UPPSALA
UNIVERSITET

Parallel QR-factorisation for Power System State Estimation

Fredrik Nysjö
Per Uddholm

Report in Scientific Computing Advanced Course

May 2008

PROJEKTRAPPORT

Institutionen för informationsteknologi

Abstract

Parallelisation of power system state estimation (PSSE), implemented through the weighted least squares (WLS) method, is considered. The system state variables are estimated from the measurement equations by means of QR-factorisation through Givens rotations. The investigated effects include those of column and row re-numbering, those of different data types, and those of replacing classical Givens rotations with fast Givens rotations.

While our original implementations are too slow to be of any practical use in real-scale problems, the application of the approximate minimum degree (AMD) ordering algorithm and parallelisation yields quite competitive execution times.

Keywords: Power system state estimation, QR-factorization, Givens rotations, parallelism, OpenMP

1 Introduction

Power system state estimation (PSSE) is an important part of power grid monitoring systems, and may be implemented as weighted least squares (WLS) state estimation. Such PSSE implementations thus estimate the system state variables from a set of measurements with known (Gaussian) error probability density distributions (pdf). These equations will henceforth be referred to as the measurement equations. Similarly, the least squares methods, including the WLS method, lead to systems of equations, known as the normal equations.

The measurement equations consist of one equation per measurement. In order to emphasise precise measurements, each equation is multiplied by the inverse standard deviation of the error of that particular measurement. While this increases the relative weight of the more reliable measurements, it also deteriorates the conditioning of the normal equations. This is particularly important when the measurement error may be considered to be zero, e.g. when power flows or injections may be considered to be known exactly. A simple example is an unloaded transmission line, where the power flow must be zero.

In many important cases the normal equations obtained by PSSE are poorly conditioned. However, direct QR-factorisation of the measurement equations yields a matrix equation involving only the Cholesky factor of the matrix in the normal equations. The condition number is thereby reduced to the square root of the original condition number. The QR-factorisation may be performed through Givens rotations, Householder reflections, and the classical and modified Gram-Schmidt orthogonalisation methods.

The classical Gram-Schmidt orthogonalisation method is well known to be numerically unstable (Demmel [1] p. 108, Golub and Van Loan [2] p. 231) and thus inefficient, as it requires frequent re-orthogonalisation. The modified Gram-Schmidt orthogonalisation method, represents an improvement to its classical counterpart, but may still yield poorly orthogonalised Q-matrices. For instance, Golub and Van Loan [2] (p. 232) point out that the deviation from orthogonality of the modified Gram-Schmidt orthogonalisation method is a factor

$|\kappa|$ larger than for standard Householder reflections, where $|\kappa| \geq 1$ is the matrix condition number. Those methods are thus unsuitable for poorly conditioned systems ($\kappa >> 1$), and will accordingly not be considered further.

QR-factorisations of large sparse matrices, by means of Householder reflections, are considered in some depth by Matstoms [3].

Standard Givens rotations are notoriously slow and about half as efficient as Householder reflections (Demmel [1] p. 123) for full matrices. A very attractive feature, however, is their numerical robustness. Furthermore, improved (fast Givens) algorithms exist (e.g. Appendix B in Monticelli [4]), where the number of multiplications has been reduced from four to two and all (or almost all) square root calculations have been suppressed. Thus Vempati et al. [5] demonstrate that the QR-factorisation approach, based on fast Givens rotations, may achieve performance comparable to the normal equations method. An additional advantage of Givens rotation is their feature of sequentially adding/removing measurements without re-orthogonalisation, which is very useful for bad-data analysis.

There also exist fast Householder reflections (Diniz [6] p. 351 and refs. [27] through [29] therein). While their applicability certainly is worth studying, such an undertaking would fall beyond the scope of this report.

Vempati et al. [5] also discuss row-oriented (see also Soman et al. [7] p. 90ff) and column-oriented ([7] p. 92ff) approaches, and conclude that row-oriented elimination is more efficient. However, they do not present any data in direct support of that conclusion. Instead, it appears to be based on the argument, that by row-wise elimination, repetitive fetching and storing of partially processed rows can be avoided. Moreover, by storing partially processed rows, intermediate fill-ins (see Section 4) must be accommodated. This requires continual allocation and deallocation of memory for those fill-ins. Yet, some support may be drawn from their Table V, which indicates that their row-oriented scheme is some 40 percent faster than the normal equations method. While this Table says nothing explicitly about the column-oriented approach, the mere fact that performance comparable to the normal equations method could be achieved by Givens rotations, represents a major improvement. They also compare different row- and column-ordering schemes, and identify row-ordering by the minimum degree criterion and column-ordering according to maximum column number as the most efficient ordering scheme (in cpu time). No improvement is obtained by including average path length minimisation row-ordering as a tie-breaker in the row-ordering scheme. Schemes using the row count (i.e. the number of non-vanishing elements on the row) or the sum of the column numbers as tie-breaks, yield essentially indistinguishable improvements.

In contrast, Pandian et al. [8], conclude that a row-oriented approach may be faster. These authors utilise the VPAIR (Variable pivot strategy for sequencing of row PAIRs) row-ordering method and the minimum degree algorithm (MDA) for the column permutation.

The matrices which appear in PSSE are extremely sparse, so the data structure is expected to have a major effect on performance. This project thus aims to develop sparse QR-factorisation routines for several data structures from scratch, compare the efficiency of those routines, and analyse the results. In addition to the comparison of the different data structures, different processing methods will be compared, i.e. row- and column-oriented schemes.

The implementation is in C with OpenMP.

This report is organised as follows. Section 2 presents the fundamentals of PSSE, with particular reference to the application of QR-factorisation. The following Section discusses the standard and fast Givens rotations. Issues related to sparsity preservation are addressed in Section 4. In Section 5 row- and column-oriented elimination schemes are considered. The two data-structures investigated in the report are presented in Section 6. The parallelisation is discussed in Section 7. The results are listed in Section 8 and a discussion is provided in Section 9.

2 Fundamentals of power system state estimation

2.1 Power flow problems

In a power system the state variables are the voltage amplitudes E_i and phase angles $\theta_i = \arg(E_i)$, where E_i is the voltage at bus i. In order to estimate the power system state from a set of power measurements, the state variables must be related to the power flow. Thus a few comments on the power flow problems are in order before discussing PSSE.

In a linear grid currents and voltages are related through the equation

$$I = YE, \tag{1}$$

where I is the vector of currents I_i entering bus i, E is the vector of voltages E_i at bus i, and Y is the network matrix. However, problems involving the determination of the system state from given power fluxes are quite non-linear, as the apparent power S_i entering node i is essentially the product of voltage and current

$$S_i \equiv P_i + jQ_i = E_i \sum_k Y^*_{i,k} E^*_k, \tag{2}$$

where P_i is the real power, Q_i is the real power, $j = \sqrt{-1}$ is the imaginary unit, and the asterisk denotes complex conjugation. This kind of problem is thus quadratically non-linear in the voltage amplitudes and trigonometrically non-linear in the phase angles. This conclusion may be clarified by separating real and complex parts in $Y_{i,k} = G_{i,k} + jB_{i,k}$ to obtain

$$P_i = \sum_k |E_i||E_k| \left[G_{i,k}\cos(\theta_i - \theta_k) + B_{i,k}\sin(\theta_i - \theta_k)\right]$$

$$Q_i = \sum_k |E_i||E_k| \left[G_{i,k}\sin(\theta_i - \theta_k) - B_{i,k}\cos(\theta_i - \theta_k)\right]. \tag{3}$$

No general analytical solution to these equations exists. In order to solve the full power flow equations, iterative methods like Gauss-Seidel or Newton-Raphson usually have to be employed.

A number of simplifying approximations may be made, leading to methods such as the decoupled power flow and the “DC” power flow. These methods are discussed by e.g. Wood and Wollenberg [9] (Chapter 4), and will not be dwelled upon here. A point of some importance, however, is that the linearised equations used in the Newton-Raphson method, depend on the current approximation of the state, so the Jacobian matrix must be continually recalculated. A similar situation occurs in power state estimation. PSSE is accordingly an iterative procedure, where the updated power system state is used to update the full AC system (3), or the simplified equations (e.g. the “DC” power flow equations), until the solution has converged. This report is not concerned with this iteration, but restricts itself to studying the performance of the QR-factorisation of given measurement matrices H (below). The full iterative procedure is listed by Vempati et al. [5].

2.2 Power system state estimation

The non-linear equations, for the power flow through the power meters, may be formally cast in the form

$$f_i(x^*) = z_i, \tag{4}$$

where $f_i(x)$ is the theoretical value of the power flux through meter i for the power system state x, x^* is the true state, and z_i is the reading on meter i. This may be linearised about the current power system state estimate x. The result is the system (the measurement equations)

$$H\Delta x = \Delta z, \tag{5}$$

where H is the measurement matrix, $\Delta x = x^* - x$ is the correction to the current state x, and Δz is the measurement vector. The latter is obtained as the difference between the actual measurement and the power flux corresponding to the current state x, i.e. $\Delta z_i = z_i - f_i(x)$.

Since the quality of the instrumentation varies, measurement errors are assumed to be random and follow the Gaussian distribution $N(0, R_i)$. In order to weight measurements according to reliability, the ith equation in (5) is multiplied by the reciprocal to the standard deviation for the corresponding meter, i.e. $R_i^{-1/2}$

$$H_1\Delta x = \Delta z_1, \tag{6}$$

where $H_1 = R^{-1/2}H$ and $\Delta z_1 = R^{-1/2}\Delta z$, and $R^{-1/2}$ is the diagonal matrix with element $R_i^{-1/2}$ at position (i, i).

The WLS algorithm minimises the function

$$g(\Delta x) = (H_1\Delta x - \Delta z_1)^T(H_1\Delta x - \Delta z_1), \tag{7}$$

with the solution given by the normal equations

$$H_1^T H_1 \Delta x = H_1^T \Delta z_1. \tag{8}$$

A more careful derivation of this result is provided by Wood and Wollenberg [9] (Chapter 12.3), employing the maximum likelihood principle.

Although the above discussion may give the impression that only power measurements occur, measurements of other quantities, e.g. voltage, can also be included.

2.3 Estimation through QR-factorisation

In certain cases the normal equations are ill-conditioned. Then the fact that the condition number of the Cholesky factor L

$$H_1^T H_1 = L^T L, \tag{9}$$

where L is upper triangular, is much smaller than that of the normal equations matrix $H_1^T H_1$, i.e.

$$\kappa(L) = \sqrt{\kappa(H_1^T H_1)} << \kappa(H_1^T H_1), \tag{10}$$

can be used. The important point is now that the the renormalised measurement matrix H_1 may be QR-decomposed as

$$H_1 = QU = Q \begin{bmatrix} L \\ 0_{(m-n)\times n} \end{bmatrix}, \tag{11}$$

where m is the number of measurements and $n < m$ is the number of state variables, Q is an $m \times m$ orthogonal matrix ($Q^T Q = QQ^T = I_m$, I_m being the $m \times m$ identity matrix), U is an upper triangular $m \times n$ matrix, and $0_{(m-n)\times n}$ is the $(m-n) \times n$ zero matrix. Equation (8) can now be rewritten as

$$L^T L\Delta x = L^T \Delta\hat{z} \Leftrightarrow L\Delta x = \Delta\hat{z}, \tag{12}$$

where $\Delta\hat{z}$ consists of the first n elements in $Q^T\Delta z_1$.

This represents a considerable improvement of the conditioning of the equation for the correction Δx.

As discussed below, the columns in Q may appear with arbitrary signs, however, so in some positions U may differ from L by sign.

This report thus adopts PSSE standard notation, where the factors in the QR-factorisation are denoted by Q and U, and R is the diagonal matrix of inverse measurement variances.

The observation that Q essentially consists of the Cholesky factor L is of some value, as it allows the use of symbolic Cholesky factorisation to predict the non-zeros in matrix U. It is only entirely true if $H_1^T H_1$ is SPD (symmetric positive definite), however. Attempts to apply symbolic Cholesky factorisation to $H_1^T H_1$, which were only PSD (positive semi definite), led to minor discrepancies between the structure of U. Those differences vanished when a small SPD part $10^{-12} I$ was added to $H_1^T H_1$.

2.4 A note on the uniqueness of the QR-factorisation

The QR-factorisation of any full column rank $m \times n$ matrix A is mathematically equivalent to the Gram-Schmidt orthogonalisation procedure in a vector space spanned by the columns in A. The columns of matrix Q then constitute an orthonormal basis of the linear vector space spanned by the columns in A while U is the coefficient matrix

$$A(:,j) = \sum_{k=0}^{j} Q(:,k) U(k,j) \tag{13}$$

where $0 \leq j < n$ and Matlab's colon notation is used. Thus, assuming the $Q(:,k)$ have been determined uniquely for $0 \leq k < j-1$, it follows $U(i,j) = Q(:,i)^T A(:,j)$, i.e. the elements in U are uniquely determined. The remainder

$$Q(:,j) U(j,j) = A(:,j) - \sum_{k=0}^{j-1} Q(:,k) U(k,j) \tag{14}$$

is now orthogonal to the previously determined $Q(:,k)$ $(0 \leq k < j-1)$. It may thus be incorporated in the orthonormal basis, by normalisation. This determines $U(j,j)$ and $Q(:,j)$ to within the factor ± 1. Induction now proves that the elements in U are determined to within the sign, and uniquely determined if the diagonal elements are positive. No diagonal element in U can vanish, as $U(j,j) = 0$ would contradict the full rank property, by eq. (14).

The statement above that U equals the Cholesky factor is accordingly only true to within the signs of its elements. This is of no consequence for the positions of the non-zeros.

On the other hand, if A is not full column rank, there exist constants c_k such that $\sum_k c_k Q(:,k) = 0$. Then also

$$A(:,n) = \sum_{k=0}^{n-1} Q(:,k)(U(k,n) + c_k), \tag{15}$$

so the QR-factorisation is only unique (to within signs) for full column rank matrices.

The matrix $H_1^T H_1$, in the normal equations (8), is guaranteed to be SPD. Otherwise a unique solution would not exist. Hence the QR-factorisation is unique, and the structure of U can be obtained by symbolic Cholesky factorisation.

3 The Givens rotation

3.1 The standard Givens rotation

The Givens rotation mathematically expresses the fact that for any $m \times n$ matrix A ($m \geq n$), element $A_{i,j}$ may be zeroed through multiplication with the $m \times m$ orthogonal matrix

$$Q^{(i,j)} = \begin{bmatrix} I_{j-1} & 0 & 0 & 0 & 0 \\ 0 & c & 0 & s & 0 \\ 0 & 0 & I_{i-j-1} & 0 & 0 \\ 0 & -s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I_{m-i} \end{bmatrix} \tag{16}$$

where c and s are scalars, I_k is the $k \times k$ identity matrix, zeros represent zero block matrices, and $i > j$ was assumed, as this report is concerned with the elimination of sub-diagonal elements. In the product matrix $A' = Q^{(i,j)}A$ all rows are identical to the corresponding row in A, except rows j and i for which the linear combinations

$$\begin{aligned} A'_{j,:} &= cA_{j,:} + sA_{i,:} \\ A'_{i,:} &= -sA_{j,:} + cA_{i,:} \end{aligned} \tag{17}$$

holds, where the colon subscript denotes any column index between 0 and $m-1$ (with C-style indexation). Row i is said to be rotated with row j. Since $Q^{(i,j)}$ is devised to eliminate $A'_{i,j}$, orthogonality yields

$$c = \frac{A_{j,j}}{\sqrt{A_{i,j}^2 + A_{j,j}^2}} \text{ and } s = \frac{A_{i,j}}{\sqrt{A_{i,j}^2 + A_{j,j}^2}}. \tag{18}$$

Givens rotations thus eliminate selected elements and is thus suitable for sparse matrix computations. Thus the full QR-factorisation is accomplished by zeroing all sub-diagonal elements through Givens rotations. That is, starting with $Q = I_m$ and $U = A$, for each non-vanishing sub-diagonal element (i, j) in U update $U := Q^{(i,j)}U$ and $Q := QQ^{(i,j)T}$. In practice, however, Q is generally a full matrix. In many applications it is not explicitly required, and therefore not necessary to store. In other cases Q is needed to compute $\hat{z}$ in eq. (12). This computation may be done simultaneously with the QR-factorisation, i.e. by initialising $\hat{z} = z_1$ and successively updating $\hat{z} := Q^{(i,j)T}\hat{z}$. In those situations where Q is really needed, e.g. where the state is updated several times using the same QR-factorisation, Q is best saved as the c and s values and indices (i, j) of the elements eliminated in the individual rotations.

3.2 The fast Givens rotation

The standard Givens rotation requires four multiplications per non-vanishing row element plus two divisions and one square root calculation. Especially the divisions and square roots are quite time consuming. The following discussion of the fast Givens rotation will follow that of Monticelli [4].

The central idea is to keep the square roots in a diagonal matrix D which never enters the elimination of the sub-diagonal entities. Specifically, if $|c| \geq |s|$, rows and columns j and i in $Q^{(i,j)}$, are factorised as (the c-transform)

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix} \begin{bmatrix} 1 & t \\ -t & 1 \end{bmatrix}, \tag{19}$$

where $t = s/c$. Here the square root is only needed to compute c in the first factor, while the actual elimination is performed by the second factor, which only depends on $t = A_{i,j}/A_{j,j}$. If, on the other hand, $|c| < |s|$, the s-transform

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} k & 1 \\ -1 & k \end{bmatrix}, \tag{20}$$

where $k = c/s$ is used. QR-factorisation through Givens rotations, however, involves repeated application of the factors in eqs. (19) and (20). In order to bring the diagonal factors to the left of the t- and k-factors, the following observations also need to be made

$$\begin{bmatrix} 1 & t \\ -t & 1 \end{bmatrix} \begin{bmatrix} d_j & 0 \\ 0 & d_i \end{bmatrix} = \begin{bmatrix} d_j & 0 \\ 0 & d_i \end{bmatrix} \begin{bmatrix} 1 & td_i/d_j \\ -td_j/d_i & 1 \end{bmatrix} \tag{21}$$

and

$$\begin{bmatrix} k & 1 \\ -1 & k \end{bmatrix} \begin{bmatrix} d_j & 0 \\ 0 & d_i \end{bmatrix} = \begin{bmatrix} d_i & 0 \\ 0 & d_j \end{bmatrix} \begin{bmatrix} kd_j/d_i & 1 \\ -1 & kd_i/d_j \end{bmatrix}, \tag{22}$$

where d_i and d_j are the elements at positions (i, i) and (j, j) in the diagonal matrix D.

Hence the elimination of element $A_{i,j}$ requires $t = d_i A_{i,j}/(d_j A_{j,j})$ and is performed by matrix

$$\begin{bmatrix} 1 & p \\ -q & 1 \end{bmatrix} = \begin{bmatrix} 1 & td_i/d_j \\ -td_j/d_i & 1 \end{bmatrix} \tag{23}$$

in the c-transform. Thus $p = d_i^2 A_{i,j}/(d_j^2 A_{j,j})$ and $q = A_{i,j}/A_{j,j}$, while the diagonal matrix D is updated through $d_i^2 := c^2 d_i^2$ and $d_j^2 := c^2 d_j^2$, where $c^2 = 1/(1 + t^2)$.

For the s-transform, similarly, $k = d_j A_{j,j}/(d_i A_{i,j})$,

$$\begin{bmatrix} p & 1 \\ -1 & q \end{bmatrix} = \begin{bmatrix} kd_j/d_i & 1 \\ -1 & kd_i/d_j \end{bmatrix}, \tag{24}$$

$p = d_j^2 A_{j,j}/(d_i^2 A_{i,j})$ and $q = A_{j,j}/A_{i,j}$, while the diagonal matrix D is updated through $d_i^2 := s^2 d_j^2$ and $d_j^2 := s^2 d_i^2$, where $s^2 = 1/(1 + k^2)$. As all updates involve only squares of the diagonal matrix elements, no square roots need to be computed. Since the zeroing is performed by matrices (23) and (24), only two multiplications are required per non-zero element; multiplication with unity is trivial. The fast Givens rotation involves at least three divisions, which is one more than standard Givens.

4 Preservation of sparsity

During QR-factorisation certain elements change from zero to some non-zero value, so called "fill-ins". Fill-ins are unavoidable, yet undesirable, since they reduce the sparsity of the matrix and thus increase the work, and generally involve the creation and insertion of new elements in the matrix data structure.

The latter complication may be alleviated by performing a symbolic Cholesky factorisation of the normal equation matrix $H_1^T H_1$. This yields the sparsity structure of the factor U. Fill-ins in U may thus be avoided by initiating U with the final sparsity structure.

The amount of fill-in can also be controlled by permuting (or reordering) rows and columns. For example, matrices A_1 and A_2 below produce very different amounts of fill

$$A_1 = \begin{bmatrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{bmatrix} \text{ and } A_2 = \begin{bmatrix} x & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & x \\ 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & x & x \\ x & x & x & x & x \end{bmatrix}. \qquad (25)$$

In A_1 fill-ins will occur at all positions, while in A_2 no fill-in takes place. Some fill-ins are permanent while others are intermediate and will eventually be eliminated.

Rows are permuted by multiplication from the left with some row permutation matrix P_r, while columns are permuted by multiplication from the right with some column permutation matrix P_c. The sparsity structure of U is given by the symbolic Cholesky factor of $(P_r H_1 P_c)^T (P_r H_1 P_c) \equiv P_c^T H_1^T H_1 P_c$, and thus is unaffected by the row-ordering. The number of intermediate fill-ins in H depends on both the row- and column-orderings, however.

In this report columns are first permuted in order to preserve sparsity of U, using a minimum degree ordering algorithm. Rows are then ordered to minimise the number of intermediate fill-ins.

We also try the approximate minimum degree (AMD) ordering algorithm, available in the AMD-library. Since this was done late in the project, the investigation of the effects on performance from using this ordering is somewhat incomplete. The AMD-library is available from `http://www.cise.ufl.edu-/research/sparse/amd`, and further details on the algorithm can be found in

[10], [11] and [12].

Further discussions and alternative methods are provided in Section 2.3 in Matstoms [3].

5 Row- and column-oriented elimination schemes

Sub-diagonal elements may be zeroed column by column (column-oriented scheme) or row by row (row-oriented scheme). The fundamentals of those schemes are presented below.

5.1 Column-oriented scheme

Before initiating the rotations, columns are permuted to conserve sparsity based on the sparsity structure of $H_1^T H_1$. The ordering of the processing of the rows is decided continually through the orthogonalisation. Since the sparsity structure of the rows changes during the process, the row that produces the least number of fill-ins is selected, at each stage. Since sub-diagonal elements are eliminated column by column, partially processed rows are repeatedly retrieved, rotated, and stored. Each row-store must accommodate new fill-ins and deallocate memory for zeroed elements.

5.2 Row-oriented scheme

As in column-wise elimination, the sparsity structure of $H_1^T H_1$ is used to permute the columns to conserve sparsity. In contrast, however, sparsity is also conserved by means of initial row-ordering. An additional feature is the fact that matrix U is initiated, through symbolic Cholesky factorisation, as the null matrix with the structure of the Cholesky factor of $H_1^T H_1$. The QR-factorisation then proceeds as the orthogonalisation of

$$\begin{bmatrix} U \\ H_1 \end{bmatrix}, \tag{26}$$

and Q is now an $(m + n) \times (m + n)$. Since Q either is not saved at all, or as a sequence of indices of eliminated entries and the corresponding c- and s-values, the increase in size has no practical consequences.

Furthermore, fill-ins in H_1 never occur, as the rows are copied and *the copy* is then rotated with rows in U until fully processed. There is thus no need to store and retrieve partially processed rows in H_1. All fill-ins take place in the copy.

Row-ordering is simple and performed only once, before initiating the rotations. Rows may, for example, be sorted in order of ascending number of non-zeros, highest column-index, or column-index sum. In the column-oriented approach, rows must be compared with other rows throughout the process, in order to minimise the number of fill-ins.

Also, new measurements are easily incorporated in the row-oriented approach, by simply processing the resulting additional row.

6 Sparse matrix formats

6.1 CRS-format

The CRS (compressed row storage) format defines a sparse $m \times n$ matrix through three arrays:

`values` – the values (listed row-wise) of the elements in its sparsity structure.

`columns` – the column indices (listed row-wise) of the elements in its sparsity structure.

`rowIndex` – the indices of the first elements (listed row-wise) on each row. The last element is the number of entries (using C-style indexation) or that number plus one (using Fortran-style indexation).

The CRS format is a row-major (or compressed row) format. For completeness it should also be mentioned that an analogous column-major (or compressed column) format exists with obvious modifications, where array `values` is as above but an array `rows` replaces array `columns`, and an array `columnIndex` replaces array `rowIndex`. This format is called the CCS (compressed column storage) format. Further information on CRS and CCS can be found in e.g. [13].

There also exists many other storage formats similar to CRS and CCS, e.g. special variants for symmetric matrices, banded matrices and block matrices. Due to the nature of the matrices encountered in PSSE, none of these specialised formats are of interest to us and have not been considered further in this report.

6.2 Binary search tree format

While fill-ins cause a considerable amount of re-shuffling in array based formats, like the CRS-format above, this problem is less severe for balanced binary trees, e.g. Sedgewick [14]. For this reason balanced binary trees are also investigated as a storage format.

7 Parallelisation

7.1 Column-oriented approach

In column-oriented Givens, as described in Section 5, elements are eliminated column-by-column and rows are thus only partially processed for each fully processed column. When sparse matrices are used, the matrix may need to be rebuilt several times due to new fill-ins. This can be a problem in a parallel implementation when using matrix formats where non-zero elements are stored in a linear array and auxiliary index-arrays are used. Since the insertion of a new non-zero requires (at least a large fraction of) the matrix to be rebuilt, insertion becomes a strictly sequential operation. For this reason, we do not address the parallelisation of column-oriented Givens.

7.2 Row-oriented approach

Row-oriented Givens, as described in Section 5, has several advantages for a parallel implementation.

- Threads can be assigned its own set of rows, and each row will be fully processed by the thread it was assigned to.
- Intermediate fill-ins occur only in each thread's local copy of a row, not in U and H.

The pseudo-code (with the OpenMP-pragma included) for the algorithm implemented, is the following:

Algorithm 1 GIVENS(H, U)

```
#PRAGMA OMP PARALLEL FOR
for $i = 0$ to $n$ do
    $H_i \leftarrow$ COPY(H,i)
    for $j = 0$ to $m$ do
        if $H_i(j) \neq 0$ then
            LOCK($U$, $j$)
            ROTATE($H_i$, $U$, $j$)
            UNLOCK($U$, $j$)
        end if
    end for
end for
```

7.3 Parallel implementation

We have implemented the row-oriented approach described in Section 7.2. The implementations use the CRS-storage format, described in Section 6.1 and the binary tree format, described in Section 6.2, respectively. In the implementation where the CRS-storage format is used, the CCS-format is also used when loading a matrix from file and performing column-reordering, and a linked list format when performing the symbolic factorization, but all matrices are converted back into CRS format before the function where the actual QR-factorization takes place is called.

The locking-mechanism, for preventing threads from using the same row in U at the same time, was implemented in the following way:

```
while (1) {
  #pragma omp critical
  {
    if (U_lock[j] == UNLOCKED)
      U_lock[j] = own_lock; // Lock row j in U with this thread's lock.
  }
  if (U_lock[j] == own_lock)
    break;
}
```

The variable `own_lock` is assigned a unique value for each row in H, thus no threads will share this value. When a thread is done, it unlocks the row with:

```
#pragma omp critical
{
  U_lock[j] = UNLOCKED;
}
```

The program performs the following steps before the Givens rotations are started:

- The H matrix is loaded from its file into a matrix represented in the internal matrix storage format.
- The columns in H are ordered according to the MD-ordering on $H^T H$.
- The structure of U is computed by symbolic factorisation of H.
- The initial null-matrix U is constructed with the structure found by the symbolic factorisation.
- The rows in H are reordered to minimize intermediate fill-in during the factorisation.

8 Results

During the project, we had access to the following parallel computer systems at Uppmax:

> `Ngorongoro` – The Ngorongoro-system is a Sun Fire 15K server, running Solaris 9. We have conducted our tests on its `duma`-domain, which has 4 1300 MHz UltraSPARC IV+ dual core (CMP) CPUs and 16 GB of RAM.
>
> `Isis` – The Isis-cluster consists of 200 IBM x3455 servers, each such node having two dual core AMD Opteron 2220 CPUs and 4-16 GB of RAM. The cluster is running Scientific Linux.
>
> `Grad` – The Grad-cluster consists of 56 nodes, each node having two quad core Intel Xeon E5240 CPUs and 16 GB of RAM. The cluster is running Scientific Linux.

All tests of the parallel implementations were conducted on these three systems. On Ngorongoro, the Sun Studio 11 C-compiler was used to compile the code. On `Isis` and `Grad`, the code was compiled with the Portland Group C-compiler. More detailed information on the systems, with the exception of the

Table 1: Execution times in s for random test matrices in absence of initial column and row reordering.

size of H_1	symbolic Cholesky	QR	non-zeros in U
(1,000 , 600 , 4,000)	2.84×10^0	4.62×10^1	114,679
(2,000 , 1,200 , 8,000)	2.23×10^1	4.03×10^2	467,724

Table 2: Execution times in s for fast Givens rotations and random test matrices in presence of initial column and row reordering.

size of H_1	column permutation	symbolic Cholesky	QR	non-zeros in U
(1,000 , 600 , 4,000)	1.72×10^{-1}	6.41×10^{-1}	8.80×10^0	43,232
(2,000 , 1,200 , 8,000)	6.10×10^{-1}	4.30×10^0	7.69×10^1	158,766
(4,000 , 2,400 , 16,000)	3.72×10^0	3.23×10^1	7.38×10^2	626,183

Grad-cluster (which is still under construction, at this time of writing), can be found at `http://www.uppmax.uu.se/systems`.

Some tests of sequential implementations has also been done on other systems, such as the authors' laptops.

All source-code for this project, with instructions, Makefiles and test data, can be found via `http://www.it.uu.se/edu/course/homepage/projektTDB`.

8.1 Effects of column- and row permutation

Tables 1 and 2 show the effect of initial column and row permutation on the performance of the binary tree representation, with fast Givens rotations. Table 1 thus lists execution times, for the symbolic Cholesky and the QR-factorisations, as well as the number of non-zeros in the resulting U-factor, when initial column and row permutation are not performed on a random test matrix H_1 of size $(N, 0.6N, 4N)$, where the first position (N) represents the number of rows, the second $(0.6N)$ is the number of columns, and the third $(4N)$ is the number of non-zero entries. In Table 2 column and row permutation have been performed, the time consumption of the column permutation is also presented. No essential improvements due to row permutation could be detected. Results are quoted for sequential single runs on an Acer Aspire 5002 WLMI computer, but variations are within one or a few percent.

It is quite clear that the most time consuming part is the proper Cholesky factorisation. The most time demanding part here are the operations associated with memory accesses for retrieval and storage of matrix elements. Some ideas

Table 3: Execution times in s for standard Givens rotations and random test matrices in presence of initial column and row reordering.

size of H_1	column permutation	symbolic Cholesky	QR	non-zeros in U
(1,000 , 600 , 4,000)	1.72×10^{-1}	6.25×10^{-1}	9.12×10^{0}	43,232
(2,000 , 1,200 , 8,000)	6.10×10^{-1}	4.30×10^{0}	7.78×10^{1}	158,766
(4,000 , 2,400 , 16,000)	3.72×10^{0}	3.21×10^{1}	7.51×10^{2}	626,183

to enhance the performance of these functions exist, e.g. use of direct pointers to tree nodes representing the matrix elements. Those ideas have not yet been tested, however. As will be seen in the next Section, the flop count is, however, a minor consideration.

It can also be seen that although column permutation consumes comparatively little time, it pays off handsomely, by reducing the time requirement of the proper QR-transformation by a factor of about 5. Column permutation is particularly profitable as it only needs to be performed once.

The fact that the symbolic Cholesky factorisation is rather time consuming should be considered against the fact that it only needs to be performed once per network. Neither column permutation nor symbolic Cholesky factorisation are therefore included in the performance studies below.

There are several variants of column and row permutations. Here columns are ordered according to the minimum degree criterion, i.e. in ascending order of number of non-zeros in the corresponding columns in $H^T H$. It remains to investigate alternative orderings and to introduce tie-breakers. Rows are ordered in ascending order of column index, as well as number of non-zeros of last element, on the row. In both cases, row permutation is of no or little consequence. Tie-breakers are not investigated.

8.2 Performance of standard and fast Givens rotations

Table 3 presents the effect of replacing fast Givens rotations with standard Givens. The latter is only a few percent faster than the latter.

Row- and column permutations were applied, but the conclusion, that fast Givens rotations only reduces the numerical work insignificantly, holds even if no permutations take place.

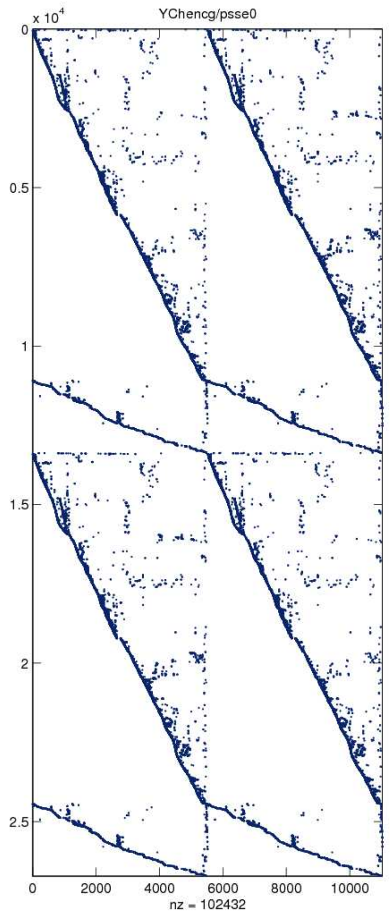

Figure 1: The structure of the YCheng/psse0-matrix. This matrix is a realistic model matrix for PSSE.

8.3 Profiling

8.3.1 Binary tree implementation

An indication of the time spent in the various parts of the binary tree implementation can be obtained by profiling its sequential version. The results listed below are for the YCheng/psse0 [15] matrix (a full rank 26,722 × 11,028 matrix with 102,432 non-zeros, see fig. 1)

```
duma> prof Tree
 %Time Seconds Cumsecs  #Calls   msec/call  Name
  41.5   95.65   95.65584096683      0.0002  setEntry
  22.7   52.20  147.85563817528      0.0001  getRowsOnColPr
  15.8   36.46  184.31348529252      0.0001  getColsOnRowPr
   7.0   16.07  200.38       1   16070.      QRStd
```

```
6.5   14.90  215.28       1  14900.      QR
4.7   10.72  226.001521602819      0.0000  _mcount
0.9    2.13  228.1318550250      0.0001  insertPr
0.6    1.42  229.55       2    710.      symbChol
0.1    0.31  229.86 4015752      0.0001  getRowsAndColsPr
0.1    0.26  230.12  754716      0.0003  inTree
0.1    0.16  230.28  860242      0.0002  deallocateME
0.0    0.03  230.31  931742      0.0000  insert
0.0    0.02  230.33       1     20.      permuteCols
0.0    0.01  230.34   46630      0.0002  mergeSort
0.0    0.00  230.34       1      0.      cmpMatr
0.0    0.00  230.34       1      0.      main
0.0    0.00  230.34       1      0.      readInputFile
0.0    0.00  230.34       1      0.      permuteRows
0.0    0.00  230.34       1      0.      calcMTMSymb
0.0    0.00  230.34      14      0.0     getRowsAndCols
```

The importance of an appropriate data structure and efficient data access is thus demonstrated by the fact that some 80 percent of the execution time is spent on retrieving and storing data. In particular, function `setEntry`, which accounts for approximately 40 percent of the entire time consumption, is used to save processed rows in U. It utilises a tree search for each stored element, and therefore has a considerable potential for further optimisation. Moreover, function `getRowsOnColPr` is the recursive part of a function that localises the row numbers of the non-zeros on a column. It is only used, however, in the column permutation. Since this can be put outside the main iteration, it only needs to be done once. Its relative time consumption in an iterative procedure, involving a large number of QR-factorisations, is thus small. Considerably larger gains may be accomplished by optimising function `getColsOnRowPr`. This function, which is used in order to fetch non-zeros on a given row, is used repeatedly in the QR-factorisations.

The numerical work is primarily done in functions `QR` and `QRStd`, which only account for about 7 percent of the execution time each. The implementation of fast Givens rotations, is thus hardly worth while.

8.3.2 CRS-based implementation

As for the binary tree based implementation, the profiling for the CRS-based implementation is done with the YCheng/psse0-matrix as input matrix.

When the MD-ordering is used, the CRS-based implementation yields the following profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 59.12    106.50   106.50   102430     0.00     0.00  insert_new_element
 18.31    139.48    32.98                                   _mcount2
 12.68    162.33    22.85 1550400457     0.00     0.00  get_element
  5.00    171.34     9.01 226474083     0.00     0.00  set_element_noalloc
  3.15    177.02     5.68          1     5.68    24.58  qr_givens
  0.53    177.98     0.96          2     0.48     4.83  symmat_std_to_sym
  0.29    178.51     0.53          1     0.53   111.41  symmat_perm_cols
  0.24    178.94     0.43 34441656     0.00     0.00  try_insert
  0.24    179.38     0.43                               __rouinit
  0.15    179.64     0.27                               __pgio_environ
  0.07    179.78     0.14                               __rouexit
  0.05    179.86     0.08                               __set_element_noallocEND
  0.04    179.93     0.07  1581529     0.00     0.00  insert_new_element_noalloc
  0.04    180.00     0.07        1     0.07     0.50  symmat_symb_chol
  0.02    180.03     0.03   102432     0.00     0.00  set_element
  0.01    180.05     0.02  1122593     0.00     0.00  symmat_insert_nonzero
  0.01    180.07     0.02        1     0.02     0.02  create_nm_from_t_symb
  0.01    180.09     0.01                               __try_insertEND
  0.01    180.10     0.01   336033     0.00     0.00  md_comp
  0.01    180.11     0.01        1     0.01     0.01  perm_rows
  0.01    180.12     0.01        1     0.01     0.01  read_sparse_rowwise
  0.01    180.13     0.01        1     0.01     0.02  symmat_symb_lt_mtm
  0.01    180.14     0.01        1     0.01     0.02  symmat_symb_lt_mtm2
  0.00    180.15     0.01                               __set_elementEND
  0.00    180.15     0.00    75500     0.00     0.00  free_nonzeros
  0.00    180.15     0.00    26726     0.00     0.00  free_matrix
  0.00    180.15     0.00        4     0.00     0.00  symmat_alloc
  0.00    180.15     0.00        4     0.00     0.00  symmat_free
  0.00    180.15     0.00        1     0.00     0.00  alloc_matrix
  0.00    180.15     0.00        1     0.00   146.22  main
  0.00    180.15     0.00        1     0.00     0.00  mergeSort
```

Using the AMD-ordering instead of MD-ordering one obtains the profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 37.35     13.99    13.99        1    13.99    13.99  reorder_columns_amd
 33.84     26.66    12.68                               _mcount2
 17.49     33.22     6.55 595190627     0.00     0.00  get_element
  5.15     35.14     1.93 63505738     0.00     0.00  set_element_noalloc
  4.00     36.64     1.50        1     1.50     6.80  qr_givens
  1.04     37.03     0.39        1     0.39     3.63  symmat_std_to_sym
  0.44     37.20     0.17                               __rouinit
  0.29     37.31     0.11                               __pgio_environ
  0.19     37.38     0.07  1973646     0.00     0.00  insert_new_element_noalloc
  0.11     37.42     0.04                               __rouexit
  0.06     37.44     0.02                               __set_element_noallocEND
  0.03     37.45     0.01        1     0.01     0.01  create_nm_from_t_symb
 0.03     37.46     0.01                                  amd_2
 0.00     37.46     0.00   589457     0.00        0.00  try_insert
 0.00     37.46     0.00   413779     0.00        0.00  symmat_insert_nonzero
 0.00     37.46     0.00    37750     0.00        0.00  free_nonzeros
 0.00     37.46     0.00    26724     0.00        0.00  free_matrix
 0.00     37.46     0.00        2     0.00        0.00  free_matrix_cco
 0.00     37.46     0.00        2     0.00        0.00  symmat_alloc
 0.00     37.46     0.00        2     0.00        0.00  symmat_free
 0.00     37.46     0.00        1     0.00        0.00  flip_ccs_to_crs
 0.00     37.46     0.00        1     0.00       24.44  main
 0.00     37.46     0.00        1     0.00        0.00  read_sparse_cco
 0.00     37.46     0.00        1     0.00        0.00  symmat_symb_chol
 0.00     37.46     0.00        1     0.00        0.00  symmat_symb_lt_mtm
```

We may note that the function that performs column reordering, `symmat_perm_cols`, has a very long execution time in the MD-version. However, this is because the column reordering is done on a matrix in CRS-format, instead of a matrix in CCS-format. This has been changed in the version of this implementation where AMD-ordering is used, with the result that execution time for this step is considerably lower.

It is difficult to compare these two versions of the implementation from these profiles, since the `main`-function has been altered between the different versions. Row reordering, for example, was omitted in the version where AMD-ordering is used. To get more comparable profiles, the MD-ordering should be moved into the AMD-version and replace the call to `amd_order` (not visible in the profile above).

8.4 Effect of number of threads

8.4.1 CRS-implementation

Both the CRS- and tree-versions have been parallelised. For the former, application to the YCheng/psse0 [15] matrix suggests that execution times are almost inversely proportional to the number of threads, see fig. 2.

Equivalently, fig. 3 indicates that speedup is virtually proportional to the number of threads.

However, a programming error, in the function where column-ordering on H is computed from minimum degree ordering (MD-ordering) on H^TH, resulted in a column-order for which symbolic factorization on H gave an U with 4,677,319 non-zero elements. When this error was corrected, column-ordering followed by symbolic factorization instead gave an U with 276,473 non-zeros. At that time, we no longer had access to the Grad-cluster, so subsequent tests have been conducted on `Isis`, for up to 4 threads. See Figure 4 for tests of the parallel implementation with MD-ordering of columns prior to the factorization.

We have also tested the parallel CRS-based implementation with AMD-ordering, see Figure 5. The AMD-ordering resulted in a U with only 70,803

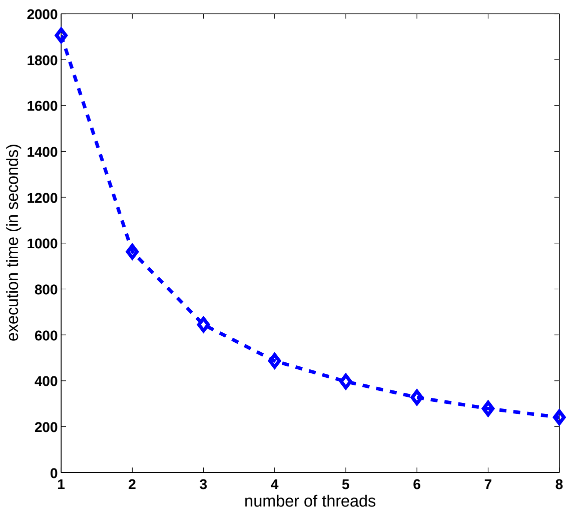

Figure 2: Execution times for the original CRS-based parallel implementation, with inadequate column reordering of columns due to a programming error, when factorizing the YCheng/psse0-matrix on a node on `Grad`. Execution times are for performing the Givens rotations only.

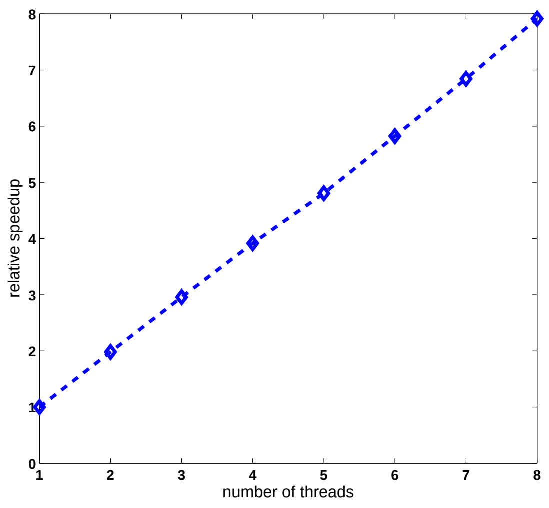

Figure 3: Speedup, as a function of number of threads, for the original CRS-based parallel implementation, with inadequate column reordering of columns due to a programming error, when factorizing the YCheng/psse0-matrix on a node on `Grad`. Speedup is measured for performing the Givens rotations only.

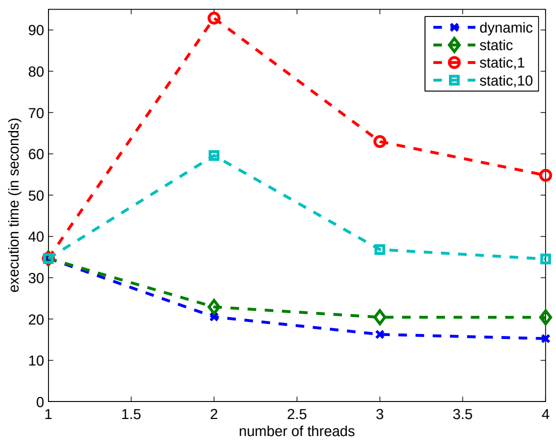

Figure 4: Execution times for the CRS-based parallel implementation, with corrected column reordering, when factorizing the YCheng/psse0-matrix on a node on `Isis` and using different OpenMP schemes for partitioning the rows in H between the threads. The MD-ordering prior to the factorization has been corrected, and the structure of U computed by symbolic factorization has 276,473 non-zeros.

non-zeros.

In the version with MD-ordering, the OpenMP scheme `dynamic` provides the fastest execution times. The `dynamic`-scheme prevents, at least to some extent, threads from leaping far ahead of the others, so row-ordering prior to the factorization may still have effect when this scheme is used.

In the version with AMD-ordering, the `dynamic`-scheme gives slower execution times than the `static`-schemes. A possible explanation for this may be that no row-ordering was done prior to the factorization, and thus there was no benefit in processing the rows in order. Communication is also higher for the `dynamic`-scheme, so the overhead from such communication may become visible when the number of non-zeros in U are few and rows in H can be processed faster.

The maximum speedup achieved is approximately 2.5 on 4 threads, for both the MD- and AMD-version.

8.4.2 Tree implementation

The parallelisation of the binary tree algorithm exhibits very different features from that of the original CRS-version. While serial execution times on the `duma`-domain on the `Ngorongoro` multiprocessor computer at Uppsala university are reduced to 50 - 60 s, the introduction of OpenMP pragmas still results in exe-

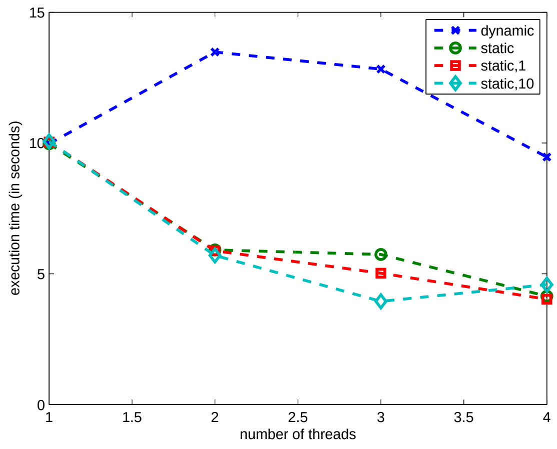

Figure 5: Execution times for the CRS-based parallel implementation, with column reordering based on AMD-ordering instead of the MD-ordering used in previous versions, when factorizing the YCheng/psse0-matrix on a node on `Isis` and using different OpenMP schemes for partitioning the rows in H between the threads. This version uses AMD- instead of MD-ordering prior to the factorization, and the structure of U computed by symbolic factorization has 70,803 non-zeros.

cution times of at least 200 s. The reasons for this are not understood. Figure 6 shows two series of measurements of execution time t_{exe} as a function of the number of threads n. Very long execution times are obtained for $n = 3$ threads, and even for $n = 8$ threads the parallel implementation is around four time slower than the serial one.

Further reductions in the time demand for a single sequential QR-factorisation to 30 - 40 s are obtained on a node on the `Isis`-cluster at UPPMAX. Figure 7 thus presents two series of measurements of the execution times for the QR-factorisation on `Isis`.

These results are quite different. The time consumption of the sequential and single thread parallel implementations are quite similar. There is also a significant speedup for up to four threads, where a QR-factorisation requires about 16 - 18 s. For higher numbers of threads, however, a sharp decline in efficiency occurs.

8.5 Comparison between using CRS- and binary tree formats with row-oriented schemes

By employing the tree structure to the YCheng/psse0 matrix, execution times of about 130 s for the column permutation, 60 s for the symbolic Cholesky factorisation, and 297 s for the fast Givens based QR-factorisation are obtained, with

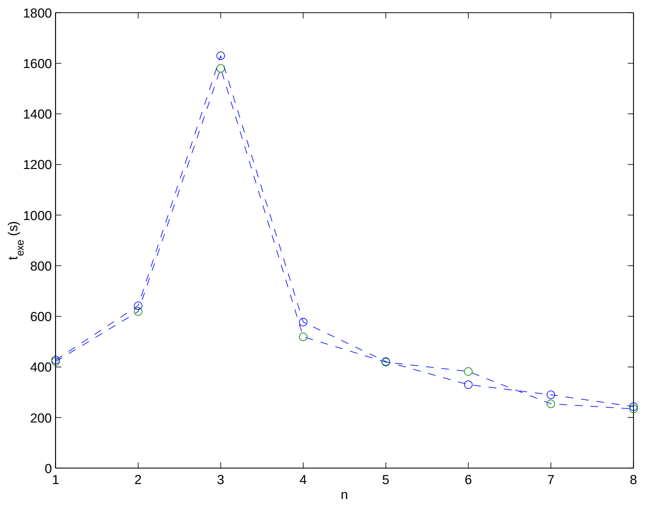

Figure 6: Execution time t_{exe} of parallelised QR-factorisation as a function of number of threads n on `duma`.

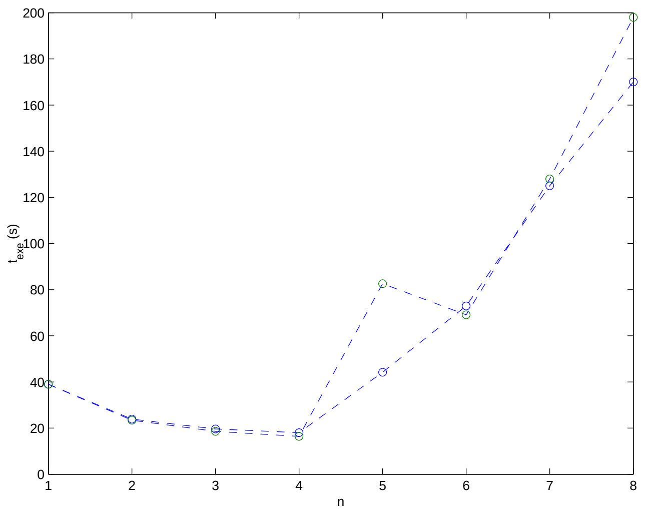

Figure 7: Execution time t_{exe} of parallelised QR-factorisation as a function of number of threads n on `Isis`.

a sequential implementation on an Acer Aspire 5002 WLMI machine. Replacing fast Givens with standard Givens rotations, increased execution times to 317 s. The maximum norm difference between the results from the two methods was around 5.5×10^{-10}, indicating that both methods converge to the same result.

This is an improvement by a factor of about six, compared to the original CRS-implementation, which was run on a node on the `Grad`-cluster.

As mentioned above, further improvements were obtained on computers `Ngorongoro` and `Isis`. Performance is accordingly very machine dependent, so the data quoted above must be read with this fact in mind.

Figure 7 reports results with dynamic scheduling, and are quite similar to the corresponding results in fig. 4.

A point of some interest is the observation that considerably longer execution times would be obtained for the much smaller random matrices used in Tables 1 through 3, by increasing the row number N. Those matrices were constructed to possess approximately four elements per row, just as the YCheng/psse0-matrix. This is a vivid illustration of the fact that matrix structure is as important as matrix size. The resulting U-factor has 276,473 non-zeros. Comparison with Tables 1 through 3 suggests that this number is a crucial parameter for the execution time.

9 Discussion and conclusions

While a comparison between row- and column-oriented methods was included in the project description, time only admitted the implementation of row-oriented schemes. This comparison must accordingly be left to later investigations. Because column-oriented elimination leaves partially processed rows for each fully processed column, a parallel version can be tricky to implement for sparse matrices, especially if one wishes to avoid re-building the matrix during the factorization.

Substantial enhancements of performance due to initial column permutation are achieved. Permutation according to AMD-ordering proves to be especially efficient in preserving the sparsity of U, and this also results in faster symbolic factorization. No improvements can be seen when row permutation is applied. This does not, however, mean that row reordering is without effect, only that no effects can be discerned for the rather limited variations that have been tested. Yet, in parallelised implementations, threads tend to select rows in a somewhat randomised order, depending on when they complete processing of previous rows. This reduces any gain due to row reordering. The introduction of barriers would counteract such effects but, on the other hand, reduce performance as threads would be idle, while waiting for the last ones to catch up.

For the original CRS-implementation almost perfect scalability is observed, as shown in figs. 2 and 3. This is not the case for the tree-implementation, where on the `duma`-domain on `Ngorongoro` parallelisation implies considerably longer execution times and poor scalability, cf. fig. 6. On `Isis` no increase in execution time can be detected, and for small numbers of threads a significant improvement of efficiency is observed.

Replacement of standard Givens rotations with fast Givens rotations only yields slight improvements. In the binary tree implementation, fast Givens rotations also tends to leave residual round-off errors at the positions of eliminated fill-ins. For some reason, this occurs less frequently for standard Givens rotations. Hence, components emerging from the rotations with values smaller than some threshold (e.g. 1×10^{-12}), are set to zero. In order to clarify the point about the eliminated fill-ins, the Givens rotation is devised to set the first element in the rotated row in H zero. It is, therefore, always set to zero to save flops and avoid round-off errors. However, additional entries in the rotated row occasionally also vanish, and as they may reside as round-off errors, all elements in the rotated row in H are set to zero if smaller than some threshold value.

The most time consuming operations are the fetching and storage of data. At least in the binary tree implementations, better routines for the storage of rotated rows in U and track-keeping of the first non-zero element in the row should be investigated.

Acknowledgments

Thanks are due to Magnus L. Johansson at ABB for supplying us with valuable reference material and guidance through the project. Maya Neytcheva at Uppsala University is also thanked for discussions and assistance with various practical details during the project. We also thank UPPMAX for access to their parallel computers.

References

[1] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia (1997), ISBN-10 0-89871-389-7.

[2] G. H. Golub and C. F. Val Loan, *Matrix Computations* 3rd Ed., The John Hopkins University Press, Baltimore (1996), ISBN 0-8018-5414-8.

[3] P. Matstoms, *Sparse QR-Factorization with Applications to Linear Least Squares Problems* Linköping Studies in Science and Technology. Dissertations. No. 337. Department of Mathematics, Linköping University, SE 581 83 Linköping. Linköping (1994).

[4] A. Monticelli, *State Estimation in Electric Power Systems Analysis: A Generalized Approach*, Kluwer Academic Publishers Group, Dordrecht (1999), ISBN 0-7923-8519-5.

[5] N. Vempati, I. W. Slutsker, and W. F. Tinney, *Enhancements to Givens rotations for Power System State Estimation* IEEE Transactions on Power Systems, Vol. 6, No. 2, May 1991, pp. 842 - 849.

[6] P. S. R. Diniz, *Adaptive Filtering: Algorithms and Practical Implementations* 2nd Ed., Kluwer Academic Publishers Group, Dordrecht (2002), ISBN 1-4020-7125-6.

[7] S. A. Soman, S. A. Khaparde, and S. Pandit, *Computational Methods for Large Sparse Power Systems Analysis: An Object Oriented Approach* 2nd Ed., Kluwer Academic Publishers Group, Dordrecht (2002), ISBN 0-7923-7591-2.

[8] A. Pandian, K. Parthasarathy, and S. A. Soman, *Towards Faster Givens Rotations based on Power System State Estimator* IEEE Transactions on Power Systems, Vol. 14, No. 3, August 1999, pp. 837 - 843.

[9] A. J. Wood and B. F. Wollenberg, *Power Operation, Generation, and Control* 2nd Ed., Wiley, New York (1996), ISBN 0-471-58699-4.

[10] P. Amestoy, T. A. Davis, and I. S. Duff, *Algorithm 837: AMD, An approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software, vol 30, no. 3, Sept. 2004, pp. 381-388.

[11] P. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, vol 17, no. 4, pp. 886-905, Dec. 1996.

[12] T. A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, Sept. 2006. Part of the SIAM Book Series on the Fundamentals of Algorithms.

[13] Appendix A in http://www.intel.com:80/software/products/mkl/docs/WebHelp/mklrefman.htm

[14] R. Sedgewick, *Algorithms in C++: Parts 1-4* 3rd Ed., Addison-Wesley, Boston (1998), ISBN 0-201-35088-2.

[15] http://www.cise.ufl.edu/research/sparse/matrices/YCheng/