STUDIA INFORMATICA 2003
Volume 24 Number 4 (56)

Zdzisław SZCZERBIŃSKI
Polish Academy of Sciences, Institute for Theoretical and Applied Computer Science
Gliwice

# PARALLEL COMPUTING APPLIED TO SOLVING LARGE MARKOV CHAINS. A FEASIBILITY STUDY

**Summary.** The article is concerned with parallel computation issues arising in numerical solution of systems of linear equations which describe stationary probabilities of states in large Markov chains. Upon introduction to the subject of Markov chains and their solution, several adequate solution methods are surveyed, from the classical through projection to decompositional ones. Each algorithm is accompanied by a study of its suitability to parallel computing (multi- and vector processing). Additional opinions on aspects of the potential for parallelization in the discussed methods are contained in the conclusion.

**Keywords:** parallel computing, Markov chains, iterative methods, systems of linear equations

# OBLICZENIA RÓWNOLEGŁE W ZASTOSOWANIU DO ROZWIĄZYWANIA DUŻYCH ŁAŃCUCHÓW MARKOWA. STUDIUM WYKONALNOŚCI

**Streszczenie.** Artykuł jest poświęcony zagadnieniom obliczeń równoległych, występującym w trakcie numerycznego rozwiązywania układów równań liniowych, opisujących stacjonarne prawdopodobieństwa stanów w dużych łańcuchach Markowa. Po wprowadzeniu do tematyki łańcuchów Markowa, dokonano przeglądu wybranych metod rozwiązywania, począwszy od klasycznych, poprzez projekcyjne, do metod dekompozycyjnych. Dla każdego algorytmu została dokonana analiza, na ile nadaje się on do wykonania w trybie równoległym (wieloprocesorowym lub wektorowym). Dodatkowe uwagi dotyczące możliwości zrównoleglania dla omawianych metod zawarto w części końcowej.

**Słowa kluczowe:** obliczenia równoległe, łańcuchy Markowa, metody iteracyjne, układy równań liniowych

## 1. Introduction

We are concerned with Markov processes with discrete state spaces and continuous time (i.e. a state may change at any real-valued time instant). Such processes are called *continuous-time Markov chains*. We further limit our attention to *homogeneous* chains, i.e. Markov chains whose probabilities are stationary with respect to time.

Assume a continuous-time Markov chain, represented by a family of random variables $X(t)$, takes values from the set $\{x_1, x_2, \ldots, x_n\}$. If $X(t) = x_i$ then the chain is said to be in state $x_i$. For a homogeneous chain, the probability of transition from state $x_i$ to state $x_j$ in a very small time interval $\Delta t$ is linear:

$$p_{ij}(\Delta t) = q_{ij}\Delta t,$$

where $q_{ij}$ represents the transition rate between states $x_i$ and $x_j$. A homogeneous continuous-time Markov chain is represented by

- a set of states, and
- an *infinitesimal generator matrix* $Q = [q_{ij}]$ whose entries are the transition rates, except for the diagonal elements whose values are such that the following holds: $q_{ii} = -\sum_{j,j\neq i} q_{ij}$.

We are interested in computing the probabilities of the Markov chain being in respective states $x_1, x_2, \ldots, x_n$; once these values are known, it is analytically possible to gather vital characteristics of the system modelled by the chain.

It may be shown that the *stationary probability vector* $\pi$, a vector of length $n$ whose $k$-th element denotes the stationary probability of state $x_k$, can be obtained by solving the system of equations[1]

$$\pi^T Q = 0^T$$

The above system has an infinite number of solutions since the matrix $Q$ is singular. We are interested in finding the unique solution for which the sum of the probabilities (elements of vector $\pi$) is equal to 1. The above condition (*conservation of probability*) may be directly introduced into the system by replacing one equation (e.g. the last one) with $\pi^T e = 1$, where $e$ is a vector whose elements are all equal to 1. $Q$ is now non-singular and the system of equations takes the form

$$\pi^T Q = b^T, \tag{1}$$

[1] We assume columnwise orientation of vectors, e.g. $\pi$ is a column vector and $r^T$ is a row vector.

where $b$ is a vector whose all elements are equal to 0 except one element which is equal to 1. For such a system, the unique solution which satisfies conservation of probability may be analytically computed. Let us note that (1) may be written as

$$Q^T\pi = b,$$

which allows for the application of general numerical methods of solving linear systems of the form $Ax = b$.

Alternatively, the stationary probability vector $\pi$ can be obtained by using $P = [p_{ij}]$ rather than $Q$. The corresponding system of equations involving $\pi$ is then

$$\pi^T P = \pi^T \tag{2}$$

or, in the form usually required by numerical methods

$$P^T\pi = \mathrm{diag}(P^T),$$

where $\mathrm{diag}(P^T)$ is the vector formed of the diagonal elements of $P^T$.

The above problems appear simple from the mathematical standpoint in view of the extensive family of methods for solving systems of linear equations. However, the technical side of the solution process is far more complicated, mainly because of the large number of states which usually occur in real-world systems modelled by Markov chains. Besides, $Q$ (or $P$) is usually (very) sparse, which limits the effectiveness of classical methods developed for dense systems of equations, and necessitates employment of special techniques to deal with sparsity.

In the paper we show how the solution of the above and related problems can benefit from applying the techniques of *parallel computing*, by which we mean both *multiprocessing* (also known as *true parallel computing*) and *vector processing*. In section 2, a general introduction to the idea of solving linear systems is made. The next three sections describe various methods for obtaining solutions to such systems, with discussion on their potential for being parallelized or vectorized. Concluding remarks and directions for future research are contained in section 6.

## 2. Solution methods for linear systems

Solution methods for

$$Ax = b \tag{3}$$

are classified into two categories: *direct* and *iterative*. Direct methods give exact solutions and the number of operations they involve is fixed for a given system size. A common feature of these methods is the factorization of the matrix $A$. Classical methods, such as Gaussian elimination, are based on LU factorization [11]; other useful factorizations are QR [7] and WZ [9], developed especially for parallel computers. These techniques are effective for medium-sized and dense or narrowly banded systems. The obstacle to using them for large sparse systems is the presence of numerous non-zero entries in the factors, known as *fill-in*. There are methods for reordering matrices so as to reduce fill-in; however, they inevitably require additional computer memory which, for many problems, exceeds allowable limits.

Iterative methods generate a sequence of approximate solutions until a desired accuracy is reached. Their advantage over direct methods is thus that the system can be solved to a predetermined accuracy and so the number of operations required can be far less than that in direct solvers. Iterative methods are also preferred for other reasons. Usually, the only operation in which the matrix $A$ (or a matrix easily derived from $A$) is involved is multiplication by vector(s); such operations do not alter the matrix, which is important for large and sparse matrices because compact storage schemes may now be implemented. Additionally, since the matrix is never altered, there is no rounding error propagation which characterizes direct methods. From the point of view of parallel computing, the matrix-vector products inherent in practically every iterative solver are ideal for *vectorization*, which makes these methods well suited to implementation on parallel architectures. A major disadvantage of iterative methods is their frequent bad convergence (prohibitively long time is required to reach a solution with accepted accuracy) or even inconvergence, for an ill-conditioned matrix $A$. By contrast, in direct methods an upper bound on the time required to obtain the solution is easily determined before the actual calculation; besides, the solution is (theoretically) always accurate. Nevertheless, for a whole spectrum of applications including, among others, Markov chains, iterative solvers are more effective than direct solvers.

## 3. Classical iterative methods

The oldest classical methods converge linearly and rather slowly. They include the Jacobi and Gauss-Seidel algorithms, often accelerated by using a relaxation technique.

### 3.1. The Jacobi method

The matrix $A$ may be written as

$$A = D + L + U$$

where $D, L, U$ are, respectively, the diagonal, lower triangular and upper triangular parts of $A$. The system (3) has now the form

$$(D + L + U)x = b$$

or

$$x = D^{-1}b - D^{-1}(L + U)x$$

The above forms the basis for an iteration process with the iteration step expressed componentwise as

$$x_i^{(k+1)} = \frac{b_i}{a_{ii}} - \sum_{j=1, j\neq i}^{n} \frac{a_{ij}}{a_{ii}} x_j^{(k)}, i = 1, 2, \ldots, n, k = 0, 1, 2, \ldots$$

The initial vector is chosen arbitrarily, e.g. $x^{(0)} = 0$ or $x^{(0)} = b$. The process is terminated when an adopted criterion for solution accuracy is satisfied. The most common criteria here are:

- absolute error criterion:
  $$\max_{1\leqslant i\leqslant n}|x_i^{(k+1)} - x_i^{(k)}| < \delta \text{ for a given } \delta > 0,$$
- relative error criterion:
  $$\max_{1\leqslant i\leqslant n}|x_i^{(k+1)} - x_i^{(k)}| \leqslant \max_{1\leqslant i\leqslant n}|x_i^{(k+1)}|\epsilon \text{ for a given } \epsilon > 0,$$
- $k > k_0$, where $k_0$ is a preselected maximal number of iterations (often predicted by some estimate).

Unfortunately, for many cases where the system (3) does not satisfy *convergence criteria* (these are based on comparisons of certain expressions involving elements of $A$ and $b$), the Jacobi method is useless (it is sometimes possible to fulfill the criteria by forming linear combinations of the equations). Besides, even if convergence exists, it is usually very slow.

A common technique to increase the rate of convergence is to "relax" the Jacobi method. The iteration rule can be written in matrix form as

$$x^{(k+1)} = x^{(k)} - (D^{-1}Ax^{(k)} - D^{-1}b)$$

We can thus see $x^{(k+1)}$ as a change of $x^{(k)}$ towards the final solution:

$$x^{(k+1)} = x^{(k)} + z^{(k)},$$

where $z^{(k)} = D^{-1}(b - Ax^{(k)})$ is called the *correction vector*. In a *relaxation method*, $x^{(k)}$ is improved by using $\omega z^{(k)}$ instead of $z^{(k)}$, $\omega > 0$. The *relaxation coefficient* $\omega$ is chosen in such a way that the rate of convergence increases when compared with that of the Jacobi method. The iteration step now becomes

$$x^{(k+1)} = x^{(k)} + \omega z^{(k)} = x^{(k)} + \omega D^{-1}(b - Ax^{(k)}) \tag{4}$$

and is called *underrelaxation* if $0 < \omega < 1$ and *overrelaxation* if $\omega > 1$. If $\omega$ has the optimal value for a given linear system then convergence is significantly improved. Unfortunately, for wrong values of $\omega$, the opposite may be true. There is no theory for determining the optimal value of $\omega$ for arbitrary $A$; usually, such values are obtained from repetitive numerical experiments.

We note that the Jacobi method is inherently parallel: there is no data dependence between values $x_i^{(k+1)}$ in iteration step $k$, so they can be computed simultaneously. The distribution of computations among $p$ processors is rowwise i.e.

$$x_{1:m}^{(k+1)} = x_{1:m}^{(k)} + \omega z_{1:m}^{(k)} \qquad \text{(processor 1)}$$

$$x_{m+1:2m}^{(k+1)} = x_{m+1:2m}^{(k)} + \omega z_{m+1:2m}^{(k)} \qquad \text{(processor 2)}$$

$$\vdots$$

$$x_{(p-1)m+1:n}^{(k+1)} = x_{(p-1)m+1:n}^{(k)} + \omega z_{(p-1)m+1:n}^{(k)}, \qquad \text{(processor } p\text{)}$$

where $m = \lceil \frac{n}{p} \rceil$ and the subscript expression $i : j$ denotes indices $i, i+1, \ldots, j$; for vectors $x^{(k)}, x^{(k+1)}$ and $b$ (which is used in $z^{(k)}$) this subscript indicates the corresponding elements whereas for matrices $A$ and $D$ (used in $z^{(k)}$) the subscript denotes all columns in rows from $i$ to $j$ (since $D^{-1}$ is diagonal, the above refers to respective elements of the diagonal). Note that the whole vector $x^{(k)}$ is needed in $z^{(k)}$ for each processor. The parallel algorithm may be summarized as follows. Each processor updates the section of $x^{(k)}$ assigned to it, thus forming the corresponding section of $x^{(k+1)}$. In the process, it uses the corresponding set of rows of $A$ and sections of vector $b$ and the diagonal of $D^{-1}$. All these processes are overseen by a master process which tests for convergence.

Besides the rowwise distribution of the matrix $A$ and computations, there is another level of parallelism in the Jacobi algorithm. Note that $x^{(k+1)}$ in (4) is computed from

$x^{(k)}$ by a matrix-vector multiplication, a vector subtraction, a vector-vector scalar multiplication ($D^{-1}$ may be treated as a vector formed by the diagonal), a scalar-vector multiplication and a vector summation. All these operations are easily *vectorized* i.e. translated into instructions of a *vector computer*. Hence, the Jacobi method is well suited for vector architectures, including many "classical" supercomputers and minisupercomputers. Most of such systems feature machine implementation of some typical sequences of operations, including the so-called *linked triad*, arising in many numerical problems, where a vector is first multiplied by a scalar and then added to another vector. Note that a sequence of this type is formed by the last two operations in the above description of computations in the iteration step of Jacobi.

Special attention is required by the main computation in the whole sequence, the matrix-vector multiplication. Vectorization of this operation is of great importance since such multiplication is the most costly part of the execution time in each iteration. In order to vectorize the multiplication in an optimal way, one has to take into account the structure of the matrix $A$. As stated earlier, for Markov chains this matrix is usually very sparse. We shall first see how the matrix-vector multiplication is vectorized when the matrix is full and, basing on this, treat the problem for large and sparse matrices.

In computing a matrix-vector product, say $c = Ar$, on a vector computer, the *columnwise form* is preferred, following a well-established tradition of coding programs for such machines in Fortran. The resulting algorithm is:

```
do i = 1, n
  c(i) = 0.0
enddo
do k = 1, n
  do i = 1, n
    c(i) = c(i) + a(i,k)*r(k)
  enddo
enddo
```

The reason is that, for this "orientation" of algorithm, the elements of $A$ in the innermost loop (which is translated into a vector instruction) are in contiguous storage locations (this follows from Fortran coding), which is, by the nature of vector computer organization, required to provide an optimal data flow from memory to processor while executing the resulting vector instruction.

If the matrix is sparse, it could be theoretically treated as a full one. However, sparse matrices in real-world problems (e.g. Markov chain models of computer networks) are

usually *very large*, e.g. a Markov chain may have 10000 states which translates into the matrix $A = Q^T$ of 100 million elements, of which perhaps "only" 100000 are not zeroes. One should therefore try to store only the non-zero elements of $A$, and do it in such a way that the matrix-vector product is still easily vectorized. The matrix is usually stored either by rows or by columns, in a compressed form; without loss of generality we can assume that it is stored by rows. There are two possibilities to compress a row: to *pack* it by a *mask vector* or to *gather* it by an *index vector*, see e.g. [18] for details. The question of which method to use is open for discussion. Generally, the former one is preferred if the matrix is of less or medium sparsity while the latter if the matrix is very sparse. Further, there are two approaches to performing matrix-vector multiplication. In the first one, a row of $A$ is unpacked with the mask vector (or *scattered* with the index vector) and the scalar product is computed with this full decompressed vector. In the second approach, the vector $r$ is also compressed by the same vector (mask or index) as the row of $A$ and the scalar product is computed with two packed (gathered) vectors: the row of $A$ and $r$. Generally, the latter approach is more efficient and often preferred in practice.

### 3.2. The Gauss-Seidel and SOR methods

The *Gauss-Seidel* iterative method differs only slightly from the Jacobi method described earlier: the calculated values of $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are used when calculating the value $x_i^{(k+1)}$. In other words, the newly computed results in an iteration are used in *the same* iteration to compute the remaining results, whereas in the Jacobi method all results in an iteration are derived from results produced in *the previous* iteration. The iteration step now becomes

$$x_i^{(k+1)} = \frac{b_i}{a_{ii}} - \sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^{(k+1)} - \sum_{j=i+1}^{n} \frac{a_{ij}}{a_{ii}} x_j^{(k)}, i = 1, 2, \ldots, n, k = 0, 1, 2, \ldots \tag{5}$$

or, in matrix form

$$x^{(k+1)} = (D + L)^{-1}(b - Ux^{(k)})$$

As for the Jacobi method, there exist convergence criteria for this method; their detailed description is beyond the scope of this paper. Let us only note that the Gauss-Seidel method is guaranteed to converge if the matrix $A$ is positive definite. Also, similarly to Jacobi, it is possible to accelerate the Gauss-Seidel algorithm's convergence by "weighting" the previously and currently computed elements towards an average, i.e.

$$x_i^{(k+1)} = \omega \tilde{x}_i^{(k+1)} + (1 - \omega) x_i^{(k)}, i = 1, 2, \ldots, n; k = 0, 1, 2, \ldots$$

where $\tilde{x}_i^{(k+1)}$ is the (new) element computed according to the Gauss-Seidel procedure i.e. $\tilde{x}_i^{(k+1)}$ is identical with $x_i^{(k+1)}$ in (5). As before, $x_i^{(k)}$ is the element from the previous iteration. The above may be written in matrix form as

$$x^{(k+1)} = \omega(D + \omega L)^{-1}b - (D + \omega L)^{-1}[\omega U + (1 - \omega)D]x^{(k)}$$

It has been proved that convergence can be achieved only if $0 < \omega < 2$. Usually, $\omega$ is in the range $(1; 2)$, hence the above extrapolation technique is called *successive overrelaxation* (*SOR*). Note that for $\omega = 1$ the relaxation method becomes the Gauss-Seidel algorithm.

There are certain classes of matrices for which there is a simple formula for the optimal overrelaxation factor $\omega$. However, for general matrices there exist no analytical methods to find the optimal value of this parameter and implementations of SOR use heuristic approach.

By contrast with the Jacobi procedure, the Gauss-Seidel and SOR methods are not parallel in their form. The reason is that the new values of the elements of $x$ can only be calculated one after the other and not simultaneously since the calculations depend on one another: the computation of $x_i^{(k+1)}$ requires that all the elements $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ have already been computed. Hence, these methods are, generally, not suited for execution on parallel systems by rowwise partitioning as in the Jacobi case. There are modifications of SOR towards at least partly parallelization, as e.g. in [6] where an iteration is divided into two phases, of which the former is easily parallelizable; the latter phase, however, is sequential and time-consuming (solution of a lower triangular systems of equations by a direct method). In the case of large and sparse matrices $A$, a fortunate situation may occur where a multitude of zero entries in the matrix leads to some elements of the new iterate not necessarily dependent on previous elements. By reordering the equations it is sometimes possible to make updates to groups of elements in parallel [4].

Vectorization of the SOR method (and Gauss-Seidel as its special case) is far more difficult than in the Jacobi algorithm. There arises the technical problem of "non-smooth" data flow from memory to processor since the elements $a_{ij}$ in (5) are not in contiguous memory locations (as elements of a row of an array they are equidistant from their neighbours in the row, with distance $n$) and it is not easy to restructure the computations into a columnwise form which would cancel the problem. Additionally, for Markov chains, the usually irregular, large sparsity of $A$ introduces compression/decompression problems due to separation of the matrix-vector product into parts involving "old" and "new" values in the vector.

## 4. Projection methods

*Projection methods*, also known as *Krylov subspace techniques*, are a popular class of iterative solvers for large systems of linear equations. Their effectiveness has been tested on a wide range of science and engineering applications including fluid dynamics, atmospheric modelling, structural analysis and finite element analysis. They feature fairly good convergence, are competitive with classical iterative methods in terms of memory utilization, and are well suited to implementation on parallel computers.

### 4.1. The conjugate gradient method

By far the most widely known projection method is the *conjugate gradient* algorithm developed in the early 1950s by Hestenes and Stiefel [16] and used to solve linear systems where the matrix $A$ is symmetric positive definite. The method has since been generalized to allow for arbitrary matrices. Below we shall first present the general principle of the basic method and then describe its variants which are best suited to solving large Markov chains.

The simplest conjugate gradient algorithm [14] is based on the idea of minimizing the function

$$f(x) = \frac{1}{2}x^T Ax - x^T b,$$

which, for symmetric positive $A$ is minimized when the gradient

$$\nabla f = Ax - b = -r$$

is zero, which is equivalent to (3). The minimization consists in generating a succession of search directions $p^{(k)}$ (with $p^{(0)} = r^{(0)}$) and improved solutions $x^{(k)}$ according to the formulae:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} A p^{(k)}$$

$$p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}$$

The coefficients $\alpha^{(k)}$ and $\beta^{(k)}$ are chosen so as to minimize $f$ in $x^{(k+1)}$ over the whole subspace $(p^{(0)}, p^{(1)}, \ldots, p^{(k)})$ and to ensure that the direction vectors forming the above subspace are conjugate with respect to $A$ i.e. $p^{(k)T} A p^{(j)} = 0$ for $j < k$. It has been proved

in the literature that after at most $n$ steps the algorithm terminates with the solution found i.e. $x^{(k)}$ is the solution for some $k \leqslant n$. However, this can occur only with exact arithmetic. In practice, roundoff errors make the procedure iterative and the termination is allowed after some error criterion is met.

In our study, the large and sparse matrix $A$ is not necessarily positive definite or symmetric (in fact, it is symmetric in very rare cases). Therefore we turn our attention to generalizations of the classical conjugate gradient method for arbitrary linear systems of equations.

### 4.2. The biconjugate gradient method

The *biconjugate gradient* method was developed by Fletcher [10] from an algorithm for tridiagonalization of nonsymmetric matrices (due to Lanczos), applied to solve nonsymmetric systems of equations. This method uses two residual vectors rather than one as in the classical conjugate gradient algorithm; likewise, two direction vectors are formed instead of one. Denoted respectively by $r^{(k)}$, $\widehat{r}^{(k)}$, $p^{(k)}$ and $\widehat{p}^{(k)}$, these vectors satisfy the following conditions:

- biorthogonality — $r^{(i)T}\widehat{r}^{(j)} = \widehat{r}^{(i)T}r^{(j)} = 0,\ j < i$
- biconjugacy (with respect to $A$) — $p^{(i)}A^T\widehat{p}^{(j)} = \widehat{p}^{(i)}Ap^{(j)} = 0,\ j < i$
- mutual orthogonality — $r^{(i)T}\widehat{p}^{(j)} = \widehat{r}^{(i)T}p^{(j)} = 0,\ j < i$

The algorithm may be written as follows.

1. Choose an initial approximate solution $x^{(0)}$.
2. Compute the residual $r^{(0)} = b - Ax^{(0)}$.
3. Set $p^{(0)} = \widehat{p}^{(0)} = \widehat{r}^{(0)} = r^{(0)}$, and $k = 0$.
4. Perform iteratively the following sequence of computations

$$\alpha^{(k)} = \frac{\widehat{r}^{(k)T}r^{(k)}}{\widehat{p}^{(k)T}Ap^{(k)}}$$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)}p^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)}Ap^{(k)}$$

$$\widehat{r}^{(k+1)} = \widehat{r}^{(k)} - \alpha^{(k)}A^T\widehat{p}^{(k)}$$

$$\beta^{(k)} = \frac{\widehat{r}^{(k+1)T}r^{(k+1)}}{\widehat{r}^{(k)T}r^{(k)}}$$

$$p^{(k+1)} = r^{(k)} + \beta^{(k)}p^{(k)}$$

$$\widehat{p}^{(k+1)} = \widehat{r}^{(k)} + \beta^{(k)}\widehat{p}^{(k)}$$

increment $k$

As in the conjugate gradient method, the coefficients $\alpha^{(k)}$ are chosen to ensure the biorthogonality condition, and $\beta^{(k)}$ — the biconjugacy condition. There is a danger of the algorithm breaking down; this happens (fortunately, rarely) when one of the denominators is zero. Otherwise, the method converges after $m \leqslant n$ iterations (i.e. step 4 is performed at most $n$ times) with $r^{(m+1)} = \hat{r}^{(m+1)} = 0$ and $x^{(m+1)}$ being the solution. Again, the above convergence rule is theoretical only. In practical applications, the algorithm proceeds beyond $n$ iterations, until a convergence test determines that it should terminate, e.g. $r^{(k)T} r^{(k)} \leqslant \epsilon$ where $\epsilon$ is a tolerance criterion.

It may be seen from the above that the most costly computation in the biconjugate gradient procedure are the two matrix-vector multiplications. Of much interest to us is that they can be executed *simultaneously*, for example an a dual-processor system. For a large and sparse $A$ this will be done after decompressing $A$ and $A^T$. Note that *both* $A$ and $A^T$ will have been compressed earlier. It is sufficient to compress $A$ alone only if diagonal storing is possible i.e. non-zero diagonals of $A$, rather than rows or columns, are stored; however, no assumption as to diagonal dominance of the matrices derived from Markov chains can be made.

As regards vectorization, we are fortunate to have all the crucial operations (matrix-vector and vector-vector multiplications, and linked triads) fully vectorizable here. Thus, this method (incidentally, along with the classical conjugate gradient algorithm) is *ideally suited* for vector computers.

Besides being fairly effective in its own right, the biconjugate gradient method is significant because it led directly to the development of several techniques with faster convergence, the most noted of which are described in the following sections.

### 4.3. The conjugate gradient squared method

In the biconjugate gradient algorithm described in the preceding section, it can be shown that the residual $r^{(k)}$ is computed as the initial residual $r^{(0)}$ multiplied by a matrix polynomial of degree $k$. The *conjugate gradient squared* method was derived by Sonneveld [26] from the above algorithm by simply squaring the residual polynomial, which results in "contracting" the residual and reducing it faster than in the original method. The algorithm has the following form.

1. Choose an initial approximate solution $x^{(0)}$.
2. Compute the residual $r^{(0)} = b - Ax^{(0)}$.
3. Set $q^{(0)} = p^{(-1)} = 0$, $\rho^{(-1)} = 1$, and $k = 0$.
4. Perform iteratively the following sequence of computations

$$\rho^{(k)} = r^{(0)T} r^{(k)}$$

$$\beta^{(k)} = \frac{\rho^{(k)}}{\rho^{(k-1)}}$$

$$u^{(k)} = r^{(k)} + \beta^{(k)} q^{(k)}$$

$$p^{(k)} = u^{(k)} + \beta^{(k)}(q^{(k)} + \beta^{(k)} p^{(k-1)})$$

$$w^{(k)} = Ap^{(k)}$$

$$\sigma^{(k)} = r^{(0)T} w^{(k)}$$

$$\alpha^{(k)} = \frac{\rho^{(k)}}{\sigma^{(k)}}$$

$$q^{(k+1)} = u^{(k)} - \alpha^{(k)} w^{(k)}$$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)}(u^{(k)} + q^{(k+1)})$$

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} A(u^{(k)} + q^{(k+1)})$$

increment $k$

The conjugate gradient squared method converges whenever the biconjugate gradient method does i.e. both methods have the same break-down conditions; unfortunately, these conditions are not precisely known. The newer method requires more calculations (actually, twice the amount of computational work necessary for the basic conjugate gradient algorithm). However, convergence is now much faster. Another advantage is that the need to use the transpose of $A$ is now avoided. In terms of computation cost, multiplication with $A^T$ is replaced by additional matrix-vector multiplication involving $A$. Nevertheless, in view of the problems stemming from compression/decompression of $A^T$ (see the preceding section) it is truly beneficial to use this method in cases where $A$ is large and sparse.

### 4.4. The generalized minimum residual method

Another variant of the biconjugate gradient algorithm corresponds to a symmetric but not necessarily positive definite matrix $A$. In step 3, $\tilde{r}^{(0)}$ is set to $Ar^{(0)}$ rather than $r^{(0)}$. Further, for all $k$, $\tilde{r}^{(k)} = Ar^{(k)}$ and $\tilde{p}^{(k)} = Ap^{(k)}$. This approach is known as the *minimum residual* method since it successively minimizes the function

$$f(x) = \frac{1}{2} r^T r = \frac{1}{2} |Ax - b|^2$$

over the same set of search directions $p^{(k)}$ as the ones generated in the conjugate gradient method. The method is due to Paige and Saunders [21]. It has been generalized in various

ways for nonsymmetric matrices. The most robust of these variations is possibly the *generalized minimum residual* method developed by Saad and Schultz [24]. Without going to excessive detail, we can outline the algorithm as follows.

In iteration $k$, the approximate solution $x^{(k)}$ is calculated as

$$x^{(k)} = x^{(0)} + z^{(k)},$$

where $z^{(k)}$ (the "correction" vector) is chosen from the Krylov subspace spanned by vectors $r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}$ so that it minimizes the two-norm of the residual in this iteration

$$||r^{(k)}||_2 = ||b - A(x^{(0)} + z^{(k)})||_2 = ||r^{(0)} - Az^{(k)}||_2$$

In other words, determining the correction $z^{(k)}$ involves solving a $k$-dimensional least-squares problem.

The generalized minimum residual method is expensive in terms of memory usage because all consecutive vectors forming the Krylov subspace must be maintained throughout the algorithm, and their number increases with each iteration. On the other hand, the method is optimal in the sense that it provides the smallest residual in a fixed number of iteration steps. Its convergence is not necessarily strictly monotonic, though, i.e. $||r^{(k+1)}||_2 \leqslant ||r^{(k)}||_2$ rather than $||r^{(k+1)}||_2 < ||r^{(k)}||_2$. In practice, the method is usually *restarted* every $m$ iterations, with $x^{(m)} = x^{(0)} + z^{(m)}$ as the new "initial" approximation for the first restart, $x^{(2m)} = x^{(0)} + z^{(2m)}$ for the second and so forth; this constrains excessive storage. Unfortunately, the implementations done so far have confirmed that $m$ may not be reasonably large or else the resulting program runs out of space.

The generalized minimum residual method is amenable to both parallelization and vectorization. A recent example is shown in [22] where this method was implemented on the Convex C3840, a quad-processor vector minisupercomputer, and used to solve example, large Markov chains, achieving 20-fold speedup compared with the non-optimized, single processor version of the program for the same machine.

### 4.5. Other projection methods

Besides the ones described so far, there are several other methods stemming from the biconjugate gradient algorithm. A number of them are known and used more widely while some are only gaining popularity, albeit fairly slowly. The *biconjugate gradient squared stabilized* algorithm [15, 25] "smoothes" the uneven convergence of the conjugate gradient squared method by using two different polynomials to reduce $r^{(k)}$ instead of

applying one polynomial twice: the first one is the same as that used in the biconjugate gradient method and the second is derived from the generalized minimum residual method. This results in better convergence due to better local minimization. Another form of the biconjugate gradient method gave rise to the *conjugate residual squared* algorithm [19] which is in fact a modification of the conjugate gradient squared method, also leading to smoother reduction of the residual error. Likewise, the *generalized conjugate residual* method [8] can be used to solve linear systems with general matrices; this method, however, is costly in terms of storage. The *quasi-minimal residual* algorithm [13] is a derivative of the generalized minimum residual method which also takes a least squares approach to minimizing residuals while using the biorthogonal basis of the biconjugate gradient method, resulting in, generally, less storage at the cost of maintaining and using the transpose of $A$. A similar approach applied to the conjugate gradient squared method led to the *transpose-free quasi-minimal residual* algorithm [12], which, as the name suggests, has the advantage of not requiring multiplication with $A^T$. A good, hierarchical survey of projection methods may be found in [17].

From the point of view of parallel computing, the above methods retain good suitability to both parallelization and vectorization, which is characteristic of *all* algorithms related to conjugate gradients (see remarks in section 4.2). Moreover, the methods derived directly from the biconjugate gradient algorithm (i.e. performing operations on $A^T$) feature mutual independence of two computationally intensive matrix-vector multiplications in each iteration, thus augmenting the potential for parallel calculations.

### 4.6. Preconditioning

Generally, all projection methods work well for systems of linear equations whose matrices are well-conditioned. In order to reduce the condition number of the system's matrix, and thus improve the convergence rate, one may apply *preconditioning*, consisting in replacing the original equation (3) by the system

$$\tilde{A}^{-1}Ax = \tilde{A}^{-1}b,$$

where $\tilde{A}^{-1}$ approximates $A^{-1}$ and $\tilde{A}$ is chosen such that it is relatively inexpensive to compute $\tilde{A}^{-1}w$ for any vector $w$. The idea here is that, thanks to commutativity of matrix multiplication, $\tilde{A}^{-1}A \approx I$, which allows the algorithm to converge in fewer iterations. There is, obviously, additional computational cost inherent in preconditioning. For example, in the *preconditioned biconjugate gradient* method an additional set of vectors $z^{(k)}$

and $\tilde{z}^{(k)}$ is introduced, defined by (see section 4.2. for comparison)

$$\tilde{A}z^{(k)} = r^{(k)}, \quad \tilde{A}^T\tilde{z}^{(k)} = \tilde{r}^{(k)} \tag{6}$$

and the calculations are modified as follows

$$\alpha^{(k)} = \frac{\tilde{r}^{(k)T}z^{(k)}}{\tilde{p}^{(k)T}Ap^{(k)}}$$

$$\beta^{(k)} = \frac{\tilde{r}^{(k+1)T}z^{(k+1)}}{\tilde{r}^{(k)T}z^{(k)}}$$

$$p^{(k+1)} = z^{(k)} + \beta^{(k)}p^{(k)}$$

$$\tilde{p}^{(k+1)} = \tilde{z}^{(k)} + \beta^{(k)}\tilde{p}^{(k)}$$

The main problem here is that it is now necessary to solve the additional equations (6) in each iteration. However, since the matrix $\tilde{A}$ (called the *preconditioner*) is selected so as to simplify matrix-vector multiplications involving $\tilde{A}^{-1}$, it is relatively easy to obtain $z^{(k)} = \tilde{A}^{-1}r^{(k)}$ (the same holds for $\tilde{A}^T$, $\tilde{z}^{(k)}$ and $\tilde{r}^{(k)}$). Moreover, all the operations are fully vectorizable. A common technique is to use the diagonal part of $A$ as the preconditioner. This further simplifies the solution of (6) since $\tilde{A}^{-1}$ is now trivial and the whole procedure is reduced to multiplying corresponding elements of two vectors (the diagonal of $\tilde{A}^{-1}$ is stored as a vector), which is an elementary operation on a vector computer. More refined preconditioning techniques include the *incomplete LU factorization* (ILU) [20] and its variants ILUTH and ILUK [27]. They work fine when implemented on sequential computers but are not directly vectorizable since they involve classical LU factorization which requires back-propagation, basically a serial operation. However, with some effort, these techniques may be transformed into a scheme which vectorizes the computations. For details, see e.g. [19] where ILU is vectorized by using the Neumann series expansion of the preconditioner, or one of the papers describing the wavefront method for vectorization of the LU factorization process, e.g. [2].

## 5. Decompositional methods

The so-called *decompositional methods* of solving large Markov chains are based on the principle of *divide and conquer*, where a problem is partitioned into subproblems which are next solved and the solutions are eventually combined into a solution to the original

problem. The approach of divide and conquer is, in our case, as follows: the (long) Markov chain is divided into smaller subchains, and these subchains are solved individually. In order for such a technique to work, the subchains must be independent of one another. This is very rare in practical Markov chains. However, in several cases, the subchains may be considered *nearly* independent, with strong interactions among their internal states and weak interactions among the subchains themselves. This gave rise to the category of *nearly completely decomposable* systems, with first applications in economy and later extension to Markov chains analyzed in performance evaluation of computer systems [5]. The approach implies that the above character of interactions (strong internal vs weak external) enables an ordering of states which results in a *block structure* of the stochastic matrix of transition probabilities $P$:

$$P = \begin{pmatrix} P_{11} & P_{21} & \dots & P_{1m} \\ P_{21} & P_{22} & \dots & P_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ P_{m1} & P_{m2} & \dots & P_{mm} \end{pmatrix}$$

The blocks $P_{ii}$ are square and of order $n_i$, $i = 1, 2, \dots, m$, $\sum_{i=1}^{m} n_i = n$. In this form of $P$, the non-zero elements of the off-diagonal blocks are small (interactions between subchains) compared to those of the diagonal blocks (interactions within subchains). Hence, an approximation is made in which the system is treated as completely decomposable, i.e. $P_{ij} = [0]$ for $i \neq j$. The vector $\pi$ in (2) is then partitioned in conformance with $P$: $\pi = [\pi_1^T, \pi_2^T, \dots, \pi_m^T]^T$ where $\pi_i$ is a vector of length $n_i$. $\pi$ is obtained by solving

$$P_{ii}^T \pi_i = \mathrm{diag}(P_{ii}^T)$$

for $i = 1, 2, \dots, m$ and combining "subvectors" $\pi_i$. Pure concatenance is not possible because the elements of each subvector sum to one. In order to cause the elements of the whole vector $\pi$ to sum to one, the subvectors are weighted, with weights equalling the probabilities of the modelled system being in one of the states comprising corresponding subchains.

From the point of view of parallel computing, decompositional methods exhibit three issues where parallelization is possible.

1. The states of the Markov chain must be properly ordered so as to achieve the required block structure of $P$. This is best accomplished by treating the chain as a directed graph and applying graph search algorithms, several of which have been successfully parallelized, see e.g. [1, 23].

2. Solutions for blocks $P_{ii}$ are individual and independent of one another so it is natural that they can be obtained in parallel tasks, with each task performing computations on separate blocks and subvectors.
3. Within each of the tasks mentioned above, a linear system of equations is solved. There is thus possible further parallelization and/or vectorization, depending on the solution method chosen. We must, however, underline that the underlying matrix is now dense, and a solution method will be selected which is (probably) outside the categories described in this paper; in this case, a direct method rather than an iterative one will possibly be used.

## 6. Conclusion

We have discussed solution methods for large Markov chains as seen from the viewpoint of a researcher into parallel computation. There have been studied three distinct families of methods: classical, projection and decompositional. Each algorithm has been accompanied by a study of its suitability to parallel computing. It appears that there is no general rule as to which class of methods provides the greatest potential for parallel computing. Our opinion on these classes is the following.

- Classical methods must be treated individually. The Jacobi method, with its poor convergence rate, is a very good candidate for both parallelization and vectorization. It is a good point of reference for more advanced solution techniques. Clearly, if an algorithm reaches the solution in a longer time than the fully parallelized and vectorized Jacobi, then the algorithm is actually inefficient. Parallelization of the Gauss-Seidel / SOR group of methods is practically ineffective due to data dependences between computations for different equations in the linear system.
- Projection methods are generally well suited to implementation on parallel systems, especially vector computers. Their most important feature is that they are solely composed of simple operations on vectors, which are vectorizable by definition, and matrix-vector multiplications, which are both parallelizable and vectorizable. Moreover, the methods related to biconjugate gradients feature an additional level of parallelism since calculations for the two gradients may be performed simultaneously. We are planning further research into effective implementation of selected projection methods on a workstation cluster (parallelization) and a shared-memory multiprocessor vector (mini)supercomputer (parallelization and vectorization on the same computing platform).

- Decompositional methods, even in purely sequential form, are still more in the phase of experiments than sustained practice. Incidentally, we are aware of no practical parallel implementations of these methods. In the paper we have presented our own theoretical concept for parallelization. The subject is undoubtedly interesting and should be gratifying to potential researchers. Delving into it requires, however, deeper studies into, among others, graph theory and process scheduling.

Finally, let us make an interesting remark which came to mind while studying preconditioning in projection methods. As mentioned in the corresponding section, ILU and its variants, arguably the most popular preconditioning techniques today, are based on a direct solution of an auxiliary linear system. The solution can be, albeit not trivially, vectorized. This bridges the gap between direct and iterative methods; let us digress that the two classes are also brought together in the concept of *iterative refinement* of direct methods. It is therefore interesting to reconsider the possibility of applying direct methods to solving Markov chains. In section 2, prompted by disqualifying opinions from authoritative authors, we have distanced ourselves from this approach. Several researchers maintain, however, that direct methods need not necessarily be dismissed in solving large systems of equations. There is, for example, ongoing research into refinement of typical (e.g. Gauss-Jordan elimination) direct methods to very efficient parallel algorithms, with variants for large and sparse systems, see e.g. [3]. Techniques for solving large Markov chains could possibly benefit from applying this approach.

## REFERENCES

1. Akl S. G.: Parallel Computations: Models and Methods. Prentice Hall, New Jersey 1997.
2. Amestoy P. R., Duff I. S.: Vectorization of a multiprocessor multifrontal code. International Journal of Supercomputer Applications, 1989, Vol. 3, pp. 41-59.
3. Amestoy P. R., Duff I. S., L'Excellent J. Y.: Multifrontal solvers within the PARASOL environment. In: Kagstrom B., Dongarra J., Elmroth E., Waśniewski J. (Eds.): Applied Parallel Computing PARA'98. Springer Verlag, Berlin 1998, pp. 7-11.
4. Barrett R., Berry M., Chan T., Demmel J., Dorato J., Dongarra J., Eijkhout V., Pozo R., Romine C., Van der Vogt H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia 1994.

5. Courtois P. J.: Decomposability: Queueing and Computer System Applications. Academic Press, Orlando 1977.
6. Da Cunha R. D., Hopkins T.: Parallel overrelaxation algorithms for systems of linear equations. In: Welch P. et al (Eds.): Transputing '91. IOS Press, 1991, pp. 159-169.
7. Dryja M., Jankowska J., Jankowski M.: Survey of Numerical Methods and Algorithms. Part 2. WNT, Warsaw 1982.
8. Eisenstat S., Elman H., Schultz M.: Variational iterative methods for nonsymmetric systems of linear equations. SIAM Journal on Numerical Analysis, 1983, Vol. 20, pp. 345-357.
9. Evans D.: Parallel algorithms in computational linear algebra. In: Van Leeuven J., Lenstra J. (Eds.): Parallel Computers and Computations. Centrum voor Wiskunde en Informatica 1985.
10. Fletcher R.: Conjugate Gradient Methods for Indefinite Systems. Springer Verlag, Berlin 1976.
11. Fortuna Z., Macukow B., Wąsowski J.: Numerical Methods 4th ed. WNT, Warsaw 1998.
12. Freund R. W.: A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. SIAM Journal on Scientific and Statistical Computing, 1996, Vol. 14, pp. 470-482.
13. Freund R. W., Nachtigal N. M.: QMR: a quasi-minimal residual method for non-Hermitian linear systems. Numerische Mathematik, 1991, Vol. 60, pp. 315-339.
14. Golub G. H., Van Loan C. F.: Matrix Computations. The Johns Hopkins University Press, Baltimore 1983.
15. Gutknecht M, H.: Changing the norm in conjugate gradient type algorithms. SIAM Journal on Numerical Analysis, 1993, Vol. 30, pp. 40-56.
16. Hestenes M. R., Stiefel E.: Methods of conjugate gradients for solving linear systems. J. Research Natl. Bur. Standards, 1952, Vol. 49, pp. 409-436.
17. Knottenbelt W. J.: Parallel Performance Analysis of Large Markov Chains. Ph. D. Thesis, University of London, Imperial College of Science, Technology and Medicine 1999.
18. Kozielski S., Szczerbiński Z.: Parallel Computers: Architecture, Elements of Programming 2nd ed. WNT, Warsaw 1994.

19. Ma S., Chronopoulos A. T.: Implementation of iterative methods for large sparse nonsymmetric linear systems on parallel vector computers. International Journal on Supercomputing, 1990, Vol. 4, pp. 9-24.
20. Meijerrink J., Van der Vorst H.: An iterative solution method for linear systems for which the coefficient matrix is a symmetric m-matrix. Mathematical Computation, 1977, Vol. 31, pp. 148-162.
21. Paige C. C., Saunders M. A.: Solution of sparse indefinite systems of linear equations. SIAM Journal on Numerical Analysis, 1975, Vol. 12, pp. 617-624.
22. Pecka P.: An Object-Oriented Multithreaded System for Modelling Transient States in a Computer Network with Markov Chains. Ph. D. Thesis, Polish Academy of Sciences, Institute for Theoretical and Applied Computer Science, Gliwice 2002 (in Polish).
23. Quinn M. J., Deo N.: Parallel graph algorithms. Computing Surveys, 1984, Vol. 16, pp. 319-348.
24. Saad Y., Schultz M. H.: GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 1986, Vol. 7, pp. 856-869.
25. Sleijpen G. L., Fokkema D. R.: BiCGSTAB(L) for linear equations involving unsymmetric matrices with complex spectrum. Electronic Transactions on Numerical Analysis, 1983, Vol. 1, pp. 11-32.
26. Sonneveld P.: CGS, a fast Lanczos-type solver for nonsymmetric systems. SIAM Journal on Scientific and Statistical Computing, 1989, Vol. 10, pp. 36-52.
27. Stewart W. J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton 1994.

Recenzent: Dr inż. Ewa Starzewska-Karwan

Wpłynęło do Redakcji 21 listopada 2002 r.

## Omówienie

W artykule przeprowadzono analizę możliwości zastosowania obliczeń równoległych do efektywnego rozwiązywania dużych łańcuchów Markowa, rozumianego jako rozwiązywanie układów równań liniowych (1) opisujących stacjonarne prawdopodobieństwa sta-

nów takich łańcuchów. Dokonano przeglądu znanych metod iteracyjnych, stosowanych do rozwiązywania takich układów; każdej metodzie towarzyszy analiza możliwości zrównoleglenia, prowadzącego do wykonania adekwatnego programu w trybie wieloprocesorowym lub wektorowym. W pierwszej części omówiono klasyczne metody iteracyjne: Jacobiego (słaba zbieżność metody, duże możliwości zrównoleglenia) i Gaussa-Seidela (lepsza zbieżność, zrównoleglenie praktycznie niemożliwe). W kolejnej części przedstawiono metody projekcyjne, wykorzystujące podprzestrzenie Kryłowa. Dla algorytmów wywodzących się z metody gradientów sprzężonych wykazano duże możliwości wektoryzacji (kluczowe operacje obliczeniowe to działania na wektorach) oraz dodatkowo, w metodzie gradientów dwusprzężonych - zrównoleglenia na dwa procesory. W dalszej części przeglądu omówiono metody dekompozycyjne, polegające na podziale długiego łańcucha Markowa na podłańcuchy w celu ich oddzielnego rozwiązania a następnie scalenia wyników, co prowadzi do uzyskania rozwiązania dla pierwotnego łańcucha. Wyróżniono trzy aspekty możliwego zrównoleglenia: w algorytmach przeszukiwania grafów, w dekompozycji układu równań opisujących długi łańcuch oraz w trakcie rozwiązywania poszczególnych (po dekompozycji) mniejszych układów równań. W końcowej części artykułu zawarto dodatkowe uwagi na temat zrównoleglania omówionych metod oraz wskazano możliwości dalszych badań.

Adres

Zdzisław SZCZERBIŃSKI: Instytut Informatyki Teoretycznej i Stosowanej PAN, ul. Bałtycka 5, 44-100 Gliwice, Polska, zdzich@iitis.gliwice.pl