Extracting State-Based Performance Metrics using Asynchronous Iterative Techniques

Douglas V. de Jager and Jeremy T. Bradley

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, United Kingdom

Abstract

Solution of large sparse linear fixed-point problems lies at the heart of many important performance analysis calculations. These calculations include steady-state, transient and passage-time computations in discrete-time Markov chains, continuous-time Markov chains and semi-Markov chains. In recent years, much work has been done to extend the application of asynchronous iterative solution methods to different contexts. This work has been motivated by the potential for faster solution, more efficient use of the communication channel and access to memory, and simplification of task management and programming. In this paper, we show how the key performance metrics mentioned above can be transformed into problems which can be solved using asynchronous iterative methods with guaranteed convergence—using the full breadth of Chazan and Miranker's classes of asynchronous iterations. We introduce the application of asynchronous iterative solution methods within this context by applying several algorithm variants to steady-state analysis of a GSPN model of a flexible manufacturing system. We show that for varying numbers of processors and different problem sizes one of these algorithm variants offers consistently better wall time until convergence, a consistently better communication profile, and often requires fewer update iterations than a standard parallel Jacobi algorithm, seemingly benefiting from a form of Gauss–Seidel effect.

Key words: Asynchronous iterative solution; performance analysis; continuous-time Markov chain; semi-Markov chain; steady-state analysis; transient analysis; passage-time analysis

1 Introduction

Extracting performance measures from performance models is a computationally intensive process. Typically a performance model will consist of a large Markov

Email addresses: `dvd03@doc.ic.ac.uk` (Douglas V. de Jager), `jb@doc.ic.ac.uk` (Jeremy T. Bradley).

chain in excess of 10^9 states, and producing even a simple steady-state vector for a utilisation metric will use large amounts of computing resources.

Service-level agreements and quality of service metrics, for example that, 93% of all file chunk downloads in a peer-to-peer network should be completed within 20 seconds, require first passage-time measures on Markov chains. Passage-time calculations are an order of magnitude more difficult again. If a more expressive performance model is deployed in the form of a semi-Markov chain, then this too adds another order of magnitude to the difficulty of the problem.

One potential approach is to devise solution algorithms which allow calculation to be divided efficiently over multiple processors, perhaps on an homogeneous cluster or even over a heterogeneous computation environment. Traditionally, such algorithms for performance analysis have been parallel algorithms, in which iterative updates to vector entries happen synchronously. These algorithms, given their inherent need to synchronise at each iteration step, often perform poorly as they scale, particularly over heterogeneous environments where differing interconnect networks and processor speeds create bottlenecks. In this paper, we show that distributed asynchronous algorithms are possible, where each computational node can perform calculation iterations without having to synchronise frequently with other nodes and thus create communication bottlenecks. This approach was originally proposed by Chazan and Miranker [1] for linear system solution, and has since been shown successful in many contexts—for example, non-linear problems [2], partial differential equations [3], convex programming [4] and optimisation [5].

In this paper, we show that steady-state, transient and passage-time performance measures for a variety of performance models, can be reformulated as linear fixed-point problems of the form $M\overline{x}+\overline{b} = \overline{x}$. The reformulated linear systems are further shown to be amenable to asynchronous iterative solution, which allows them to be distributed across massively distributed heterogeneous clusters. In order to apply asynchronous iterative theory without restriction, we show that the performance measures generating the fixed-point equations satisfy the condition that $|M|$ has a spectral radius of strictly less than one. We also introduce here three particular variants of an asynchronous iterative solution algorithm.

The paper is organised as follows. In Section 2 we introduce Markov chain and semi-Markov chain definitions and show how steady-state and transient and passage-time analysis can be reduced to two particular forms of fixed-point problem. In Section 3 we review the traditional synchronous iterative solution methods for general linear fixed-point problems. In Section 4 we provide a mathematical overview of asynchronous iterative theory and we review the potential advantages of asynchronous iterative solution relative to synchronous iterative solution. In Section 5 we review a partial application of asynchronous iterative theory to the problem of solving for dominant eigenvectors, of which steady-state vectors are a special case. We then present an unrestricted application of asynchronous iterative theory to this problem. In Section 6, we show that asynchronous iterative solution theory can be applied to transient analysis and passage time problems. Finally in Section 7, we present three particular solution algorithms. We consider a variety of FMS (Flexible Manufacturing System) models to compare wall time until convergence, together with calculation and communication overheads, for the asynchronous iterative algorithms relative to a standard parallel Jacobi algorithm.

2 Performance Problems as Fixed-Point Problems

In this section, we show that steady-state, transient and first passage-time analysis of continuous-time Markov chains (CTMCs) and semi-Markov processes (SMPs) can be transformed into two classes of fixed-point problem, whereby we will show subsequently that we can employ totally asynchronous solution methods which are guaranteed to converge appropriately.

In the first instance, we use the standard results that steady-state analysis of continuous-time Markov chains and semi-Markov chains can be expressed as the familiar DTMC eigenvector problem $\overline{x} = M\overline{x}$ where $M \in \mathbb{R}^{n\times n}$ is a column-stochastic square matrix [6,7].

In the second case, we show that passage-time and transient analysis of continuous-time Markov chains and semi-Markov chains can be expressed as the more general fixed-point problem $\overline{x} = M\overline{x} + \overline{b}$ where $M \in \mathbb{C}^{n\times n}$ is a complex square matrix and $\overline{b} \in \mathbb{C}^n$ is a complex column vector.

This is a new approach to transient and passage-time analysis of CTMCs which would usually be performed through uniformisation [8]. However, our observation is that asynchronous techniques cannot be applied directly to uniformisation. To the best of our knowledge, the application of asynchronous techniques to the analysis of semi-Markov processes is also wholly new.

We note that experimental work has previously been presented by Fischer and Kemper [9] in which an asynchronous block two-stage method [10,11] is applied to solve for CTMC steady-state vectors, but without a guarantee of convergence. Such a guarantee is important, as discussed in [12,13], since asynchronous methods may or may not converge when applied, without reformulation, to steady-state analysis problems. We will seek to provide precisely such a convergence guarantee in this paper.

The next sections show how the equations for the steady-state, transient and passage-time analysis of CTMCs and SMPs can be transformed into the linear fixed-point systems as stated above. The remainder of the theoretical contribution of the paper is devoted to proving that these performance-based constructions are amenable to asynchronous iterative solution techniques.

2.1 Semi-Markov Processes

Consider a Markov renewal process $\{(X_n, T_n) : n \geq 0\}$ where T_n is the time of the nth transition ($T_0 = 0$) and $X_n \in \mathcal{S}$ is the state just after the nth transition. Let the kernel of this process be:

$$R(n, i, j, t) = \mathbb{P}(X_{n+1} = j, T_{n+1} - T_n \leq t \mid X_n = i) \tag{1}$$

for $i, j \in \mathcal{S}$. The continuous time semi-Markov process (SMP), $\{Z(t), t \geq 0\}$, defined by the kernel R, is related to the Markov renewal process by:

$$Z(t) = X_{N_s(t)} \tag{2}$$

where $N_s(t) = \max\{n : T_n \leq t\}$, i.e. the number of state transitions that have taken place by time t. Thus $Z(t)$ represents the state of the system at time t.

In an SMP the kernel, $R(n, i, j, t)$, is independent of the transition number, n:

$$\begin{aligned} R(i, j, t) &= \mathbb{P}(X_{n+1} = j, T_{n+1} - T_n \leq t \mid X_n = i) \quad n \geq 0 \\ &= p_{ij} H_{ij}(t) \end{aligned} \tag{3}$$

where $p_{ij} = \mathbb{P}(X_{n+1} = j \mid X_n = i)$ is the one-step state transition probability between states i and j and $H_{ij}(t) = \mathbb{P}(T_{n+1} - T_n \leq t \mid X_{n+1} = j, X_n = i)$, is the sojourn time distribution in state i when the next state is j.

2.2 Continuous-Time Markov chains

We can manipulate any CTMC into a stochastically equivalent semi-Markov process since SMPs are a superset of CTMCs. Consider a Markov process $\{X'_n : n \geq 0\}$ where $X'_n \in \mathcal{S}$ is the state just after the nth transition. It is usual to represent a CTMC with a generator matrix, A, and elements a_{ij} representing the transition rate from state i to state j.

An equivalent SMP can be produced from this generator matrix A by setting the kernel of the SMP, $R(i, j, t)$, appropriately:

$$R(i, j, t) = \begin{cases} -\frac{a_{ij}}{a_{ii}}(1 - e^{a_{ii}t}) & : |a_{ii}| > 0 \\ 0 & : a_{ii} = 0 \end{cases} \tag{4}$$

where $a_{ii} = -\sum_{j \neq i} a_{ij}$ is the value of the ith leading diagonal element in the matrix, A. By translating a CTMC into a semi-Markov process in this fashion, we can also get results for steady-state, transient and passage-time analysis for CTMCs that are achievable using SMPs.

2.3 Steady-State Solution of Semi-Markov Processes

For an SMP with state space $\mathcal{S}$ and kernel $R(i, j, t) = p_{ij}H_{ij}(t)$, the p_{ij} values constitute the elements of the transition matrix P of the embedded DTMC. The steady-state solution of this DTMC is given by the vector $\overline{\pi}$, solved by $\overline{\pi}P = \overline{\pi}$. The SMP has steady state probability distribution vector, $\overline{q}$ (from, for example, [7]) where:

$$q_i = \frac{\pi_i \bar{h}_i}{\sum_{j \in S} \pi_j \bar{h}_j}$$

where $\bar{h}_i$ is the mean sojourn time of being in state i, and:

$$h_i(t) = \sum_{j \in S} p_{ij} h_{ij}(t) = \sum_{j \in S} R'(i, j, t)$$

and $R'(i, j, t)$ is the derivative of $R(i, j, t)$.

2.4 Steady-State Solution of CTMCs

A CTMC with generator matrix A has steady-state vector $\overline{q}$, as given by a unique solution to the equation $\overline{q}A = \overline{0}$. If we treat a CTMC with generator matrix A as a special case of an SMP as in Section 2.2, then we can easily relate the steady-state solution of a CTMC with the steady state solution of the SMP with $\bar{h}_i = -\frac{1}{a_{ii}}$, giving the solution vector $\overline{q}$ by:

$$q_i = \frac{\pi_i / a_{ii}}{\sum_{j \in S} \pi_j / a_{jj}}$$

$\overline{\pi}$ is calculated as before from the embedded DTMC, P, of the SMP. Here the elements of P, $p_{ij} = -a_{ij}/a_{ii}$ for $i \neq j$ and $p_{ii} = 0$.

Thus for steady-state solution of a CTMC also, we have shown that the principle calculation involves the solution of the fixed-point problem $\overline{\pi}P = \overline{\pi}$. Note the P matrix produced here represents a less stiff embedded DTMC than the standard uniformisation transformation would generate (a small improvement on [14]).

2.5 First passage times in Semi-Markov Processes

The formulation of semi-Markov process analysis was presented seminally by Pyke [15]. However it was only through transformations presented in [16,17] that it was realised that passage-time and transient analysis could be of a form appropriate for asynchronous computation. We summarise this briefly below.

Consider a finite, irreducible, continuous-time semi-Markov process with N states. Recalling that $Z(t)$ denotes the state of the SMP at time t ($t \geq 0$), the first passage time from a source state i at time t into a non-empty set of target states $\vec{j}$ is:

$$P_{i\vec{j}} = \inf\{u > 0 : N_s(u) \geq M_{i\vec{j}}\} \tag{5}$$

where the SMP is time-homogeneous and $M_{i\vec{j}} = \min\{m \in \mathbb{Z}^+ : X_m \in \vec{j} \mid X_0 = i\}$ is the transition marking the terminating state of the passage. $M_{i\vec{j}}$ is a random variable that finds the transition number of the transition that takes the given trace into the target set of states for the first time. $P_{i\vec{j}}$, applied to the same trace, finds the shortest passage time, by selecting the first instant that the system reaches the target set. This formulation of the random variable $P_{i\vec{j}}$ applies to an SMP with no immediate (that is, zero-time) transitions.

We represent the Laplace transform of the passage-time random variable $P_{i\vec{j}}$ with $L_{i\vec{j}}(s)$, which can be calculated by:

$$L_{i\vec{j}}(s) = \underbrace{\sum_{k \notin \vec{j}} r^*_{ik}(s) L_{k\vec{j}}(s)}_{(*)} + \underbrace{\sum_{k \in \vec{j}} r^*_{ik}(s)}_{(\dagger)} \quad : \text{for } 1 \leq i \leq N \tag{6}$$

where $r^*_{ik}(s)$ is the Laplace–Stieltjes transform (LST) of $R(i, k, t)$ from Section 2.1 and is defined by:

$$r^*_{ik}(s) = \int_0^\infty e^{-st}\, \mathrm{d}R(i, k, t) \tag{7}$$

The move into Laplace space to calculate the passage time quantity makes use of the Laplace transform's pleasant properties under convolution. Equation (6) essentially creates a recursive definition for $L_{i\vec{j}}(s)$ where the passage from i to $\vec{j}$ can be accomplished in $(*)$ by progressing one transition to an intermediate state $k \notin \vec{j}$ with distribution $r^*_{ik}(s)$ and convolving that with the passage from k to $\vec{j}$, $L_{k\vec{j}}(s)$. Alternatively, in $(\dagger)$ the passage takes the transition from i directly to the target set $\vec{j}$, if such a transition exists, $r^*_{ik}(s)$ where $k \in \vec{j}$.

Equation (6) has an immediately derivable matrix–vector form of:

$$\underline{L}_{\vec{j}}(s) = U' \underline{L}_{\vec{j}}(s) + \underline{b}_{\vec{j}} \tag{8}$$

where $\underline{L}_{\vec{j}}(s) = (L_{1\vec{j}}(s), \ldots, L_{N\vec{j}}(s))^T$, $\underline{b}_{\vec{j}} = (\sum_{k \in \vec{j}} r^*_{1k}(s), \ldots, \sum_{k \in \vec{j}} r^*_{Nk}(s))^T$ and:

$$U'_{ij} = \delta_{j \notin \vec{j}}\, r^*_{ij}(s) \tag{9}$$

where $\delta_X = 1$ if X is true and 0 otherwise.

Since semi-Markov passage-time equation Equation (8) is in the standard fixed-point form, $\underline{x} = U'\underline{x} + \underline{b}$, no further rearrangement is required. It is shown in Section 6 that the principal matrix, U', satisfies the spectral radius requirement necessary for asynchronous solution, in all but two specific cases. In one of those cases, the passage-time can be stated without calculation; in the other case, a simple passage-time preserving aggregation can be applied to the semi-Markov process to recover the original spectral radius condition.

2.6 Transient Analysis of Semi-Markov Processes

We look at a related problem of finding the transient probability distribution in a semi-Markov chain. From [17], we obtained the following equation for the transient probability, $\pi_{i\overline{j}}(t)$, of being in a state in $\overline{j}$ at time t having started from a state i and time 0. $\pi^*_{i\overline{j}}(s)$ represents the Laplace transform of $\pi_{i\overline{j}}(t)$ below:

$$\pi^*_{i\overline{j}}(s) = \delta_{i \in \overline{j}} \overline{F}^*_i(s) + \sum_{k=1}^{N} r^*_{ik}(s) \pi^*_{k\overline{j}}(s) \tag{10}$$

$\overline{F}^*_i(s) = \frac{1}{s}(1 - h^*_i(s))$ is the Laplace transform of the reliability function, where $h^*_i(s)$ in turn represents the Laplace transform of the state sojourn distribution $h_i(t)$.

Thus extracting a matrix–vector form we get:

$$\overline{\pi}_{\overline{j}} = R\overline{\pi}_{\overline{j}} + \overline{f}_{\overline{j}} \tag{11}$$

where $\overline{f}_{\overline{j}} = (\delta_{1\in\overline{j}}\overline{F}^*_1(s), \ldots, \delta_{N\in\overline{j}}\overline{F}^*_N(s))^T$ and $R_{ij} = r^*_{ij}(s)$.

As before Equation (11) is a fixed-point problem of the sort $\overline{x} = R\overline{x} + \overline{b}$ that can be tackled by asynchronous iterations, this time if the matrix R satisfies certain properties (as discussed in Section 6).

2.7 First passage times and Transient analysis in CTMCs

As with steady-state analysis, the technique here is to translate the CTMC to an SMP and apply the passage-time or transient procedure for an SMP. The standard way to perform passage-time analysis on a CTMC is to modify the transition matrix so that the target states of the passage are absorbing and perform uniformisation [18,8,19] on the resulting chain. While transient analysis of CTMCs relies directly on the uniformisation technique without modification. However, the uniformisation technique is not amenable to being transformed into a fixed-point problem and thus we are forced to use the SMP conversion route again.

From Section 2.2, a CTMC with generator matrix A once transformed into an SMP has $r^*_{ij}(s)$ entries:

$$r^*_{ij}(s) = \begin{cases} \frac{a_{ij}}{s-a_{ii}} & : |a_{ii}| > 0 \\ 0 & : a_{ii} = 0 \end{cases} \tag{12}$$

We can now rewrite the passage time formula from Equation (6) as:

$$L_{i\overline{j}}(s) = \sum_{k\notin\overline{j}} \frac{a_{ik}}{s-a_{ii}} L_{k\overline{j}}(s) + \sum_{k\in\overline{j}} \frac{a_{ik}}{s-a_{ii}} \qquad : \text{for } 1 \le i \le N \tag{13}$$

This too is a fixed-point problem of the sort $\overline{x} = U'\overline{x} + \overline{b}$.

A similar argument can be made for performing transient analysis on CTMCs. We specialise Equation (10) to the case where the semi-Markov process represents a CTMC as above to get an expression for the transient distribution of the CTMC:

$$\pi^*_{i\overline{j}}(s) = \frac{\delta_{i\in\overline{j}}}{s - a_{ii}} + \sum_{j=1}^{N} \frac{a_{ij}}{s - a_{ii}} \pi^*_{k\overline{j}}(s) \quad : \text{for } 1 \leq i \leq N \tag{14}$$

Again, this is a fixed-point problem of the sort $\overline{x} = U\overline{x} + \overline{c}$.

2.8 Solution of passage time and transient problems

We have produced fixed-point equations of the form $\overline{x} = U'\overline{x} + \overline{b}$ or $\overline{x} = U\overline{x} + \overline{c}$, for a passage-time analysis or transient analysis respectively. From here we will generate and solve approximately $30 \times m$ such equations, where m is the number of time points in the inversion: each equation represents a distinct s-value. These s-values are chosen deliberately to match the requirements of the Laplace inversion algorithm e.g. Laguerre [20], Euler [21], so that a time-based inversion of the resulting passage-time or transient distribution can be produced. Further details of this numerical inversion approach to solving passage-time equations can be found in [16].

3 Traditional Iterative Solution for General Linear Fixed-Point Problems

In this section we review the traditional iterative solution methods for linear fixed-point problems.

Traditional synchronous iterative methods are variants of the power method [6]. These methods are stable, and relatively easy to implement over parallel systems. This makes these iterative methods particularly well-suited to problems with very large state spaces.

As variants of the power method, we may think of synchronous iterative methods as being those which involve construction of vector sequences, $\langle\overline{x}^k\rangle$, where $\overline{x}^{(0)} \in \mathbb{C}^n$ and $\overline{x}^{(k+1)} = f(\overline{x}^{(k)})$ for some function $f : \mathbb{C}^n \to \mathbb{C}^n$ (where f is sometimes referred to as the *iteration mapping*) [22]. The two principal types of synchronous iterative method are *Jacobi* and *Gauss–Seidel*. Jacobi methods involve updating all vector components at each iteration. Specifically, the Jacobi iteration mapping is given by $\overline{x}^{(k+1)} = M\overline{x}^{(k)}$. Gauss–Seidel methods, by contrast, involve updating each of the vector components (or, blocks of vector components) consecutively. Assuming the vector components are updated in index order, the Gauss–Seidel iteration mapping may be given by $x_i^{(k+1)} = \sum_{j=1}^n M_{ij} x_j^{(k)}$ if component i is being updated and $x_i^{(k+1)} = x_i^{(k)}$ if component i is not being updated.

Within a uni-processor setting, Gauss–Seidel methods are known to be preferable to Jacobi methods. This follows by the classical Stein–Rosenberg theorem [23]. By this theorem, it is well known that the Jacobi and Gauss–Seidel iterations converge at a geometric rate to the solution vector. Now, assuming that a Jacobi iteration requires approximately the same amount of time as n Gauss–Seidel iterations, the rate of Gauss–Seidel convergence is significantly faster.

Within a parallel set-up, this advantage is often reversed. If the dependency graph associated with a parallel implementation is sufficiently complete—so few Gauss–Seidel iterations can occur concurrently—then the average time taken to complete a Gauss–Seidel iteration is greater than or equal to the average time taken complete a Jacobi iteration (not least because of the messaging overheads). When this is the case, then by theoretical work presented in[24] we know that Jacobi methods are often faster, in terms of wall time, than Gauss–Seidel methods.

4 Asynchronous Iterative Solution of General Linear Fixed-Point Problems

In this section we outline the general theory of asynchronous iterative solution [1,25,23], noting the potential advantages of asynchronous iterative solution relative to synchronous iterative solution.

The theory of asynchronous iterative solution was originally presented by Chazan and Miranker in [1]. Fundamentally, this theory provides a solution method for fixed-point problems. Chazan and Miranker concerned themselves only with linear problems, but subsequent work extended the theory to non-linear problems also (for example, [25]). In this paper, we need only to concern ourselves with the theory governing solution of linear problems. This may be outlined as follows.

4.1 Formal Description

Consider the general fixed-point problem:

$$\overline{x} = A\overline{x} + \overline{b}, \text{ where } A \in \mathbb{C}^{n \times n}, \overline{x}, \overline{b} \in \mathbb{C}^n. \tag{15}$$

To solve for the unique fixed-point $\overline{x}$, we generate a convergent sequence of iterates from the corresponding *class of asynchronous iterations*.

The class of asynchronous iterations is given by the class of sequences, $\langle \overline{x}^{(t)} \rangle$, of column vectors in $\mathbb{C}^n$ recursively defined:

$$x_i^{(t+1)} := \begin{cases} \sum_{j=1}^{n} A_{ij} x_j^{(t-d(t,i,j))} + b_i & \text{if } i \in U(t) \\ x_i^{(t)} & \text{if } i \notin U(t) \end{cases} \tag{16}$$

where U, d are functions satisfying a set of conditions. Function $U : \mathbb{N} \to \mathbb{P}\{1, \ldots, n\}$, which we may refer to as the update function, gives the set of vector entries to be updated at each step (where $\mathbb{P}\{1, \ldots, n\}$ denotes the power set of $\{1,\ldots,n\}$). Function $d : \mathbb{N} \times \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathbb{N}$, which we may refer to as the delay function, gives the relative "age" of the entries used in the update of each vector entry at each step. These functions satisfy the following conditions:

$\mathcal{CM}$1: Each vector entry i $(1 \leq i \leq n)$ features in an infinite number of update sets.

$\mathcal{CM}$2: For each pair of vector entries i, j $(1 \leq i, j \leq n)$, we have that:

$$\begin{cases} (t - d(t,i,j)) \to \infty \text{ as } t \to \infty; \\ \forall t (d(t,i,j) \leq t). \end{cases}$$

Condition $\mathcal{CM}1$ prevents a step being reached after which one of the vector entries, say i, is no longer updated. Condition $\mathcal{CM}2$ prevents information from a particular step being used repeatedly in updates *ad infinitum* instead of "newer" information.

Generalising the fundamental result of Chazan and Miranker's seminal paper [1], we get the following convergence guarantee for generated sequences [26]. It allows us to know pre-generation whether sequences are convergent.

Theorem 1 *Consider the class of asynchronous iterations corresponding to equation Equation (15). If $\rho(|A|) < 1$, then every sequence of iterates within the class converges to the unique fixed-point $\overline{x}$ as $t \to \infty$; if $\rho(|A|) \geq 1$, then some non-convergent sequence of iterates exists within the class.*

In this theorem, $|A|$ denotes the matrix whose entries are the absolute values of the corresponding entries in A—and we term $|A|$ the modulus matrix of matrix A. The theorem is a generalisation on three fronts: delay is permitted to depend both on the entry being updated and on the entry being used in the update; multiple entries are permitted to be updated at any particular time; and delay does *not* require a fixed, finite bound.

Theorem 1 gives a sufficiency condition for sequence convergence. This may surprise the reader given how lax are class-defining conditions $\mathcal{CM}1$ and $\mathcal{CM}2$. They impose no restrictions on how often a vector entry is updated: an entry may be updated every step, or it may be updated once every million steps. They permit update information to be "chronologically" mal-ordered: for instance, a vector entry at a given step, say $x_j^{(2)}$, may be used to update some other entry, i, before an earlier step of the updating entry, say $x_j^{(1)}$, is used to update entry i. And, they permit update information to be lost: an entry at a given step, say $x_j^{(3)}$, may never be used to update vector entry i, or, indeed, to update any other vector entry.

Whilst theorem 1 gives a necessary condition for convergence of all sequences within the particular class, it does not give a necessary condition for convergence of any particular sequence. If the condition is contravened, we cannot claim that all sequences within the class do not converge. Consider, for example, any sequence for which $\overline{x}^{(0)} = \overline{x}$. Such a sequence will clearly converge no matter the spectral radius of the matrix $(\forall t(\overline{x}^{(t)} = \overline{x}))$. All we can claim, if theorem 1's convergence condition is contravened, is that some non-convergent sequence exists within the class. This is an important point which will be revisited in the next section.

4.2 Advantages of Asynchronous Techniques

The advantages of asynchronous iterative methods as compared with synchronous methods are well-documented [26–28]. Asynchronous techniques provide a faster wall-time solution than is seen in synchronous techniques such as Jacobi or Gauss–Seidel methods [22]. This is especially true when processors perform iterations at different speeds or inter-processor communication times are inhomogeneous; both events will likely cause inactive delays in synchronous methods as processors are forced to wait for updates before moving on to the next iteration. Lubachevsky and Mitra [28], present an analogous account of solution speed-up for implementations over single processor systems: in particular, where the size of the principal matrix is substantially larger than the storage capacity of the main memory.

It has also been observed that asynchronous algorithms can have faster asymptotic convergence properties, requiring fewer iteration steps to converge than their synchronous counterparts [29].

Synchronous iterative solvers tend to be more difficult to implement than their asynchronous equivalents, especially when being implemented over large parallel or distributed environments; increasingly required for analysis of massive linear system solution [16]. There are three main reasons for this. In synchronous algorithms it is harder to manage global synchronisation, lossy communication channels, and restarts in distributed environments.

Finally, asynchronous solvers are expected to make significantly more efficient use of the communication channel and memory. By definition, synchronous, iterative solvers tend to have bursty communication and memory-access patterns. This may yield communication or memory-access delays that asynchronous solvers do not encounter.

5 Asynchronous Iterative Solution of Dominant Eigenvector Problems

Asynchronous techniques have been applied to many application areas (as referenced in Section 1). However, as noted by Bertsekas and Tsitsiklis in [22]: “Much remains to be done to enlarge the already substantial class of problems for which asynchronous [iterative] algorithms can be correctly applied.”

In this section we consider the application of asynchronous iterative solution theory to the problem of finding the unique dominant eigenvectors which are associated with positive real dominant eigenvalues.

Let $M \in \mathbb{C}^{n\times n}$ be a matrix with such a positive real dominant eigenvalue and unique associated dominant eigenvector. Without loss of generality we may assume this dominant real eigenvalue is equal to one. Then we are considering using the class of asynchronous iterations to solve for unique $\overline{x}$ in:

$$\overline{x} = M\overline{x} \tag{17}$$

5.1 Non-Trivial Application to Dominant Eigenvector Problems

If the class of asynchronous iterations corresponding to Equation (17) is to provide the possibility of faster solution than corresponding classes of synchronous iterations, then we require a method for knowing pre-generation that the asynchronous sequences which we might generate are convergent. Unfortunately, theorem 1 does not (in any obvious way) provide us with the requisite convergence guarantee.

Firstly, $\overline{x}$ is not a unique fixed-point for Equation (17). For any $\lambda \in \mathbb{R}$, $\lambda\overline{x}$ is also a fixed-point. That is to say, vector $\overline{x}$ is unique only up to scalar multiples.

The second concern is that we do not have strictly sub-unit spectral radius for the associated modulus matrix.

It is well known [30] that,

$$\rho(|M|) \geq \rho(M) = 1. \tag{18}$$

and, by definition of M^*,

$$\rho(|M^*|) \geq \rho(M^*) \geq 1. \tag{19}$$

Two approaches present themselves for resolving these difficulties. Recalling that theorem 1 does not give necessary conditions for sequence convergence, the first approach begins by noting that for many matrices M, within the class of asynchronous iterations corresponding to Equation (17), there are many non-constant sequences ($\overline{x}^{(0)} \neq \lambda\overline{x}, \lambda \in \mathbb{R}$) which do converge to non-zero, real multiples of the steady-state distribution $\overline{x}$. (Actually, the only sequences which converge to $\overline{0}$ are those for which $\overline{x}^{(0)} = \overline{0}$.) Given this, the aim is to find a set of conditions which picks out a significant number of the non-zero, convergent sequences.

This set of conditions would then serve as a convergence guarantee in theorem 1's stead. This is the approach adopted in [28,31,32].

The second approach has as its aim to find a vector $\overline{v} \in \mathbb{C}^n$ and a matrix $\hat{M} \in \mathbb{C}^{n\times n}$ such that

$$\rho(|\hat{M}|) < 1 \tag{20}$$

and the unique steady-state distribution $\overline{x}$ of Equation (17) is also the unique fixed-point of

$$\overline{x} = \hat{M}\overline{x} + \overline{v}. \tag{21}$$

If this is achievable, then we can generate sequences from the entire class of asynchronous iterations corresponding to Equation (21) and we can employ theorem 1 as a convergence guarantee for these generated sequences.

We consider the two approaches in turn.

5.2 Partially Asynchronous Perron–Vector Solution

In this subsection we consider the partially asynchronous method presented in [28]. Two other partially asynchronous methods may be found in [31,32], both of which appeal to the concept of *paracontractions*.

Lubachevsky and Mitra [28] present an asynchronous solution framework for the fixed-point problem:

$$\overline{y} = U\overline{y}, \tag{22}$$

where unique vector $\overline{y} \in \mathbb{R}^n$ is both positive and of unit one-norm and matrix $U \in \mathbb{R}^{n\times n}$ is non-negative and is irreducible.

Strictly, their framework also requires U to have a non-zero diagonal entry. But, this requirement is easily transcended through a simple pre-processing step.

Lubachevsky and Mitra [28] point out that for any $\alpha \in (0, 1]$, Equation (22) can be rewritten as:

$$\overline{y} = (\alpha U + (1 - \alpha)I)\overline{y} \tag{23}$$

In this well-known Equation (23) from [33], matrix $(\alpha U + (1 - \alpha)I)$ is clearly irreducible. So, if U does not comprise a positive diagonal entry, then we can easily find another irreducible matrix which has both the same dominant eigenvector and has a positive diagonal.

Given a suitable U, we find in [28] a set of conditions which picks out a subclass from the corresponding class of asynchronous iterations. If we let $\tilde{\imath} \in \{1, 2, \ldots, n\}$ be some particular index such that $U_{\tilde{\imath}\tilde{\imath}} > 0$, then these conditions are:

$\mathcal{LM}$1: $y_{\tilde{\imath}}^{(0)} > 0$
$\mathcal{LM}$2: Delay is bounded by some finite Δ.
$\mathcal{LM}$3: There is some finite update bound Σ such that every vector entry is updated at least once every Σ consecutive updates.

$\mathcal{LM}$4: $\forall s (d(s, \tilde{\imath}, \tilde{\imath}) = 0)$

This subclass of asynchronous iterations which is picked out may be referred to as being *partially* asynchronous [23]. This is because condition $\mathcal{LM}3$ imposes an update bound. Asynchronous subclasses which are not subject to such an update bound are referred to in the literature as *totally* asynchronous.

Associated with the subclass of asynchronous iterations which is picked out, Lubachevsky and Mitra provide us with the following convergence guarantee.

Theorem 2 *Consider the subclass of asynchronous iterations, as picked out by the four conditions, which corresponds to equation Equation (22). If $\rho(U) \leq 1$, then for every sequence within the subclass there is some finite, positive, real λ such that the sequence converges to $\lambda \overline{y}$ as $t \to \infty$.*

5.3 Totally Asynchronous Dominant Eigenvector Solution

In the previous subsection we considered an asynchronous iterative solution method for dominant eigenvectors with two key limitations. The method may only be applied to non-negative matrices. The method also has limited application across fully distributed architectures as it requires that both delay and update frequency have fixed bounds. In [28] Lubachevsky and Mitra point out a third, less significant, difficulty: "It is correct to say that the effective restriction implied by [$\mathcal{LM}4$] is that the coordinate with index $\tilde{\imath}$ is assigned to only one processor."

In this subsection we seek an asynchronous iterative method which avoids the three difficulties mentioned above.

Our aim is to reformulate Equation (17) to be of the same type as Equation (15), where crucially the reformulation yields strictly sub-unit spectral radius for the modulus matrix. If this is achievable, then we will through Theorem 1 be able to employ the entire class of corresponding asynchronous iterations.

5.3.1 Theorem of Matrix–Vector Splitting

Key to the reformulation is the following theorem.

Theorem 3 *Let the following hold.*

($\mathcal{MV}$1) *$M \in \mathbb{C}^{n \times n}$ is irreducible.*
($\mathcal{MV}$2) *$\rho(|M|) = 1$.*
($\mathcal{MV}$3) *$\overline{v} \in \mathbb{C}^n$.*
($\mathcal{MV}$4) *$V^{(K)} \in \mathbb{C}^{n \times n}$ $(K \subseteq \{1, 2, \ldots, n\})$ is defined by:*

$$V_{ij}^{(K)} := \begin{cases} v_i & \textit{if } j \in K \\ 0 & \textit{if } j \notin K \end{cases}$$

($\mathcal{MV}$5) *$\overline{v}, K$ are chosen such that $(M - V^{(K)})$ has zero entries and it is possible to replace some of the zero entries in $(M - V^{(K)})$ with $\alpha > 0$ to form a matrix, $(M - V^{(K)})^{(\alpha)}$, with irreducible modulus equivalent where $\rho(|(M - V^{(K)})^{(\alpha)}|) \leq 1$.*

Then, we know the following to be true.

(C1) *$\rho(|M - V^{(K)}|) < 1$;*

(C2) *If 1 is an eigenvalue of M, then it is a simple eigenvalue of M and there is a corresponding right eigenvector $\overline{x}$ of M, which comprises no zero entries, and is the unique fixed-point of*

$$\overline{x} = (M - V^{(K)})\overline{x} + \overline{v}. \tag{24}$$

Proof. Taking each case in turn:

(C1) Suppose there is such a α. Then $|(M - V^{(K)})^{(\alpha)}|$ is irreducible and also $\rho(|(M - V^{(K)})^{(\alpha)}|) \leq 1$. If we now let α tend to zero then we may bound $\rho(|(M - V^{(K)})^{(\alpha)}|)$ strictly away from 1, by the Perron–Frobenius Theorem [34]. By continuity of spectral radius with respect to matrix entries [35]—in particular, continuity when entries become zero and irreducibility is potentially compromised—we have:

$$\rho(|(M - V^{(K)})|) < 1. \tag{25}$$

(C2) Suppose $\overline{y} = M\overline{y}$. Then:

$$\overline{y} = (M - V^{(K)})\overline{y} + (\sum_{k \in K} y_k)\overline{v}. \tag{26}$$

The last equality holds because:

$$V^{(K)}\overline{y} = \begin{pmatrix} v_1 \sum_{k \in K} y_k \\ \vdots \\ v_n \sum_{k \in K} y_k \end{pmatrix} = (\sum_{k \in K} y_k)\overline{v}$$

Now, by the preceding, we know that $\rho(|M - V^{(K)}|) < 1$. We also know from [30] that $\forall N \in \mathbb{R}^{n \times n}(\rho(|N|) \geq \rho(N))$. So, 1 is not an eigenvalue of $(M - V^{(K)})$. This implies that $\sum_{k \in K} y_k \neq 0$. Now, $\forall \alpha \neq 0$, we have that $\overline{z} := (M - V^{(K)})\overline{z} + \alpha\overline{v}$ and that $\overline{z} = \alpha(I - (M - V^{(K)}))^{-1}\overline{v}$. The latter is valid since $\rho(M - V^{(K)}) < 1$. Accordingly, the fixed-point $\overline{x}$ of $\overline{x} = (M - V^{(K)})\overline{x} + \overline{v}$ is a scalar multiple of $\overline{y}$, and thus $\overline{x}$ is a right eigenvector of M corresponding to 1.

Further, given non-singularity, $\overline{x}$ is the unique fixed-point, and so 1 is a simple eigenvalue of M. Finally, $\overline{x}$ comprises no zero entries. This is known because we may choose $V^{(K)}$ to comprise exactly one non-zero entry, so the non-zero coefficient $(\sum_{k \in K} y_k)$ in Equation (26) reduces to a non-zero coefficient y_k, and, further, by irreducibility, we may choose $V^{(K)}$ such that the index k corresponding to y_k is any index from 1 through n.

5.3.2 Example Splittings

Whilst remarkably powerful, the theorem above, with $\mathcal{MV}5$, is not immediately computationally useful. To illustrate the possibilities it presents, let us consider one of several possible stronger versions of the condition $\mathcal{MV}5'$. $\overline{v}$ and K are chosen such that:

(1) $\forall i, j \ (0 \leq |M_{ij} - V_{ij}^{(K)}| \leq |M_{ij}|)$;
(2) $\exists i, j \ (0 \leq |M_{ij} - V_{ij}^{(K)}| < |M_{ij}|)$.

This more clearly illustrates what can be done.

Let us choose $\overline{v}, K$ such that $V^{(K)}$ comprises just a single non-zero entry and this non-zero entry equals the corresponding non-zero entry in M (of which there is clearly one). This satisfies our $\mathcal{MV}5'$. It allows us, in effect, to "zero" any entry of our choice in the coefficient matrix of the problem which we are tackling.

Let us choose $\overline{v}$ such that each entry in the vector is equal to the corresponding entry in the first column of matrix M. This satisfies our $\mathcal{MV}5'$. Similarly to the first choice of $\overline{v}, K$, this particular choice allows us "zero" any column of our choice in the coefficient matrix of the problem which we are tackling.

5.3.3 Further Implications of Matrix–Vector Splitting

The theorems presented in this chapter are significant for at least three reasons—beyond, of course, that they open the door to totally asynchronous iterative solution, as required.

Unique Solution for Complex Matrices By the Perron–Frobenius theorem, we know that if M is real and non-negative then there is necessarily a unique (up to scalar multiples) dominant eigenvector $\overline{x}$ and this eigenvector comprises only positive entries. Theorem 3 does not necessitate that there is a solution vector $\overline{x}$, if M is comprised of negative entries or has entries with imaginary parts. Consider for example,

$$\begin{pmatrix} -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \tag{27}$$

This matrix has no eigenvalue 1, so no corresponding dominant eigenvector $\overline{x}$. What Theorem 3 does tell us is that if there is an $\overline{x}$ when M has negative entries, e.g.

$$\begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \tag{28}$$

or entries with imaginary parts, then this eigenvector $\overline{x}$ is unique (up to scalar multiples). Further, the vector comprises only non-zero entries, and it can be derived using Theorem 3.

Minimising Spectral Radius It is well-known that for Equation (26), the smaller $\rho(|M - V^{(K)}|)$, the faster we may expect solution convergence [36]. Building on Theorem 3, a worthwhile future research endeavour would be to investigate strategies for choosing $K, \overline{v}$ so as to minimise $\rho(|M - V^{(K)}|)$.

Optimal Sparsity Patterns With a view toward accelerating iterative linear system solution methods, much effort is currently being expended investigating graph and hypergraph partitioning strategies for matrices over several processes [37–40]. These strategies aim to balance computation work-load appropriately over the processes and to minimise inter-process communication. By suitably choosing $K, \overline{v}$ as permitted by $\mathcal{MV}5$, we may maximise the sparsity $\rho(|M - V^{(K)}|)$, and, in conjunction with (or instead of) graph/hypergraph partitioning strategies we may optimise process work-load and minimise inter-process communication. Quite how best to choose $K, \overline{v}$ for these ends is as yet unknown.

6 Asynchronous Iterative Solution of Passage Time and Transient Analysis Problems

In this section we show that the spectral radius requirement may be satisfied so that asynchronous iterations may be employed to solve passage-time and transient analysis problems.

Before doing so, however, we put forward a lemma. We will appeal to this lemma in the proof of the subsequent proposition.

Lemma 4 *If* $\mathrm{Re}(s) > 0$, $|h^*_{ij}(s)| = 1$ *if and only if* $h(t) = \delta(t)$.

Proof. Assume $h^*_{ij}(s) = \int_0^\infty h_{ij}(t)e^{-st}\,\mathrm{d}t = 1$ for some s, where $\int_0^\infty h_{ij}(t)\,\mathrm{d}t = 1$ and $h_{ij}(t) \geq 0$ for all $t \geq 0$. Subtracting the two integrals, gives:

$$\int_0^\infty h_{ij}(t)(1 - e^{-st})\,\mathrm{d}t = 0 \tag{29}$$

This can be true if $s = 0$, but restricting ourselves to $\mathrm{Re}(s) > 0$, implies that $\mathrm{Re}(1 - e^{-st}) > 0$ for all $t > 0$. Given that $h_{ij}(t) \geq 0$ for $t \geq 0$, Equation (29) can only evaluate to 0 if $h_{ij}(t) = 0$ for all $t > 0$. If $h_{ij}(t)$ is to be a valid probability density function and satisfy $\int_0^\infty h_{ij}(t)\,\mathrm{d}t = 1$ also then the only candidate for $h_{ij}(t)$ is the Dirac delta function. □

Let us now recall from our passage time derivations of Equations (6) and (13) that we have a fixed-point problem of the sort:

$$\overline{x} = U'\overline{x} + \overline{a} \tag{30}$$

Let us recall also from our transient probability derivation of Equations (10) that we have a fixed-point problem of the sort:

$$\overline{x} = R\overline{x} + \overline{b} \tag{31}$$

These equations are solvable by the method of asynchronous iterations as given by Theorem 1 if, respectively, $\rho(|U'|) < 1$ and $\rho(|R|) < 1$.

We will show that the first inequality holds, by way of the following proposition. We may show that the second inequality holds in analogous fashion—the proof of the following proposition makes no appeal to the difference between U' and R, namely the replacement of some entries with 0.

Proposition 5 $\rho(|U'|) < 1$ *unless either of the following are true:*

(1) All the transitions in the SMP are immediate; or
(2) There is an absorbing strongly-connected subcomponent of the state space, which comprises states all of which have immediate transitions ($h_{ij} = 1$ for all i, j in the subcomponent).

Proof. From [36], we know that $\rho(|U'|) < 1$ if we can construct a matrix M from U' by replacing some of the zeros in U' with non-zeros, where the newly constructed M satisfies all three of the following:

($\mathcal{PT}$1) $\|M\|_\infty \leq 1$
($\mathcal{PT}$2) $\exists k \sum_j |M_{kj}| < 1$
($\mathcal{PT}$3) M is irreducible

If this matrix can be constructed[1] then by reasoning similarly to the proof of Theorem 3, we know that

$$\rho(|U'|) \leq \rho(|M|) < 1$$

The entries of U' consist of $r^*_{ij}(s)$ or 0 as defined by Equation (9). Now $r^*_{ij}(s) = p_{ij}h^*_{ij}(s)$ where $0 \leq p_{ij} \leq 1$ is a probability. From [41, Lemma 1], we know that $|h^*_{ij}(s)| \leq 1$ for all i, j. Therefore:

$$\sum_j |U'_{ij}| \leq \sum_j |r^*_{ij}(s)| \leq 1$$

If we have a situation where $\sum_j |U'_{ij}| < 1$, then we can generate a complete dense row M_i such that $\sum_j |M_{ij}| < 1$ as per the requirement for constructing M.

[1] Note that we only require that it is possible to construct such a matrix M. We do not require such a matrix to have any meaning in the original context of an SMP or CTMC.

We know from Lemma 4 that for $\mathrm{Re}(s) > 0$, $|h^*_{ij}(s)| = 1$ if and only if $h(t) = \delta(t)$, a Dirac delta function at 0, i.e. only if the transition from i to j is immediate.

With this result, we can say that for some i, $\sum_j |U'_{ij}| = 1$ if and only if all of the out-transitions from state i are immediate.

With this information, we can state that the only two cases where we cannot construct such a matrix M are as stated in the proposition. $\square$

Clearly, if all the transitions in the SMP are immediate, then all passage times are 0 without calculation. So, the first exceptional case has a trivial associated result.

Let us now consider the second exceptional case. This case has two subcases: the subcase where the subcomponent contains one or more target states for the passage, and the subcase where it does not. Whichever is true, we can aggregate the irreducible absorbing subcomponent of the SMP into a single state with no out-edges, but maintaining all the previous in-edges to any state of the subcomponent.

If none of the target states was in the absorbing subcomponent, the passage time will be unaffected. If one or more of the target states for the passage was in the subcomponent, the passage time is also unchanged since the additional time from the boundary of the subcomponent to any of the target states is by definition zero.

Once this aggregation has been carried out, M can be notionally generated albeit for an aggregated, but passage-time preserving, system, and the result $\rho(|U'|) < 1$ has been achieved. Thus showing that asynchronous iterations can be applied to passage-time calculations in SMPs and CTMCs.

7 Evaluation of Three Types of Asynchronous Algorithm

In this section, we present three particular types of totally asynchronous algorithm, and we present empirical test results concerning the performance of these algorithms relative to a Jacobi algorithm when solving for the steady-state vectors of flexible-manufacturing-system models. Comparison is made in terms of the number of updates, the number of messages sent, and wall time before algorithm termination.

7.1 Three Algorithms

We have shown earlier in this paper that the full breadth of Chazan and Miranker's class of asynchronous iterations may be used to solve for dominant eigenvectors. This opens the door to a great many different possible solution algorithms.[2]

[2] It is perhaps worth noting that Jacobi and Gauss–Seidel methods both conform to Chazan and Miranker's class definition

```
(1) Apply a matrix-vector splitting to the problem matrix, $M$,
    to yield an equation of the form Equation (26), defined by a
    matrix, $M'$, equal to $(M - V^{(K)})$, and a vector, $\overline{v}$
(2) Given some initial vector, $\overline{x}^{(0)}$, let each process construct a
    terminating sequence of local vector restrictions, $\langle \overline{x}|^{(k)}_{\texttt{local}} \rangle$,
    by messaging each other a start command, and then looping
    one of the following three if statements until no process is
    updating to produce the next vector or sending/receiving a
    message

  • Asynchronous Eager:
    if (new message received from remote process) {
            update($\overline{x}$)
    }
  • Asynchronous Reluctant:
    if (new message received from remote process) {
        until ($\|\overline{x}|^{(\texttt{current})}_{\texttt{local}} - \overline{x}|^{(\texttt{before latest update})}_{\texttt{local}}\| < \epsilon_2$)
            update($\overline{x}$)
    }
  • Asynchronous Variable Threshold:
    if (new message received from remote process) {
        until earlier of ($\|\overline{x}|^{(\texttt{current})}_{\texttt{local}} - \overline{x}|^{(\texttt{before latest update})}_{\texttt{local}}\| < \epsilon_2$,
        $N_I$ iterations within until loop)
            update($\overline{x}$)
    }

  define update($\overline{x}$) {
     Update all local vector entries, $x_{i_{\texttt{local}}}^{(\texttt{current})}$,
     by way of single Gauss-Seidel iteration,
     and then normalise;
     if ($\|\overline{x}|^{(\texttt{updated})}_{\texttt{local}} - \overline{x}|^{(\texttt{previous message sent})}_{\texttt{local}}\| > \epsilon_1$)
          message $x_{i_{\texttt{local}}}^{(\texttt{updated})}$ to remote process responsible
          for $x_{j_{\texttt{remote}}}^{(\texttt{next})}$ such that $M'_{j_{\texttt{remote}}\, i_{\texttt{local}}} \neq 0$
  }
```

Fig. 1. A description of the three asynchronous iterative algorithms

In Figure 1 we introduce three particular solution algorithms—the third actually being a spectrum of particular algorithms. The three algorithms differ with respect to the frequency with which messages are sent.

Different Asynchronous Variable Threshold algorithms may be generated by varying the threshold value given by iteration N_I. In particular, at the boundaries, when $N_I = 0$ the Asynchronous Variable Threshold is equivalent to the Asynchronous Eager, and when $N_I = \infty$ the Asynchronous Variable Threshold is equivalent to the Asynchronous Reluctant.

In the descriptions of the three algorithms it is stated that a message is sent when the change to the local portion of the vector in terms of some suitable norm—typically, an induced p-norm ($1 \leq p \leq \infty$) [42]—is greater than some specified threshold value. In addition, each local update cycle in the Asynchronous Variable Threshold and in the Asynchronous Reluctant algorithms completes when the change to the local portion of the vector in terms of some suitable norm is greater than some threshold. This second threshold need not to be equal to the messaging threshold. Also, the norm used to conclude each update cycle need not to be the same as the norm used for messaging.

We note that the algorithms presented here are sender-centric, analogous algorithms may be constructed whereby processes read, rather than send, information from each other according to the steps of the algorithm. Examples of reader-centric algorithms (applied to non-singular linear problems) may be found in [43].

For these algorithms to adhere to the Chazan–Miranker definition, we require that each and every vector entry is continuously updated, and we require also that newer information is used to update each of the vector entries—by which we mean that the information used in the update of any of these vector entries is not bounded above by some time step.

The Asynchronous Eager and Variable Threshold algorithms clearly satisfy these conditions. Each update cycle within the Asynchronous Reluctant algorithm may be shown to conclude, and thereby it follows easily that the conditions are also satisfied. Local convergence of the Asynchronous Reluctant algorithm follows from the Perron–Frobenius theorem together with the continuity of spectral radius using a similar argument to the proof of theorem 3.

Because these three algorithms would converge even if some processes ceased to make further vector updates at some particular point, then we know from work presented in [27] that the algorithms are all guaranteed to terminate in finite time.

7.2 Test Framework

In the following three subsections we consider the asynchronous algorithms as tested empirically on steady-state calculations based around a model of a flexible manufacturing system—the FMS model [44].

Details of the empirical testing framework are as follows. The algorithms are implemented in C++ using the MPI2 asynchronous communication framework. Algorithm comparison is made in terms of the number of updates, the number of messages sent, and wall time before algorithm termination. The results we present were produced across two Linux clusters. The *luna* cluster has 8 homogeneous dual-processor nodes with Infiniband interconnect running at 2Gbs^{-1}. The *vertex* cluster has 64 heterogeneous dual-core nodes with standard 1Gbs^{-1} ethernet backbone. A simple matrix–vector splitting, which accords with theorem 3, was used throughout the testing: given a transition matrix M, we set $\overline{v}$ to be equal to the first column of M, i.e. $\forall i(v_i := M_{i1})$ and we set K to be the first column, i.e. $K = \{1\}$. The initial vectors used in the testing were randomly generated, with the same initial vectors being used when comparing algorithms. We used the one-norm to determine both whether a message ought to be sent according to the three algorithms and also whether each update cycle had concluded according to the Asynchronous Reluctant and Asynchronous Variable Threshold algorithms. We set $\epsilon_1 = \epsilon_2 = 0.005$.

The flexible manufacturing system itself is parametrised by the number of unprocessed parts, k. In our tests, we considered values of k, $6 \leq k \leq 9$, which generate models with between $537,768$ and $11,058,185$ states. The state-space of these systems was partitioned over multiple processors (between 8 and 60 processors) and analysed for its steady-state vector in comparison with a parallel implementation of the Jacobi algorithm. We examined several configurations of this experimental setup to establish the differences in algorithm performance relative to the size of the system being analysed and the number of processors over which the problem is partitioned. Each test was repeated three times, and the results were averaged.

The default partitioning strategy which we used in our tests was the block distribution. We may describe this as follows. We define the block size as being the number of problem rows divided number of processes (+1 if the number of rows is not divisible by the number of processes). With this definition, we map problem rows to processes by:

$$\texttt{process} = \left\lceil \frac{\texttt{problem row}}{\texttt{block size}} \right\rceil$$

We also map problem rows to local rows (on particular processes) by:

$$\texttt{problem row} = \texttt{problem row} \bmod \texttt{block size}$$

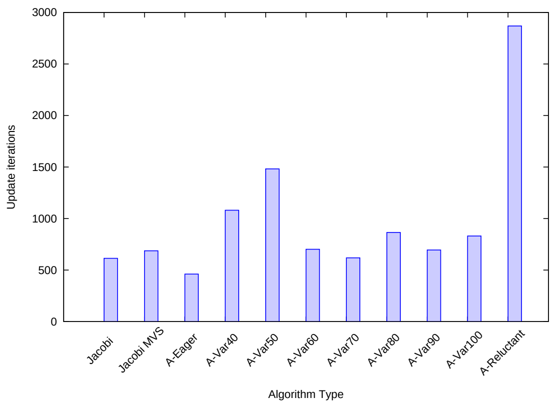
A second partitioning strategy is the row cyclic distribution. We may describe this as follows. We map problem rows to processes by:

$$\texttt{process} = \texttt{problem row} \bmod \texttt{number of processes}$$

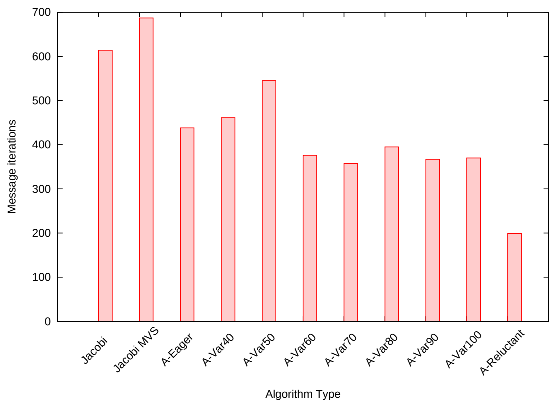
We map problem rows to local rows (on particular processes) by:

$$\texttt{local row} = \left\lceil \frac{\texttt{problem row}}{\texttt{number of processes}} \right\rceil$$

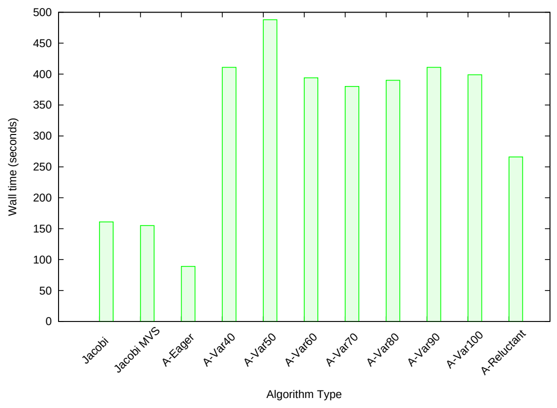
The scalability we look for in the experiments refers not only to the time taken to solve a given problem, although obviously important, but also to factors such as the number of local iterations performed, or the number of update messages sent between processors.



(a) Average number of update iterations per process



(b) Average number of message iterations per process



(c) Average wall time to convergence

Fig. 2. Execution properties of a variety of different synchronous and asynchronous algorithms

7.3 First Test: a Range of Algorithms

We considered a particular steady-state analysis problem defined by a square matrix of size (height) 1,639,440 and with 13,552,989 non-zeros. We fixed the distribution of the problem to be across 8 processors within the luna cluster. We then ran to convergence both the Asynchronous Eager and Asynchronous Reluctant solvers. In addition we ran several of the intermediate Asynchronous Variable Threshold algorithms, with threshold values ranging from 40 through 100. As our yardstick, we ran the regular parallel Jacobi solver and the Jacobi solver which employs pre-processing via matrix-vector splitting.

In Figure 2a, we show the average number of local update iterations at each processor until convergence where, during each update iteration, each local vector entry is updated once. In Figure 2b, we show the average number of message iterations until convergence where, during each message iteration, each process requiring information from a particular process is messaged once. Finally, we show the wall time until convergence in Figure 2c.

In the update plot (Figure 2a) we see that the best performing algorithm was Asynchronous Eager, which required 75% of the number of update iterations as compared with the standard Jacobi. This implies some form of Gauss–Seidel effect. The worst performing was Asynchronous Reluctant, which required just over four and a half times the number of update iterations as compared with the standard Jacobi. Interestingly, the preprocessed Jacobi required 11% more update iterations than standard Jacobi. Also, the Asynchronous Variable Threshold algorithms did not require steadily more update iterations as the threshold number increased. For example, when the threshold was set to 70 the algorithm required the fewest number of update iterations, and when the threshold was set to 50 the algorithm required the highest number of update iterations. This suggests that not all update iterations by the Asynchronous Variable Threshold algorithm, and also the Asynchronous Reluctant algorithm, are equally helpful in terms of ultimately converging to the solution vector. This is because if we cannot trivially relate threshold values within Asynchronous Variable Threshold algorithms to the number of update iterations required until convergence, then it must be that some update iterations by the Asynchronous Variable Threshold algorithm, and also the Asynchronous Reluctant algorithm, cause additional oscillation by update vectors about the solution vector, whilst other update iterations productively lead the algorithm toward convergence.

In the messaging plot (Figure 2b) we see that the best performing algorithm was Asynchronous Reluctant, which required 32% of the number of messaging iterations as compared with the standard Jacobi. Asynchronous Eager also outperformed the standard Jacobi, requiring 71% of the number of messaging iterations as compared with the standard Jacobi. Again, the Asynchronous Variable Threshold algorithms did not require steadily more iterations as the threshold number increased. There was a degree of oscillation, with lowest messaging number again being when the threshold was set to 70.

In the wall-time plot (Figure 2c), we see that convergence for Asynchronous Eager required 55% of the wall time relative to convergence for standard Jacobi. If we recall that the number of Asynchronous-Eager updates and messages were approximately 73% of the number of Jacobi updates and messages, then it would appear that about 60% of the improved time was down to a Gauss–Seidel effect and 40% of the improved time was down to reduced idle time and better use of the communication channel (less bursty messaging).

Asynchronous Reluctant performed less well than standard Jacobi in terms of wall time, requiring 1.6 times the wall time for convergence. If we recall the update and message plots, then we could reasonably argue that over networks where messaging is particularly expensive relative to local calculation, Asynchronous Reluctant would have outperformed all the other algorithms in term of wall time. Of course, this is not necessarily so. This is because, in a system with particularly expensive communication, it is possible that Asynchronous Reluctant would oscillate in a more extreme way about the solution vector, thereby markedly increasing the number of message iterations required for convergence.

Interestingly all the Asynchronous Variable Threshold algorithms performed noticeably less well than Asynchronous Reluctant in terms of wall time. Interestingly also, the preprocessed Jacobi algorithm performed better than the standard Jacobi in terms of wall time.

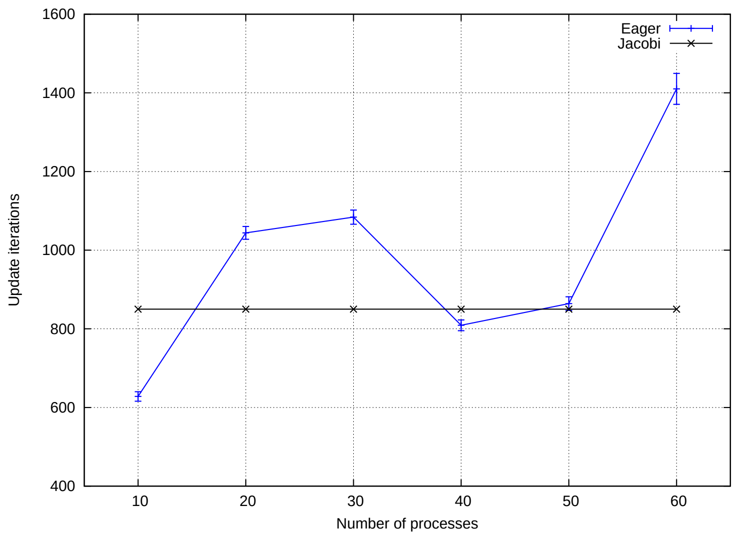
7.4 Second Test: Varying the Number of Processors

We considered a particular problem defined by a square matrix of size 4,459,455 and with 38,534,004 non-zeros. We then varied the distribution of this problem to be across 10, 20, 30, 40, 50 and 60 processors within the Vertex cluster. We found that by increasing the number of processors, the average number of local matrix non-zeros relative to remote matrix non-zeros on each process ($M'_{ij} \neq 0$ is deemed local if vector entries $x_i^{(k)}, x_j^{(k)}$ (for any k) are updated by the same process and remote otherwise) dropped as follows: 86.4%, 22.5%, 10.6%, 7.0%, 5.0%, 3.9%. With each distribution we ran to convergence both the Asynchronous Eager and Asynchronous Reluctant solvers. To compare, we ran the Jacobi solver which employs pre-processing via matrix-vector splitting.

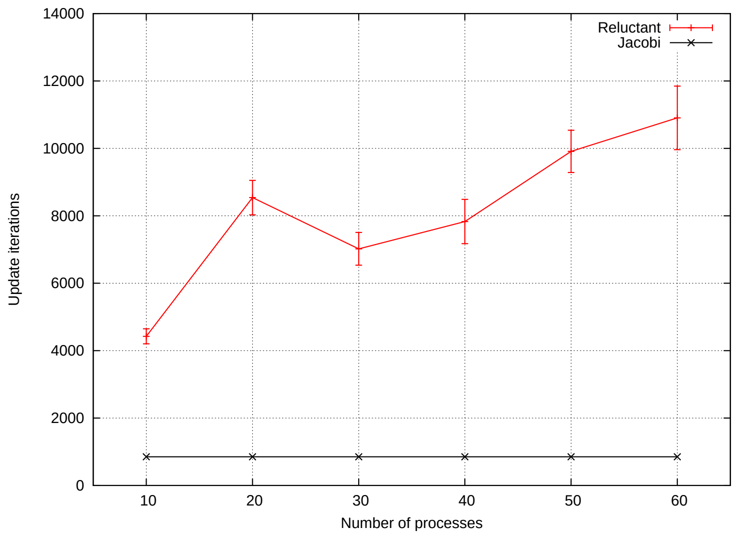
We show in the plots the average number of local update iterations at each processor until convergence (Figure 3) where, during each update iteration, each local vector entry is updated once; we show in Figure 4 the average number of message iterations until convergence where, during each message iteration, each process requiring information from a particular process is messaged once; and finally in Figure 5, we show the wall time until convergence. We also present the standard deviation (along with the associated coefficients of variation) of each setup as an average of the standard deviation over three separate executions. In the following figures, c_v(X) represents the coefficient of variation of the algorithm, X.

In the update plot (Figure 3) we see that Asynchronous Eager and Pre-processed Jacobi alternate in terms of being the best performing algorithm. Their relative performance in terms of update iterations ranges from Asynchronous Eager requiring 74% of the number of iterations as compared with Pre-processed Jacobi, when there are 10 processors, to Asynchronous Eager requiring 1.66 times the number of update iterations as compared with Pre-processed Jacobi when there are 60 processors. We attribute the outperformance by Asynchronous Eager of Pre-Processed Jacobi to a Gauss–Seidel effect.

The worst performing of the algorithms in terms of update iterations, as expected, was Asynchronous Reluctant. At best Asynchronous Reluctant required 5.21 times the number of update iterations as compared with Pre-Processed Jacobi—when there were 10 processors. At worst Asynchronous Reluctant required 12.83 times the number of update iterations as compared with Pre-Processed Jacobi—when there were 60 processors. With the exception of the 20 processor test, Asynchronous Reluctant performed progressively less well in terms of update number as the processor number increased.



(a) Eager Asynchronous v Jacobi: max. $c_v(\text{Eager}) = 2.8 \times 10^{-2}$



(b) Reluctant Asynchronous v Jacobi: max. $c_v(\text{Reluctant}) = 8.6 \times 10^{-2}$

Fig. 3. Number of update iterations per process as the number of processes varies, showing mean and ± 1 standard deviation for asynchronous algorithms

During the 20-processor test we noted that one of the processors was under heavy external load from a competing process. It is possible that the increased iteration number was partly due to this variation in processor load.

In the message plot (Figure 4) we see that Asynchronous Reluctant was always the best performing algorithm and Pre-Preprocessed Jacobi was always the worst performing algorithm. Both Asynchronous Eager and Reluctant show oscillating messaging performance. Both have lowest message number when there are ten processors: Reluctant requiring 30% and Eager requiring 68% of the number of messaging iterations relative to Pre-Processed Jacobi. Both have highest message number when there are twenty processors: Reluctant requiring 81% and Eager requiring 94% of the number of messaging iterations relative to Pre-Processed

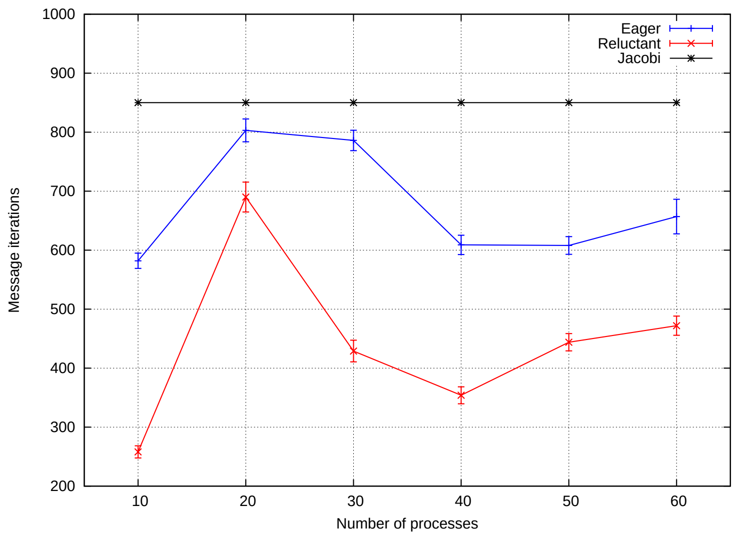


Fig. 4. Number of message iterations per process as the number of processes varies, showing mean and ±1 standard deviation for asynchronous algorithms. Max. c_v(Eager) $= 4.5 \times 10^{-2}$, max. c_v(Reluctant) $= 4.3 \times 10^{-2}$.

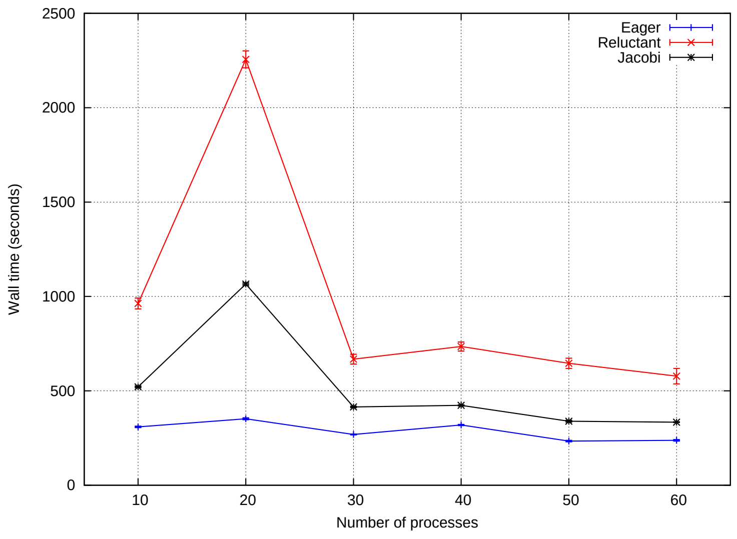


Fig. 5. Wall time in seconds as the number of processes varies, showing mean and ±1 standard deviation for all algorithms. Max. c_v(Eager) $= 1.7 \times 10^{-2}$, max. c_v(Reluctant) $= 4.2 \times 10^{-2}$, max. c_v(Jacobi) $= 1.5 \times 10^{-2}$.

Jacobi. Again, we note that one of the processors was under heavy load during the twenty-processor test. It is possible that this impacted on both asynchronous algorithms.

In the wall-time plot (Figure 5) we see that Asynchronous Eager was always the best performing, and Asynchronous Reluctant was always the worst performing. If we exclude the performance over 20 processors, Asynchronous Eager required on average 67% of the wall time to converge as compared with Pre-Processed Jacobi. Again, if we exclude the performance over 20 processors, Asynchronous Reluctant

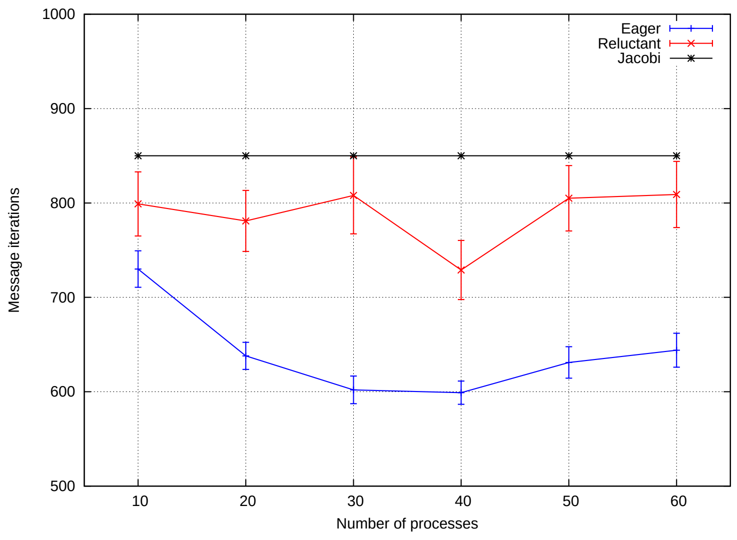


Fig. 6. Number of message iterations per process as the number of processes varies, using a *row-cyclic distribution* of entries across processes. Shows mean and ± 1 standard deviation for asynchronous algorithms. Max. $c_v(\text{Eager}) = 2.8 \times 10^{-2}$, max. $c_v(\text{Reluctant}) = 5.0 \times 10^{-2}$.

required on average 1.77% of the wall time to converge as compared with Pre-Processed Jacobi.

Over twenty processors, we note that both Pre-Processed Jacobi and Asynchronous Reluctant performed rather poorly—at least doubling the required wall time. We mentioned earlier that one of the processors was under heavy load during the test, and so it does not surprise that these two algorithms suffered, as their performance is limited by the worst performing processor. Interestingly, the work-load imbalance barely impacted on the wall-time performance of the Asynchronous Eager algorithm. This suggests that the updating work done by the other processes whilst waiting for information from the process under load was helpful in terms of ultimately reaching convergence. This, in turn, means that these updates probably did not lead to notable oscillation about the solution vector, despite their making use of old information from the process under load.

Excluding the twenty-processor test, wall time gradually decreased as processor number increased. Wall time with 60 processors relative to wall time over 10 processors was 77%, 60% and 64% for Asynchronous Eager, Asynchronous Reluctant and Pre-Processed, respectively.

As mentioned earlier, as the process number increases, the ratio of local to remote matrix non-zeros drops significantly. This results in more information being communicated between processes, so whilst update cost per iteration per process decreases as we distribute the particular problem across more processes, the messaging cost per iteration per process increases as we distribute the particular problem across more processes. It is for this reason that wall time does not halve, for example, when we double the number of processes employed to solve the particular problem.

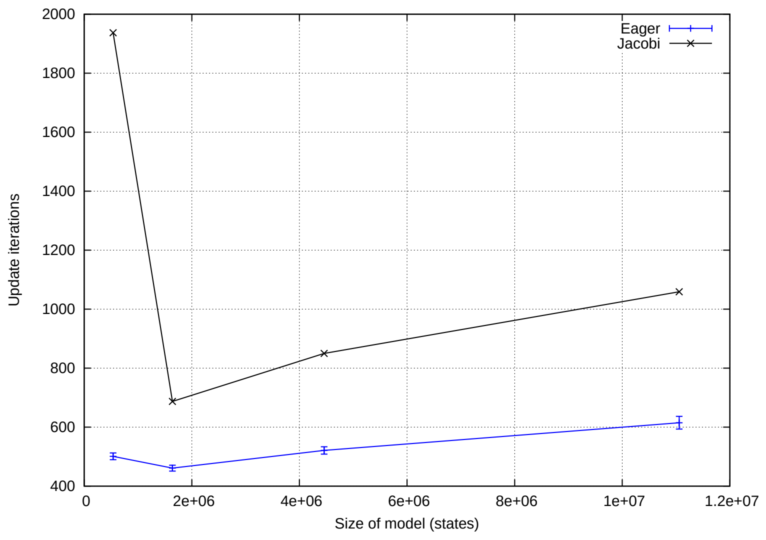
One may wonder what impact different distribution schemas might have when comparing the performance of the different algorithms. Alongside the block-distribution tests presented in this paper, we also ran analogue tests with a row-cyclic distribution. We do not show the results of these here, because somewhat surprisingly, the distribution, with the exception of one notable point, seemed to have little impact on the relative performance of the algorithms (in terms of update iterations, message iterations and wall time). We thought this surprising because the FMS model tends to generate problems with non-zeros congregating predominantly around the diagonal. This means that a row-cyclic distribution accords substantially less well with the structure of the underlying problem (e.g. with 10 processes, a row-cyclic distribution yields an average of 394,550 local matrix non-zeros and 3,458,850 remote matrix non-zeros). The exceptional point we refer to concerns the message iteration count for Asynchronous Reluctant. We show the plot as processor number varies with a row-cyclic distribution in Figure 6. In this plot we see that whilst still outperforming Pre-Processed Jacobi, Asynchronous Reluctant performed less well than Asynchronous Eager.

7.5 Third Test: Varying the Problem Size

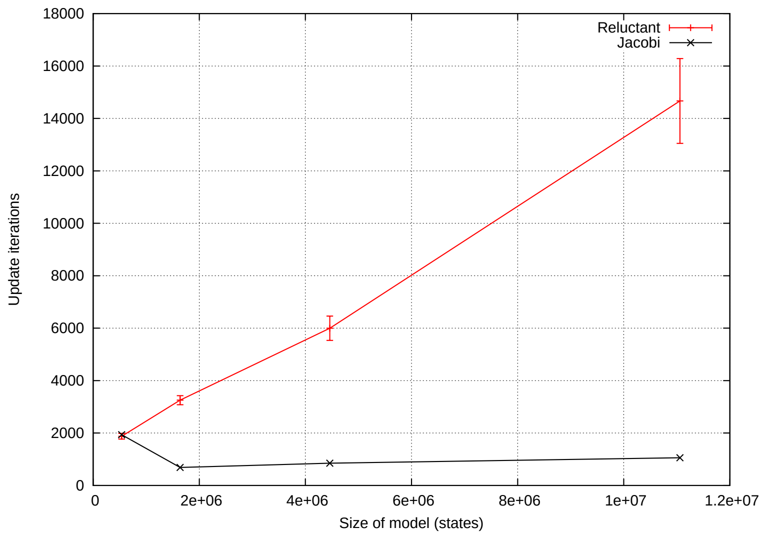
We considered four problems, defined by matrix of size 537,768, 1,639,440, 4,459,455, and 11,058,185, and with 4,205,720, 13,552,989, 38,534,004 and 89,986,704 matrix non-zeros, respectively. As with the first test, we distributed these across 8 processors within the luna cluster. The problem distributions yielded ratios of local matrix non-zeros relative to remote matrix non-zeros of just under 90%. To compare we ran the Jacobi solver which employs pre-processing via matrix-vector splitting.

We show in the plots the average number of local update iterations at each processor until convergence (Figures 7a and 7b) where, during each update iteration, each local vector entry is updated once; we show in Figure 8 the average number of message iterations until convergence where, during each message iteration, each process requiring information from a particular process is messaged once; and we show the wall time until convergence in Figures 9a and 9b. In each case we present results for Asynchronous Eager and Asynchronous Reluctant strategies. As before, we also present the standard deviation of each setup as an average of the standard deviation over three separate executions along with associated coefficients of variation.

In the update plots (Figures 7a and 7b) we see that the best performing algorithm was Asynchronous Eager. This was both in absolute terms and in terms of rate of growth as the problem size grows. This would seem to imply a more notable Gauss–Seidel effect as the problem size grows. By some distance, the worst performing was Asynchronous Reluctant, both in absolute terms and in terms of rate of growth. It is interesting to note that for the smallest problem, both the Asynchronous



(a) Eager Asynchronous v Jacobi: max. $c_v(\text{Eager}) = 3.5 \times 10^{-2}$



(b) Reluctant Asynchronous v Jacobi: max. $c_v(\text{Reluctant}) = 11.0 \times 10^{-2}$

Fig. 7. Number of update iterations per process as the model size (number of states) varies, showing mean and ± 1 standard deviation for asynchronous algorithms

Eager and Jacobi algorithms performed poorly relative to performance over larger problems—the Jacobi algorithm in particular.

In the message plot (Figure 8) we see that Asynchronous Reluctant was always the best performing algorithm in absolute terms and Pre-Preprocessed Jacobi was always the worst performing algorithm in absolute terms. In terms of rate of growth as the problem size grows, it would appear that both Asynchronous algorithms outperformed the Jacobi algorithm—the Eager more clearly than the Reluctant.

In the wall-time plots (Figures 9a and 9b) we see that Asynchronous Eager was always and definitively the best performing, in absolute terms and in terms of rate time increase as the problem size grows. Asynchronous Reluctant, by contrast, was

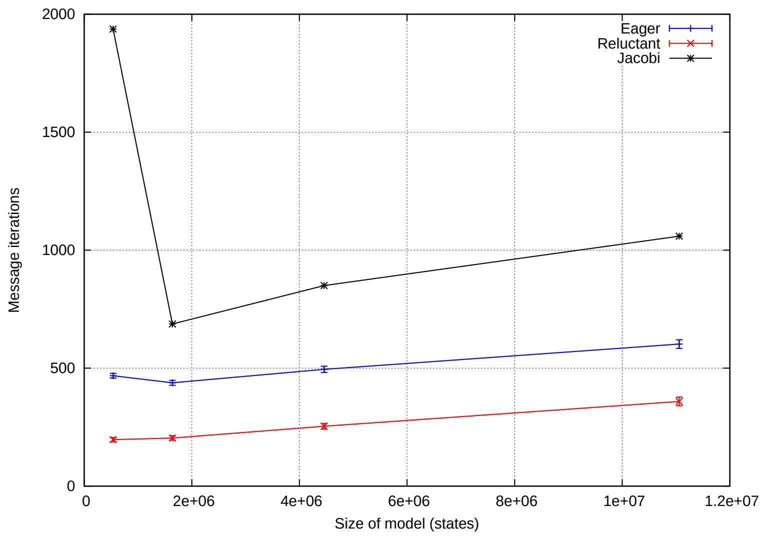


Fig. 8. Number of message iterations per process as the model size (number of states) varies, showing mean and ± 1 standard deviation for asynchronous algorithms. Max. $c_v(\text{Eager}) = 3.0 \times 10^{-2}$, max. $c_v(\text{Reluctant}) = 5.1 \times 10^{-2}$.

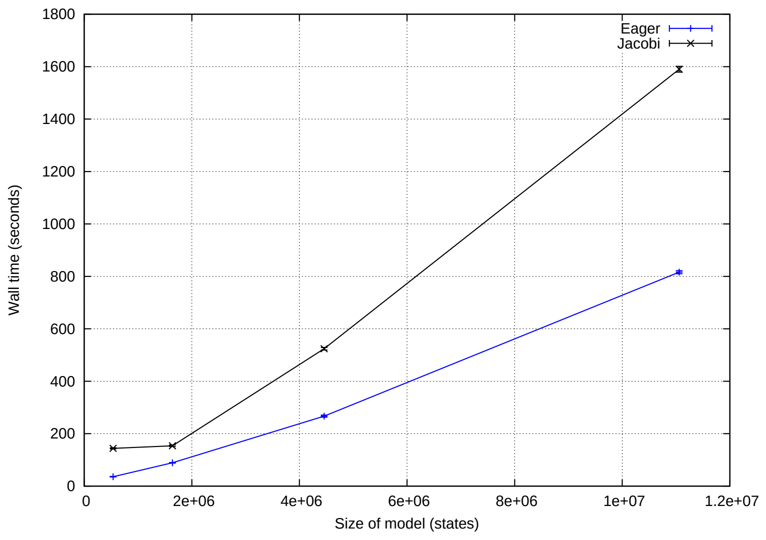
always the worst performing, in absolute terms and in terms of rate time increase as the problem size grows. The rate of time increase relative to increased problem size was approximately 1:1, 1.1:1 and 3:1 for Asynchronous Eager, Jacobi and Asynchronous Reluctant, respectively.

8 Conclusions

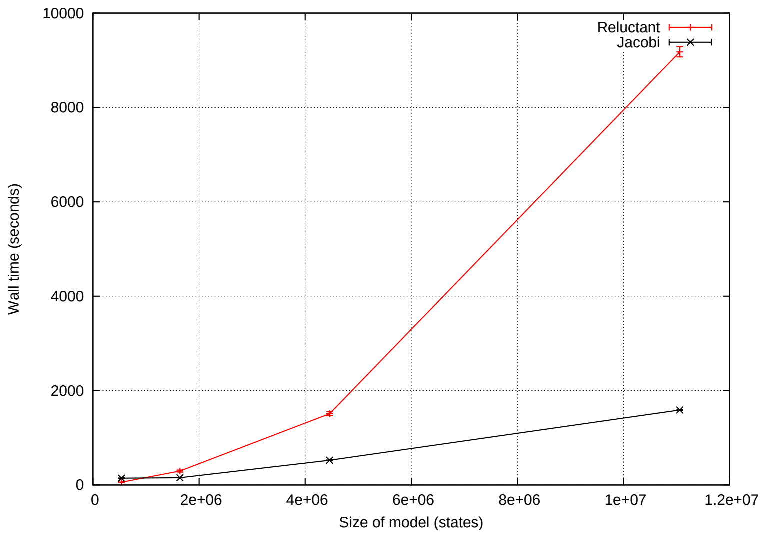
In this paper, we have shown how steady-state, transient and first passage-time calculations, for discrete-time Markov chains, continuous-time Markov chains and semi-Markov chains, can be expressed as fixed-point problems which can be solved using necessarily convergent totally asynchronous iterations.

This combination of results means that for the first time totally asynchronous iterative algorithms can be brought to bear on a wide range of performance problems. This is significant since it opens the way for massively distributed computation engines which can compute passage-time, transient and steady-state quantities over massive state space models and heterogeneous architectures. This potentially creates a single abstract computation framework for the distributed calculation of state-based performance metrics for the major types of analysis and models.

In this paper we introduced a family of totally asynchronous iterative solution algorithms which have been implemented in C++ with an MPI2 communication framework. Having applied these to the solution of a GSPN model of a Flexible Manufacturing System, we were able to compare their performance relative to that of a parallel Jacobi algorithm, in terms of wall time until convergence, and the associated calculation and communication overhead.



(a) Eager Asynchronous v Jacobi: max. c_v(Eager) $= 2.8 \times 10^{-2}$, max. c_v(Jacobi) $= 1.9 \times 10^{-2}$



(b) Reluctant Asynchronous v Jacobi: max. c_v(Reluctant) $= 7.5 \times 10^{-2}$, max. c_v(Jacobi) $= 1.9 \times 10^{-2}$

Fig. 9. Wall time (secs) as problem size (number of vertices) varies, showing mean and ± 1 standard deviation for all algorithms

We showed that for varying numbers of processors and different problem sizes, one of the asynchronous approaches, according to which messages are sent after most update iterations, consistently outperformed the Jacobi algorithm in terms of wall time until convergence, in some cases requiring half the wall time. This approach also consistently required significantly fewer messages to be sent before convergence. Interestingly, the approach also often required fewer update iterations, apparently benefiting from accelerated convergence through a form of Gauss–Seidel effect.

We showed also that another asynchronous approach, according to which messages are sent only after several update iterations have been completed, often sustained the lowest message transmission count (although, not always). This suggests that this algorithm could be used when messaging is substantially more expensive than local calculation, for example where a calculation is being carried out across clusters and the inter-cluster link is relatively slow. An exact strategy for algorithm selection will clearly depend on the architecture of the computational environment.

Finally, it was noted that when one of the processors was under heavy load (that is, an external load from a competing process), the convergence of the frequently messaging asynchronous algorithm was barely impacted upon, in contrast with the other algorithms (Jacobi, in particular). This suggests that the updating work done by the other processors whilst waiting for information from the processor under load, during execution of this asynchronous algorithm, was helpful in terms of ultimately reaching convergence. This, in turn, means that these updates probably did not lead to notable oscillation about the solution vector, despite their making use of old information from the processor under load. This suggests a particular suitability of this asynchronous algorithm to heterogeneous distributed frameworks.

The experiments we presented in this paper may be regarded as companion work to the theoretical work in [27], in which comparison is drawn between synchronous and asynchronous algorithms, in terms wall time until convergence. In future work, we aim to investigate solution of larger problems, distributed across many more processors, whilst also keeping a fine-grained record of the low-level run-time characteristics of the computation environment (CPU usage, memory, bandwidth, etc.). Our aim will be to investigate the extent to which, if we increase the problem size, we can maintain the efficiency of the different algorithms by increasing the number of processors. This will allow us to draw specific scalability/cost-optimality conclusions, with respect to the different asynchronous algorithms.

Acknowledgements

We would like to thank Helen Wilson for her invaluable help in proving Lemma 4. We would also like to acknowledge the very diligent efforts of our three anonymous referees, who helped us substantially improve the paper. Bradley is supported in part by the EPSRC on the Analysis of Massively Parallel Stochastic Systems (AMPS) project, EP/G011737/1. `http://aesop.doc.ic.ac.uk/projects/amps/`.

References

[1] D. Chazan, W. L. Miranker, Chaotic relaxation, Linear Algebra and Its Applications 2 (1969) 199–222.

[2] J. Bahi, J.-C. Miellou, K. Rhofir, Asynchronous multisplitting methods for nonlinear

fixed point problems, Numerical Algorithms 15 (1997) 315–345.

[3] D. Amitai, A. Averbuch, M. Israeli, S. Itzikowitz, Implicit–explicit parallel asynchronous solver of parabolic PDEs, SIAM Journal on Scientific Computing 19 (4) (1998) 1366–1404.

[4] X.-C. Tai, P. Tseng, Convergence rate analysis of an asynchronous space decomposition method for convex minimization, Tech. Rep. 120, University of Bergen (1998).

[5] D. El Baz, P. Spiteri, J.-C. Miellou, Distributed asynchronous iterative methods with order intervals for a class on nonlinear optimization problems, Journal of Parallel and Distributed Computing 38 (1996) 1–15.

[6] W. J. Stewart, Introduction to the Numerical Solution of Markov Chains, Princeton University Press, 1994.

[7] S. M. Ross, Stochastic processes, Wiley Series in Probability and Mathematical Statistics, John Wiley & Sons, 1982.

[8] B. Melamed, M. Yadin, Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes, Operations Research 32 (4) (1984) 926–944.

[9] M. Fischer, P. Kemper, Distributed numerical Markov chain analysis, in: Proceedings of the Euro PVM/MPI, no. 2131 in LNCS, Springer-Verlag, 2001, pp. 272–279.

[10] A. Frommer, D. B. Szyld, Asynchronous two-stage iterative methods, Numerische Mathematik 69 (1994) 141–153.

[11] V. Migallón, J. Penadés, D. B. Szyld, Block two-stage methods for singular systems and Markov chains, Numerical Linear Algebra with Applications 3 (1996) 413–426.

[12] D. B. Szyld, The mystery of asynchronous iterations convergence when the spectral radius is one, Tech. Rep. 98-102, Department of Mathematics, Temple University, Philadelphia, Pa. (October 1998).

[13] A. Frommer, D. B. Szyld, On asynchronous iterations, Journal of Computational and Applied Mathematics 123 (2000) 201–216.

[14] P. Buchholz, M. Fischer, P. Kemper, Distributed steady state analysis using Kronecker algebra, in: NSMC'99, Proceedings of the 3rd International Workshop on Numerical Solutions of Markov Chains, 1999, pp. 76–95.

[15] R. Pyke, Markov renewal processes with finitely many states, Annals of Mathematical Statistics 32 (4) (1961) 1243–1259.

[16] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt, H. J. Wilson, Hypergraph-based parallel computation of passage time densities in large semi-Markov models, Journal of Linear Algebra and Applications 386 (2004) 311–334.

[17] J. T. Bradley, N. J. Dingle, P. G. Harrison, W. J. Knottenbelt, Distributed computation of transient state distributions and passage time quantiles in large semi-Markov models, Future Generation Comp. Sys. 22 (7) (2006) 828–837.

[18] J. K. Muppala, K. S. Trivedi, Numerical transient analysis of finite Markovian queueing systems, in: U. N. Bhat, I. V. Basawa (Eds.), Queueing and Related Models, Oxford University Press, 1992, pp. 262–284.

[19] G. Bolch, S. Greiner, H. de Meer, K. S. Trivedi, Queueing Networks and Markov Chains, Wiley, 1998.

[20] J. Abate, G. L. Choudhury, W. Whitt, On the Laguerre method for numerically inverting Laplace transforms, INFORMS Journal on Computing 8 (4) (1996) 413–427.

[21] J. Abate, W. Whitt, The Fourier-series method for inverting transforms of probability distributions, Queueing Systems 10 (1) (1992) 5–88.

[22] D. P. Bertsekas, J. N. Tsitsiklis, Some aspects of parallel and distributed iterative algorithms – A survey, Automatica 27 (1) (1991) 3–21.

[23] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and distributed computation: Numerical methods, Prentice-Hall, 1989.

[24] J. N. Tsitsiklis, A comparison of Jacobi and Gauss–Seidel parallel iterations, Applied Mathematics Letters 2 (1989) 167–170.

[25] G. M. Baudet, Asynchronous iterative methods for multiprocessors, Journal of the ACM 25 (2) (1978) 226–244.

[26] D. V. de Jager, Asynchronous iterative solution for dominant eigenvectors with applications in performance modelling and PageRank, Ph.D. thesis, Imperial College London (May 2009).

[27] D. P. Bertsekas, J. N. Tsitsiklis, Convergence rate and termination of asynchronous iterative algorithms, in: ICS'89, Proceedings of the 3rd International Conference on Supercomputing, ACM, 1989, pp. 461–470.

[28] B. Lubachevsky, D. Mitra, A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius, Journal of the ACM 33 (1) (1986) 130–150.

[29] J. M. Bull, T. L. Freeman, Numerical performance of an asynchronous Jacobi iteration, in: Conference on Algorithms and Hardware for Parallel Processing, 1992, pp. 361–366.

[30] R. A. Horn, C. R. Johnson, Matrix analysis, CUP, New York, 1986.

[31] M. Pott, On the convergence of asynchronous iteration methods for nonlinear paracontractions and consistent linear systems, Linear Algebra and Its Applications 284 (1998) 1–33.

[32] J. Bahi, Algorithmes parallèles asynchrones pour des systèmes singuliers, Academie des Sciences Paris Comptes Rendus Serie Sciences Mathematiques 326 (1998) 1421–1425.

[33] A. Berman, R. J. Plemmons, Nonnegative Matrices in the Mathematical Sciences, 3rd Edition, SIAM, Philadelpha, 1994.

[34] R. S. Varga, Matrix Iterative Analysis, 2nd Edition, Springer-Verlag, 2000.

[35] R. Naulin, C. Pabst, The roots of a polynomial depend continuously on its coefficients, Revista Colombiana de Matemáticas 28 (1) (1994) 35–37.

[36] O. Axelsson, Iterative solution methods, Cambridge University Press, 1994.

[37] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the ACM/IEEE Supercomputing Conference, ACM/IEEE, 1995, p. 28.

[38] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed Computing 48 (1) (1998) 96–129.

[39] U. V. Catalyürek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Transactions on Parallel and Distributed Systems 10 (7) (1999) 673–693.

[40] A. Trifunovic, Parallel Algorithms for Hypergraph Partitioning, Ph.D. thesis, Imperial College London (February 2006).

[41] J. T. Bradley, H. J. Wilson, Iterative convergence of passage-time densities in semi-Markov performance models, Performance Evaluation 60 (1–4) (2005) 237–254.

[42] L. Uko, J. C. Orozco, Some p-norm convergence results for Jacobi and Gauss–Seidel iterations, Revista Colombiana de Matemáticas 38 (2) (2004) 65–71.

[43] D. El Baz, A. Frommer, P. Spiteri, Asynchronous iterations with flexible communication: Contracting operators, Journal of Computational and Applied Mathematics 176 (1) (2005) 91–103.

[44] G. Ciardo, K. S. Trivedi, A decomposition approach for stochastic reward net models, Performance Evaluation 18 (1) (1993) 37–59.