalising constants, while MVA directly computes mean performance indices recursively. The LBANC algorithm uses recursion to solve a variant of eq. (5.3.1), where the queues are added and not removed:

$$G(\Delta\vec{m}+1_k,\vec{N}) = G(\Delta\vec{m},\vec{N}) + \sum_{r=1}^{R} D_{k,r} G(\Delta\vec{m}+1_k,\vec{N}-1_r) \quad (5.3.3)$$

The above equation holds for any choice of queue k, $k=1,2,..,M$. This equation is named in [4] as the **Convolution Expression**. It is significantly more efficient than the Convolution algorithm for cases of $q \ll M$, where $q \leq M$ is the number of queue types for the current population. As is also the case with the MVA algorithm, LBANC's computational cost grows exponentially with the number of classes.

5.3.e Brief comparison

All of above described algorithms feature a high (exponential) complexity, either in the number of different classes (Convolution , MVA and LBANC algorithms) or in the number of queues (RECAL). For example, Convolution exhibits $O(N^R)$ space and time asymptotic complexity, whereas RECAL exhibits $O(N^M)$. Therefore, none of these algorithms is suitable to be used in the evaluation of the models used to represent modern systems. These models may contain hundreds or thousands of users and tens of different classes, facts that may make their computational cost prohibitive in practice.

For the special case of the MVA algorithm, heuristic techniques can be used to assist in reducing the search space and therefore the computational cost [17]. This approach is not so readily available in the other algorithms, as only MVA directly computes values that have a direct physical meaning in our model. However, these techniques still do not improve the worst case cost and in many practical cases they do not provide sufficient speedup.

The high computational cost associated with the existing algorithms highlights the importance of the Method of Moments algorithm, which is presented in the next section.

5.4. Method of Moments (MoM)

5.4.a Introduction

The Method of Moments (MoM) is a queueing network performance evaluation algorithm introduced in [4] and further optimised in [3]. The fact that the established algorithms already presented were unable to evaluate closed queueing networks with large population sizes led to the introduction of MoM, which is defined as a recursion of the higher-order moments of queue lengths, which it is proved that they determine the model performance at equilibrium. These are calculated at every iteration using a linear system of equations. Thus, exact analysis is available at a significantly reduced cost, compared to the MVA algorithm.

The higher-order moment approach means that in the resulting recursion we need to remove only one job at a time from the network and also the number of evaluated higher-order moments per step is fixed. Therefore, the algorithm needs to perform a number of steps that grows linearly with the population size; this is a unique advantage when contrasted with the exponential growing of the recursion tree in the established methods for evaluating queueing networks.

The algorithm's time complexity is log-quadratic in the total computation size, whereas its size complexity is log-linear.

5.4.b Presentation

For the formulation of the algorithm we follow the notation used in [4] and presented in the beginning of this chapter (p. 19).

Essentially, the MoM algorithm utilises the Population Constraint and the Convolution Expression formulas simultaneously to efficiently compute the normalising constant. For a model with R classes, we can define **basis of higher-order moments** (Fig. 3) as the set:

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

where $v(\vec{n}) = \left\{ G(\Delta\vec{m}, \vec{n}) \middle| \sum_{k=1}^{M} \Delta m_k \leq R, \Delta m_k \geq 0 \right\}$ is the set of normalising constants of all possible models with population $\vec{n}$ and $l=\sum_k \Delta m_k$ added queues for all $0 \le l \le R$.

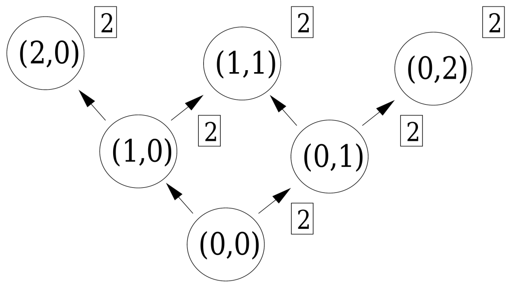


Fig. 3: *The basis $V(\vec{N})$ for $M=R=2$. Each node represents a different value of $\Delta\vec{m}$. The boxed numbers represent the number of constants $G(\Delta\vec{m},\cdot)$ in $V(\vec{N})$.*

If we define a vector ordering, then we will be able to define an **ordering of the basis of moments** as well. Therefore, it is decided that vectors of equal length are ordered ascendantly by the sum of their elements; if they have the same sum, the one with the leftmost non-zero element is considered greater. The canonical ordering of the basis of moments is achieved by first ordering the normalising constants $G(\Delta\vec{m},\vec{N})$ by their multiplicity vector $\Delta\vec{m}$ and afterwards sorting the constants with the same $\Delta\vec{m}$ according to their population vector $\vec{N}$.

It has also been proved in [4] that there always exist two square matrices $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$, which are defined by the coefficients of the Population Constraint (eq. 3.2.2) and the Convolution Expression (eq. 4.2.3) and which relate in a linear expression all and only the normalising constants in the bases $V(\vec{N})$ and $V(\vec{N}-\vec{1}_R)$. Equivalently, $V(\vec{N})$ satisfies the matrix difference equation:

$$\boldsymbol{A}(\vec{N})V(\vec{N})=\boldsymbol{B}(\vec{N})V(\vec{N}-\vec{I}_R), \tag{5.4.1}$$

where both $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$ have order independent of the total population N.

The pseudo-code given below (Algorithm 5.4.1), which generates the matrices $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$, has been adapted from [4]. It is of significant importance to note that the structure of the matrices can be updated from population $\vec{N}$ to $\vec{N}+\vec{I}_R$ by only changing the values N_R in $\boldsymbol{A}(\vec{N})$ to N_R+1_R, as this can reduce the number of operations necessary between iterations corresponding to the same class.

```
Algorithm 5.4.1: Generation of the matrices $A(\vec{N})$ and $B(\vec{N})$

Input: Population $\vec{N}$, Current class examined $s$
Let $A(\vec{N})$, $B(\vec{N})$ be square matrices of order $\binom{M+R}{R}R$ with element $a_{ij}$ on row $i$
        and column $j$
Initialise all $a_{ij}$, $b_{ij}$ with $0$
$i \leftarrow 0$
for $l$ from $0$ to $R-1$
        for all $\Delta\vec{m}$ with $M$ non-negative elements and $\sum_{k=1}^{M}\Delta m_k=l$
```

```
                    if $\sum_{k=1}^{M} \Delta m_k = R-1$
                    for $k$ from $1$ to $M$ /* Add CE for queue k */
                        $i \leftarrow i+1$
                        $j \leftarrow$ index of $G(\Delta\vec{m}+1_k, \vec{N})$ in $V(\vec{N})$
                        $a_{ij} \leftarrow 1$
                        $j \leftarrow$ index of $G(\Delta\vec{m}, \vec{N})$ in $V(\vec{N})$
                        $a_{ij} \leftarrow -1$
                        for $r$ from $1$ to $R-1$
                            $j \leftarrow$ index of $G(\Delta\vec{m}+1_k, \vec{N}-1_r)$ in $V(\vec{N})$
                            $a_{ij} \leftarrow -D_{kr}$
                        end for
                        $j \leftarrow$ index of $G(\Delta\vec{m}+1_k, \vec{N}-1_R)$ in $V(\vec{N}-1_R)$
                        $b_{ij} \leftarrow D_{kR}$
                    end for
                end if
            end for
        end for

        for $l$ from $0$ to $R-1$
            for all $\Delta\vec{m}$ with $M$ non-negative elements and $\sum_{k=1}^{M} \Delta m_k = l$
                if $\sum_{k=1}^{M} \Delta m_k = R-1$
                        for $r$ from $1$ to $s-1$ /* Add PC of class r */
                        $i \leftarrow i+1$
                        $j \leftarrow$ index of $G(\Delta\vec{m}, \vec{N})$ in $V(\vec{N})$
                        $a_{ij} \leftarrow N_r$
                        $j \leftarrow$ index of $G(\Delta\vec{m}, \vec{N}-1_r)$ in $V(\vec{N})$
                        $a_{ij} \leftarrow -Z_r$
                        for $k$ from $1$ to $M$
                            $j \leftarrow$ index of $G(\Delta\vec{m}+1_k, \vec{N}-1_r)$ in $V(\vec{N})$
                            $a_{ij} \leftarrow -(m_k+\Delta m_k) D_{kr}$
                        end for
                    end for
                end if
                for $r$ from $1$ to $s-1$ /* Add PC of Class R */
                    $i \leftarrow i+1$
                    $j \leftarrow$ index of $G(\Delta\vec{m}, \vec{N}-1_s)$ in $V(\vec{N})$, where $\vec{N}-1_0=\vec{N}$
                    $a_{ij} \leftarrow N_R$
                    $b_{ij} \leftarrow Z_r$
                    for $k$ from $1$ to $M$
                        $j \leftarrow$ index of $G(\Delta\vec{m}+1_k, \vec{N}-1_s-1_R)$ in $V(\vec{N}-1_R)$
                        $b_{ij} \leftarrow (m_k+\Delta m_k) D_{kR}$
                    end for
                end for
            end for
        end for
```

```
**while** $i \le \binom{M+R}{R} R$
        $a_{ii} \leftarrow 1$
        $b_{ii} \leftarrow 1$
        $i \leftarrow i+1$
**end while**
```

From the above algorithm becomes evident that the maximum absolute value of an element encountered in matrices $\boldsymbol{A}(\vec{N})$ or $\boldsymbol{B}(\vec{N})$ can be bounded by:

$$M(\boldsymbol{A}(\vec{N})) = M(\boldsymbol{B}(\vec{N})) = D_{max}(M(\vec{m}) + R), \tag{5.4.2}$$

where D_{max} is the maximum model value:

$$D_{max} = max_{k,r}\{D_{kr}, Z_r\}$$

and $M(\vec{m})$ denotes the maximum queue multiplicity value encountered in the model.

From the definition of the basis, we can observe that the knowledge of $V(\vec{N})$ and $V(\vec{N} - \vec{1}_R)$ implies the knowledge of $G(\vec{N})$, $G(\vec{N} - 1_r)$, $1 \le r \le R$ and $G(\vec{1}_k, \vec{N} - 1_r)$, $1 \le k \le M$, $1 \le r \le R$. These normalising constants are enough to calculate performance indices. The remaining constants in the vectors $V(\vec{N})$ and $V(\vec{N} - \vec{1}_R)$ can be used to estimate variances and covariances of the performance metrics [4].

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:

$$V(\vec{N}) = \boldsymbol{A}^{-1}(\vec{N})\boldsymbol{B}(\vec{N})V(\vec{N} - \vec{\boldsymbol{I}}_R) \tag{5.4.3}$$

If $N_R = 0$, we can apply a similar recursion to the class $R-1$. The recursion terminates when $V(\vec{0})$ is reached, since it contains only normalising 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.

The general steps in the MoM iterations can be summarised in the following high-level algorithm 5.4.2, which is taken from [4]:

```
Algorithm 5.4.2: Method of Moments (MoM)

Compute $V(N_1, 0, \ldots, 0)$ using an efficient single class method (e.g. Convolution or LBANC)
for $r$ from 2 to $R$
        Initialize the elements of $V(N_1, \ldots, N_{r-1}, 0, \ldots, 0)$ based on the previous class results
        and the termination conditions $G(\Delta\vec{m}, \vec{0}) = 1$ or $G(\Delta\vec{m}, \vec{0} - \vec{1}_R) = 0$
        for $n_r$ from 1 to $N_r$
                Setup and evaluate eq. (5.4.3) obtaining $V = (N_1, \ldots, N_{r-1}, n_r, 0, \ldots, 0)$
        end for
end for
Return mean performance indices using known equations and the normalising constants in
$V(N_1, \ldots, N_R)$ and $V(N_1, \ldots, N_R - 1)$
```

The above pseudocode is written at a high level, far from a concrete implementation. As it has been implemented as part of the current project, it is thought that it can be beneficial to provide the reader with a more concrete and lower-level pseudocode, such as the following algorithm 5.4.3:

```
Algorithm 5.4.3: Method of Moments (MoM)

/* $\vec{a}(i)$ denotes the i^th element of vector $\vec{a}$ */
$r \leftarrow 0$
/* Initializing data structures of class $r+1$ */
Generate matrices $\boldsymbol{A}$, $\boldsymbol{B}$ using Algorithm 4.5.1 with input $\langle [1,0,0,0,\dots,0], 1 \rangle$, where
$[1,0,0,0,\dots,0]$ a vector of $R$ elements.
$\vec{N}_0 \leftarrow \vec{0}$
$N_{0r} \leftarrow 1$
$\vec{V}_{cur} \leftarrow \vec{1}$
for $r$ from $1$ to $R$
        /* Processing class $r$ */
        for $n_r$ from $N_{0r}$ to $\vec{N}(r)$
                $\boldsymbol{A}' \leftarrow \boldsymbol{A}$
                Add $n_r - 1$ to all $N_R$ values of $\boldsymbol{A}'$
                $\vec{b} \leftarrow \boldsymbol{B} \cdot \vec{V}_{cur}$
                $\vec{V}_{prev} \leftarrow \vec{V}_{cur}$
                $\vec{V}_{cur} \leftarrow (A')^{-1} \cdot \vec{b}$
                /* Population $n_r$ completed */
        end for
        $\vec{N}_0(r) \leftarrow \vec{N}(r)$
        /* Class $r$ completed */
        if $r < R$
                $\vec{N}_0(r+1) \leftarrow 1$
                $N_{0r} \leftarrow 1$
                /* Initializing data structures of class $r+1$ */
                Generate Matrices $\boldsymbol{A}$, $\boldsymbol{B}$ using Algorithm 4.5.1 with input $\langle \vec{N}_0, r+1 \rangle$
        end if
end for
Return mean performance indices using known equations and the normalising constants in
$\vec{V}_{cur} = V(N_{1,} \dots, N_R)$ and $\vec{V}_{prev} = V(N_{1,} \dots, N_R - 1)$
```

It is evident from the above algorithms that MoM requires only N steps to solve a queueing network model. Furthermore, the number of normalising constants calculated in each step remains fixed during the recursion and is equal to the cardinality of $V(\vec{N})$, whereas in the case of the RECAL algorithm, the number of calculated constants grows combinatorially.

MoM manages to improve performance over the existing methods by simultaneously exploiting the Population Constraint and the Convolution Expression formulas, instead of exploiting just one of them, as does RECAL (only PC) and LBANC (only CE). Adding queue replicas and thereby increasing the number of shared normalising constants between the equations allows us to formulate a linear system and use it to estimate several unknown normalising constants. This is feasible as the number of equations grows faster than the number of unknowns when using MoM's recursion method.

In practice, MoM can achieve better performance than all existing methods for large enough networks, e.g. a few tens of jobs and more than two or three classes [4].

One of the main limitations of MoM is the fact that it does not seem to generalise to load-dependent queues. Furthermore, computation using the MoM algorithm requires exact arithmetic, as rounding errors can result in numerical instabilities. The the algorithm deals with extremely large numbers, much larger than the datatypes defined in most programming languages, and therefore one must implement or use special libraries to support arbitrary length arithmetic. The overhead added by such an approach is regarded as small and the algorithm produces impressive results when implemented efficiently [4].

The first approach to deal with this linear system with large number is presented in [4] and uses special libraries supporting arbitrary length arithmetic. Optimisations available, such as low-cost recalculation of the LU decomposition due to the special structure of the matrix $\boldsymbol{A}$, further increase MoM's speed.

Among the goals of this project, an alternative method in dealing with this core part of the MoM algorithm is examined, implemented and tested. More specifically, using modular arithmetic to solve the resulting linear system over finite fields and recombining the results afterwards can lead a further increase in algorithm efficiency in some cases.

5.5. Extensions of the Method of Moments

Apart from the presented MoM algorithm, several other similar algorithms have been presented which evaluate analytically queueing networks by using the same concept of relying on higher-order moments of queue lengths instead of mean values, as for example done by the MVA algorithm. This allows these algorithms to reduce remarkably the computational cost, especially on models with large number of jobs.

One of this algorithms is the Class-Oriented Method of Moments (CoMoM) [19], which is computes performance indices in closed multi-class queueing networks and scales efficiently as the number of classes increase. In general, the MoM and CoMoM algorithms feature similar data structures and characteristics, as well as the same need for exact arithmetic. However, CoMoM scales better than MoM as the number of classes R increases, whereas MoM scales better as the number of queues M increases. This happens because the two algorithms differ in the choice of the basis of unknowns that is computed at every iteration. MoM evaluates at each step the normalising constants of populations $\vec{N}, \vec{N}-\vec{1}_1, \ldots, \vec{N}-\vec{1}_{R-1}$ and increases the number of queue replicas by R queues or more, whereas CoMoM evaluates the normalising constants of a much larger set of populations, up to M jobs less, but with the addition of at most one queue replica.

This makes the MoM algorithm preferable for models with several queues, whereas CoMoM is best used for models with several classes. CoMoM includes a set of techniques to address singularity cases, which can be caused by interdependence between the PC and CE equations; in one of this techniques, a Hybrid MVA/CoMoM algorithm is used [19].

Another approach presented is the Generalised Method of Moments algorithm, which was introduced in [20]. This algorithm combines the approaches of MoM and Convolution by extending the MoM recursion with multiple recursive branches; each of these branches evaluates a model with different number of queues. This approach is more efficient than the original recursive structure that is used for MoM in this project.

6. Solution of Linear Systems using Modular Arithmetic

6.1. Introduction

It is well known in the cryptography community that many large linear systems can be solved using modular arithmetic. It is common in such problems to need to solve large linear systems with integer coefficients. However, the solution of such a linear system is a rational number. The techniques applied can be extended to handle real coefficients, are parallelisable with great efficiency and can be adapted to handle sparse matrices.

The approach presented in this section will be used to solve the MoM linear system (5.4.1):

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

As it has been already presented, the need to solve such large integer linear systems exactly is evident in the MoM algorithm. Furthermore, because MoM's computational cost does mainly depend on the time needed to solve such linear systems, the implementation proposed in this project would benefit by a multiprocessing approach. As it will be seen in this chapter, there exists a method of reducing one linear system with integer coefficients to a number of other same-sized linear systems that are the modular representations (i.e. are calculated using the modulo operation) of the initial one. Then, these produced linear need to be solved; this may be preferable to solving the initial one when done in parallel. Finally, the results can be combined using the Chinese Remainder Theorem. Parallelisation of the solving algorithm is relatively straightforward, as each of the produced "residual" linear systems can be assigned to a different processor. Implementation in distributed memory computing systems has led to efficient results [21].

It has to be noted however, that any solver that needs to be used by the MoM algorithm cannot be a simple straightforward solver, like the ones used in practice. It is usual for MoM to need to solve linear systems where the matrix may be singular. This means that several values of the solution vector may be unable to be computed. Furthermore, these values may propagate and may be used as the input of a subsequent linear system's solution. These values are usually unnecessary for MoM to successfully compute the normalising constant and the performance indices of a queueing network model; however, the solver has to be able to recognise them and recover from such ill-conditioned systems. Much effort has been put in the design, implementation and evaluation of a solver exhibiting such robust characteristics, which is documented in the following sections.

6.1.a Definitions

The Two Definitions of Modulo

There exist two possible definitions of the modulo operation $a \ mod \ n$, where $a\in\mathbb{R}$ and $n\in\mathbb{N}$; the two definitions differ in the sign of the remainder. The usual one, which always results in a non-negative remainder, is named **Euclidean definition** and was introduced in [22]. Let $q\in\mathbb{Z}$ be the quotient of a over n. Then, the result of the operation is a number $r\in\mathbb{R}$ where:

- $a=nq+r$ and
- $0\leq r\leq|n|$.

The other definition of the modulo operation does not require a non-negative result in all cases. More specifically, the result of the operation must have the same sign as the number a. Therefore, the following must hold:

- $a=nq+r$,
- $sgn(r)=sgn(a)$ and
- $|r|\leq|n|$.

The result of both operations is unique. Both definitions of the modulo operation can be used in our approach to build, solve and recombine the results of some residual systems. However, the second definition, which allows a negative modulo, is preferable as a matter of computational efficiency.

This is true, because the linear system built by the MoM algorithm in every iteration usually contains small negative numbers, of the same magnitude as the maximum value appearing in the model description; usually a few tens or a few hundreds. However, the divisor n can be arbitrarily large. Using the first definition of the modulo operation would result in "artificially" filling the linear system with very large numbers, which would cause an added computational cost as will be explained in more detail later. The second definition of the modulo in the field of exact solution of integer linear system using residual techniques was first documented in [23], however its choice was caused by the need for programming convenience and not efficiency or speed.

Most programming languages use one of these two definitions, whereas others, such as FORTRAN, MATLAB or PROLOG, provide two different operators – sometimes called mod and rem –, one for each definition.

The interest in using the modulo operation stems from our aim to build a Residue Number System. A Residue Number System basically represents large integers using a set of smaller ones; this can result in faster computations in practice, whereas it permits parallel computations as well. The maximum representational efficiency is achieved when the smaller integers are prime and therefore do not share common factors. The key theorem enabling this splitting of computations is the Chinese Remainder one. In the case of large integer linear systems solution, using the same theorem we are able to represent the large initial system using a set smaller numbers, thus enhancing performance in many cases. After each smaller system has been solved, the splitting is reversible if several conditions hold; in such a case one can easily obtain the initial solution.

We will now define these concepts; afterwards, a detailed presentation of our approach will follow.

Congruence Relation in Modular Arithmetic

Two integers a and b are said to be **congruent modulo** $\boldsymbol{n}$, $n\in\mathbb{N}^*$, symbolised as:

$$a\equiv b(mod\ n) \tag{6.1.1}$$

if their difference $a-b$ is an integer multiple of n. The number n is called the modulus of the congruence. Equivalently, both numbers have the same remainder when divided by n. By convention: $a\equiv b(mod\ 1)\Leftrightarrow a=b$.

Modular Multiplicative Inverse

The modular multiplicative inverse of a number $a\in\mathbb{Z}$ modulo a number $n\in\mathbb{N}$ is a number $x\in\mathbb{Z}$ such that $ax\ \ mod\ \ m=1$, or equivalently using the congruence relation:

$$ax\equiv 1\ (mod\ n) \tag{6.1.2}$$

The most efficient algorithm to calculate the modular multiplicative inverse is the Extended

Euclidean algorithm.

Coprime Integers

Two integers are said to be **coprime** or **relatively prime** if their greatest common divisor is 1. An efficient algorithm to discover the greatest common divisor of two integers and, therefore, whether they are coprime is the Euclidean one.

Chinese Remainder Theorem

The Chinese Remainder Theorem states that, if we have k positive integers $n_{1,} n_{2,} \ldots, n_k$ ($n_i \in \mathbb{N}^* \; \forall \, i=1,2,\ldots,k$) which are pairwise coprime, then, for any given integers $a_{1,} a_{2,} \ldots, a_k$, there exists an integer x solving the system of simultaneous congruences:

$$\begin{gathered} x \equiv a_1 \, (mod\, n_1) \\ x \equiv a_2 \, (mod\, n_2) \\ \vdots \\ x \equiv a_k \, (mod\, n_k) \end{gathered}$$

Furthermore, it is proved that all solutions x to this system are congruent modulo the product $N = n_1 n_2 \ldots n_k$:

$$x \equiv y \, (mod\, n_i) \, \forall \, i=1,2,\ldots,k \Leftrightarrow x \equiv y \, (mod\, N)$$

After we have split the initial linear system to a set of new ones, we arrive at the set of their solutions. The Chinese Remainder Theorem allows us to recombine these solution and reduce it to the solution of the initial system. Without the existence of this theorem, the solution of such systems would not be feasible in such a parallel way.

6.2. Solution Procedure

In this section, the procedure followed by the solver in order to compute the solutions of the linear system produced by the MoM algorithm in every iteration is outlined. The reader is reminded that the linear system required to be solved by MoM is given by the equation (5.4.1):

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

and it is a critical requirement of the MoM for the solution to be provided using arbitrary precision exact arithmetic.

The matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are calculated by MoM, whereas $V(\vec{N} - \vec{I}_R)$ is the solution vector of the previous linear system and $V(\vec{N})$ the solution vector of the current one. Of course, it is the responsibility of MoM to calculate the first such vector. In order to provide a good design from the viewpoint of software engineering, the solver is as decoupled as possible from the MoM implementation. These qualities of the implementation will be described in a following chapter 7.

For matters of clarity, the current presentation as well as the solver itself uses a different notation and considers the linear system in the usual form:

$$\boldsymbol{A} \vec{x} = \vec{b} , \tag{6.2.1}$$

where $\boldsymbol{A}$ is an $n \times n$ matrix and $\vec{x}$, $\vec{b}$ are n element vectors.

Using modular arithmetic, the initial system is solved in several residual ones. Each of the residual systems is then solved as usual and afterwards the results are recombined.

Below we will describe some of the unique characteristics of the matrices that define the linear system as well as outline a procedure for the solution of the said system. The main idea of the process followed in the current project is similar to procedures have been presented, among others, in [24] and [25] A more implementation-oriented approach can be found in [23]. Other important approaches, more similar to the one followed in this project, have been presented in [26], [27] and [21].

However, the used to solve the current problem has similarities but is different in many aspects from the ones presented in the literature. For example, efficiency issues have been the cause of many design decisions and application of different mathematical tools than other implementations. Another key characteristic which heavily influenced the formulation of the followed approach is the requirement for the solver to be robust, in the sense that it has to process and compute exactly the solutions of an ill-defined, singular linear system.

Therefore, the current approach forms a sufficient basis for implementation and introduction of more optimisations in the future, such as the experimentation with different solution methods. Lastly, the decoupled modular design makes the solution procedure below applicable to other, more optimised versions of MoM or even other algorithms in different scientific fields in the future.

6.2.a Unique Characteristics and Requirements of MoM

1. One of the most important requirements of MoM is the need to perform exact arithmetic [4]. Due to the fact that it is common for operations between numbers with very different magnitude – i.e. more that a thousand orders of magnitude – to arise, the usage of inexact, floating precision arithmetic would introduce errors due to truncation and approximation. If these errors are multiplied by another very large number, they may become large enough to influence not only the correctness of the results but also the execution stability of the MoM algorithm.
2. Another key feature of the MoM algorithm is that, in its simplest implementation, one produces a matrix $\boldsymbol{A}$ that contains columns filled with zeros; these columns correspond to the class r with population $N_r=0$. This means that the matrix $\boldsymbol{A}$ that defines the linear system can be singular, i.e. with zero determinant. The normal approach in such a case if a usual solver is used, would be to stop the solution of such a system and the execution of the algorithm. However, our solution process has been design in such a way so as to be able to handle such cases, discovering the solutions of as many unknowns as possible.

   An example of such a linear system is the following:

   $$\begin{bmatrix} 6 & 0 & 0 & 0 \\ 6 & 0 & 0 & -1 \\ -7 & 4 & 0 & 10 \\ 0 & 0 & 0 & 2 \end{bmatrix} \cdot \vec{x} = \begin{bmatrix} 6 \\ 2 \\ 41 \\ 8 \end{bmatrix}$$

   One can easily determine in the above linear system that $x_1=1$, $x_2=2$ and $x_4=4$, but it is unable to determine the value of x_3; all relevant information has been lost.

   A normal solver would halt its solution, for example as soon as it evaluated a zero determinant. However, the value of x_3 – and any other such elements – may be unnecessary for MoM to continue its operation and evaluate the model. The solver that was designed as part of this project and is presented below must be able to return the values of all computable elements of the solution vector and flag the element x_3 as indeterminable. It is now the responsibility of MoM to consider whether it can continue from such a state.
3. Imagine now that MoM decides that the execution can continue under these circumstances; if not other techniques dealing with singularity need to be used, which are presented in [3] and

[4]. The previous solution vector, corresponding to $V(\vec{N}-\vec{I}_R)$ and which now contains indeterminable values, will therefore be multiplied with another matrix $\boldsymbol{B}$, as per equation (5.4.1). This means that unless all elements of the respective 3rd column of $\boldsymbol{B}$ are equal to zero, which is seldom the case, the uncomputable values will propagate even more. As a result, the next linear system that the solver is required to process may look like the following:

$$\begin{bmatrix} 6 & 0 & 0 & 0 \\ 7 & 0 & 0 & -8 \\ -7 & 5 & 0 & 10 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \vec{x} = \begin{bmatrix} 26 \\ \cdot \\ \cdot \\ 0 \end{bmatrix},$$

where the dot symbol represents an indeterminable value. Clearly, the situation for the solver is now more difficult.

4. It has to be noted however that all element of every solution vector and vector $\vec{b}$ of equation (6.2.1) can only contain non-negative and uncomputable values. Negative values will never arise due to the properties of MoM. The values of the solution vector represent a **basis of moments** (see section 5.4.b, p. 22) which is a set of normalising constants; a negative normalising constant has no natural meaning.

The above characteristics of MoM have greatly influenced the design of this solver and are some of the main reasons that this implementation differs from the existing ones.

6.2.b Brief Presentation

The procedure outlined here is used to solve a linear system $\boldsymbol{A}\vec{x}=\vec{b}$, where $\boldsymbol{A}$ an integer matrix and $\vec{b}$ a vector of non-negative elements. This linear system conforms to the requirements and characteristics presented in the previous section. The procedure extends to rational numbers by scaling. In order for the presentation to be concise and short, many important parts of the procedure are omitted; among them, the most important is the way the solver deals with singularity case and indeterminable values. This omission does not mean that their implementation is a trivial matter. Much effort has been put to adapt the linear system solution process using modular arithmetic to the ill-conditioned linear systems defined by the MoM algorithm.

Influential in the development and design of this procedure have been [26], [21] and [27]. However, the procedure is uniquely adapted to solving linear systems with requirements similar to the ones of MoM and consequently contains several novel ideas.

- Find a number $M_{threshold} \in \mathbb{N}$ such that:

$$M_{threshold} = 2\,max\left(|d|,\, n(n-1)^{(n-1)/2} M(\boldsymbol{A})^{n-1} M(\vec{b})\right), \tag{6.2.2}$$

  where n the number of equations and $d = det(\boldsymbol{A})$. Also, for a matrix $\boldsymbol{B}=[b_{ij}]$, we denote as $M(\boldsymbol{B})$ the maximum absolute element:

$$M(\boldsymbol{B}) = max_{i,j} |b_{ij}| \tag{6.2.3}$$

  A similar definition exists for the maximum absolute element of vector $\vec{b}$.

- Choose a set of moduli $m_1, m_2, \ldots, m_s \in \mathbb{N}$, where $M = m_1 m_2 \ldots m_s \geq M_{threshold}$. The moduli must be coprime:

$$gcd(m_i, m_j) = 1 \; \forall i \neq j \tag{6.2.4}$$

  and must satisfy:

$$gcd(M, d) = 1 \tag{6.2.5}$$

- Formulate the s residue systems of equations: $A\vec{x} = \vec{b} \ (mod\ m_i)$ for each modulo m_i.

  This can be accomplished not only by using the Euclidean definition of the modulo operator, but also by using the one that allows for a negative result.

- Solve each of the s residue systems using a method similar to Gaussian Elimination. From each residue system obtain the absolute value of the residue representation d_k of the determinant:

  $$d_k = |det(A(mod\ m_k))| \ , \tag{6.2.6}$$

  as well as a vector $\vec{y}_k$ of the residue system's solutions. This vector contains the residue representation of the final solution vector.

- By the Chinese Remainder Theorem, we can recombine the residue representations to obtain the initial ones. This can be done using several methods, the preferable of which is in our case the Single-Radix Conversion Algorithm. An other method which can be applied is the Mixed-Radix Conversion Algorithm. Both these methods are presented in more detail in Section 6.3.f (p. 49).

- If the moduli were chosen so as to satisfy (6.2.2), the final vector of solutions $\vec{x}$ will now have been determined.

As it can be seen, the process can be parallelised; each residual linear system can be assigned to a different processor. This way, each linear system that will be solved in parallel will contain smaller numbers than the initial one.

6.3. Building the Method

In this section the procedure followed by the solver will be presented in more detail. The reasoning behind several design choices will be explained and literature references will be provided. For matters of improved clarity and reader comprehension, the procedure will be presented in parallel with a simple example. More specifically, after each step of the solution is discussed a simple example highlighting the most important details of the step will follow.

This approach is not only a way to document and present what has been done; it can also function as means for a reader to understand precisely how linear systems can be solved efficiently using linear algebra and residual arithmetic. Such a step-by-step approach does not exist in the literature, therefore its documentation can be valuable for people who want to apply the same techniques on a different sector or people who may want to augment and improve the current project and MoM in general.

As it has been said before, it is common for the linear system constructed by MoM to be defined by singular matrices, i.e. matrices the determinant of which is zero. Such a matrix usually contains one or more columns filled exclusively with zero elements. Furthermore, some of the elements of the right hand side of the equations may be indeterminable. In any case, the algorithm must be robust and stable enough to return the values of as many different variables as possible.

A typical linear system one can examine to understand all aspects of the solver is the following, where the dots in the right-hand side vector represent indeterminable elements existing as a result of indeterminable unknowns of a previous MoM iteration:

$$A\vec{x}=\vec{b}\Rightarrow\begin{bmatrix}6 & 0 & 0 & -3 & 0 & 0 & 1\\ 0 & 0 & 3 & 2 & 0 & 0 & 8\\ -1 & 2 & 0 & 0 & 0 & 0 & 0\\ 0 & -2 & 0 & 7 & 0 & 0 & 4\\ 0 & 0 & 5 & 3 & -1 & 0 & 0\\ 3 & 7 & 12 & 0 & 0 & 0 & 0\\ 0 & 1 & 2 & 0 & 0 & 0 & 1\end{bmatrix}\cdot\vec{x}=\begin{bmatrix}27\\ 18\\ \cdot\\ 11\\ \cdot\\ 101\\ 13\end{bmatrix} \tag{6.3.1}$$

It can be easily seen that the determinant of A is zero. We will now solve this system step by step, presenting the procedure in the meanwhile.

6.3.a Step 1: Determining a lower bound for the product of moduli

The first operation it is necessary to do is to find the number $M_{threashold}$, which is the lower limit for the product of moduli. This is accomplished by using equation (6.2.2):

$$M_{threshold}=2\,max\left(|d|,\,n(n-1)^{(n-1)/2}M(\boldsymbol{A})^{n-1}M(\vec{b})\right)$$

However, it is necessary to notice that several different formulas for calculating $M_{threashold}$ exist in practice. The formula used here is encountered in [27] and was preferred to other formulas because of the fact that it leads to superior performance in practice and it is simple to calculate.

Among the other formulas that were considered are:

- The formula $M_{threshold}=2\,max\left(|d|,\,M(\boldsymbol{A})\right)$, which is presented in [26]. This formula is simpler to calculate than (6.2.2). However, it can be easily seen that the limit defined by this equation is lower; this in practice leads in many cases to corrupted results. This is true, because in the case of MoM the systems formulated contain numbers of far greater magnitude in the right hand side $\vec{b}$, compared to those encountered in matrix A, which are relatively small numbers. This fact is ignored by this equation and is the main reason that it does not perform satisfactory in this problem.

- The formula $M_{threshhold}=2\prod_{i=1}^{n}\left(\sqrt{\sum_{j=1}^{n}a_{ij}^2}\right)\prod_{k=1}^{n}|b_i|$, which is presented in [23]. The limit defined by this formula is a bit more complicated to calculate than the previous formulas, but it always manages to recombine the residual results and evaluate the final ones. However, the main drawback of this technique is that the limit defined can be several orders of magnitude larger than the other methods, as it can be easily verified.

  This usually means that we may need to select more and bigger moduli at a great computational cost: moduli are usually selected, for reasons that will be explained in the following step, as very large prime numbers [21]. In order to verify that such a number is prime, several primality tests need to be used, such as the Miller-Rabin primality test or the Quadratic Sieving technique. These tests may need considerable time to run, especially on large systems.

- The formula $M_{threshold}=2\,max\left(|d|,M\left(A^{-1}\vec{b}\right)\right)$, which is used by [21]. This is a formula that gives a satisfactory limit, i.e. one that can be applied to tackle our problem correctly. However, it is inefficient to compute because it involves evaluating the inverse matrix A^{-1}.

  Due to the fact that this is a simple and intuitive formula, which will be recognised when one has a full view of the entire procedure, several approximations of the maximum possible element of $A^{-1}\vec{b}$ where introduced in order to be able to use it in practice. These approximations took advantage of useful inequalities between the various norms of a real matrix. The reader can find useful references regarding such operations and bounds of normal, inverse and classical adjugate matrices in [28]. Introducing approximations results however in values of $M_{threshold}$ that are close to (6.2.2) and therefore provide a similar performance. Therefore, the use of (6.2.2) is preferred as it is much more straightforward to compute.

Evaluating (6.2.2) however involves the evaluation of the determinant of the matrix A. Evaluation of the determinant is a lengthy process, characterised by $O\left(n^3\right)$ time complexity. It is usually involves almost solving the linear system using regular arithmetic, making our modular arithmetic approach unnecessary. Therefore one can not count on actually evaluating the determinant on real-world scenarios. As no bounds for the absolute value of the determinant of matrix A are provided by the MoM algorithm [4], one can only resort in bounding this value using an inequality. A widely used one is Hadamard's Inequality, which for real valued matrices consisting of n column vectors v_j states that:

$$|det(A)|\leq\prod_{j=1}^{n}\|v_j\|\,, \tag{6.3.2}$$

where $\|v_j\|$ the Euclidean length of the column vector in $\mathbb{R}^n$. Furthermore, if entries on the $n\times n$ matrix are bounded by the maximum element $M(A)$, then:

$$|det(A)|\leq n^{n/2}M(A)^n \tag{6.3.3}$$

Therefore, equation (6.2.2) becomes:

$$M_{threshold}=2\,max\left(n^{n/2}M(A)^n,\,n(n-1)^{(n-1)/2}M(A)^{n-1}M\left(\vec{b}\right)\right) \tag{6.3.4}$$

This equation can be computed with little computational cost and leads to good results in practice. It can be straightforwardly compute using $O\left(n^2\right)$ operations.

If we had to solve a single linear system, this is the point where the current step would end and we would have to begin choosing the specific set of moduli. However, due to the fact that selection of the moduli may only be performed at a significant computational cost, it was decided for the usual case of the MoM algorithm to perform these calculations only once. The consequence is that we have to adapt (6.3.4) so as to provide a threshold value big enough for all possible MoM linear systems based on the specific model.

The reader may be reminded that the normal procedure to evaluate a queueing network model using the Method of Moments is to solve as many linear systems as the total population of jobs. The size of the linear system remains constant throughout the execution and only the contents of the matrices change. However, it is possible to bound the maximum absolute value contained in every matrix A and B of MoM based on a particular model (5.4.b, p. 22):

$$M\left(A\left(\vec{N}\right)\right)=M\left(B\left(\vec{N}\right)\right)=D_{max}\left(M(\vec{m})+R\right), \tag{5.4.2}$$

where D_{max} is the maximum model value and is defined by $D_{max}=max_{k,r}\{D_{kr},Z_r\}$.

One can also provide a maximum limit on the value of normalising constant G, as its maximum number of digits has been presented in (5.1.1) (p. 20):

$$n_{max}=\left\lceil \log_{base}\left(G_{max}\right)\right\rceil = N\left\lceil \log_{base}\left(D_{max}\left(N+M+R\right)\right)\right\rceil \Rightarrow$$

$$G_{max}=base^{N\left\lceil \log_{base}\left(D_{max}\left(N+M+R\right)\right)\right\rceil}, \tag{6.3.5}$$

where N the total population of the queueing network model and $base$ the base of the numerical system we use for representation, i.e. $base=2$ for the binary system and $base=10$ for the decimal one.

In order to be able to perform this step only once, one has to provide an adequate bound for the maximum element of vector $\vec{b}$, which in our case is the right hand side of equation (5.4.1):

$$\vec{b}=\boldsymbol{B}\left(\vec{N}\right)V\left(\vec{N}-\vec{I}_R\right)$$

$V\left(\vec{N}-\vec{I}_R\right)$ is actually the collection of normalising constants for a network involving one population less. It is known that the values of normalising constants are at least weakly increasing as the total population increases. As the true maximum element of matrix $\boldsymbol{B}$ remain below $M\left(\boldsymbol{B}\right)$, which is constant throughout the execution, one can bound the maximum element of all $\vec{b}$ vectors as the maximum element of vector $\vec{b_{max}}=\boldsymbol{B}\left(\vec{N}\right)V\left(\vec{N}\right)$. 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:

$$M\left(\vec{b_{max}}\right)\leq n\,M\left(\boldsymbol{B}\right)G_{max}, \tag{6.3.6}$$

Where n the linear system order. As $M\left(\vec{b}\right)\leq M\left(\vec{b_{max}}\right)$, we can consider in the general case that:

$$M\left(\vec{b}\right)=n\,M\left(\boldsymbol{B}\right)G_{max} \tag{6.3.7}$$

After all these values and bounds have been defined, we are now able to present a final formula for $M_{threshold}$, one that is 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 eq. (6.3.4):

$$M_{threshold}=2\,max\left(n^{n/2}M\left(\boldsymbol{A}\right)^n, n\left(n-1\right)^{\left(n-1\right)/2}M\left(\boldsymbol{A}\right)^{n-1}M\left(\vec{b}\right)\right), \tag{6.3.4}$$

n is the linear system order. Substituting (5.4.2), (6.3.5), (6.3.7) in the previous formula leads us to:

$$M_{threshold}=2\,max\left\{\begin{matrix} n^{n/2}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]^n, \\ n^2\left(n-1\right)^{\left(n-1\right)/2}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]^{n-1}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]2^{N\left\lceil \log_2\left(D_{max}\left(N+M+R\right)\right)\right\rceil}\end{matrix}\right\} \tag{6.3.8}$$

Example

In this example we will evaluate $M_{threshold}$ for the linear system (6.3.1). To present the procedure in more clarity, we will use eq. (6.2.2) instead of (6.3.8):

$$M_{threshold}=2\,max\left(\left|d\right|, n\left(n-1\right)^{\left(n-1\right)/2}M\left(\boldsymbol{A}\right)^{n-1}M\left(\vec{b}\right)\right) \tag{6.2.2}$$

In this case we have $n=7$, $M\left(\boldsymbol{A}\right)=12$ and $M\left(\vec{b}\right)=101$. When computing the maximum elements we consider For the computation of the maximum absolute value of $\vec{b}$ we disregard any indeterminable values.

The value of the determinant that is needed for the calculation of $M_{threshold}$ cannot be straightforwardly evaluated in this case, and this is common in the linear systems defined by MoM. That is true, as the exact value of the determinant is zero due to the 6th column. In practice, such a result would greatly reduce the correctness of our results.

As it has already been said, it is usual for an MoM linear system to be defined by a singular matrix. This makes solution using the known and available methods impossible. Therefore, the modular technique defined in this chapter has been adapted to be able to cope with this singularity: we approximate the maximum value of the determinant as (eq. (6.3.3)):

$$|d| = n^{n/2} M(\boldsymbol{A})^n \tag{6.3.9}$$

If the matrix was not singular we could provide a better approximation of the determinant using equation (6.3.2) or even a full computation of it using one of the known methods. However, the reader should be reminded that this procedure has been designed with the goal of minimising the time required by the total of the MoM evaluation and not the time required to solve a single system, so this computation must be performed only once. Computation using (6.3.9) is therefore the only viable method.

So, in the case of system (6.3.1) we have:

$$M_{threshold} = 2\,max\left(n^{n/2} M(\boldsymbol{A})^n, n(n-1)^{(n-1)/2} M(\boldsymbol{A})^{n-1} M(\vec{b})\right) \Rightarrow$$

$$M_{threshold} \simeq 4.56 \cdot 10^{11}$$

This is quite a big value. The algorithm could solve the current linear system with smaller threshold values. However, as the procedure is adapted for MoM and in MoM the digits involved grow exponentially in value (linearly in length) to the total population, the reader has to understand that this value is selected with MoM in mind.

6.3.b Step 2: Determining the moduli

In this step we must choose a set of moduli $m_1, m_2, \ldots, m_s \in \mathbb{N}$. Several important theoretical conditions must hold.

- The product of all the moduli must be greater than $M_{threshold}$:

$$M = m_1 m_2 \ldots m_s \geq M_{threshold}$$

- The moduli must be coprime for the Chinese Remainder Theorem to be applied and lead to recombination of the results (eq. (6.2.4)):

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

- Lastly, the following property must hold (eq. (6.2.5)):

$$gcd(M, d) = 1$$

Satisfying and verifying the two last conditions can be time consuming. To ease this procedure, there is a consensus in the sector toward selecting very large primes as moduli. Such primes satisfy the coprimality requirement by definition and greatly reduce the probability of encountering a case where $gcd(M, d) = 1$ ([29], [30] and [31]).

In general, there are many alternative strategies one can use in order to select the specific moduli that result in the minimum total runtime. The optimal strategy for a given problem depends on important algorithmic details and design choices of subsequent steps, as well as the advantages and disadvantages of the underlying computing system. One should consider as well any how any computational benefits measure to the computational cost and complexity of the selection process itself; in many such cases behaviour similar to the "law of diminishing marginal returns" can be expected.

The literature sources contain no specific set of guidelines regarding when it may be preferential to use many and smaller moduli and when less in number but bigger ones, or even a completely different (mixed) selection approach. Several approaches have been examined in practice and were considered when designing this solution algorithm and evaluating the experimental results. Each one of them has unique advantages and disadvantages:

- Some researchers ([26]) prefer the selection of moduli smaller than the maximum number representable in the computer's word size W: $m_i \le W$. This approach has the advantage that some of the subsequent computations may be able to be performed using floating-point arithmetic. This approach is not applicable in all problems; for example, MoM requires usage of exact arithmetic in all steps of the computation, so it would not benefit from such an approach. Furthermore, the word sizes of modern computer systems may be hundreds orders of magnitude smaller than the values of $M_{threshold}$ encountered in typical problems. This means that if relatively small moduli were to be used, i.e. with length comparable to the word size, then this would mean that the initial system would be split in a great number of residual systems. Thus, the runtime may not benefit at all from the usage of inexact arithmetic.

  However, this approach may be preferential if used in computing systems consisting of many different processing nodes, as for example Massively Parallel Computers. Such systems feature rich interconnection networks and high-bandwidth distributed memory. On the contrary, the current implementation targets primarily the simple multi-core Personal Computer or small server hardware, where at most 8-16 different processing cores are currently available and the memory bandwidth is shared between all of them. This part will be elaborated more on the next chapter, which discusses more implementation-specific challenges.

- Other researchers ([12]) prefer using a pre-determined set of moduli for all cases, acknowledging that in several instances the method may fail to produce correct results. This is unacceptable in MoM, especially since it may be computationally costly to verify during the runtime whether the results of every linear system are valid or not. Furthermore, there is no unique – or even set of unique – moduli set that can be used and lead to efficient solution of the linear systems in all real-world cases MoM will be used.

- Another approach that has been examined in practice has been to have a list that contains precomputed candidate moduli in ascending order [23]. The algorithm is able to calculate $M_{threshold}$ and then multiplies the moduli from the smallest one to the biggest, accumulating the product. When the product exceeds $M_{threshold}$, then enough moduli have been selected. This approach has the advantage that it relies on a precomputed list, therefore lifting the requirement of performing primality tests on each candidate modulo during the runtime. However, it tends to generate more residual linear systems than the minimum possible. This approach may be sufficient when implementing the algorithm using a serial programming paradigm, however it does not scale well during parallelisation. The typical number magnitude encountered in each of the residual systems that must be solved during a particular invocation may exhibit high diversity, resulting in a high deviation in the runtime needed to solve each of them: for example, one parallel unit may be assigned the solution of an "easier" system that contains smaller numbers whereas another may be assigned a "harder" one. Inevitably, one processor may need to wait while another one needs more time to produce results. This time waste cannot be "filled" using techniques applicable in other problems, such as preparation of the arguments of subsequent invocations; the current version of MoM is inherently serial in the manner in which it performs the population recursion. The next linear system remains largely unknown until the previous one has been solved. Lastly, such an approach is not consistent with the efficient parallel programming approach, which aims to maximise the processor utilisation throughout the system.

One could argue that using a precomputed list of candidate moduli is impossible in our case, as the moduli's magnitudes depend on the model's arithmetic qualities. This is true, however one could produce such a list for models that contain typical delay times, service demands as those encountered in practice. Such an approach would be beneficial when dealing with a wide gamut of networks. However, on bigger models it may be impossible to use only precomputed moduli; some of them may need to be computed during the runtime.

For the above reasons, the most efficient strategy in our case is to try to select as many moduli with equal magnitude as the number of processing cores; this approach minimises the number of linear systems that need to be solved in parallel and maximises the percentage of runtime in which the processors are doing useful work.

In theory, this strategy would scale ad infinitum if the necessary primality tests that need to be performed for each candidate prime modulo were fast. In reality, the time needed for such tests when using Java's arbitrary large prime generator does not scale well enough for increasing desired length of the primes, as it can be seen in Fig. 4 (p. 41). Therefore, an adaptation of this strategy is performed when dealing with very large models:

Suppose that the parallel code is running on n different processors. Then, our fist approach is to select moduli with bit length of approximately $b=\frac{\lceil \log_2 (M_{threshold}) \rceil}{n}$. However, this bit length b can be quite large in some cases; the maximum limit for b in our case based from implementation testing and benchmarking was set at 2.500 bytes. If the calculated b value is above that, we are forced to choose between two options:

- A first approach could be to disregard the maximum limit and select arbitrarily large moduli. This is not the preferable approach, as the amount of time needed to select a prime number of a given size quickly grows in comparison to its bit length, as the primality tests can be expensive to perform, as presented in Fig. 4.

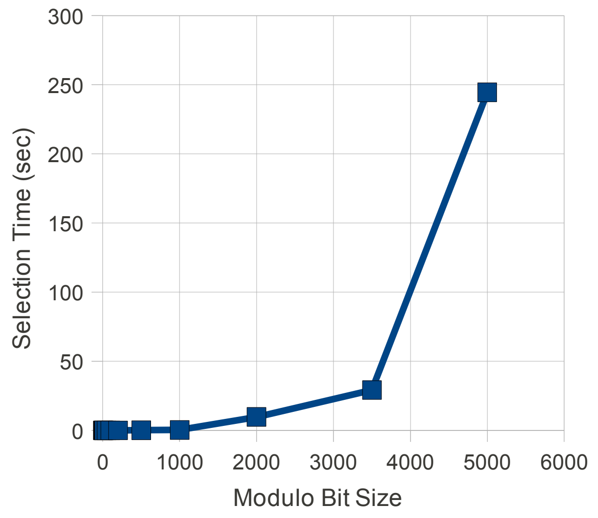


Fig. 4: *Modulo selection time vs. its bit size. We can verify the high growth rate of the selection time as the desired bit size increases. The exact selection time in not of interest, as it was determined on a different machine than the testing host.*

Following this approach may lead us on losing too much time on the moduli selection operation; therefore, this strategy may not lead to the best result in all cases.

- Thus, a different "divide and conquer"-like approach could be tried. For example, we could start with an initial pair of $\langle b_{0,} n_0 \rangle = \left\langle \left\lceil \frac{\log_2 (M_{threshold})}{n} \right\rceil, n \right\rangle$, where n the number of initially desired moduli (number of processors), and at each iteration set:

  $$\langle b_{i+1}, n_{i+1} \rangle = \left\langle \left\lceil \frac{b_i}{2} \right\rceil, 2\text{n} \right\rangle, \quad i = 0, 1, 2, \ldots,$$

  until $b_k \le b_{threshold}$, where $b_{threshold} = 2{,}500$ in our case. The total number of moduli selected will be $s = n_k$. This approach minimises 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, even if each CPU core maybe required to solve more than one residual linear systems per iteration.

One could further reduce the $b_{threshold}$ value, in order to select smaller moduli or even moduli fitting within the computer's word size. However, using so small moduli does not yield improved results, as the amount of residual linear systems that must be solved in such cases becomes very large.

This adapted version can make a more efficient use of processor caches and memory buses as well, as the maximum numbers existing in the linear system are bounded by the respective modulo. Thus, a lower modulo results in lower numbers, thus in simpler systems.

Example

In the previous step we had calculated the desired minimum product of moduli as equal to $M_{threshold} \simeq 4.56 \cdot 10^{11}$. Let us suppose that we want to produce two residual systems, i.e. we want to select two moduli. Two good prime candidates are $m_1=755{,}239$ and $m_2=1{,}942{,}111$, where $m_1 m_2 \simeq 1.47 \cdot 10^{12}$. Both moduli are far below the bit length threshold which would cause us to perform the iterations of the adapted strategy and select 4 or more moduli.

What we can see from this example is that even though we were initially considering a linear system where 11 digits were needed to evaluate the model, we have now reduced it to two systems which only need 6 and 7 digits, respectively.

Even though the two moduli m_1 and m_2 may seem very large compared to the ones contained in the linear system, however the reader must take into account that the modulo selection strategy is tailored towards efficient support of the MoM algorithm. In this algorithm, the numbers encountered in the vector $\vec{b}$ grow exponentially after every invocation; selecting so big moduli is therefore a good strategy in practice.

6.3.c Step 3: Linear System Sanitisation

One important and necessary step of the solution process is the Linear System Sanitisation. By this we mean a set of methods, rules and techniques which are applied to the initial linear system to transform it to a new one, in which all elements can be computed using simple adaptations of the usual techniques. This particular step is independent of the use of modular arithmetic; it may be beneficial to apply similar techniques even if one wants to perform a simple serial solution of the linear system.

Therefore, Linear System Sanitisation detaches MoM from the core part of the solver. It enables the easy integration of other solution core algorithms than Gaussian Elimination in the future, without the need for the designer to deal with the technicalities of an ill-conditioned linear system. It also adheres to a key high performance computing: when facing a design trade-off, one should be concerned with making the common case more efficient instead of adapting the design to handle unusual situations in the same layer, thus making common cases slower. In our case, detaching the two parts of the sanitiser and the solver allows each of them to become efficient in its own right.

It also has to be noted that this approach is not encountered in literature, however it is intuitive, it can be merged without many modifications with the rest of the procedure and it can be very useful in practice.

In short, what this part does is to process the initial $n \times n$ matrix A and the n-element vector $\vec{b}$ and return a new matrix A' and a new vector $\vec{b}'$. The dimensions of A' are $m \times l$, where $l \leq m \leq n$, whereas the length of $\vec{b}'$ is m.

The transformation of A and $\vec{b}$ to A' and $\vec{b}'$ can be described using the following steps:

1. Remove all indeterminable elements of $\vec{b}$ and the corresponding rows of A.

   This is necessary as each row of A and $\vec{b}$ defines a weighted summation of unknowns. If the respective element of $\vec{b}$ is indeterminable, then this equation can be completely dropped from the linear system.

2. On the edited matrix, remove all zero-colums, i.e. columns that contain only zero elements. Mark the respective unknowns in the solution vector $\vec{x}$ as indeterminable.

   The removal of rows may seem to create more indeterminable unknowns than those of the initial system. However, this is not true; the values of these unknowns cannot be determined in any way.

This approach of row removals decreases the size of the initial system, therefore reducing the amount of operations required to solve it.

There are several repercussions that needed to be considered during the design and implementation phase of the project. First of all, the algorithm may produce a new matrix $\boldsymbol{A}'$ that is not square. In such a case:

3. If $m < l$ the solution of the linear system cannot proceed due to singularity. Several means of recovery from such cases are documented in [3].
4. If $m \geq l$ then the matrix has row rank, i.e. number of linearly independent rows, that is less than l:

$$rank(\boldsymbol{A}') \leq l \tag{6.3.10}$$

To continue from this point onwards, one could consider leaving the matrix as it is, in a non-square shape, and adapt all methods used in the rest of the solution process to work on such non-square matrices. Of course, the total cost of operations during the system solution remains $O(n^3)$. However, this choice raises two important issues: the adaptation of Gaussian Elimination to operate satisfactory in non-square matrices and, primarily, the definition of a concept similar to the determinant on non-square matrices. It is known that the determinant is defined only on square matrices and its value and sign evaluation are interwoven with the solution of linear systems using modular arithmetic.

Example

Recall the initial linear system (6.3.1):

$$\boldsymbol{A}\vec{x}=\vec{b} \Rightarrow \begin{bmatrix} 6 & 0 & 0 & -3 & 0 & 0 & 1 \\ 0 & 0 & 3 & 2 & 0 & 0 & 8 \\ -1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 7 & 0 & 0 & 4 \\ 0 & 0 & 5 & 3 & -1 & 0 & 0 \\ 3 & 7 & 12 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \vec{x} = \begin{bmatrix} 27 \\ 18 \\ \cdot \\ 11 \\ \cdot \\ 101 \\ 13 \end{bmatrix}$$

1. At first we observe that the 3rd and the 5th rows must be removed from the system:

$$\boldsymbol{A}' = \begin{bmatrix} 6 & 0 & 0 & -3 & 0 & 0 & 1 \\ 0 & 0 & 3 & 2 & 0 & 0 & 8 \\ 0 & -2 & 0 & 7 & 0 & 0 & 4 \\ 3 & 7 & 12 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \vec{b}' = \begin{bmatrix} 27 \\ 18 \\ 11 \\ 101 \\ 13 \end{bmatrix}$$

2. We remove the 5th and 6th columns of $\boldsymbol{A}'$, as they contain only zero vectors. By this step we know that the 5th and 6th elements of the solution vector $\vec{x}$ will be indeterminable:

$$\boldsymbol{A}' = \begin{bmatrix} 6 & 0 & 0 & -3 & 1 \\ 0 & 0 & 3 & 2 & 8 \\ 0 & -2 & 0 & 7 & 4 \\ 3 & 7 & 12 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 \end{bmatrix} \quad \vec{b}' = \begin{bmatrix} 27 \\ 18 \\ 11 \\ 101 \\ 13 \end{bmatrix} \quad \vec{x} = \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ \cdot \\ \cdot \\ ? \end{bmatrix}$$

In the solution vector $\vec{x}$ we denote by " $\cdot$ " a definitely indeterminable element, whereas by “ $?$ ” one that may or may not be determinable. If the row rank of A' is equal to its number of columns, then all “ $?$ ” will be able to be computed; however, we do not check this condition.

3. Not applicable as $m=l=5$ in our case.
4. The matrix A' is square, but this is just a coincidence. If the 3rd element of the initial vector $\vec{b}$ was not indeterminable, then we would not have received a square matrix in this step.

6.3.d Step 4: Formulating residual systems of equations

During this step we produce s new linear systems, one for each selected modulo $m_1, \ldots, m_s$. These new linear system are named residual linear systems. More specifically, for each modulo m_k a new residual matrix A_k is calculated, where every element $a_k(i,j)$ of it is evaluated as:

$$a_k(i,j)=a(i,j) \quad mod \quad m_k \tag{6.3.11}$$

where $a(i,j)$ is the respective element of the matrix A' resulting from the previous step. A similar definition is used for the residual vector $\vec{b}_k$:

$$b_k(i)=b(i) \quad mod \quad m_k \tag{6.3.12}$$

where $b(i)$ is the ith element of the vector $\vec{b}'$ of the previous step.

From a design point of view, there are two important factors that influence the efficiency of the algorithm in practice:

1. The specific behaviour of the modulo operation used when evaluating negative elements.

   Let us consider the modulo operation using as dividend a number $x \in \mathbb{R}^-$ and as divisor a number $n \in \mathbb{N}^*$. If we use the Euclidean definition of modulo, which does not allow for negative results, then the following property holds:

   $$x \, mod \, n=(n+x) \, mod \, n \ ,$$

   where $n+x \leq n$. This has the consequence that if x is a relatively small number compared to n, then after the modulo operation it becomes comparable to n. As it has been already discussed, the common case for MoM is for the matrix 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, if one uses the definition of modulo that allows for negative numbers, then the magnitude of a negative element does not increase. Consider this definition using x as dividend and n as divisor, where x, n the same as above. In this case the following property holds:

   $$x \, mod \, n=-[(-x) \, mod \, n]$$

   It can be seen that in this case the result has the same magnitude as x. This greatly influences the total procedure's runtime, especially during the next step, which is the solution of the residual systems. It is important to note however that one can use whichever definition preferable for the modulo; the algorithm can be easily adapted in order to produce correct results in any case.

Lastly, one has to consider that the definition of the modulo operation used can influence the efficiency and optimality of the moduli selection strategy used in Step 2 (6.3.b, p. 38).

2. If the modulo definition allowing negative results is used and the minimum selected modulo $m_{min}=min(m_{1,} m_{2,} \ldots, m_s)$ satisfies the following property:

$$m_{min} > M(\boldsymbol{A}) ,$$

then for any matrix $\boldsymbol{A}$ produced during MoM's iterations the residual elements of matrix $\boldsymbol{A}$ need not be computed in any case. That is because their values do not change with respect to the exact modulo m_k or even the modulo operation itself. Such a strategy allows one to perform $O(s n^2)$ operations less during each MoM iteration, where s the number of selected moduli.

Example

In a previous step we selected as moduli $m_1=755{,}239$ and $m_2=1{,}942{,}111$.

- If the negative definition of the modulo is used, then the residual matrices $\boldsymbol{A_1}$ and $\boldsymbol{A_2}$ are the same as in the previous step:

$$\boldsymbol{A}_1=\boldsymbol{A}_2=\begin{bmatrix} 6 & 0 & 0 & -3 & 1 \\ 0 & 0 & 3 & 2 & 8 \\ 0 & -2 & 0 & 7 & 4 \\ 3 & 7 & 12 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 \end{bmatrix} ,$$

whereas the residual vectors $\vec{b}_1$ and $\vec{b}_2$ coincidentally remain the same as the previous $\vec{b}\,'$:

$$\vec{b}_1=\vec{b}_2=\begin{bmatrix} 27 \\ 18 \\ 11 \\ 101 \\ 13 \end{bmatrix}$$

This coincidence means that in this particular case the usage of modular arithmetic does not provide any meaningful performance gain, as the residual systems are as complex as the initial one. However, this should not be considered a problem; this process has been optimised with the objective of solving the entire series of MoM iterations as fast as possible and not with the objective of solving a single stand-alone linear system as fast as possible. In order to do so, one should select new moduli during every MoM iteration. Such an approach was tested and found to impose a severe overhead for each iteration. The total runtime for all iterations was much greater on the computing system architecture targeted by MoM.

- If the Euclidean definition of the modulo operation is used, which only allows non-negative results, the residual matrix $\boldsymbol{A_1}$ becomes:

$$A_1 = \begin{bmatrix} 6 & 0 & 0 & 755236 & 1 \\ 0 & 0 & 3 & 2 & 8 \\ 0 & 755237 & 0 & 7 & 4 \\ 3 & 7 & 12 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 \end{bmatrix}$$

It is now evident that usage of the usual Euclidean definition of modulo can make the solution process much more complex, as the residual matrix $\boldsymbol{A_k}$ may contain much bigger elements than before. Calculations between larger values are more expensive computationally.

6.3.e Step 5: Solving the residual systems

At this point, each one of the k linear systems must be solved. In general, one could use several different solution techniques, such as Gaussian Elimination or LU Decomposition. For this project, implementation using Gaussian Elimination was preferred, as the Project mainly focused on the research regarding linear system solution using residual systems. Depending on the specific problem, a reader attempting to apply this solution technique at a different problem in the future may prefer the usage of a different method depending on the problem requirements, the hardware architecture and the impact of possible optimisations.

At this step, two operations must be completed:

1. The algorithm should attempt to calculate the value of each unknown of the residual solutions vector $\vec{x}_k$, where $\boldsymbol{A_k}\vec{x}_k = \vec{b}_k$.

   Gaussian Elimination can be applied to calculate the values of the unknowns, however one needs to be cautious as the matrix $\boldsymbol{A_k}$ may not be square. Therefore, during each iteration of Gaussian Elimination one must eliminate all rows of the matrix. Of course, the resulting matrix if the operation succeeds will not be upper triangular, as is the case with normal Gaussian Elimination. It will contain an upper triangular matrix on the upper part and some rows that will be full of zeros (eliminated elements), except the last column. This way one can calculate the values of all unknown elements, unless the matrix is singular.

2. The algorithm should also attempt to calculate the residual determinant d_k of $\boldsymbol{A_k}$.

   There is an important consideration here: one can easily evaluate the determinant of a square triangular matrix using $O(n)$ operations, as the product of the diagonal elements; however, if the matrix is not even square, how can one define the concept of a determinant? Furthermore, even if one defined a determinant, how could the sign of this determinant be evaluated?

   The reader is reminded that in the simple Gaussian Elimination algorithm on a $n \times n$ square matrix $\boldsymbol{S}$, the value of the determinant can be computed after the entire procedure has finished as:

$$det(\boldsymbol{S}) = (-1)^J \left[\prod_{k=1}^{n} a_{kk} \right],$$

   where J is the number of row exchanges performed during the elimination process.

   As it has been explained before, a key point that distinguishes the current linear system solution algorithm to the ones commonly used is the fact that the linear system is ill-conditioned. The case of modular linear system solvers tackling such systems has not been researched enough in practice, as most solvers are able to process only well-defined, non-singular linear systems. In our case, one can easily end up with a linear system defined by a non-square matrix. However, an intuitive, simple and fast solution was found to evaluate the needed residual determinant in such cases.

To begin, one should note that it is not that the matrix $\boldsymbol{A_k}$ needs inherently to be non-square. It 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. The new square matrix will have full row rank, i.e. all its rows will be linearly independent.

Therefore, we can define for each residual system a canonical form: a **canonical linear system** $A_C \cdot \vec{x} = \vec{b_C}$ is defined by a square matrix 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. Using this definition of the canonical linear system, 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, one can introduce an important abstraction:

If we can produce the correct solution vector using any representation of $\boldsymbol{A_k}$ that reduces its rank, why not suppose that the system is actually in the canonical form? By this way, we could keep the results and simply evaluate the determinant. Because we supposed that the system is canonical, we should expect a positive determinant; if it is negative, we can just negate it to become positive. This negation can be performed, as it is equivalent to performing one more row swap on the initial $\boldsymbol{A_k}$ and re-applying Gaussian Elimination. However, as the results would not change and only the determinant would change sign, we can simply skip the process and just return an always positive determinant. 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, as it would make the algorithm more complex than possible. However, one could argue that the process of solving the system is very similar to the one that could be used to find its canonical form. In any case, it is used only as a theoretical tool, verifying the correctness of the solution approach and limiting the simplifications one can introduce.

To conclude, after the extended version of Gaussian Elimination is performed on $\boldsymbol{A_k}$, we calculate the determinant using the triangular upper part of it and return its absolute value. So, the "canonical" determinant calculated from a given $m \times l$, $m \leq l$ matrix A_k is:

$$det(\boldsymbol{A_k}) = \left| \prod_{i=1}^{l} a_{ii} \right|$$

The followed approach could be made easier to understand if one compared the equivalent formal procedure that uses canonical linear systems (Fig. 5) with the actual one (Fig. 6).

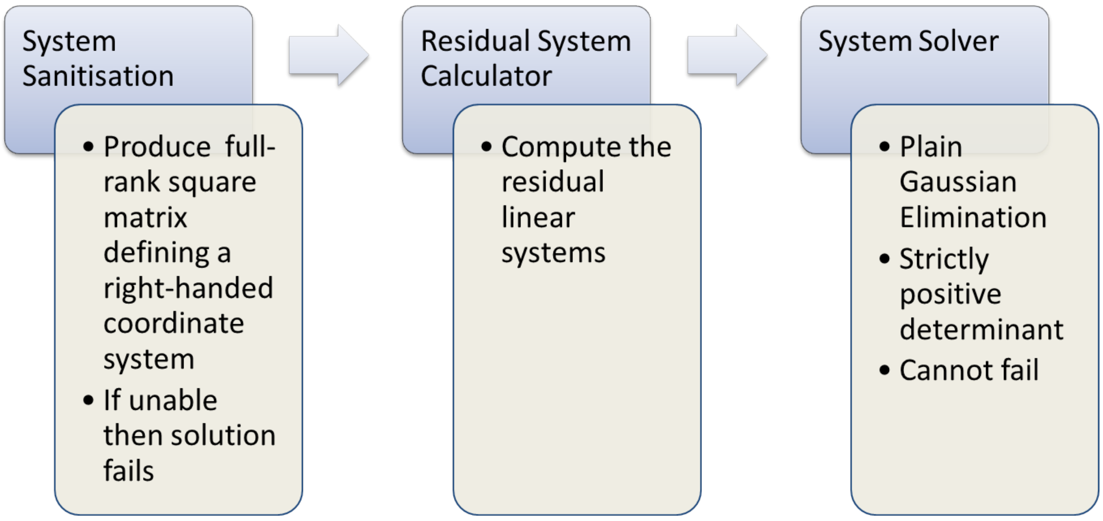

System Sanitisation
Produce full-rank square matrix defining a right-handed coordinate system
If unable then solution fails
Residual System Calculator
Compute the residual linear systems
System Solver
Plain Gaussian Elimination
Strictly positive determinant
Cannot fail

Fig. 5: *Diagram of procedure to calculate the residual determinant using canonical linear systems.*

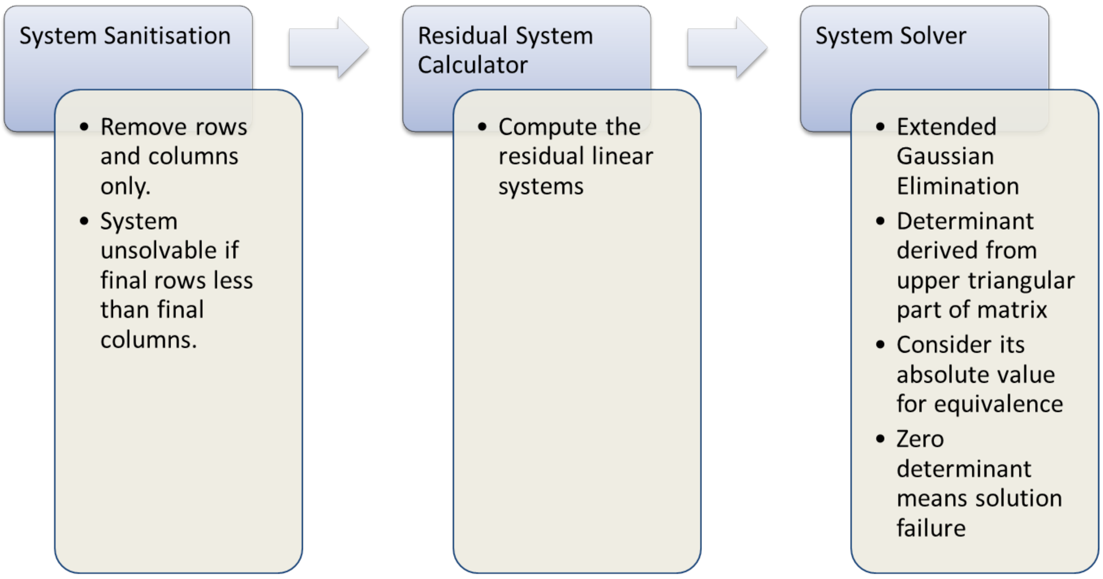

System Sanitisation
Remove rows and columns only.
System unsolvable if final rows less than final columns.
Residual System Calculator
Compute the residual linear systems
System Solver
Extended Gaussian Elimination
Determinant derived from upper triangular part of matrix
Consider its absolute value for equivalence
Zero determinant means solution failure

Fig. 6: *Diagram of the actual procedure used to calculate the residual determinant using any linear system.*

Example

Performing the extended Gaussian Elimination on both residual linear systems leads us to:

$$A_1 = A_2 = \begin{bmatrix} 6 & 0 & 0 & -3 & 1 \\ 0 & 7 & 12 & 3/2 & -1/2 \\ 0 & 0 & 24/7 & 52/7 & 27/7 \\ 0 & 0 & 0 & -9/2 & 37/8 \\ 0 & 0 & 0 & 0 & -23/216 \end{bmatrix} \quad \vec{b}_1 = \vec{b}_2 = \begin{bmatrix} 27 \\ 175/2 \\ 36 \\ -27/2 \\ 0 \end{bmatrix}$$

Using back-substitution we calculate the residual vector of solutions for both systems as:

$$\vec{x}_1 = \vec{x}_2 = \begin{bmatrix} 6 \\ 5 \\ 4 \\ 3 \\ 0 \end{bmatrix}$$

and the canonical determinant as $d_1 = d_2 = 69$ (no negation is needed).

6.3.f Step 6: 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. However, each solution vector $\vec{x}_k$ of the previous step may not contain information about all unknowns of the initial linear system produced by MoM; some elements have been deemed indeterminable in previous steps. This means that at this step, we only need to process the determinable elements of $\vec{x}_k$; the indeterminable unknowns need not be processed at all.

In order to arrive to the final solution vector $\vec{x}$, one must perform the following operations ([21], [26] and [27]):

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

   $$\vec{x} = \frac{\vec{y}}{d} ,$$

   adding information about any uncomputable elements as well.

   If conditions (6.2.2), (6.2.3), (6.2.4) and 6.2.5) were respected, then $\vec{x}$ will satisfy the initial linear system $A\vec{x} = \vec{b}$.

Two methods for obtaining a final number f from a set of residue numbers f_i are presented: the single-radix conversion (Algorithm 6.3.1) and the mixed-radix one (Algorithm 6.3.2). The choice of the recombining algorithm used is very important and can influence design choices made for earlier steps. Depending on the specific problem one has to solve and the characteristics of the linear systems, one or the other algorithm can be selected.

Algorithm 6.3.1: Single-Radix Conversion Algorithm

1) Calculate $c_i = \left(\frac{M}{m_i}\right) \ mod_{-1} \ (m_i)$ for $i = 1, 2, \ldots, s$, where mod_{-1} the modular multiplicative inverse operator.

2) Calculate the final number $f \sim \{f_1, f_2, \ldots, f_s\}$ as:

$$f = \left[\sum_{i=1}^{s} \left(\frac{M}{m_i} c_i f_i\right)\right] \ mod \ \ M$$

Algorithm 6.3.2: Mixed-Radix Conversion Algorithm

1) Calculate $c_{ij} = m_i \ \ mod_{-1} \ \ m_j$ for $i, j = 1, 2, \ldots, s$, where mod_{-1} the modular multiplicative inverse operator.

2) Calculate the following coefficients $u_1, \ldots, u_s$:

$$\begin{aligned}
u_1 &= f_1 \ \ mod \ \ m_1 \\
u_2 &= (f_2 - u_1) c_{12} \ \ mod \ \ m_2 \\
u_3 &= [(f_3 - u_1) c_{13} - u_2] c_{23} \ \ mod \ \ m_3 \\
&\vdots \\
u_s &= \{\ldots[(f_s - u_0) c_{1s} - u_2 - \ldots - u_{s-1}] c_{s-1,s}\} \ \ mod \ \ m_s
\end{aligned}$$

3) Calculate the final number $f \sim \{f_1, f_2, \ldots, f_s\}$ as:

$$f = (u_1 + u_2 m_1 + u_3 m_1 m_2 + \ldots + u_s m_1 m_2 \ldots m_{s-1}) \ \ mod \ \ M$$

The conversion algorithm must be used to calculate the final determinant and the values of the determinable unknowns. Both conversion algorithms perform $O(n^2)$ operations [26], however the mixed radix conversion algorithm is more applicable to cases where the determinant and the unknowns are comparable to the computing system's word size, therefore permitting usage of inexact arithmetic in this part [26].

On the specific problem of solving the linear systems generated by MoM, usage of inexact floating-point arithmetic is not possible as the solution vector contains normalising constants for different populations and queues. This means that the final solution can only be computed using exact arithmetic and therefore the application of the Single-Radix Conversion Algorithm using exact arithmetic is necessary.

Example

In this example, recombination procedure using the Single-Radix Conversion Algorithm will be presented:

1. We know from the previous step that $\vec{x}_1 = \vec{x}_2 = \begin{bmatrix} 6 \\ 5 \\ 4 \\ 3 \\ 0 \end{bmatrix}$ and $d_k = 69$. Therefore, we obtain:

$$\vec{y}_1 = \vec{y}_2 = d_i \vec{x}_i = \begin{bmatrix} 414 \\ 345 \\ 276 \\ 207 \\ 0 \end{bmatrix}$$

2. We will now use the Single-Radix Conversion Algorithm to obtain d from d_k. Calculations are omitted as they involve large numbers, however the final result is $d=69$.
3. Recombining all $\vec{y}_k$ vectors, dividing by $d=69$ and reinserting the indeterminable vector elements which had been omitted during the system sanitisation step leads us to the following solution:

$$\vec{x} = \begin{bmatrix} 6 \\ 5 \\ 4 \\ 3 \\ \cdot \\ \cdot \\ 0 \end{bmatrix}$$

One can easily verify that the above $\vec{x}$ verifies the initial linear system $A\vec{x}=\vec{b}$.

6.4. Theoretical Properties of the Algorithm

It is important to discuss several results regarding the effects of limiting the maximum modulo length, the scalability of the algorithm and its complexity from a theoretical viewpoint, before proceeding with the implementation.

6.4.a Maximum Number of Moduli

As it has been already presented in section 6.3.b, it is not beneficial to select infinitely large moduli. In practice, we limit the maximum length at $b_{threshold}=2{,}500$. Imagine now that the initial desired length of each modulo is just above that threshold level, for example it is $b_{threshold}+1$. This would mean that our algorithm will double the number of selected moduli and halve their length, which would become:

$$\left\lceil \frac{b_{threshold}+1}{2} \right\rceil = 1{,}251$$

in this case. One could claim that not enough primes exist with such bit length; therefore their product may not be able to be larger than $M_{threshold}$, thus causing the procedure to fail.

It is important to investigate what is the maximum $M_{threshold}$ value that is feasible in such worst cases 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, however one does not need a closed expression; $\pi(x)$ has been bounded in [32] as:

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

The above bound holds for $x \geq 355{,}991$, which is more than enough for our case. We can claim that when selecting primes according to their bit length b, at least n_b primes, where:

$$n_b = L\left(2^{b+1}-1\right) - U\left(2^b\right)$$

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:

$$n_b = L\left(2^{b+1}\right) - U\left(2^b\right),$$

which can be written as:

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

Evaluating $n_{1,251}$ 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 very far above what computer systems can handle; the limit exists only in theory and does not impose any practical limitations. This means that in such a worst case, the bit-length of $M_{threshold}$ would need to be more than $1{,}251\, n_{1,251} \simeq 2.71 \cdot 10^{752}$ in order for problems to begin to appear, which could be immediately alleviated by increasing the $b_{threshold}$ value above $2{,}500$.

6.4.b Growth Rate of $M_{threshold}$

It is very important to examine the rate at which the length of $M_{threshold}$ increases. The rate of such an increase 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 in some cases then the rate of increase 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.

The problem size is defined, as far as the modular linear system solver is concerned, by the order of the linear system; we will see in the next Chapter 8, where the experimental results are presented and discussed, that the same is true for the performance of the MoM algorithm as a whole: the total runtime depends primarily upon the order of the produced linear system.

$M_{threshold}$ Is given by eq. (6.3.8):

$$M_{threshold} = 2\,max\left\{\begin{matrix} n^{n/2}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]^n, \\ n^2\left(n-1\right)^{\left(n-1\right)/2}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]^{n-1}\left[D_{max}\left(M\left(\vec{m}\right)+R\right)\right]2^{N\left\lceil\log_2\left(D_{max}\left(N+M+R\right)\right)\right\rceil}\end{matrix}\right\}$$

We will study the length of its binary representation with regard to n. We can define this length as:

$$l_{threshold} = \left\lceil \log_2\left(M_{threshold}\right)\right\rceil$$

Applying the logarithmic operator and disregarding the constant terms leads us to the following result, where n the exact order of the linear system and not the order's representation:

$$l_{threshold} = O\left(n\log\left(n\right)\right) \tag{6.4.1}$$

6.4.c Complexity

The asymptotic computational cost of the linear system solution algorithm using modular arithmetic remains the same. We are primarily interested in the number of operations performed. We will ignore the cost of primality testing, both for simplicity and because modern primality tests are of polynomial complexity (i.e. the AKS Primality Test [33], which runs in $O(\ln^{6+\varepsilon}(p))$ [34], or even $O(\ln^{4+\varepsilon}(p))$ [35], where p the exact number to be tested). This means that their complexity may be able to be reduced further in the future.

This means that the most complex operation that needs to be performed by the Linear System solver presented in this chapter is the Gaussian Elimination, which can be performed using $O(n^3)$ operations where n the linear system order. Therefore, the core complexity of this algorithm remains the same as the simple serial one, even though constants may vary.

The 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:

$$(s+1)O(n^2)=O(n^2)$$

7. *Implementation*

In this chapter, the specific architecture, design and implementation and verification of the program built as part of this project is presented. It is important to note that much effort has been put in producing a clean, modular design adhering to good software engineering practice. This was necessary not only for theoretical design purposes; the amount of work needed in order to test the different implementation options of each version of the Method of Moments algorithm and to produce an efficient implementation far surpasses the amount of work corresponding to an MSc project. Therefore, the design and the approach used in every part must be thoroughly documented in a general algorithmic level, in a more implementation-oriented level and, lastly, in code level.

7.1. Main Features

The main features of the implemented program are the following:

- Ability to process queueing network model files adhering to a common format.
- Evaluation of the queueing network model using Convolution, RECAL of Method of Moments.
- Fast non-recursive implementation of RECAL.
- Ability to use two solvers in conjunction with the MoM algorithm: the single-threaded simple solver and the multi-threaded modular one. The application can automatically select the best one to solve a particular queueing network.
- 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}).
- The user is able to define the number of threads used is the case of the parallel MoM solver.
- Output of time and memory requirements of processing a queueing network model.
- Implementation of an API (application programming interface) in order to assist integration with JMVA

7.1.a Use-Case Scenarios

Several use-case scenarios were considered for the formulation of the program's specifications and the selection of the most important features to implement:

- “User A” may need to evaluate the queueing network model using MoM on a computer with inadequate installed RAM. He/She would like to be able to complete the performance evaluation using a non-parallel solver, that has lower memory usage. This user can be both a human (command-line) user or another program taking advantage of the provided API.
- “User B” may want to evaluate a queueing network model on a computer which supports the “Hyper-Threading” technology. He/She would like to be able to tune the number of threads used in order to reduce the competition between threads for the finite CPU and memory bandwidth resources. This user can be both a human user or another program.

- "User C" may need to process fast many very simple networks, where Convolution and RECAL algorithms can achieve superior performance than MoM. This user can only be a human.

7.2. Abstract Architecture

As it has already been noted, the basic service this implementation provides is the evaluation of performance indices, such as the mean queue lengths Q and the mean throughputs X of a specific queueing network model. Several different algorithms and customisations are available for the user to select, however the final results are the same in all cases.

The operations taking place in the system during a typical usage scenario are the following, in the order they are performed:

1. Launching of the main control process and recognition of the users' input, i.e. the command line arguments, or an API-initiated call from JMVA and recognition of the arguments passed by JMVA.
2. Initialisation of the requested solver using the possible settings specified by the user.
3. Parsing of the specified queueing network model file and loading of it in memory, if no syntax or other errors are encountered.
4. Calculation of the normalising constant G for the queueing network model. The normalising constant is then stored as a property in the model's representation in memory.
5. Calculation of the performance indices for the queueing network model. The performance indices are then stored in the model's representation in memory as well.
6. Termination of the process after all results have been printed.

Each one of the above top-level operations of the queueing network evaluator 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 main systems forming the core part of the design and architecture are represented in Fig. 7.

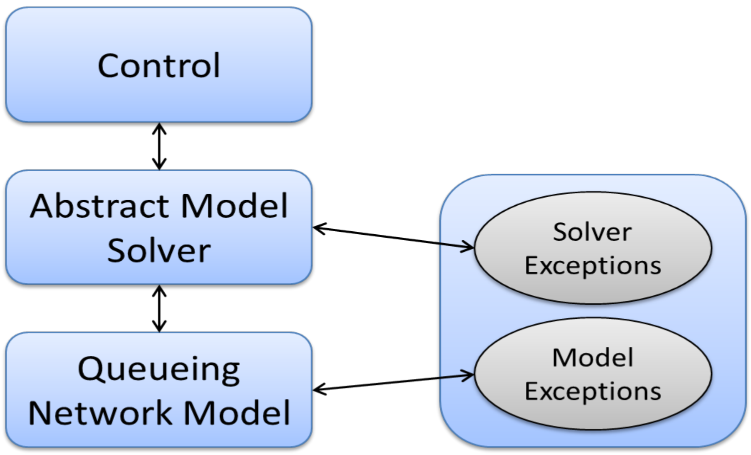


***Fig. 7:** Main systems composing the implementation*

The Control subsystem is responsible for recognising the user's input and calling and initialising the other subsystems in the correct order and using the correct arguments. Furthermore, it defines an API the JMVA application can use to "view" the current implementation as a queueing network solver, with similar operation and usage as the existing ones. To be more specific, the Control system is responsible for performing operation 1 of the above list and initiating operations 2, 4, 5 and 6. Its role is mostly "administrative", however it is fundamental for the correct communication with the user, which can be a human or another 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 normalising constant. It performs operation 3 and it supports operations 4 and 5. 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 normalising 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, which are not shown in Fig. 7. Essentially, it performs operations 4 and 5.

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.

7.3. Actual Code Design

The final implementation consists of more than 5,000 lines written in Java 6. The code is split among 29 Java classes. This makes it a relatively complex application, therefore it was tried to keep a simple design and maximise code re-use and abstractions. This is hoped to assist further contributions, in order to produce an even more complete tool.

A graphical overview of the implementation's structure is given in Fig. 8. On the topmost level, the source code is organised in 6 Java packages:

1. The Control package, which contains the Main class and is responsible to interpret the user's input and calling the other subsystems appropriately. It implements the Control component of Fig. 7. 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 the JMVA or another program. This API allows one to call the MoM algorithm using the same conventions and operations JMVA uses to call its existing solvers. This last functionality is very important, as it permits easy integration of the existing code with JMVA.
2. The DataStructures package, which 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 components of Fig. 7.
3. The Exceptions package, the classes defined in which are 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 subsystems of Fig. 7.

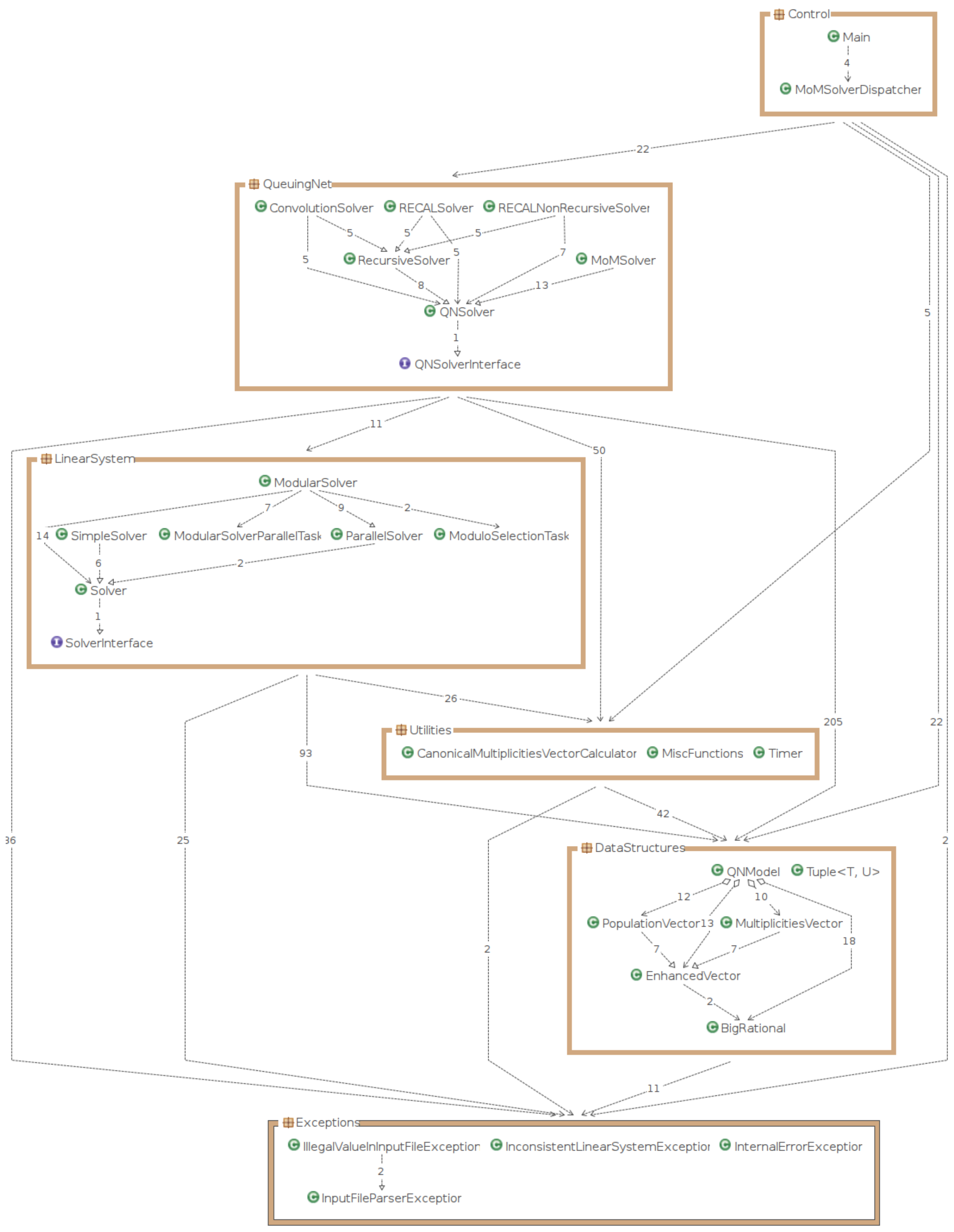


__Fig. 8:__ Structural diagram of the different packages, classes and their dependencies. We can verify that the code does not contain any tangled classes or circular dependencies, which can greatly hinder maintainability and code re-use. The diagram was produced using STAN.

4. The LinearSystem package, which contains classes that implement the various different solvers of linear systems $A\vec{x}=\vec{b}$. 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. This package supports the operations of the Abstract Solver component of Fig. 7.
5. The QueueingNet package, containing 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 of Fig. 7. All different solvers implement are designed in such a way so as to maximise code-sharing. Furthermore, they all implement a common interface, in order to be handled by the Control package in the same way.
6. The Utilities component, which contains classes implementing 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.

It is important to note that information regarding the function and usage of all classes, variables and methods have been documented in Javadoc format. This documentation can be found in the source code files of the implementation.

In the following section more specific details about the classes contained in each of the 6 packages and optimisation techniques used will be presented.

7.3.a "Control" Package

The Control package contains the Main class, which is responsible for recognising the user's arguments and performing the necessary calls to other objects in order to load the input model file into memory, evaluate it and print the results and some useful statistics about the runtime and memory usage. It can also display a message about the program's usage arguments. Furthermore, if called using particular arguments the Main class supports auto-selecting the best between the parallel of the serial MoM implementations, depending of the queueing network's characteristics, without the user's intervention. This is a direct application of the implementation characteristics discovered during the experimentation phase.

Furthermore, it contains the MoMSolverDispatcher class. This class allows usage of both the serial and the parallel Method of Moments algorithms implemented as part of this project by the JMVA implementation. Integration with JMVA is one of the main technical challenges of this project, therefore an interface needed to be implemented which would allow usage of our current solver in the same way the JMVA implementation uses its existing solver. By integrating the source code of our implementation with the JMVA code, one would be able to use an MoMSolverDispatcher object to initialise and solve a queueing network, using the same conventions, notation and method names as the other JMVA solvers contained in the jmt.analytical package of the JMT course code.

The MoMSolverDispatcher class contains a constructor, where the user needs to specify the number of classes and the total population. The rest details of the queueing network as well as the desired number of threads are passed using the input(...) method, following the convention used by JMVA. Finally, invoking the solve(...) method will result in the network's evaluation. The user is then able to access the computed results (normalising constant and performance indices) using available getter and setters, which return the values in the format required by JMVA for all solvers. The usage of the MoMSolverDispatcher object is presented graphically in Fig. 9.

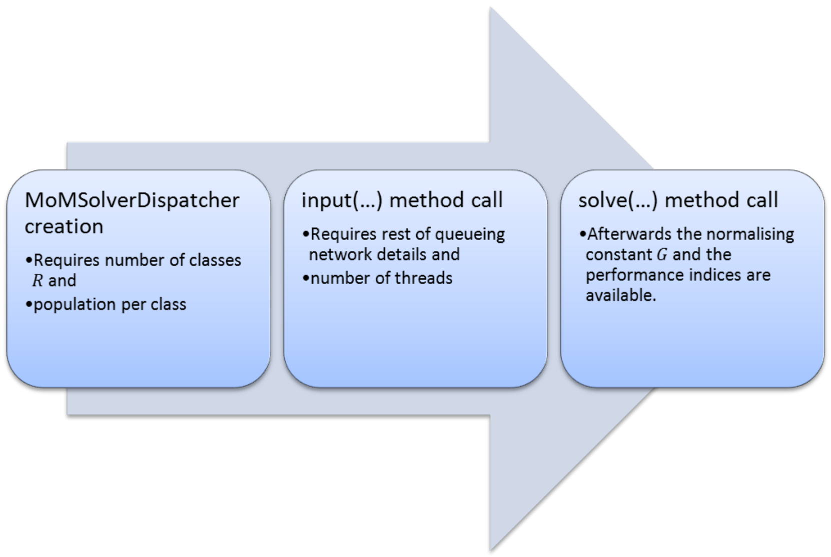

MoMSolverDispatcher creation
•Requires number of classes R and
•population per class
input(...) method call
•Requires rest of queueing network details and
•number of threads
solve(...) method call
•Afterwards the normalising constant G and the performance indices are available.

Fig. 9: *The MoMSolverDispatcher series of operations to invoke the MoMSolver. These operations must be used by an external program that makes use of the implemented API.*

7.3.b "DataStructures" Package

The DataStructures package contains 6 classes, which are used represent data the rest of the program works on as objects. This design approach was chosen in order to provided more abstract code to solve each problem, be it a linear system solution or a queueing network evaluation. This allows the code contained in the other packages to function in a very high level, similar to the one that appears in the relevant chapters 5, "Product Form Queueing Network Algorithms" and 6, "Solution of Linear Systems using Modular Arithmetic".

The following classes are contained as part of this package:

BigRational

The BigRational class is used to support all exact computations throughout the program. A special such class was deemed necessary, as the normalising constants are, in general, much bigger than the computer's word size and the maximum numbers representable by single or double precision floating point arithmetic. Furthermore, even if the normalising constants were smaller, the MoM algorithm would be unable to function using floating point arithmetic; errors introduced could propagate and lead to incorrect results, as it is an inherently arithmetically unstable algorithm.

Therefore, the BigRational class was implemented. It is able to represent arbitrarily large rational number, by storing them internally as fractions of arbitrarily long integers. All usual arithmetical and logical operations are supported, such as addition, subtraction, multiplication, division, negation, inversion, the two definitions of modulo and comparisons. Much effort has been but in optimising this library and performing an operation only if it is absolutely necessary. For example, if an operand is equal to zero then any multiplication need not be performed; this increases the throughput of operations per second, as the time needed for a multiplication is sufficiently greater than the one needed for a comparison with zero.

Due to the fact that Java does not support Operator Overloading, usage of this class for arithmetic operations may force the programmer to adapt its source code. This is however a restriction of the programming language and not of this implementation. To ease the effects of this restriction, effort has been put in using a very similar interface to Java's own implementation of arbitrary length arithmetic, namely the BigInteger and BigDecimal classes. The BigDecimal could not be used in place of our BigRational, as it cannot represent periodic numbers such as $\frac{1}{3}$.

EnhancedVector

The EnhancedVector class was built to extend the ArrayList<Integer> class with the addition of many useful methods for operations usually encountered when implementing the various algorithms that evaluate a queueing network. Such methods include easy generation of new EnhancedVector from the queueing network stored in memory, ability to find the maximum and minimum elements easily and ability to perform arithmetic operations between vectors, such as addition. Furthermore, one can easily add or subtract one from a specific element of the vector and then revert this modifications by calling a specific method; this is a very useful operation in order to efficiently implement the various recursions and iterations needed in the algorithms.

Lastly, it was chosen to base this class on the ArrayList<Integer> one instead of Vector<Integer>, as the Vector class is deprecated in current Java versions.

MultiplicitiesVector and *PopulationVector*

These two classes are based on the previously described EnhancedVector class and their contribution is primarily in enhancing type checking and promoting verification throughout the implementation. At an early stage of the implementation, it was recognised that using the same object, namely the EnhancedVector, to represent both vectors representing populations and vectors representing queue multiplicities could easily lead a programmer to errors caused by using a semantically wrong vector at a wrong position. As the Java compiler cannot recognise semantic errors, two new objects were introduced: the MultiplicitiesVector and the PopulationVector. This way the semantic difference was reflected in the syntax level and many programming errors were easily discovered during compile-time. The importance of this design choice will be made more evident when several optimisations introduced in the Convolution and RECAL algorithms are described.

QNModel

The QNModel class contains a complete representation of a queueing network model as well as all the methods that are necessary to parse the input file and provide to the rest of the implementation any needed values. This is accomplished by having several “getter” methods, that provide values such as populations, number of queues, service demands, queue multiplicities and delay times in the appropriate object type (i.e. as integers, BigRationals or other data types). Furthermore, it provides several “setter” methods that allow code using this QNModel to store the computed normalising constant and performance indices.

Apart from input file parsing and these more administrative tasks, the QNModel class is responsible to provide estimations for the maximum possible normalising constant G_{max} of the model to any classes that need it.

It was chosen to implement the model representation in an independent class in order to detach the solvers and the operations on a particular model from the specific syntax of the input file or the representation of it in the memory. Therefore, the current implementation can be easily extended to support various different input file formats.

Tuple

The Tuple class creates objects representing a 2-tuple. This is used in parts of the program there exists a relation between two specific objects, and these objects may need to be processed in pairs; an example could be for each Tuple object to represent a 2-dimensional matrix element. The objects held in each Tuple can be of any type, without this compromising type safety; this is true as Java Generics have been used.

7.3.c “Exceptions” Package

The Exceptions package contains 4 classes which extend the Java Exception object. These classes allow more improved error-handling and more meaningful error messages to the users, should an error occur. The classes contained are:

IllegalValueInInputFileException

This Exception object is thrown by the input file parser of the DataStructures.QNModel class if the input file is syntactically correct, however contains illegal values, such as decimal populations or negative queue multiplicities.

InconsistentLinearSystemException

The InconsistentLinearSystemException is thrown by any linear system solver, when it arrives at a state where it cannot calculate any of the unknowns of the linear system, i.e. all unknowns have become indeterminable elements or when the sanitised system's determinant in the case of the parallel modular solver is equal to zero. Furthermore, it is also thrown when normalising constants necessary to compute the main normalising constant G of the model or its performance indices have become indeterminable.

More details on the specific conditions on which this exception object is thrown can be found on the presentation of the LinearSystem and QueueingNet packages below.

InputFileParserException

This is an Exception object thrown by the input file parser of the DataStructures.QNModel class if the input file does not comply to the proper syntax. The correct syntax is presented in p. 95.

InternalErrorException

The InternalErrorException object may be thrown from various other classes, such as the different queueing network evaluation algorithms, the linear system solvers, the EnhancedVector implementation or even some utility classes. This exception object has the meaning that something that the program has reached an illegal state from which it cannot recover.

An InternalErrorException object can reflect a hidden bug or inadequate handling of a particular case, that leads to incorrect results. However, it is usually thrown when the MoM algorithm is used to evaluate a queueing network that is described by a singular matrix; in such cases a different evaluation algorithm should be used and the exception is not a fault of the implementation.

7.3.d “LinearSystem” Package

The LinearSystem package contains all classes related to linear system solution algorithms. These solvers are needed by the MoM algorithm, however they have the ability to function on their own; this allows simpler integration and testing of new solvers. Two different solvers can be used: the plain solver which uses Gaussian Elimination and the parallel solver that uses the modular arithmetic approach described on the previous chapter, “Solution of Linear Systems using Modular Arithmetic”.

During the design and implementation of the solvers, attention was paid in producing good quality code, with reduced memory footprint and time-complexity; the creation of new objects was kept at a minimum, as it can slow-down the system a lot. Several classes are contained in this package; we will now describe the main function, role and relations between them.

SolverInterface

The SolverInterface class defines the interface every linear system solver must support. Several basic methods are supported; among others, one can specify the linear system matrices A and $\vec{b}$ and request the solution of the linear system.

This interface allows a common the MoM Solver to be agnostic of the specific implementation of the linear system solver.

Solver

The Solver class implements SolverInterface and includes declarations of several variables and implementation of methods needed by any other solver, such as timing functions, shut-down routines etc. However, it cannot solve any linear system; the actual solution code is provided by the more specific subclasses of the actual solver used.

SimpleSolver

The SimpleSolver object extends the Solver one. It implements a basic solution of the linear system using Gaussian Elimination. However, even this solver differs substantially from the usual Gaussian Elimination algorithm; this is because it has to solve singular systems and maintain robustness when encountering indeterminable values and unknowns.

ParallelSolver

The ParallelSolver class extends the Solver one and provides all the necessary declarations and definitions dealing with different threads, thread pools, shut-down routines and parallelisation. Any new Solver object that needs to be executed in parallel can rely on the implementation offered by this class.

ModularSolver

This class implements a part of procedure used to solve linear systems using exact arithmetic, as it is presented in the previous chapter, “Solution of Linear Systems using Modular Arithmetic”. In particular, the solver method performs the following two operations during the initialisation phase:

- Evaluation of a lower limit for the product of moduli ($M_{threshold}$) based on information about the maximum anticipated matrix elements and the maximum normalising constant values provided by the MoM Solver.
- Selection of the best applicable moduli by taking into account $M_{threshold}$ and the number of parallel threads requested by the user. The maximum bit-size of the modulo is limited and cannot be more than 2.500 bits; each modulo's size is derived using the iterative procedure presented in Section 6.3.b (p. 38). Afterwards, the selection is performed in parallel to reduce runtime, by invoking ModuloSelectionTask objects. The selection of moduli is performed only once per MoM invocation. Due to the fact that the modulo size is related to the magnitude of the integer in the system, which grows linearly from iteration to iteration, the moduli size would need to grow as well. Experimentation has proven that it is beneficial to perform the moduli selection only once

per MoM iteration and not for each particular model, in order to decrease the total computational cost.

After the solver has been initialised, a solution of the liner system can be requested using the solve(...) method. In order to solve the system, the solver takes advantage of several important theoretical properties than can speed up the process and reduce memory usage. This will be explained as the actions needed to solve the system are presented:

- Firstly, the solver must check if the $\vec{b}$ vector provided contains exclusively indeterminable values. This would mean that in principle, the linear system is void and contains no meaningful equations; if this is true, then solution cannot proceed.
- Secondly, system sanitisation is performed using the methods presented in the previous chapter.
- Thirdly, the solver needs to instantiate the different residual systems that need to be solved. The residual representation of matrix A is the same between all different moduli if each modulo is greater than the maximum absolute element of A and the modulo operation allowing negative results is used. Therefore, the calculation of the residual matrix can be performed only once for all residual systems and the number of necessary operations is reduces. This is not true for the residual representations of the far smaller vector $\vec{b}$, which must be calculated normally.
- The next step is to produce as many instances of residual system solver objects, i.e. objects of type ModularSolverParallelTask. Each such object can solve a residual linear system independently and return the canonical residual determinant and the residual values of the unknowns. In order to maximise efficiency, a static pool of threads, by default one per processing core, is used for all solver invocations. If the number of residual systems than need to be solved is greater than the number of cores, then no more residual systems than the number of cores are evaluated at a given time. This approach reduces memory bus congestion and cache competition between the different threads.
- Finally, the results are recombined using the Single-Radix Conversion Algorithm and returned to the caller of the solve(...) method.

ModularSolverParallelTask

The ModularSolverParallelTask object is an extension of Java's Callable object. This allows easy parallelisation using native Java libraries and allows the focus of the programmer to remain on the problem's parallelisation and not on more “administrative” tasks such as thread “house-keeping” and synchronisation.

This class can be prepared to solve a sanitised residual linear system using the prepare(...) method. Subsequent invocation leads to solution using the Extended Gaussian Elimination algorithm. After this operation has been performed, the residual values of the determinant and the unknowns are available to the caller. An InconsistentLinearSystemException object is thrown if the residual determinant is zero.

ModuloSelectionTask

Parallelisation of the modulo selection process is necessary, as it can account for a significant part of the runtime when evaluating complex models. Therefore, the selection process was parallelised using the same approach as the previously described ModularSolverParallelTask. One can prepare a ModuloSelectionTask object using the desired bit-length of the number to prime number; on invocation, a prime of such length is chosen and returned.

The Java built-in method used to select a prime is probabilistic, but it can guarantee that the result may be a composite number with probability less than 2^{-100}. This is sufficient for the requirements of this project.

7.3.e “QueueingNet” Package

The QueueingNet package contains the implementations of the different algorithms to evaluate the performance of a queueing network model. Separation of this implementation from the underlying data structures, as well as subclassing, has resulted in clean and high-level code which is independent of the actual underlying implementation. As in the case of the “LinearSystem” package, various different solver object are contained in this package; all object conform to a particular interface and hide code-sharing through subclassing. The main function and operations of each class will now be described:

QNSolverInterface

This is an interface every other queueing network solver object must implement. It defines the way several basic methods have called. For example, among the methods defined is the method that initiates the computation of a normalising constant G, the method that computes the performance indices (mean throughputs and mean queue lengths) and other methods that print useful statistics about the needed runtime and memory usage.

QNSolver

The QNSolver object implements the previous interface and declares several variables and method implementations that are common for all solvers. However, it does not provide any concrete algorithm to evaluate a queueing network or to compute a normalising constant; these are provided by QNSolver's subclasses.

RecursiveSolver

A RecursiveSolver provides some methods and optimisation infrastructure which is needed for the case of the Convolution and RECAL algorithms. There are numerous similarities between these two algorithms. For example, both Convolution and RECAL use the same initial conditions in order to compute the normalising constant; the reader can refer to p. 20. Therefore, the methods needed to calculate these initial conditions can be shared between the implementations using subclassing. Furthermore, both Convolution and the plain recursive implementation of RECAL may need the value of a particular normalising constant, i.e. the normalising constant corresponding to a particular population vector $\vec{N}$ and multiplicities vector $\Delta\vec{m}$,numerous times throughout their execution. Thus, if this constant were to be recalculated every time, the final algorithm would be much slower as many operations would need to be repeated.

As a result, an efficient infrastructure has been implemented to store every intermediate computed normalising constant. This structure relies on the implementation of Java HashMaps, which are able to store and recall pairs of a key and a related value. Addition of a new constant and recalling of an existing one are very fast operations when using HashMaps.

The reader can refer to Fig. 10 in order to understand the exact way in which this store-and-recall data structure functions. For example, if one wants to recall the normalising constant $G\left(\vec{m}_{2,}\vec{N}_2\right)$, then it must first select the population $\vec{N}_2$ at the initial level. The initial level is actually a HashMap that relates PopulationVector objects to other HashMaps that relate MultiplicitiesVector objects to normalising constants. Therefore, by selecting $\vec{N}_2$ the user can access a set of MultiplicitiesVector objects. At this second level, the user can select the specific vector $\vec{m}_2$; this results in the return of the desired normalising constant $G\left(\vec{m}_{2,} \vec{N}_2\right)$, if it has been previously computed and stored in this structure. If the desired population or multiplicities vector cannot be found, this means that this is the first time evaluation of the corresponding normalising constant is attempted; therefore calculation must be performed manually using the recursive relation of the algorithm in use, i.e. either the Convolution Expression or the Population Constraint (see p. 20), until the initial conditions are reached or another already computed constant is found.

This caching method is used for all computed constants in the case of ConvolutionSolver and RECALSolver classes. The RECALNonRecursiveSolver class uses the same structure to store the constants it needs, but it evades the need to store all computed values. The precise way this is performed will be presented below.

ConvolutionSolver and RECALSolver

These two classes extend the RecursiveSolver one and implement queueing network evaluation using the Convolution and RECAL algorithm respectively. Both algorithm are implemented in their recursive form and use the structure of Fig. 10 to store all constants they compute, greatly speeding up the computation at the cost of increased memory usage.

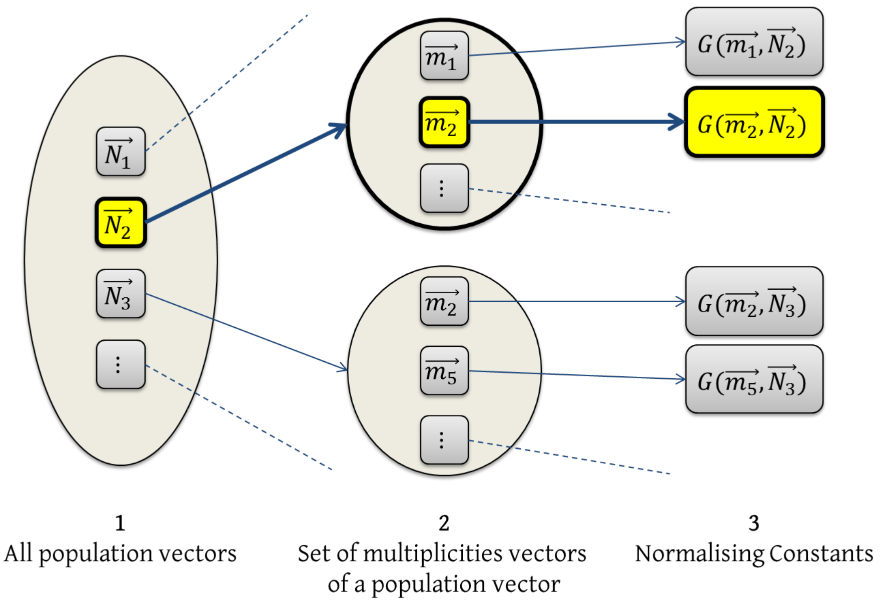


Fig. 10: *Diagram of the structure used to store computed normalising constants by the RecursiveSolver class. To recall $G\left(\vec{m}_{2,} \vec{N}_2\right)$, we select $\vec{N}_2$ at the first level and $\vec{m}_2$ at the second one.*

RECALNonRecursiveSolver

The RECALNonRecursiveSolver class implements the known RECAL algorithm that relies on the Population Constraint expression. However, instead of evaluating the final normalising constant recursively starting from the final element and gradually leading to the initial conditions as does RECALSolver, this class evaluates the same constant by evaluating the intermediate constants starting from the initial conditions and leading to the final desired one. This approach increases the speed of calculations on large models with delay times and reduces significantly the memory footprint in comparison with the RECALSolver implementation. This is accomplished as the normalising constants are computed in a specific order, level by level, thus at any time requiring the results of only the immediately previous recursion level. Essentially, it performs the regular recursion defined by the Population Constraint expression backwards. In contrast, the other RECALSolver implementation requires storing the results of all previous levels, as the constants are not considered in an ordered fashion.

RECALNonRecursiveSolver relies on the structure provided by the RecursiveSolver and picture in Fig. 10 to store the normalising constants it computes. However, the stored results corresponding to "older" recursion levels are deleted after each new level is completed.

MoMSolver

The MoMSolver class is a direct subclass of the QNSolver one and contains all the implementation of the MoM algorithm, as described in section "Method of Moments (MoM)", p. 22. Apart from the usual solver methods, this class contains an initialisation method, at which the size of the linear system that describes the queueing network and must be solved at every iteration is calculated and other variables are initialised. Furthermore, it contains a method used to generate the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ used by MoM based on Algorithm 5.4.1. Generation of these matrices can take a substantial amount of time when evaluating bigger models, therefore an optimisation was introduced:

When the MoM algorithm performs the recursion on the population of a specific class, it is known that the matrix $\boldsymbol{A}$ for succeeding populations differs only by the addition of 1 at certain matrix position. These positions are known at the time the matrix for the first population of this particular class is generated. Therefore, our implementation stores these positions at a list and updates only them by adding 1 at every iteration. When all populations of this class have been considered, a new set of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ must be generated from the beginning. This optimisation increases the speed of the algorithm, as by using it the cost of generating all matrices $\boldsymbol{A}$ is substantially reduced.

Furthermore, the MoMSolver class writes elements of matrix $\boldsymbol{A}$ by using a special method. This way, the concrete implementation of the matrix could be easily changed without modifying code other than this "writer" method. Therefore, introduction of a sparse matrix representation for this matrix in the future can be quite straightforward, requiring minimal modification of the MoM solver.

Two other important methods contained in this class are the one that computes the normalising constant of the model and the one that computes its performance indices. The normalising constant is computed using Algorithm 5.4.2. The current implementation takes advantage of the optimisation described above for easy updating of the matrix $\boldsymbol{A}$. More importantly, the code that computes the normalising constant is at a high level – close to the one presented in this report as pseudocode – and can function with any linear system solver that complies with the LinearSystem.SolverInterface interface class. However, the linear system solver must exhibit the robustness and the ability to deal with singular systems of the algorithms designed and implemented this project; one cannot simply import a new solver like the ones available in most linear algebra libraries, without modifying its core heavily.

Lastly, the current class provides a method with can calculate the maximum possible absolute element of $\boldsymbol{A}$, given the current model. We have seen that such an estimation is crucial for the selection of moduli in the case of the parallel modular linear system solver.

7.3.f "Utilities" Package

The Utilities package contains classes the methods of which perform general operations that are needed in various parts of the program. Such operations do not have any connection with a particular queueing network evaluation algorithm or any other feature of the current implementation. Furthermore, they are independent of the rest of the implementation and, therefore, could be immediately re-used in another program. Three classes are contained:

CanonicalMultiplicitiesVectorCalculator

The CanonicalMultiplicitiesVectorCalculator class implements several methods that can assist in producing an ordered set of vectors by several rules. Vectors of equal length are ordered by the sum of their elements; if they have the same sum, the one with the leftmost non-zero element is considered greater.

This method provides a canonical ordering of the vectors. It is useful, because this way one can consider all vectors summing up to a given number in an ordered manner. This class is being used in the MoM implementation in order to produce all elements of a basis of moments and to generate matrices $\boldsymbol{A}(\vec{N})$ and $\boldsymbol{B}(\vec{N})$. Furthermore, it is used in the non-recursive implementation of RECAL, in order to generate the set of vectors that correspond to each iteration.

MiscFunctions

The methods of this class are used in various points of the program, as they support general operations the application of which is not limited to a particular queueing network evaluation algorithm or linear system solver.

Examples of such operations include calculation of binomial coefficients and factorials, methods to print 1- or 2-dimensional matrices in various formats, methods to create new arrays of BigRationals from other arrays containing double precision floating point numbers or integers and methods to efficiently perform matrix-vector multiplications. Other methods contained in this class allow efficient creation of a matrix copy, i.e. copies of all elements and not a simple copy of the references – using Native System Calls of Java implemented using the Java Native Interface. This enhances the speed at which copies of very large size arrays are performed, such as the big sized MoM arrays filled with BigRational objects, without compromising portability; the Java Virtual Machine is responsible to perform the actual native calls. Lastly, there exists a method which can be used to discover the amount of memory actually in use by the Java Virtual Machine.

Timer

A Timer object can be used to easily time and benchmark any method of a class. The Timer object is created and then handled using start() and pause() method calls. When the desired operation has finished, the interval between all start() and pause() calls can be returned, either in msec or as a more readable string representing the interval in hours, minutes, seconds and subdivisions.

7.4. Testing and Verification

An important part of the implementation process was the testing and verification of the program's output. In order to verify that all different algorithms and solvers are working correctly, system-level testing was performed that involved cross-evaluation of the computed normalising constants and performance indices for models with varying number of queues, classes, populations and multiplicities. Furthermore, these results were also compared with other existing reference implementations written in C, MATLAB and MAPLE.

8. Experimental Results and Comparison

8.1. Introduction

During the design and implementation phase of this project, there were many details and contradicting approaches that needed to be investigated. The different optimisation techniques needed to be benchmarked, whereas both the time and space complexity of the different algorithms to evaluate queueing networks needed to be taken into account for the production of the final result.

In order to be able 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 needed to be created and used. This **set of Benchmark Models** should have the following characteristics:

- Some Benchmark Models should be of **comparable complexity to the requirements of queueing network models of real applications**, such as those examined in [4] and [36].
- **Several workload profiles should be defined**, corresponding to low, medium and heavy load for each model. The load is represented by the number of jobs circulating the system.
- **There should exist Benchmark Models inside the test set which can be evaluated by all different algorithms.** This is important, because the established algorithms other than MoM are usually unable to evaluate models of real-life practical interest; this is true due to the impractical time and/or space complexity of the respective algorithms.
- **Both favourable and unfavourable models should be contained for all algorithms.** It can be expected that when comparing so diverse algorithms such as Convolution, two implementations of RECAL and two implementations of MoM that use linear system solvers with difference performance and overheads, one of them cannot be the optimal choice in terms of speed in all test cases. Such a purpose-built test set could mislead us in estimating that the “optimal” algorithm would be the best case in all real-life cases as well, when actually it would be better to obtain a more balanced point of view.
- Lastly, the selection of the Benchmark Models should be **independent of the specific hardware used to quantify the implementation's behaviour**. This is more beneficial from a research point of view, as some of the algorithms presented as part of this report can be easily applied in different computing architectures. For example, parallelisation can yield quite different results when implemented on a shared memory multiprocessing environment with congested memory buses and when implemented on a distributed memory parallel architecture [37].

In order to satisfy all these requirements, a set of 48 queueing network models was used. These models 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. 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 space covered by all these different models is represented graphically in Fig. 11.

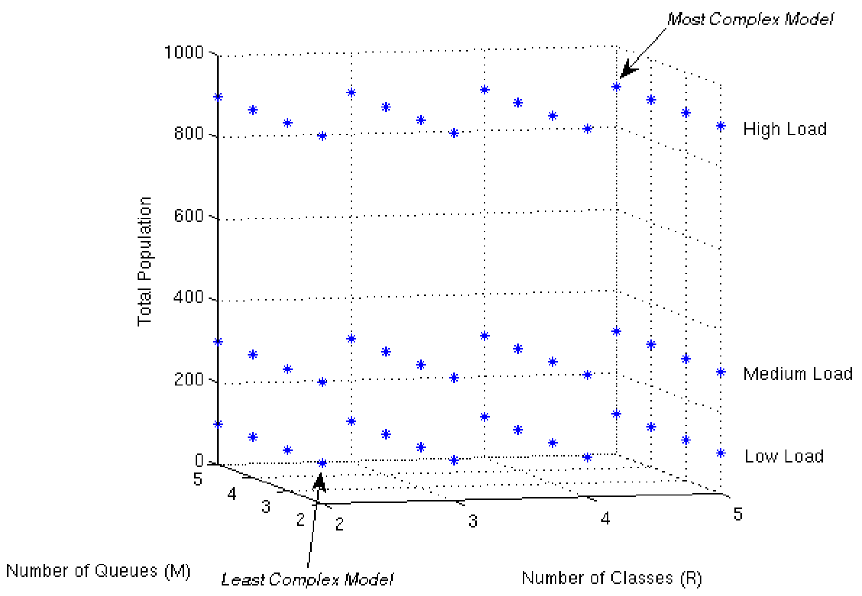


***Fig. 11:** Graphical representation of the set of queueing network models used for evaluation and testing purposes. Each star represents a model - a total of 48 models are used.*

8.2. Testing Procedure

The implementation, written on Java 6, was tested and timed at the picard01.doc.ic.ac.uk machine which has 2 Intel Xeon Processor E5540 (8M Cache, 2.53 GHz, 4 real cores each plus Hyper-Threading) and 32GB RAM. The operating system was Ubuntu 9.04 with GNU/Linux kernel 2.6.28-15-generic(x86_64) and the Java Virtual Machine used was Oracle's JVM, Version 6 Update 16. **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**. The processes were run using a non root-privileged user with niceness level (priority level) equal to 0. Other processes running on the same host simultaneously with the tested program may have influenced the runtime. Therefore, each test case was run multiple times; this led to more consistent results. 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 calculated.

8.3. Results and Comments

Before proceeding with the presentation and commenting of the results, it would be beneficial to provide the reader with an overview of the models that were able to be solved by each algorithm. In the following figures 12-15, models that were successfully evaluated within the time and memory constraints are represented by green markers, whereas unsuccessful run cases are represented by red markers. The reader is reminded that in general, the closer the marker is to the far upper right corner of the diagram, the harder the model is to evaluate.

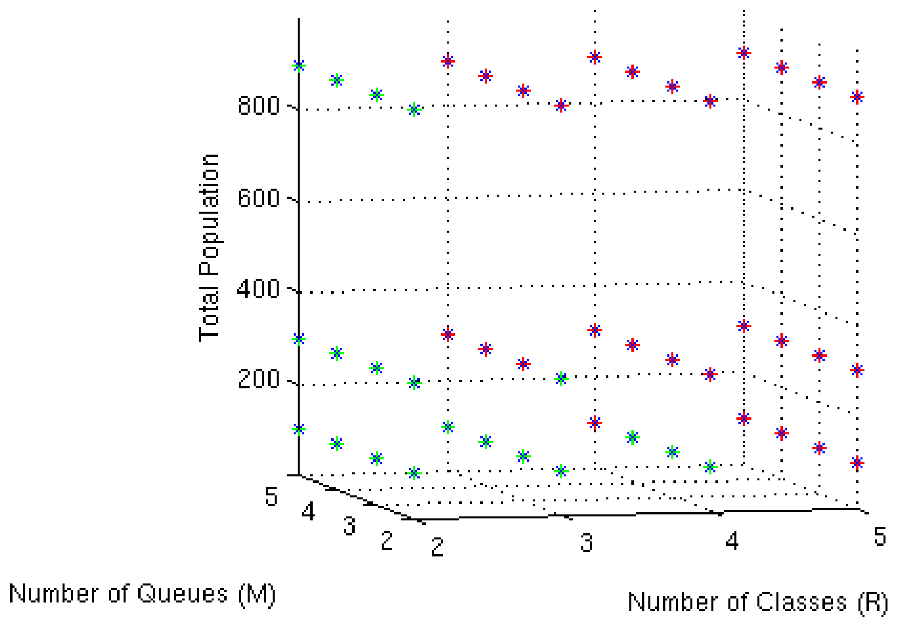


***Fig. 12:** Test-cases evaluated successfully by the Convolution algorithm are represented in green. We can verify the $O(N^R)$ complexity of the algorithm, as fewer models are feasible as the number of classes increases. 42% of the test-set models are feasible.*

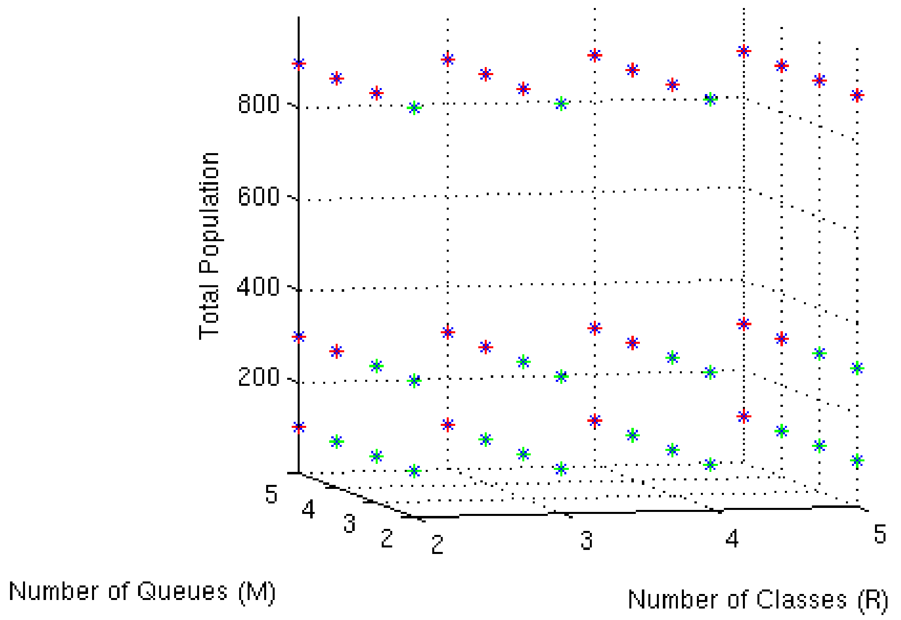


***Fig. 13:** This figure represents the same as Fig. 12 but for RECAL. The algorithm's $O(N^M)$ is confirmed, as RECAL can solve models with more classes than Convolution, but exhibits worse results as M increases. In total, 46% of the models are feasible.*

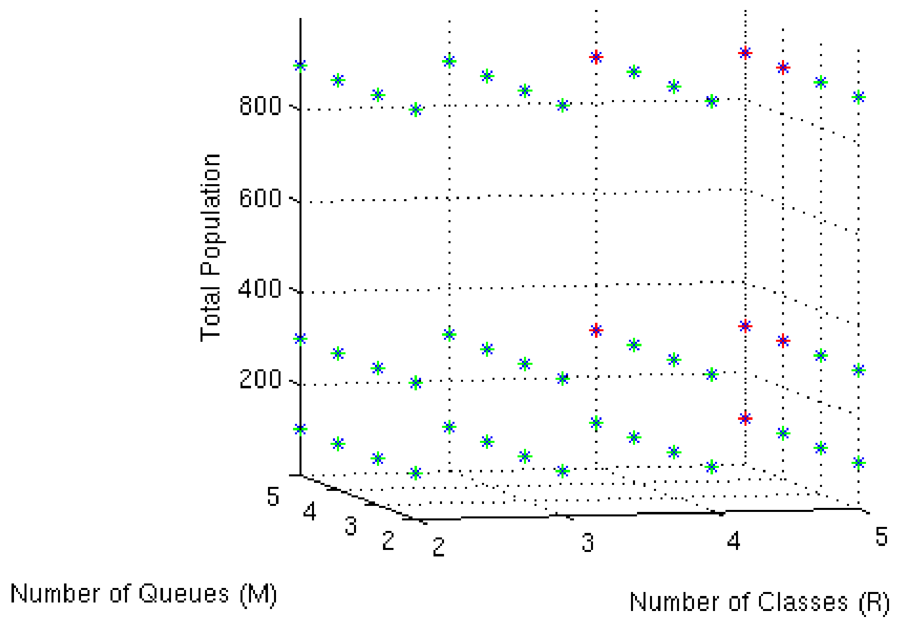


Fig. 14: *Same as previous figures but for the serial version of MoM. The algorithm is much more efficient than the previous ones. 85% of the models tested are feasible.*

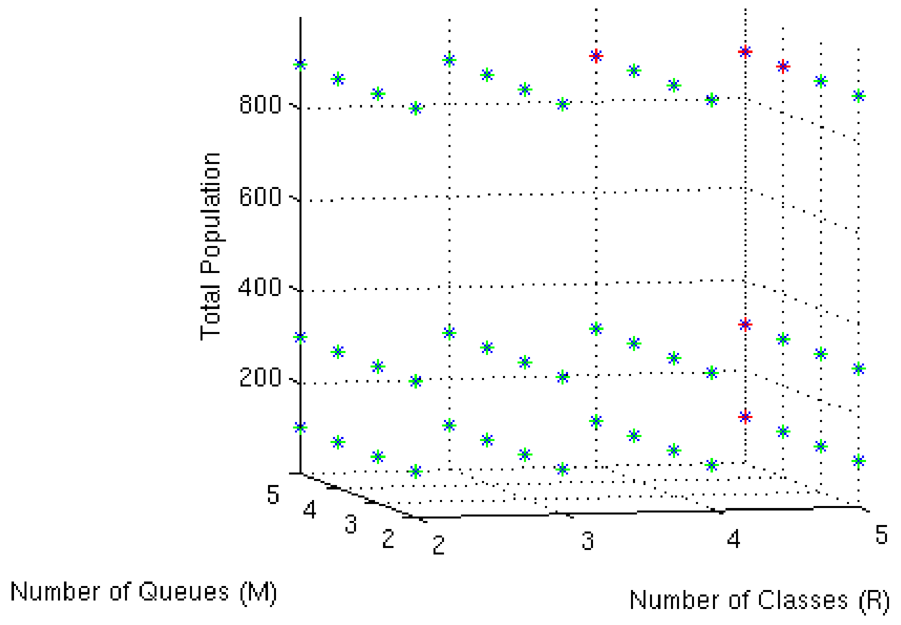


Fig. 15: *Same as previous figures but for the parallel versions of MoM, which can all evaluate the same models. The cost of solving complex models grows by* $\left[\binom{M+R}{R}R\right]^3$*, which is faster than the rate at which processors are added. 90% of the total models are feasible.*

8.3.a Convolution

As expected, 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 and space asymptotic complexity which is equal to $O(N^R)$, where N the total population of jobs.

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 ($R=2$) were able to be evaluated within the time and space constraints for all three load levels. The runtime and the memory usage for the model sets that could be completely evaluated are presented in the following Fig. 16 And 17.

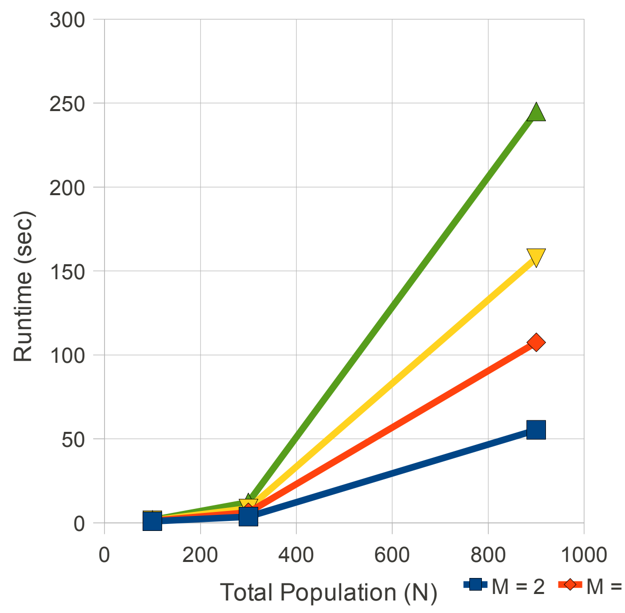


Fig. 16: *Total time needed to evaluate model with 2 classes for varying M.*

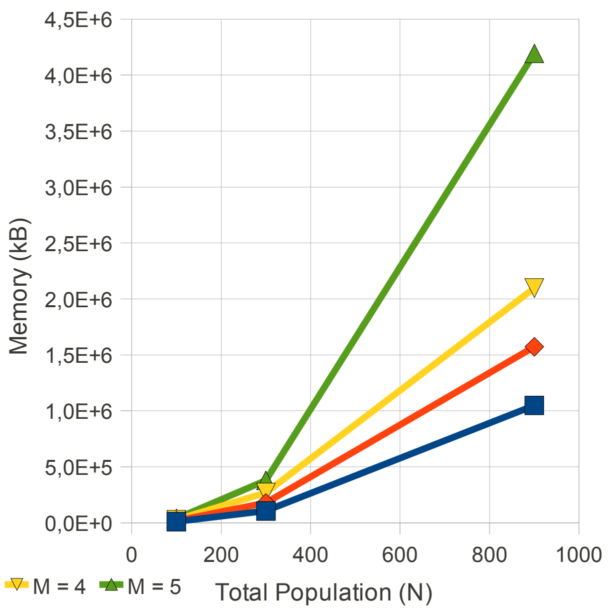


Fig. 17: *Total memory usage when evaluate model with 2 classes for varying M.*

From the above diagrams becomes evident that **the algorithm does not scale well enough**. Even though the total runtime and memory usage for the small models presented in the figures is very large but not prohibitive, one has to note it grows very fast as more classes are added. For example, out of the four queueing network models with $R=3$ are only one can be evaluated without failure for the case of medium load ($N_{tot}=300$); high load ($N_{tot}=900$) fails in all four cases.

8.3.b RECAL

The performance of the RECAL algorithm is hampered by its high asymptotic complexity, which is $O(N^M)$ for both space and time, where N the total population of jobs. **Even though RECAL in general performs better than Convolution, it still does not scale sufficiently well.** The runtime is very sensitive to the number of queues M.

Worse than Convolution, 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).

The runtime and the memory usage for the those models with $R=2$ and $R=3$ that could be evaluated are presented in Fig. 18 and 19 respectively.

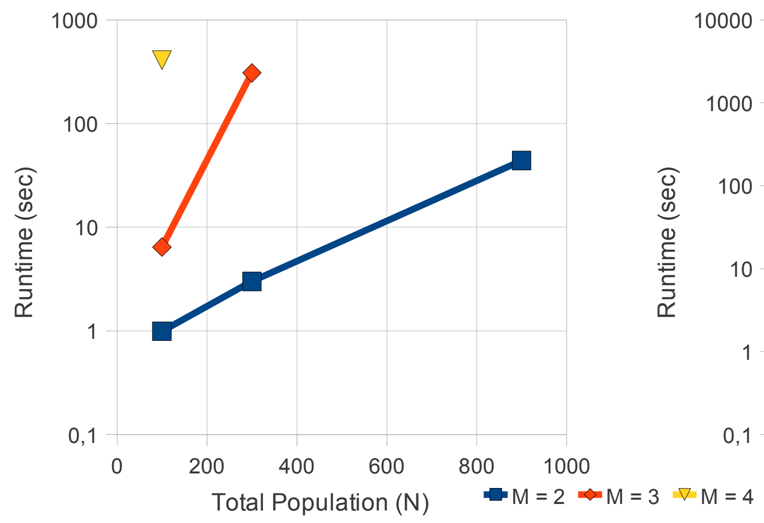


Fig. 18: *Total time needed to evaluate model with $R=2$ classes for varying M. Models that could cot be computed are not presented. No model with $M=5$ was feasible within the test constraints.*

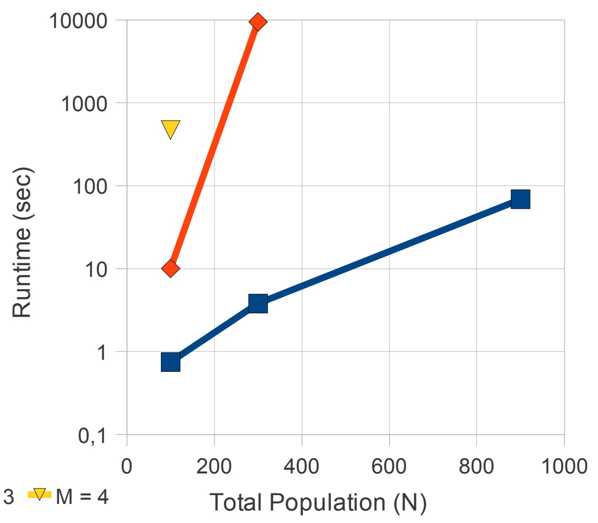


Fig. 19: *Same as Fig. 18 but for $R=3$.*

8.3.c Method of Moments

Introduction and Memory Usage

An MoM algorithm is in almost all cases **the fastest way to evaluate the queueing networks models** contained in the Benchmark Set. It is the only algorithm that has not prohibitive time requirements in order to evaluate the more complex test-cases. Furthermore, the primary computational advantage of MoM is its **low space complexity**. Even when the parallel modular linear system solver is used, which is characterised by higher space requirements that the serial version, the memory usage does not exceed a few tens of megabytes (MB). This is true even for the most complex of models. For comparison, the same memory usage on simpler models can exceed some tens of gigabytes (GB) when one uses.

This difference becomes evident if ones compares the memory requirements of the different algorithms in order to evaluate a low complexity model. This comparison is represented in Fig. 22 (p. 76); the memory is represented on a logarithmic scale.

The memory usage in the case of the MoM algorithm is mainly related to the size of the matrices A and B that are used to represent the queueing network. When the parallel solver is used, one has to store simultaneously in memory as many instances of the matrix A as the number of selected moduli; However, this does not impact negatively the memory requirements, as the number of moduli is usually limited. For example, one can verify this behaviour by comparing the memory usage of the various MoM implementations when evaluating the model with $R=4$ and $M=3$ for the tree load (population) levels; this is presented in Fig. 20.

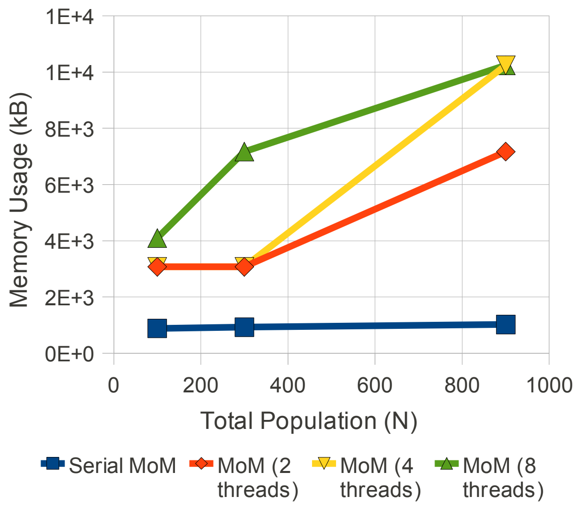


Fig. 20: *The memory requirements of evaluating the queueing network model with R=4 and M=3 are compared. The memory usage is quantised at MB units.*

Total Runtime results – Parallelisation

Without any doubt, the MoM algorithm in general is the fastest possible algorithm to evaluate queueing network model. This is reflected by the fact that the parallel MoM implementations are able to evaluate 43 out of the 48 networks contained in the test set within the 1 hour limit of maximum runtime; the serial MoM algorithm is able to evaluate 41 of the 48 models.

In order to qualitatively examine the superiority of the MoM algorithm in comparison with the established ones, one can compare the time needed by all algorithms to evaluate a single, typical, model. This model must be relatively simple, in order to be able to be evaluated by all algorithms; therefore, the case where $R=4$ and $M=3$ was chosen, with population of $N_{tot}=100$ jobs (low load level). The runtime is presented in the following Fig. 21, whereas memory usage for the same case is presented in Fig. 22; it has to be noted that this model probably brings MoM at a disadvantage, as it is not the typical case where the MoM or the parallel implementation can exhibit all its efficiency.

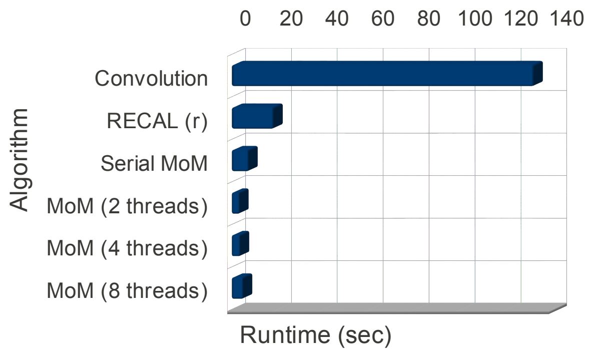


***Fig. 21:** Total Runtime needed to evaluate a queueing network with $R=4$, $M=3$ and $N_{tot}=100$. This case shows how faster all MoM implementations are in comparison with Convolution and RECAL. Even though RECAL achieves a good result in this case, it cannot scale efficiently as the number of jobs increases.*

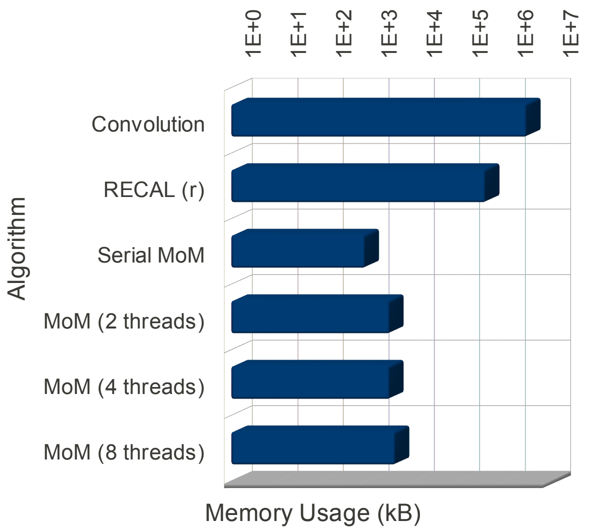


***Fig. 22:** Memory required to evaluate a queueing network with $R=4$, $M=3$ and $N_{tot}=100$. This case highlights the low space complexity of all MoM versions.*

It is very interesting to **compare the performance of MoM when using the serial Gaussian Elimination solver and when using the parallel linear system solver that uses modular arithmetic**. However, before proceeding with the comparison, there are several details about the factors that influence the performance 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.** For the reader to understand this better, we present in the following Fig. 23 the breakdown of the total runtime percentage in other functions; namely the time needed by the MoM solver itself, the time needed to select the moduli (if applicable), the time needed to solve the linear systems and the time needed to perform the matrix-vector multiplication on the right part of eq. (5.4.1):

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

  This matrix-vector is necessary in order to obtain the $\vec{b}$ vector of the linear systems representing the next population. The queueing network model tested is the one with $R=M=4$, for all three load cases. Qualitatively similar results are obtainable from any MoM execution.

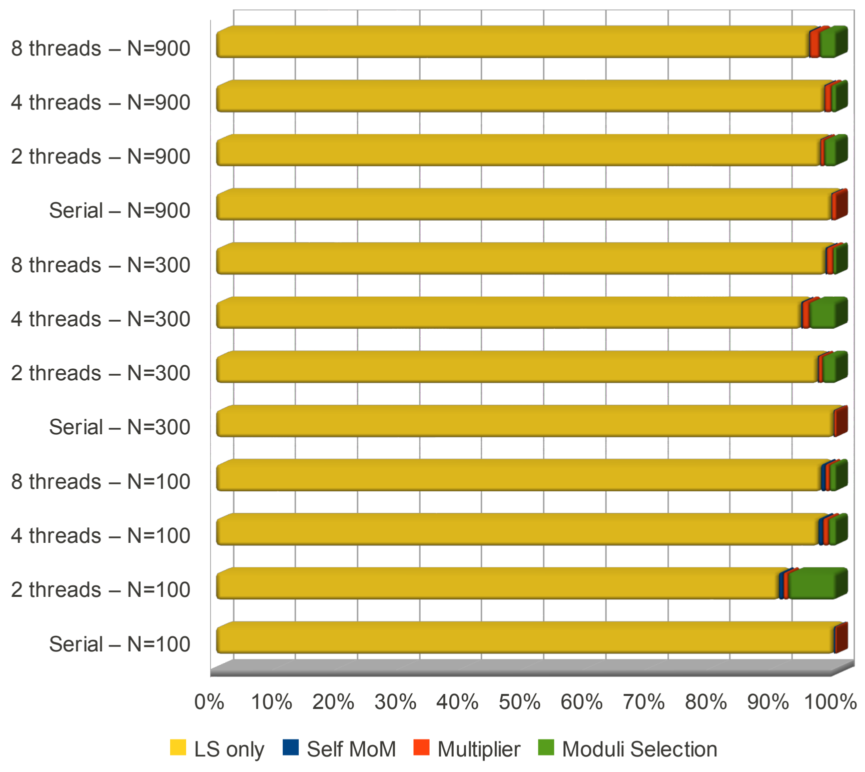


Fig. 23: *The runtime percentage breakdown on different sub-operations. The graph pertains to the model with R=M=4; however, qualitatively similar results are obtained from any MoM execution.*

As it can be seen in Fig. 23, MoM's runtime is in all cases dominated by the time needed to solve the linear systems. Other operations, such as the selection of moduli if it happens to be a parallel case or MoM's self time, usually take a little amount of time: in most cases less than 4% and a few cases between 4-10%.

The reader has to note that throughout the report except Fig. 23, the amount of time needed to select the moduli in the parallel cases is included in the linear system's solver runtime.

- **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 the magnitude of the values contained in each of them, which grows monotonically as the total number of population considered increases, thus making each subsequent solution a little bit harder.

  The runtime of the parallel MoM exhibits a mild superlinearity as the population increases. This can be explained when someone considers the modulo selection strategy used, the total number of moduli selected, the relation of each modulo to the normalising constant and the specific computational model used for the implementation, in terms of hardware, software and their inter-operation.

  The above can be seen in Fig. 24, where the runtime for each of the MoM implementations when evaluating all load cases of the same queueing network as before ($R=M=4$) is presented.

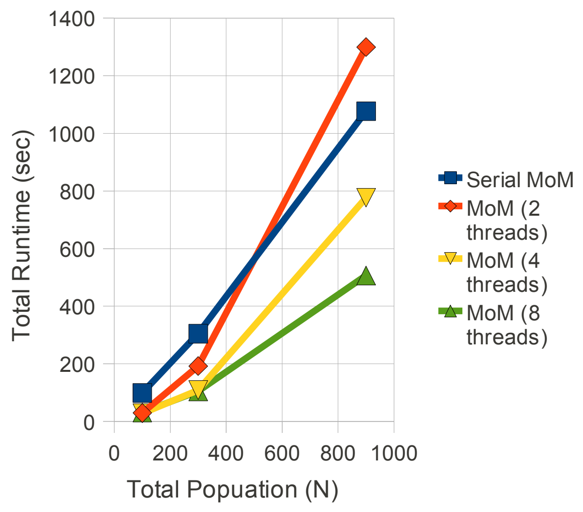


Fig. 24: *Total runtime for each MoM implementation when evaluating the queueing network model with R=M=4. The 2-threaded version results in a slowdown in the high load scenario ($N=900$), as 8 residual linear systems need to be solved in pairs. The same version selects 4 residual systems for the medium load case and still manages to be faster than the simple serial version.*

In Fig. 24 it is seen that in some extreme cases, the parallel implementation may need to solve a large number of residual systems, resulting in increased runtime when compared to the serial version. This is only encountered when using a small number of threads; adding more solves the problem and produces a substantial speedup. It must be noted that this is just an extreme case and that the average speed-up for all three load cases is higher, as presented in Fig. 26 (p. 80).

- The time complexity of solving a linear system has $O(n^3)$ asymptotic complexity, where n the size of the size of the matrix. The same asymptotic complexity is true for the case of the parallel modular arithmetic solver as well; this is because each of the residual systems solved in parallel has the same size as the initial one. However, the constant that is multiplied by $O(n^3)$ is different and causes the different performance of the parallel and the serial approach.

  As it has been presented in section 5.4.b (p. 22), the order of the linear system generated by MoM for a given network model is equal to:

$$\binom{M+R}{R}R$$

The orders of the linear systems corresponding to all queueing network models of the test-set are presented in Fig. 25.

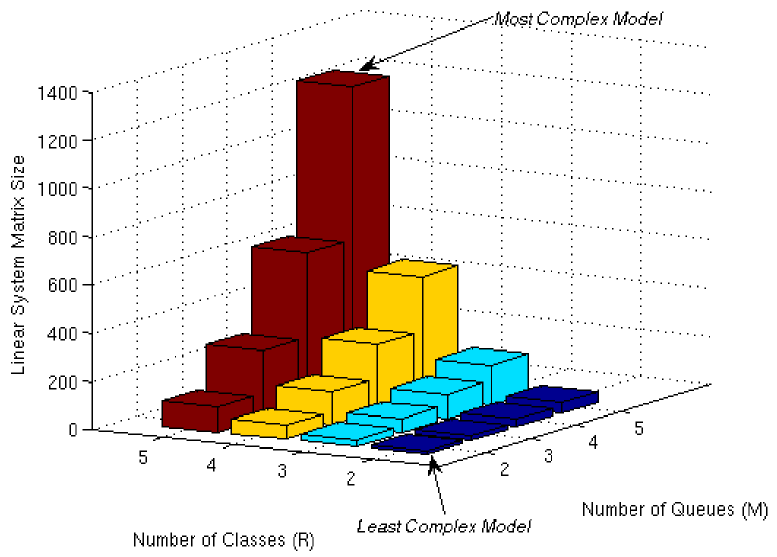


Fig. 25: *The order n of the matrices generated by MoM for each queueing network model of the test-case. The number of operations required to solve such a system is $O(n^3)$.*

At this point we will proceed with the performance comparison of the two versions of MoM based on the experimental results from our test-case set. The cases at which parallelisation is beneficial will be highlighted, the achieved speedup will be presented and justified and the limits of the implementation will be explained.

Attainable Speedup - When is the Parallel MoM preferable to the Serial one?

When testing the MoM implementations using the various test models, it was discovered that it does not exist a single solver configuration that produces the best results in all models. Which algorithm is better clearly depends on the matrix size that corresponds to the model used. For example, it was discovered that **for models small enough that their linear system size is less than approximately 120**, the parallel implementation is not beneficial at all, as it may result in a slowdown instead of a speedup. 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. 28 below:

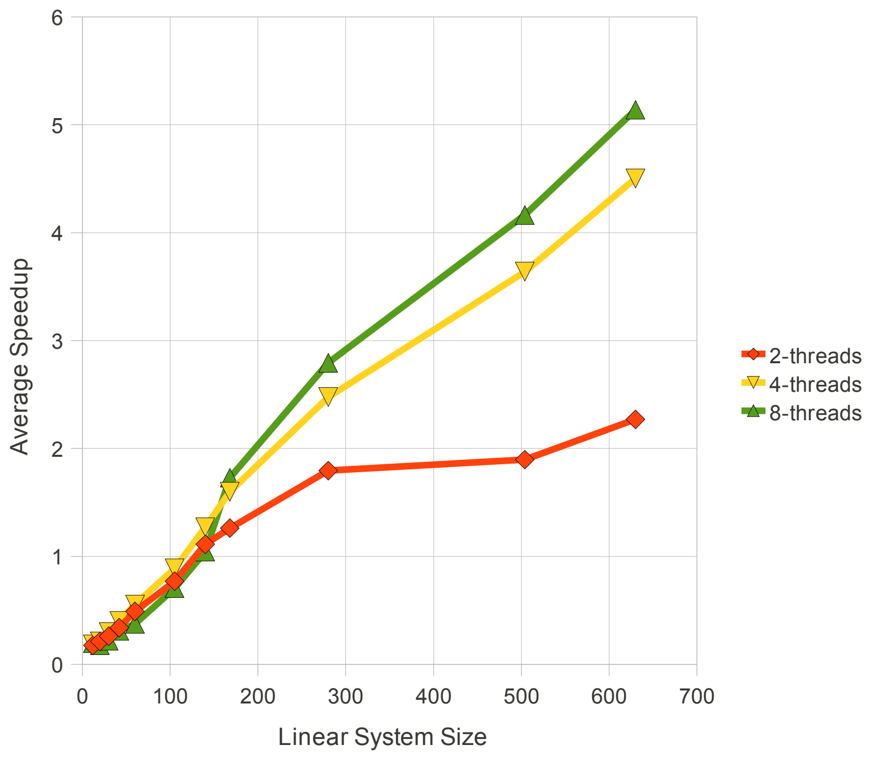


Fig. 26: *Average attainable speedup for each matrix size when using the parallel algorithm with 2, 4 and 8 threads.*

There are many trends and behaviours one should comment:

5. Overall, **the performance scales well**, both as an average for all threads and in the case of 8-threaded version. **In general, it seems preferable to use as many threads as the maximum number of independent cores of our machine** – 8 in our case. The 8-threaded version performs similarly to the best one for small matrix sizes and is the best one for large matrix sizes, i.e. with order of more than 140.
6. There exist some superlinearities in the case of the 2-threaded version at matrix size 280, as well some smaller ones in the case of the 2- and 4-threaded versions for matrix size 630. This behaviour will be explained in the following section; in short, it is not caused by superlinear performance of the parallel algorithm but by imperfect scaling of the serial one due to not optimal cache memory usage. Superlinear efficiencies of approximately the same magnitude have been reported in [21] as well.
7. In general, we can see from the diagram of Fig. 26 that the maximum parallelisation efficiency possible is achieved in the case of the 2- and 4-threaded versions for models within our test set. This is an important result and it can be speculated that the 8-threaded version should achieve such efficiency if it was applied on bigger models. Such models however were not included in our test set, as they would need more than several hours to run, especially in the case of the serial version. The current test set is more than able to allow us to compare the different approaches meaningfully.
8. **It is verified that the parallel algorithm leads to speed-up in the average case only for matrix sizes above approximately 120.** This phenomenon is expected and is attributed to several factors:
   - A first factor to consider is that the parallel solution of a linear system using modular

arithmetic is more costly, in terms of total operations performed in all parallel instances, when compared to the initial serial Gaussian Elimination solver; in short, **there exists a parallelisation overhead**. This overhead, when combined with the need to perform context switches in the parallel algorithm may pose a significant disadvantage for the parallel implementation, when running on small models. As examined in chapter 6, the parallel algorithm needs to perform several operations to solve the system. Firstly, it must evaluate the threshold value for the product of moduli, $M_{threshold}$ and select as many moduli as required. Secondly, it must calculate the residual linear systems for each invocation; even though several optimisations have been introduced, as for example the common residual matrix A_k for all moduli, the total time needed for such operations is not insignificant, especially on simpler, less computationally intensive on the Gaussian Elimination part, models. One should not forget that such operations should be repeated as many times as the total population. Lastly, the parallel algorithm must recombine the residual results at every iteration. This is also a not insignificant cost in smaller models.

- Another point to consider is the **moduli selection strategy** and, in particular, **the relation between the magnitude of the moduli and the real, unknown until the execution terminates, normalising constant of the queueing network**. As it has been highlighted in Section 6.3.b (p. 38), in some cases, depending on the arithmetic qualities of the model, the algorithm may be forced to select a particular $M_{threshold}$ value that leads us in the selection of more moduli than available processing cores. As it is preferred not to evaluate more residual systems than the number of available cores at a given time due to contention between threads, the linear systems are solved in sets.

The two above reasons, namely the overhead of parallelisation and the possible need to solve more linear systems than existing processing cores during every iteration, justify the fact that usage of the parallel algorithm may not produce a speedup when processing small-size linear systems.

To summarise, we can claim that the parallel MoM implementation is preferable to the serial one when used in queueing network models corresponding to MoM linear systems of size greater that 120. This means that **the multi-threaded implementation can assist in reducing the runtime of all practically interesting cases**; if a model is simple enough to be described by a matrix smaller than 120×120, then solving such a linear system can be quite fast even if one used the serial algorithm. The main conclusion is that parallelisation can reduce the runtime of the most complex cases, were the total benefit can be significant.

Attainable Efficiency

As it has been already presented before, the average speed-up achieved is satisfactory in the most important test-cases. It would be interesting to investigate the average parallelisation efficiency MoM attains in comparison with the linear system matrix size. This efficiency is presented in the following Fig. 27:

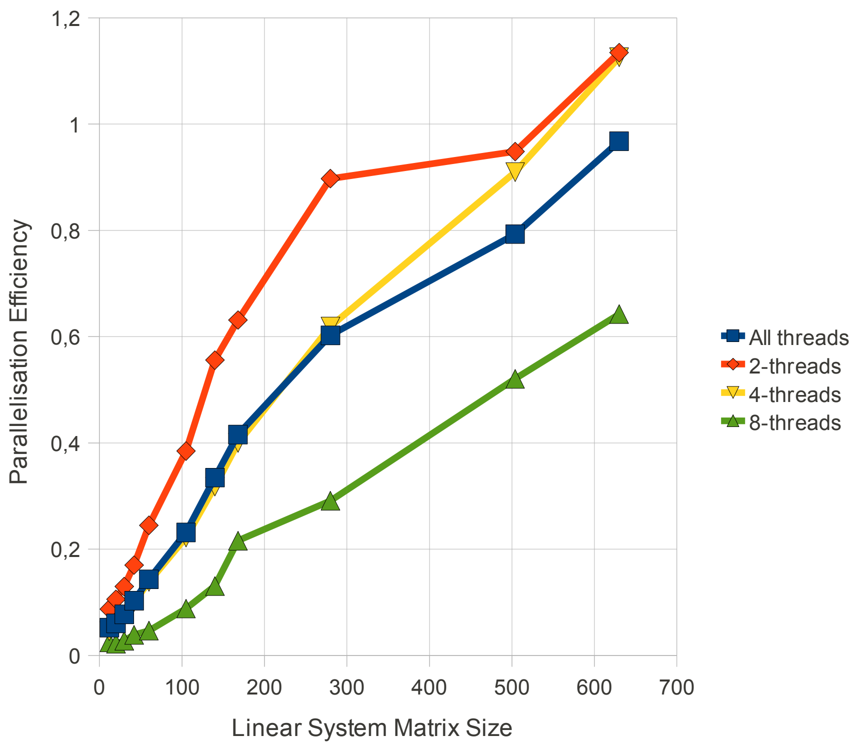


Fig. 27: *Average efficiency achieved for each matrix size when using the parallel algorithm with 2, 4 and 8 threads*

There are three main observations that can be made from Fig. 27:

1. The main observation is that **all parallel versions of MoM are able to scale good enough in practice**. For medium or large complexity models, depending on the particular case and configuration, they are able to attain efficiency that is close to the ideal value of 1.
2. When comparing the efficiency of the 2-, the 4- and the 8-threaded versions, one could argue that an implementation gets "saturated" slower when it runs in more threads. By the term "saturation" we define the maximum matrix size, above which the efficiency stops to increase a lot and, if projected asymptotically, would reach an upper bound. That upper bound in the case of the 2- and 4-threaded version is near the ideal speedup of 1. The 8-threaded version is expected to behave similarly if used in more complex models. Such models however fail to run within the time constraints for the slower serial version (and possibly the parallel ones with less number of threads) and as a result such an experimentation is out of the scope of this project.
3. A speedup more than the ideal one can be observed at the larger matrix size. This is attributed on two main factors:
   - This last experimental result is based on the runtime result of only one test model. The parallel implementations could evaluate both the low and medium load level case for that model, whereas the serial one could evaluate only the low load level within the testing constraints. Therefore, from a statistics perspective, this single result cannot be as representative as the others of the totality of the test cases and the different models.
   - Furthermore, this result and other superlinear phenomena are attributed to the way the serial MoM algorithm utilises the cache memory. One should understand the way the calculations are performed by the implementation in low level. Superlinear efficiencies of approximately the same magnitude have been as well reported in [21], which presents a dis-

tributed memory implementation.

First of all, the serial algorithm does not perform calculations in modulo. This means that **the numbers encountered in each linear system in the serial case can be quite larger than those encountered in the corresponding parallel case**, where each number of the residual system is smaller than the respective modulo.

The implications of this observation are more powerful than one would initially estimate: it is proven that even though the number of steps of **Gaussian Elimination** are polynomially bounded, these steps can deal with long operands. In the worst case, **the operands can be of exponential length [38]**. This may not be a problem when one is using floating point arithmetic , as in such cases all operands have a maximum representation length, which is the precision length of the data type. However, when one uses exact arithmetic, as it is necessary in the case of MoM, the operands can attain this exponential length in the worst case. This is the main reason behind the speedup achieved by the modular approach in general; it also means that the efficiency of the modular approach would be limited if we had to solve linear systems using floating point arithmetic in all cases, as in modern RISC processors the such floating point operations take a constant number of CPU ticks to be completed [39]. Even though IA-32 and AMD64 are CISC instruction sets, they are translated to RISC micro-operations within the processor [39].

To summarise, a linear increment in the maximum modulo length – or in the maximum contained number length for the serial case - **can, in the worst case, cause a superlinear increment in the maximum operand length during Gaussian Elimination.** As the time needed for infinite precision arithmetic is proportional to the length of the operands' representations, **one would on one hand need to perform polynomially more operations, but on the other hand, the worst-case amount of time for such operations can grow superlinearly.**

Furthermore, there are two main limitations of the computing system we used to implement this project. One is software related and one is hardware/architecture related.

As it has become clear, the arbitrary precision rational number implementation used in this project builds upon the arbitrary length integer arithmetic classes provided by Java. These classes, in order to adhere to the Object Oriented Programming Paradigm and contrary to the way operations regarding primitive Java data-types are performed, produce a new object for each operation. For example, if $a=2$, $b=3$ and $c=0$ and we try to make the assignment $c=a+b$, the result $c=5$ will be return as a new object, situated at a new memory address; it will not be in the previous place of c , as c is actually only a reference (pointer), hidden by Java's syntax. This pointer will now contain a new memory address.

The above behaviour stems from the Object Oriented Design of Java; one could transform the libraries in order to perform operations in-place, but this would not adhere to the programming paradigm and could cause many programming errors when combined with the lack of operator overloading. In order to make this more clear, executing a command like $c=a.add(b)$ (lack of operator overloading) would change both the values of c (desired) and a (not desired – misleading). In order to solve this problem, one could investigate porting the application to other languages or investigate the feasibility of implementing a new library for arbitrary precision arithmetic in Java that performs in-place operations. Both approaches were out of the scope of this project.

Lack of in-place computations increase the rate at which objects are created and rendered unused. This increases the strain on the Garbage Collector and on the cache memory. As far as the cache memory is concerned, this means that different instances of the same variable, i.e. an accumulator variable inside a loop, would have to exist simultaneously in memory. The processor caching system treats them as different variables, as they are situated in different memory addresses. Therefore, in certain cases – depending on the specific hardware specifications – the rate at which operations are performed is finally limited by the rate at which the processor can perform write-back operations. This is an important bottleneck that can affect all MoM implementations of this project and, in general, any numerically intensive Java program which does not use primitive data-types.

All the above factors are combined by the fact that the objects representing the operands can grow very large – even several kB long – leads us to the conclusion that the Java implementation makes sub-optimal usage of the processor's cache memory. This is not an algorithmic problem in the part of MoM or the parallel solver, but rather a limitation stemming from the high abstraction level in which a Java program operates. Larger objects means that fewer of them will be able to be cached; the fact that a new object is created for each operation's result means that even fewer actually distinct object will be able to remain cached.

The above limitations are software related. There is one more limitation, which stems from the fact that we are applying the Gaussian Elimination algorithm on a shared memory multiprocessing computer system. As the Gaussian Elimination needs to access $O(n^3)$ memory distinct elements during its operation (not storage complexity – this remains $O(n^2)$), its runtime contains a substantial part, more than the one of other typical algorithmic operations, during which the processor waits for a memory operation to complete after a cache miss [40]. In this implementation, several "residual" instances of the serial Gaussian Elimination are run in parallel; therefore, cache contention between threads is increased, as does memory bus congestion. This can hinder performance in some cases, even though the Intel "Nehalem" processor architecture used in the testing computer features a distinct memory controller per core [41]. This phenomenon can be observed in several cases where the parallelisation speedup achieved for 2 or 4 threads is lower than the one achieved if more threads are used. In Fig. 26 (p. 80), one can observe that this is an issue only for linear systems of order less than 168.

8.4. Related Results

In this section several related experimental results will be presented. These results either support theoretical properties proved in previous chapters of provide useful insight in several choices made regarding the algorithm's design and implementation.

8.4.a Word-sized Moduli

In this section we will investigate the effects of selecting moduli that fit within the word-size of the machine, i.e. moduli that have a length of at most 64 bits. This is an approach that is proposed several researchers ([26]), primarily because if one had a linear system with small enough numbers to fit into the word-size, then inexact floating point arithmetic could be used. However, **using inexact arithmetic has not been possible with MoM for neither the initial nor the residual linear systems**, even if when one takes into account that several of the computed results need exact representation and therefore must be stored using exact representation. Therefore, the only reason for such a selection in our case would have been the hypothesis that the smaller modulo size may take better advantage of the processor cache memory and, thus, be much faster than using a larger modulo. This is, however, not enough; in order to be preferable to select the smaller modulo size, the solution of each system should be fast enough to nullify the disadvantage that we would have to solve far more residual linear systems pre MoM iteration. It was proven from experimentation that this is not the case.

From early experimental analysis when the various design aspects of the parallel linear system solving algorithm were being determined, such an approach was deemed preferential only if one used the usual definition of the modulo operator that does not allow negative results. In any case, it was preferential only as long as this modulo definition was used; the implementation using the improved definition that allows negative results is far faster.

We will present the performance gap between the two approaches by using the runtime comparison of all MoM implementations for the medium load case of the model with $R=M=4$. This case is representative of the total picture.

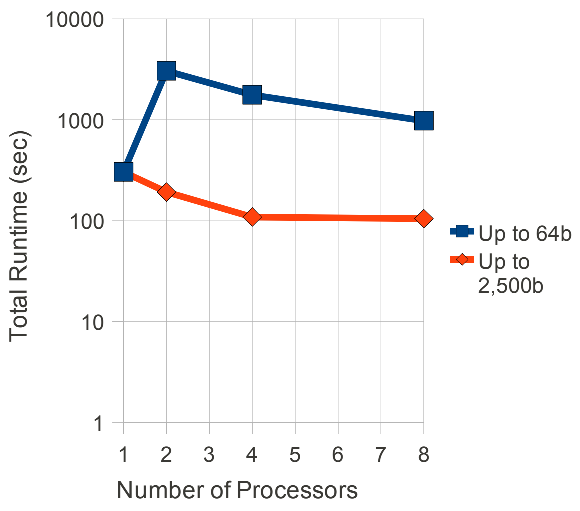


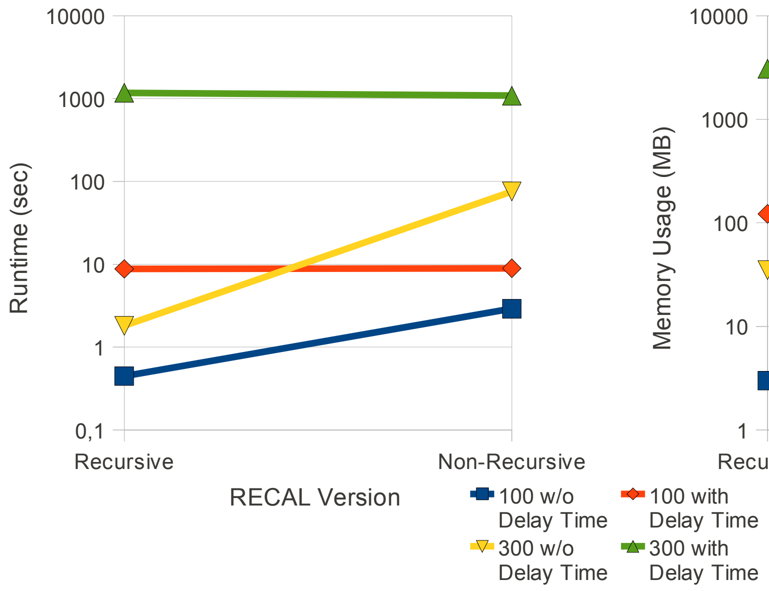
Fig. 28: *Total runtime vs. number of processors for two modulo selection strategies. The case of 1 processor corresponds to the serial MoM implementation. We can verify that selecting very small moduli may result in a slowdown.*

In the above diagram Fig. 28 it can be seen that **the time needed to solve the linear system when smaller moduli are selected is far larger than when larger ones are selected**. This is true primarily because the selection of so small moduli minimises the runtime per residual system and not the runtime for the initial linear system; when using the smaller modulo size, 128 residual systems must be solved. In general however, one has to take as well into account the cost of primality tests; this cost is included in the above runtimes, which present the total time to produce results. This cost may become significant for moduli sizes of more than a few thousand bits, as presented in Fig. 4 (p. 41), and is the main reason we limit the maximum modulo length at 2,500 bits.

8.4.b Comparison of RECAL Implementations

In this section we will compare the time and memory needed by the two existing implementations of the RECAL algorithm to compute a network's normalising constant. As a benchmark model, we will use the small queueing network with $R=M=2$ and the load scenarios corresponding to 100 and 300 jobs. The model will be tested in two versions, one with and one without delay times. The high load case is not used as it cannot be evaluated for all cases and therefore is not meaningful for comparison.

The runtime results and the memory usage is represented in the following figures 29 and 30:



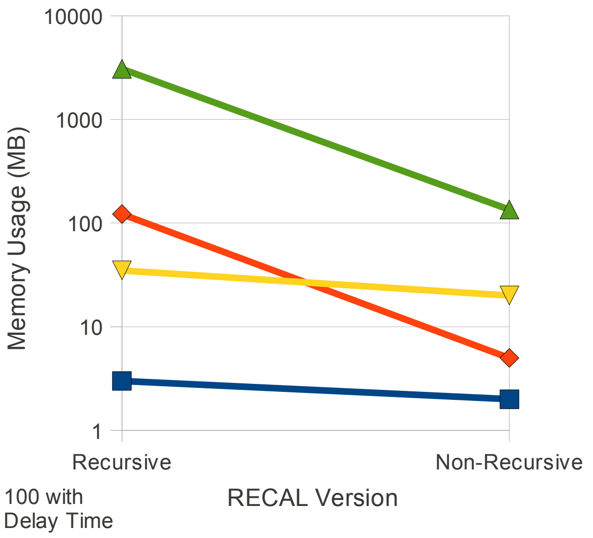


Fig. 29: *The figure presents the time the two RECAL implementations need to compute the normalising constant of a model with* $R=M=2$ *for varying populations and existence or not of delay times. We can see that if no delay times exist, the recursive version is faster as it is able to prune the computation of some constants. If, however, delay times exist, the non-recursive version is a little faster.*

Fig. 30: *This figure represents memory usage for the same cases as Fig. 29. We can verify that if no delay times exist, the non-recursive version is a little more memory efficient, whereas if delay-times exist the same version is much more efficient.*

As it can be verified from the above figures, the performance and memory requirements of RECAL are substantially influenced by the existence of delay times in the queueing network model. For networks without delay times, the recursive version is faster but at the cost of a little higher memory requirements. This happens due to the fact that the recursive version is able to prune big top-level branches of the recursion process if no delay times exist, achieving this performance benefit. For networks with delay times, the non-recursive version is a little faster, but has much less memory requirements.

This different behaviour of RECAL depending on the existence or not of delay times is not replicated in MoM. This is very important, as MoM proves to be an algorithm with much more consistent performance.

8.4.c Most Complex Queueing Network Feasible by MoM

As it has been already discussed, the test-set used to evaluate and compare the performance of the different algorithms involved a runtime constraint of 1 hour. 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. This 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 picard01.doc.ic.ac.uk host when imposing the maximum memory usage limit of 25GB **was the one representing the queueing network with**

$\boldsymbol{R=M=7}$, which is 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 $102{,}960$. The memory was inadequate to store the approximately 10^{10} arbitrarily long rational numbers.

8.4.d Experimental Growth Rate of the Normalising Constant

In the progress report we had verified experimentally that the number of digits of the normalising constant grows almost linearly with regard to the total population of the model. We can verify this growth rate in the experimental results of the test-cases.

Because we need to compare the three load profiles of all different model sets, we used as a base the number of digits of the low load case throughout the test-sets. The number of digits of the medium and high load profiles are represented with regard to this base. The averaged growth rates for all test-cases are presented in the following Fig. 31, where one can verify the linear growth.

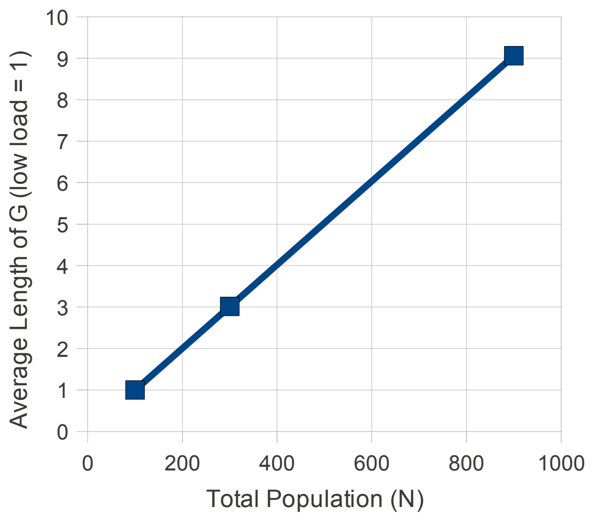


Fig. 31: *Average growth rate of the normalising constants for all test models. For each load set, the length of the normalising constant corresponding to the low load scenario is set as the base, i.e. equal to 1.*

8.4.e Experimental Growth Rate of $M_{threshold}$

Furthermore, it is interesting to investigate the bit size of $M_{threshold}$ as the queueing network model becomes more complex. As fas as MoM is concerned, the complexity of a queueing network is quantified by the size of the linear system representing it. It is important to remember that even though the value of $M_{threshold}$ can be independent of the number of moduli we select, the optimal strategy in practice is not to select arbitrarily large moduli but to select moduli with bit length lower than a certain limit value – 2,500 for the current implementation –. The relation between the bit size b of $M_{threshold}$:

$$b=\lceil \log_2(M_{threshold}) \rceil$$

and the linear system matrix size is presented in the following Fig. 32:

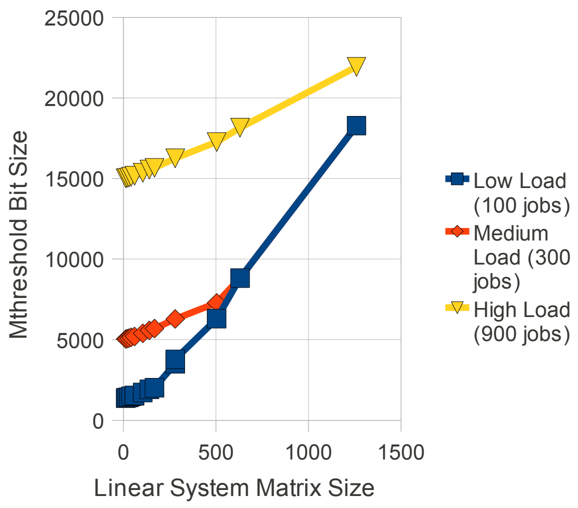


***Fig. 32:** Relation between the linear system matrix size and the bit size of $M_{threshold}$. We can verify the theoretical result of $O(n \log n)$ rate of growth.*

The above figure leads us to the important conclusion that the total required bit size of the product of moduli scales very well in practice as the MoM linear system describing this model increases in size. This rate of growth was proven theoretically for large models in section 6.4.b to be equal to $O(n \log n)$ and is confirmed by the experimental results. However, the growth curve is very close to being linear. It is an important characteristic of the parallel linear system solver that was described in Chapter 6, as it proves its scalability. Even if the optimal maximum bit length per moduli remains constant at 2,500 in the future, one would have to add more processors at a close to linear $O(n \log n)$ rate in order to maintain the same parallelisation efficiency as the queueing model complexity increases.

This is a result independent of the specific implementation of MoM used, as it is a characteristic of the linear system solver. If the same parallel solver is used together with an optimised version of MoM or even with another algorithm, it will be able to maintain the same parallelisation efficiency as the problem size increases, by increasing the number of processors almost linearly.

9. Conclusions and Future Work

In this chapter, the main conclusions reached as part of this project will be presented. Furthermore, throughout the project's progress, several other important areas of interest have been identified; it would be beneficial to research and examine these areas in the future, in order to further improve the performance of the algorithms used to evaluate queueing network models or the parallel linear system solver which uses modular arithmetic.

At the time of the submission of this report, all necessary interfaces necessary to integrate the source code as a new solver in the JMVA software have been implemented. The source code and the interfaces will be sent to the maintainer of the JMT project – part of it is JMVA – in order to perform the actual merging.

9.1. Current Advantages and Limitations

In order to summarise the entire project's outcome and to identify the areas which could be improved in the future, it is first necessary to present the advantages and the limitations of our current approach. Both the theoretical algorithmic part and the implementation will be examined. The analysis will be focus on three main parts: the Parallel Linear System Solver of Chapter 6, the Method of Moments algorithm and the software architecture and implementation.

9.1.a Parallel Linear System Solver

Overall, the parallel solver designed and implemented as part of this project exhibits very good results, both regarding its theoretical-algorithmic qualities and its performance.

- First and foremost, it exhibits a **very good speedup and efficiency when compared with the serial linear system solver**. This is the most important result, as it highlights the quality of both the algorithm's design and the implementation in practice.
- Secondly, it exhibits **good scalability** properties. Due to the limit we imposed on the maximum length of each modulo and the fact that the rate at which we need to add new moduli in order to maintain the same runtime grows at a tractable rate of $O(n \log n)$, the algorithm can scale efficiently both towards more complex linear systems, and thus more complex queueing networks, and towards execution over an increased number of processors. Usage of more processors decreases the parallelisation overhead, as it enables selection of smaller moduli.
- Lastly, it is a **robust solver** that can solve the sometimes singular linear systems outputted by this version of MoM. It is able to withstand the propagation of indeterminable elements from iteration to iteration; solution may only fail if an indeterminable value is explicitly needed by the MoM algorithm to proceed with the network evaluation.
- On the other hand, when evaluating complex models – much more complex than those encountered in the test set for the 8-threaded version - the $M_{threshold}$ value can become large enough to **require solution of multiple residual linear systems per processing core** due to the maximum modulo size limit. This is a situation rarely encountered in practice and even if encountered, **it may still be beneficial to solve more smaller residual systems in parallel than to use the serial algorithm** (see Fig. 24), as the cost of solving a more complex, in terms of number length, linear system grows faster.

This behaviour is a direct result of the fine-tuning of the parallel solver and the modulo selection strategy to behave optimally on models representing real-world cases.

9.1.b Method of Moments

The MoM algorithm and its extensions are the most efficient algorithms one can use to perform analytical evaluation of a queueing network model.

- It is **much faster than the other existing algorithms** for all models and exhibits **much lower space complexity**. This makes MoM very efficient in evaluating models of practical interest.
- Its performance does not exhibit substantial variations depending on whether delay times are included in the network model. This is true, as the order of the linear system defined by MoM depends only on the number of classes R and number of queues M.
- **The runtime of MoM is dominated by the time it takes to solve the linear system in each iteration.** This is an important advantage, as many techniques and optimisations for the most efficient solution of linear systems are available. Furthermore, this makes a substantial part of the runtime parallelisable, as was exhibited in this project.
- On the other hand, the implementation of MoM and its extensions can be more complex than those of RECAL or Convolution. However, the achievable performance gain compensates for the increase in the implementation difficulty.
- Furthermore, the cost of solving the linear system can grow fast as the number of classes R and number of queues M is increased; this increase is however far smaller than the one exhibited by the other established algorithms. Other, more optimised versions of MoM are more efficient [3].

9.1.c Software Architecture and Implementation

Much attention was paid for the most efficient implementation of the algorithms, given the programming system's and time constraints. The experimental result highlight this efficient performance of the code. However, the there are many good qualities of the code which may remain unseen by the end-user.

- **The entire implementation is built using a modular, object-oriented approach, that features good code quality without tangled classes or packages and circular dependencies**, which can hinder maintainability and code modification. This approach makes the system easy to integrate in other programs, as for example JMVA.
- **Existing components can be re-used** by new programs, whereas new functionality and optimisations can be easily introduced with minimal changes, taking advantage of the already built infrastructure.
- It is a **portable implementation**, as it is written in Java.
- On the other hand, selection of Java to solve this numerically intensive problem has introduced several constraints, with most important the **inability to perform in place exact arithmetic computations**. However, portability and integration with the JMVA tool was deemed more important than the achievement of the optimal performance. Furthermore, the parallelisation efficiency of the algorithm has not been reduced by this implementation choice, as both the serial and the parallel solvers use the same library we implemented for exact computations.

9.2. Future Work

In order to achieve improved performance in the future, the most beneficial step can be the integration of the existing parallel solver with an **optimised version of MoM**, such as the one presented in [3], or the **Class-Oriented MoM** (CoMoM), presented in [19]. The key point to understand here is that the main computational cost of evaluating a queueing network using MoM stems from the linear system solution; therefore, any decrease in the linear system size is expected to produce a cubic ($O(n^3)$) decrease in the algorithm's runtime. Such a benefit cannot be ignored, as it is the most effective way to increase the envelope of queuing network models that can be evaluated in practice.

The implementation of a more efficient MoM algorithm can be performed simultaneously with the **improved handling** of the cases where we need to evaluate a queueing network defined by a **singular matrix**. Several related techniques are presented in [3]. Furthermore,

After the linear system size has been reduced, one could investigate representation of the linear system using **sparse matrices**. This is expected to substantially reduce the memory requirement of the implementation, making possible the evaluation of even larger models. Of course, in such a case a **matrix re-ordering** using the **Reverse Cuthill – McKee algorithm [42]** or the **Approximate Minimum Degree algorithm [43]** may be necessary to reduce the number of performed operations and, most importantly, **to reduce the number of "fill-ins"**, i.e. the number of matrix elements that were initially zero but became non-zero due to row operations.

To further reduce the parallelisation overhead, the possibility of using a **precomputed prime moduli list** could be investigated. Such approach may be preferable to be used in conjunction with an increase in the maximum modulo size, enabling usage of one residual system per processor for even more complex models. In any case, **faster primality tests** should be investigated, such as an adaptation of the **Agrawal – Kayal – Saxena (AKS) primality test** which is presented in [35] and features $O(\ln^{4+\varepsilon}(p))$ time complexity.

Furthermore, ability of producing a **stricter bound for the** $M_{threshold}$ value used by the parallel linear system solver should be investigated. When used in conjunction with MoM, discovering such a stricter bound may involve producing an easily computable **stricter bound for the maximum normalising constant** of the queueing network model than the one of eq. (5.1.1) (p. 20) or better bounding the value of the maximum possible determinant of the linear system.

Lastly, one could as well investigate using or producing a library that can perform **exact arithmetic operations in-place**, at least as long as the result does not exceed a maximum pre-allocated length. The fact that the numbers contained in a residual system are bounded in length can assist in this process. However, achieving such fine-grained control over the computer hardware may not be possible in Java; such an approach **may require porting the implementation in another programming language**, like C or C++. However, using such a language can be beneficial, as efficient linear algebra libraries and implementations of most of the algorithms proposed above are readily available for C++. In this case, integration with JMVA could be preserved by calling one application from within the other. This would also allow the **parallelisation in a distributed memory system**, for example by using MPI and would reduce the memory bus congestion that can be an issue in the parallel solver.

In any case, many areas open for future improvement have been highlighted by our current approach. It is important to note that one of most important characteristics of our current implementation, namely **the independence of the MoM algorithm implementation from the parallel linear system solver, can greatly assist in improving one of these two parts with no regard of the other**. Lastly, it is inherently possible to test various network evaluation algorithms using the same solver and, conversely, to use the same network evaluation algorithm to test various solvers.

10. Appendix

10.1. Software Tools for Evaluating Queueing Networks

Queueing networks are very popular and powerful tools that can assist in the design, evaluation and improvement of existing system offering a finite resource to a number of users, such as communication networks, computer systems, etc. With users ranging from academics and researchers to industrial and governmental organisations, it is not surprising that a wide variety of software tools are available for modelling a system and evaluating the resulting network. Each system focuses on satisfying a different set of requirements, but all make heavy use of optimised algorithms and techniques to increase efficiency. A brief presentation of the most significant such tools will follow. More details will be given for the JMT and JMVA systems, as this project aims to augment their functionality.

10.1.a Java Modelling Tools (JMT)

The Java Modelling Tools is s a suite of applications developed by Politecnico di Milano and released under GPL license. It offers a complete framework for performance evaluation, system tuning, capacity planning and workload characterisation studies. One of the main goals of its development was portability between different operating systems, so Java was the programming language of choice. The complete suite encompasses six different applications:

1. JSIMgraph and JSIMwiz: Queueing network models simulator with graphical user interface and a wizard-based user interface, respectively.

   These tools generate the XML specifications of the simulation models used by the discrete-state simulation engine. Also, it visualises complex networks, assists in debugging of problems in the model and gives feedback to the user about the current state of the simulation and the estimated performance indices.

2. JMVA: Mean Value Analysis of queueing network models

   Using the JMVA application, a user can easily define a network model by specifying parameters such as mean service demands, arrival rates and population sizes or import a model created graphically using JSIMgraph or JSIMwiz. The model is then evaluated by the analytical engine of JMT, which uses mainly – but not only - the MVA algorithm. Goal of the current project is the integration of the MoM algorithm into this application. The JMVA engine is usually faster than the JSIMengine of JSIMgraph and JSIMwiz for models which contain less that 3-4 classes, but at the cost of increased memory usage. This is a common difference between simulation and analytical for performance evaluation. Integration of the MoM algorithm into JMVA can assist in reducing these run-time requirements.

3. JABA: Asymptotic Analysis of queueing network models

   JABA assists in detecting performance bottlenecks in multi-class closed queueing networks using a geometrical approach. This reduces the computational costs of long simulative analysis over different mixes of requests.

4. JWAT: Workload Analysis from log and usage data

   Main aim of this tool is to assist in the derivation of the parameters needed to define the network models by analysing log files and using advanced preprocessing, clustering and characterisation techniques.

5. JMCH: Markov chain simulator (teaching tool)

JMT supports a rich feature set for performance evaluation, such as solution of capacity planning models using simulation or analytical approaches, feature extraction and preprocessing of log files, clustering algorithms for the selection of the most important workload cases under which a system should be evaluated, determination of the optimal load conditions and automatic identification of bottlenecks. Furthermore, it can produce a wide variety of performance indices, such as mean queue lengths, throughputs and response times. It is specifically tailored towards the performance evaluation of IT systems, such as multi-tier architectures, storage arrays and communication networks. A distinctive characteristic of JMT is that it aims at the non-technical or new user, by being accompanied by rich documentation and a set of wizard interfaces that guide the user in the accomplishment of tasks. This hiding of the complexity reduces the learning curve of new users. As the JMT comprises of a diverse set of applications and algorithms, the integration of the applications into the tool and the communication of the GUI with the underlying algorithms is based on XML, simplifying the usage of external software and algorithms.

10.1.b Other Tools

Several other software tools used for performance evaluation are presented below in alphabetical order. Some of these tools use other approaches or different algorithms than the exact analytical modelling used by JMVA.

Möbius

The **Möbius framework** [44] is a multi-paradigm environment that allows the definition of composite performance and dependability models based on formalisms such as stochastic activity networks (SANs), fault trees and the PEPA stochastic process algebra. It is widely used both in academia and industry. One of the main features is the ability to distribute tasks (series of experiments) over a network to support more efficient evaluation of real-world systems.

OPEDo

OPEDo, a tool for the "Optimisation of Performance and Dependability Models", [45] is a software tool used for numerical optimisation of performance metrics on discrete event systems. It implements local, global and hybrid search methods to iteratively seek the optimum solution. The underlying model is regarded as a "black-box"; OPEDo interfaces to external applications for model definition and evaluation. The available black-box model definition and evaluation engines include the analytical solver of JMT, as well as simulation and other engines.

PIPE2

PIPE2, the "Platform-Independent Petri Net Editor 2" [46] enables creation and analysis of Generalised Stochastic Petri Net (GSPN) models. It provides structural analysis and performance analysis modules; new pluggable analysis modules can be added during run-time and extend the system's functionality Among the performance analysis tools included in PIPE2 is an efficient Markov chain steady-state analyser, a semi-Markov response time analyse r implemented using MPI and other tools that focus on passage time analysis.

QPME

QPME (Queueing Petri net Modelling Environment) [47] is a tool that supports the modelling and analysis of systems using queueing Petri nets. Petri nets combine the modelling power and expressiveness of queueing networks and stochastic Petri nets and are appropriate for modelling distributed systems and analysing their performance and scalability. QPME currently supports a simulation engine for evaluating such networks and development of an analytical solver is an ongoing project.

SHARPE

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [48] is a hierarchical modelling tool that focuses on analysis of reliability, availability, performance and performability using stochastic models. It includes analytical algorithms for performance evaluation of closed single- and multi-chain product form queueing networks.

TANGRAM-II

TANGRAM-II [49] is an "Integrated Modelling Environment for Computer Systems and Networks" evolving for more than 15 years. It features an analytical solution engine that supports both Markovian and non-Markovian models and a simulation engine. Calculating performance indices and having a complete documentation are among its main characteristics. The analytical solution engine can evaluate models both in steady and in transient states using several iterative methods.

10.2. User's Guide

In this section we will present the program's usage and requirements. The program is free software and is distributed under the terms of the GNU General Public License version 3, or any later version.

10.2.a Requirements and Compatibility

The source code conforms to the requirements of Java 6. However, all of the used features are available from at least the Java 5 version. It can be compiled using any Java 5 supporting compiler and can run in any Java Virtual Machine that supports at least this version of Java.

There are no minimum hardware requirements in order to run this program; any modern computer will be able to do so. However, for efficient usage of the MoM algorithm on a range of models at least 1GB of RAM memory is recommended. For the case of the Convolution and RECAL algorithms, at least 4GB should be used to be able to solve a variety of models; however, 12GB is the recommended size. Convolution and RECAL algorithms are only included for testing purposes and normally usage of MoM should be preferred.

Furthermore, the program can run in both single- and multi-processor systems. However, using the parallel version on a single-processor system will not result in any performance benefit. It is possible to run more parallel threads than the number of available processors, but this ability only exists for testing purposes.

Lastly, it is important to note that on systems supporting Intel's Hyper-Threading Technology it is recommended to instantiate as many threads as the number of real CPU cores and not logical ones. This has to be defined manually using the respective argument, as the JVM does not support any platform-independent way of distinguishing between physical and logical processors.

10.2.b Format of the Input File

The implementation takes as input the description of a queueing network model in a particular form, which is the following:

6. The 1st line contains the number of classes R.
7. The 2nd line contains the populations N_r of the classes, separated by spaces.
8. The 3rd line contains the total delay time Z_r for each class.
9. The 4th line contains the number of different queue types M.
10. Finally, a $M\times(1+R)$ matrix is contained. The first column of this matrix contains the multiplicity m_k of a particular queue k, whereas the rest of it contains the service demands D_{kr} any pair of class r and queue k.

An example of such an input file is the following, which defines a network with 3 classes and 4 queues:

R	3
$\vec{N}^{tr}$	10 10 10
$\vec{Z}^{tr}$	0 0 0
M	4
$\vec{m} \mid \boldsymbol{D}$	1 9 23 44
	1 27 33 28
	1 32 15 38
	1 2 48 45

10.2.c Command-Line Arguments

In order to evaluate a queueing network model described in an input file, one has to call the program using several command-line arguments. Usually, the program is invoked as:

```
java -jar MoM.jar <Algorithm> <Output Performance Indices> <Input File> [<Number of Threads>]
```

From the four supported arguments, only the first three are necessary; the last one, defining the number of threads is optional.

1. The first argument defines which algorithm should be used. The user can input 0 for Convolution, 1 for the recursive implementation of RECAL, 2 for non-recursive RECAL, 3 for MoM using the parallel solver, 4 for MoM using the serial solver and 5 to allow the program to select the best solver automatically depending on the linear system order. For linear systems of order less than 120 the serial solver if preferred, as discovered by the experimental results.
2. The second argument can be 0 or 1, depending on whether the user needs only the value of the normalising constant G of the model or needs the calculation of performance indices (mean throughputs and mean queue lengths) as well.
3. The third argument defines the path to the desired input file describing the queueing network.
4. The last argument specifies the desired number of threads. Unless the parallel solver is used, it is ignored. If the user does not specify the number of threads, as many threads as the number of available processors will be used by the program. This argument exists for testing purposes and for limiting the maximum number of threads on systems supporting Intel's Hyper-Threading Technology; on such systems, usage of as many threads as the number of real and not logical cores is recommended.

11. References

[1] S.S. Lavenberg, "A perspective on queueing models of computer performance.," *Performance Evaluation*, vol. 10, 1989, pp. 53–76.

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

[3] G. Casale, "An efficient algorithm for the exact analysis of multiclass queueing networks with large population sizes," *Proceedings of the joint international conference on Measurement and modeling of computer systems*, 2006, p. 180.

[4] G. Casale, "Exact Analysis of Performance Models by the Method of Moments," *under submission*, 2009.

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

[6] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, 2009, pp. 10–15.

[7] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1984.

[8] J. Zahorjan, "An exact solution method for the general class of closed separable queueing networks," *SIGSIM Simul. Dig.*, vol. 11, 1979, pp. 107-112.

[9] R.D. Nelson, "The mathematics of product form queuing networks," *ACM Computing Surveys (CSUR)*, vol. 25, 1993, p. 369.

[10] J.R. Jackson, "Jobshop-like Queueing Systems," *Management Science*, vol. 10, Oct. 1963, pp. 131-142.

[11] A. Willig, "A short introduction to queueing theory," *Technical University Berlin, Telecommunication Networks Group*, 1999.

[12] W.J. Gordon and G.F. Newell, "Closed queuing systems with exponential servers," *Operations Research*, vol. 15, 1967, pp. 254–265.

[13] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, vol. 22, 1975, p. 260.

[14] M. Reiser and H. Kobayashi, "Queuing networks with multiple closed chains: theory and computational algorithms," *IBM Journal of Research and Development*, vol. 19, 1975, pp. 283–294.

[15] J.P. Buzen, "Computational algorithms for closed queueing networks with exponential servers," *Communications of the ACM*, vol. 16, 1973, pp. 527–531.

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

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

[18] K.M. Chandy and C.H. Sauer, "Computational algorithms for product form queueing networks," 1980.

[19] G. Casale, "CoMoM: Efficient class-oriented evaluation of multiclass performance models," *IEEE Transactions on Software Engineering*, vol. 35, 2009, pp. 162–177.

[20] G. Casale, "The Multi-Branched Method of Moments for Queueing Networks," *Arxiv preprint arXiv:0902.3065*, 2009.

[21] H. González and E. Martinez, "A Parallel Code for Solving Linear System Equations with Multimodular Algebra," *Revista Investigacion Operacional*, vol. 23, 2002, pp. 175-184.

[22] R.T. Boute, "The Euclidean definition of the functions div and mod," *ACM Trans. Program. Lang. Syst.*, vol. 14, 1992, pp. 127-144.

[23] J.A. Howell, "Algorithm 406: exact solution of linear equations using residue arithmetic [F4]," *Commun. ACM*, vol. 14, 1971, pp. 180-184.

[24] H. Takahasi and Y. Ishibashi, "A new method for exact calculations by a digital computer," *Information Processing in Japan*, vol. 1, 1961, pp. 28-42.

[25] I. Borosh and A.S. Fraenkel, “Exact solutions of linear equations with rational coefficients by congruence techniques,” *Mathematics of Computation*, vol. 20, 1966, pp. 107–112.

[26] Ç.K. Koç, “A parallel algorithm for exact solution of linear equations via congruence technique,” *Computers & Mathematics with Applications*, vol. 23, 1992, pp. 13–24.

[27] M. Morháč, “Error-Free Algorithms to Solve Special and General Discrete Systems of Linear Equations.”

[28] R.A. Horn and C.R. Johnson, *Matrix analysis*, Cambridge Univ Pr, 1990.

[29] M. Newman, “Solving equations exactly,” *Journal of Research of the National Bureau of Standards, 71B*, vol. 4, pp. 171–179.

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

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

[32] P. Dusart, “Autour de la fonction qui compte le nombre de nombres premiers,” *Université de Limoges*.

[33] M. Agrawal, N. Kayal, and N. Saxena, “PRIMES is in P,” *Annals of Mathematics*, vol. 160, 2004, pp. 781–793.

[34] H.W. Lenstra and C. Pomerance, “Primality testing with Gaussian periods,” *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science*, 2002, pp. 1–1.

[35] 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.

[36] S.D. Kounev and A. Buchmann, “Performance modeling and evaluation of large-scale J2EE applications,” *CMG-CONFERENCE-*, 2003, pp. 273–284.

[37] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1998.

[38] X.G. Fang and G. Havas, “On the worst-case complexity of integer gaussian elimination,” *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, 1997, pp. 28–31.

[39] J.L. Hennessy, D.A. Patterson, and D. Goldberg, *Computer architecture: a quantitative approach*, Morgan Kaufmann, 2003.

[40] R.H. Saavedra and A.J. Smith, “Measuring cache and TLB performance and their effect on benchmark runtimes,” *IEEE Transactions on Computers*, vol. 44, 1995, pp. 1223–1235.

[41] V. Datasheet, “Intel® Xeon® Processor 5500 Series,” 2009.

[42] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” *Proceedings of the 1969 24th national conference*, 1969, pp. 157–172.

[43] P.R. Amestoy, T.A. Davis, and I.S. Duff, “An approximate minimum degree ordering algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 17, 1996, pp. 886–905.

[44] S. Gaonkar, K. Keefe, R. Lamprecht, E. Rozier, P. Kemper, and W.H. Sanders, “Performance and dependability modeling with Möbius,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, 2009, pp. 16–21.

[45] M. Arns, P. Buchholz, and D. Müller, “OPEDo: a tool for the optimization of performance and dependability models,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, 2009, pp. 22–27.

[46] N.J. Dingle, W.J. Knottenbelt, and T. Suto, “PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets,” *SIGMETRICS Perform. Eval. Rev.*, vol. 36, 2009, pp. 34-39.

[47] S. Kounev and C. Dutz, “QPME: a performance modeling tool based on queueing Petri Nets,” 2009.

[48] K.S. Trivedi and R. Sahner, “SHARPE at the age of twenty two,” *SIGMETRICS Perform. Eval. Rev.*, vol. 36, 2009, pp. 52-57.

[49] E.S. e Silva, D.R. Figueiredo, and R.M. Leao, “The TANGRAM-II Integrated Modeling Environment for Computer Systems and Networks.”