Implementing Hager's Exchange Methods for Matrix Profile Reduction

JOHN K. REID and JENNIFER A. SCOTT
Rutherford Appleton Laboratory

Hager recently introduced down and up exchange methods for reducing the profile of a sparse matrix with a symmetric sparsity pattern. The methods are particularly useful for refining orderings that have been obtained using a standard profile reduction algorithm, such as the Sloan method. The running times for the exchange algorithms reported by Hager suggested their cost could be prohibitive for practical applications. We examine how to implement the exchange algorithms efficiently. For a range of real test problems, it is shown that the cost of running our new implementation does not add a prohibitive overhead to the cost of the original reordering.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*numerical algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*sparse, structured, and very large systems*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sparse symmetric matrices, matrix profile, exchange method

1. INTRODUCTION

If Gaussian elimination is applied to a symmetric positive-definite matrix A of order n, all zeros between the first entry of a row and the diagonal usually fill in (this happens if rows 2, 3, ..., n all have at least one entry to the left of the diagonal). Therefore, variable-band software does not take any account of such zeros and the total number of entries stored in the triangular factor is the sum of the lengths of the rows of the original matrix, where each length is counted from the first entry to the diagonal. This sum is also known as the **profile**.

The success of variable-band software depends critically on finding an ordering for which the profile is small. If such an ordering can be found, the software is likely to be more efficient than software that relies on more sophisticated data structures, for example, nested dissection [George and Liu 1981], multifrontal [Duff and Reid 1983], or SuperLU [Demmel et al. 1999]. Furthermore, variable-band software is easy to organize for out-of-memory working.

A variety of methods has been proposed for choosing a permutation of the matrix that reduces the profile. Following the publication of the paper by Cuthill

J. A. Scott was funded by the EPSRC Grant GR/R46441.
Authors' address: Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England; email: jkr@rl.ac.uk; j.scott@rl.ac.uk.

and McKee [1969], graph theory and level sets, in particular, became a standard approach for both bandwidth and profile reduction. Methods proposed in the late 1970s and early 1980s include the Reverse Cuthill–McKee [George and Liu 1981], the Gibbs–King [Gibbs 1976], and the Gibbs–Poole–Stockmeyer [Gibbs 1976; Lewis 1982] algorithms. The Sloan algorithm [Sloan 1986, 1989] offered a considerable improvement over these methods by introducing a second step in which the ordering obtained using the pseudodiameter from a variant of the Gibbs–Poole–Stockmeyer algorithm was locally refined. The Sloan algorithm has been widely used and a number of enhancements to the original algorithm have been proposed (see, e.g., Duff et al. [1989], Kumfert and Pothen [1997], and Reid and Scott [1999]). A high quality implementation of an enhanced version of the Sloan algorithm is available within the mathematical software library HSL (www.cse.clrc.ac.uk/nag/hsl) as routine `MC60`.

A very different approach was described by Barnard et al. [1995]. Their spectral method was based on computing the Fiedler vector of the Laplacian matrix associated with the matrix A. A similar idea was suggested independently by Paulino et al. [1994a,b]. Kumfert and Pothen [1997] proposed combining the second step of the Sloan algorithm with the spectral ordering. For large problems, the resulting **hybrid** algorithm has been shown to give significantly better orderings than either the spectral method or the Sloan method alone. The main disadvantage is that the hybrid algorithm requires the computation of the Fiedler vector of a large matrix, and this can add considerably to the reordering cost (Hu and Scott [2001] report that the CPU time can be up to five times that for the Sloan algorithm).

Motivated by the success of multilevel algorithms for graph partitioning, Hu and Scott [2001] recently developed a **multilevel** algorithm for wavefront and profile reduction. A series of graphs is generated, each coarser than the preceding one. The enhanced Sloan algorithm is employed on the coarsest graph. The coarse graph ordering is then recursively prolonged to the next finer graph, with local refinement performed at each level. The final ordering on the finest graph gives an ordering for A. Extensive numerical experimentation has shown that the multilevel approach gives orderings of similar quality to that of the hybrid algorithm, while being significantly faster.

Hager [2000] suggested two methods for improving any given permutation for profile reduction. His down exchange algorithm involves a cyclic permutation, that is, the successive exchange of rows $(k, k+1)$, $(k+1, k+2), \ldots,$ $(l-1, l)$ of the permuted matrix and interchanging corresponding columns. For a given k, Hager finds the value of l that most reduces the profile. He performs a pass over the matrix with k taking the values $n-1$, $n-2, \ldots, 1$; he calculates l for each k and, if this gives a profile reduction, applies the corresponding permutation.

Hager's up exchange is similar, with the direction reversed. For a given k, he exchanges rows and columns $(k, k-1), (k-1, k-2), \ldots, (l+1, l)$, finding the value of l that most reduces the profile. He performs a pass over the matrix with k taking the values $2, 3, \ldots, n$.

Hager proposes using the down and up exchange schemes in an iterative fashion: the down exchange algorithm is first applied, followed by the

up exchange algorithm, followed by the down exchange algorithm, and so on.

In many applications, it is important that reordering the matrix to reduce the profile is done as quickly and efficiently as possible. If a large number of matrices having the same sparsity pattern are to be factored or if storage restrictions require the smallest possible profile, it may be worthwhile to spend a relatively large amount of time computing a permutation that minimizes the profile. However, if the matrix needs to be factored only once, the cost of reducing the profile must be compared with that required for the matrix factorization; in such circumstances, a slightly larger profile may be acceptable if it can be computed cheaply. Hager presents timings for his exchange algorithms that show they are expensive to run compared with algorithms such as the Sloan algorithm that are used to produce the initial reordering. Hager also reports that, in general, he found the down exchange algorithm to be significantly faster than the up exchange algorithm (typically by a factor of between 4 and 10). We have considered carefully how the exchange algorithms should be implemented. In Sections 2 and 3, we explain how to implement the down and up exchange methods efficiently. In particular, careful implementation of the up exchange algorithm results in it running much faster than Hager reported. Results for a set of test matrices arising from practical applications are presented in Section 4. The profile reductions achieved using the Hager exchanges are given together with the CPU times required to achieve these reductions. Concluding remarks are made in Section 5.

2. IMPLEMENTATION OF THE DOWN EXCHANGE

Any reduction in the profile leads to a corresponding increase in the total number of zeros ahead of the first entries of the rows. Therefore, we can minimize the profile by maximizing this total. By symmetry, this is also the total number of zeros ahead of the first entries of the columns. This total is unaffected by column permutations, so we can delay applying them and work solely with row permutations in the body of the code. We need a representation of the pattern of the whole symmetric matrix, including the diagonal.

We have chosen to hold the structure of the original matrix unchanged while the permutations are found. We hold the current permutation and its inverse in two integer arrays `perm` and `inv_perm` of length n and update these as each cyclic permutation (k, l) is found. This involves far less work than updating the matrix structure.

In describing the algorithm (and in writing the code) it is very important to distinguish between original row indices and permuted row indices. Unless otherwise stated, array subscripts and stored row indices refer to original indices so that they do not need to be altered with each cyclic permutation (k, l). In this description, "first," "second," "in front," "beyond," ... refer to the position in the column when the rows have been permuted. We hope that the context makes it clear whether the row index is original or permuted.

We define a gain to be a decrease to the profile and a loss to be an increase to the profile. The net gain is the difference between the gain and the loss resulting

	1	2	3	4	5	6
1	×		×	×		×
2		×				
3	×		×		×	
4	×			×	×	
5			×	×	×	×
6	×				×	×

Fig. 1(a). Original matrix.

	1	2	3	4	5	6
1		×				
2	×		×		×	
3	×			×	×	
4	×		×	×		×
5			×	×	×	×
6	×				×	×

Fig. 1(b). Row permuted matrix.

	1	2	3	4	5	6
1	×					
2		×		×	×	
3			×	×	×	
4		×	×	×		×
5		×	×		×	×
6				×	×	×

Fig. 1(c). Row and column permuted matrix.

from a permutation. Thus at each stage we are seeking the maximum net gain.

When permuted rows k and $k+1$ are exchanged, there is a gain of one for each column with its first entry in row k and its second entry after row $k+1$, and there is a loss of one for each column with its first entry in row $k+1$. As the further interchanges $(k+1, k+2), \ldots$ are performed, there is a further gain in each column that gained from the first interchange until the second entry of the column is reached. There is also a loss of one as each other first entry is encountered. Hager sweeps forward from row k accumulating the net gain at each stage until there are no columns with a potential for gain. The row for which the maximum gain is found provides the index l.

We illustrate this by the example shown in Figure 1(a). Exchanging rows 1 and 2 gives a gain in columns 1, 3, 4, and 6 and a loss in column 2. Following this with an exchange of rows 2 and 3 gives a further gain in columns 4 and 6 and a loss in column 5. Exchanging rows 3 and 4 gives a further gain in column 6 with no loss. No further gains are possible. The net gains are 3, 4, 5 so we choose l to be 4. The matrix pattern after the row exchanges is given in Figure 1(b) and Figure 1(c) shows the pattern after the row and column permutations.

An informal code to find l for a given k is as follows.

```
net_gain = 0
max_gain = 0
gain = (number of columns with first entry in row k)
do m = k+1, n
   gain = gain - (number of columns with first entry in row k
                    and second entry in row m)
   if (gain == 0) exit
   loss = (number of columns with first entry in row m)
   net_gain = net_gain + gain - loss
   if (net_gain > max_gain) then
      max_gain = net_gain
      l = m
   end if
end do
```

Note that `gain` is usually reduced to zero quite soon since the second entry of each column is usually near the first (it is often adjacent). It follows that the exit from the loop usually occurs quite early.

For efficient execution, it is clearly important to be able to find `gain` and `loss` quickly. We do this by using three integer arrays of length n.

`num_first`(i) holds the number of columns with their first entry in row i, $i = 1, 2, \ldots, n$.

`num_second`(i) holds the number of columns with their first entry in row k and second entry in permuted row i, $i = 1, 2, \ldots, n$.

`second`(j) holds the permuted index of the second entry of column j if the first entry is in row k and zero otherwise, $j = 1, 2, \ldots, n$.

We can calculate `num_second` efficiently for each row k by starting with all components having the value zero and restoring this afterwards. We search each column j with a first entry in row k to find its second entry `second`(j) and add one to `num_second(second`(j)`)`. Once l has been found, for each column j with a first entry in row k, we set `num_second(second`(j)`)` back to zero.

For each cyclic permutation (k, l), we also need to revise the array `num_first`. There is a change of first entry in a column only if its first entry is in row k and its second entry is not beyond row l. Therefore, for each column j with first entry in row k and `second`(j) $\leq l$, we subtract one from `num_first(inv_perm(`k`))` and add one to `num_first(inv_perm(second(`j`)))`.

It is clear that we need to be able to find the columns with their first entry in row k quickly. We therefore link all the first entries that lie in a given row. We do this by using two further integer arrays of length n.

`start`(i) holds the index of a column (if any) that has its first entry in row i, $i = 1, 2, \ldots, n$.

`next`(j) holds the index of another column (if any) that has its first entry in the same row as column j, $j = 1, 2, \ldots, n$.

It is easy to update this for a cyclic permutation (k, l). We run through the column indices j in the linked list for row k; if $\texttt{second}(j) \leq l$, we remove j from the list for row k and insert it at the front of the list for row `inv_perm(second(j))`. Inserting it at the front avoids the need to search the list.

3. IMPLEMENTATION OF THE UP EXCHANGE

For the up exchange algorithm, we again maximize the total number of zeros ahead of the first entries of the columns. As before, we need a representation of the pattern of the whole symmetric matrix, including the diagonal, and work with row permutations only.

We again hold the structure of the original matrix unchanged while the permutations are found. We again hold the current permutation and its inverse in two integer arrays `perm` and `inv_perm` of length n and update these as each cyclic permutation (k, l) is found.

When rows k and $k-1$ are exchanged, there is a gain of one for each column with its first entry in row $k-1$ and a zero in row k, and there is a loss of one for each column with its first entry in row k. For efficient coding of the algorithm, we found it necessary to regard this as a gain of one for each column with its first entry in row $k-1$ and a loss of one for each column with an entry in row k and its first entry in row k or $k-1$ (so that a column with an entry in row k and its first entry in row $k-1$ contributes a net gain of zero).

In each further interchange, row k (temporarily in row $m+1$) moves to row m and row m moves to row $m+1$, $m = k-2, k-3, \ldots, 1$. There is a further gain of one for each column with its first entry in row m and a further loss of one in each column with an entry in row k and its first entry in rows $k, \ldots, m$.

We illustrate this by the example shown in Figure 2(a) with $k = 5$. Exchanging row 5 with row 4 gives a gain in columns 4 and 6 and a loss in column 4. Following this with an exchange of row 5 (now in row 4) with row 3 gives a gain in column 3 and a loss in column 4. Next, the exchange of row 5 (now in row 3) with row 2 gives a gain in columns 2 and 5 and a loss in columns 2, 4, and 5. Finally, the exchange of row 5 (now in row 2) with row 1 gives a gain in column 1 and losses in columns 2, 4, and 5. The net gains are 1, 1, 0, -2, so we choose $l = k - 1 = 4$. The matrix after the row permutation is given in Figure 2(b) and Figure 2(c) shows the matrix after the row and column permutations.

An informal code to find l for a given k is shown in Figure 3.

To avoid unnecessary cycles of this loop, Hager holds the number of zeros in row k that have not already been used to reduce the profile, say in the variable `nzk`. The further improvement to the net gain is bounded by `nzk-loss` so the loop can be left if the inequality

$$\texttt{net\_gain} + \texttt{nzk} - \texttt{loss} <= \texttt{max\_gain}$$

holds. Unfortunately, `nzk` will start with a value near n, so that a large number of loop executions are likely to be needed.

Hager remarks that a row without any first entries can be skipped since it yields no gains and may yield losses; he uses a linked list to limit loop executions

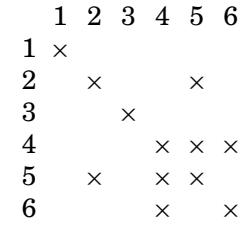

Fig. 2(a). Original matrix.

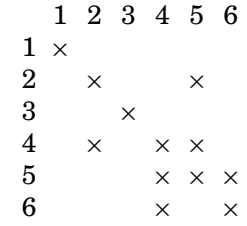

Fig. 2(b). Row permuted matrix.

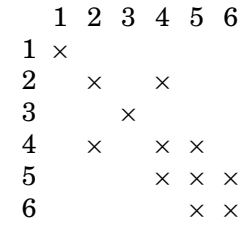

Fig. 2(c). Row and column permuted matrix.

```
net_gain = 0
max_gain = 0
loss = (number of columns with first entry in row k)
do m = k-1, 1, -1
   new_loss = (number of columns with first entry in row m
                and an entry in row k)
   loss = loss + new_loss
   gain =  (number of columns with first entry in row m)
   net_gain = net_gain + gain - loss
   if (net_gain > max_gain) then
      max_gain = net_gain
      l = m
   end if
end do
```

Fig. 3. Simple code for finding l for given k.

to rows with first entries. Since it is likely that there are many such rows, he also limits the length of each cyclic permutation to 1000.

The number of tests may be further reduced by taking advantage of the special role played by rows that have a first entry in a column that also has an entry in row k. We call these **step** rows. Their significance lies in the fact that `loss` changes only in these rows. We store the permuted indices of these rows in the array `steps` and sort them in decreasing order. To find these indices quickly we need the integer array of length n:

`first`(j) holds the index of the first entry of column j, $j = 1, 2, \ldots, n$.

To ensure that we include an interval with `loss` at its greatest (which is the number of entries in row k), we include 0 as a step row.

For efficient execution, it is important to be able to find `gain` quickly. We do this by using an array of size n:

`num_first`(i) holds number of columns with first entry in row $i, i = 1, 2, \ldots, n$.

To avoid having to cycle though the loop of Figure 3, we also hold the accumulation of `gain` over rows that are adjacent in the permuted order. We do this by using another array of size n:

`gains`(i) holds `sum(num_first(inv_perm(`$i : n$`)))`, $i = 1, 2, \ldots, n$.

For each cyclic permutation (k, l), we need to revise these arrays. There is a change of first entry in a column only if it has an entry in row k and its first entry is not in front of row l. Therefore, for each column j with an entry in row k and `perm(first(`j`))` $\geq l$, we add one to `num_first(inv_perm(`k`))`, subtract one from `num_first(first(`j`))`, and reset `first(`j`)`. Only components k, $k-1, \ldots, l+1$ of `gains` change and these can be recalculated in the loop that resets the changed components of the permutation and inverse permutation vectors.

The `loss` value is fixed for each interval between step rows, that is, for m in the range

$$p = \texttt{steps}(i) + 1 > m \geq q = \texttt{steps}(i+1) + 1$$

and the net gain is

$$\texttt{net\_gain}(m) = \texttt{net\_gain}(p) + \texttt{gains}(m) - \texttt{gains}(p) - \texttt{loss} * (p - m).$$

The inequality

$$\texttt{net\_gain}(m) \leq \texttt{net\_gain}(p) + \texttt{gains}(q) - \texttt{gains}(p) - \texttt{loss} * (p - m)$$

may be deduced. If m is to be advantageous, the inequality

$$\texttt{net\_gain}(m) > \texttt{max\_gain}$$

must hold, where `max_gain` is the best net gain found up to and including row p. A fortiori, the inequality

$$\texttt{net\_gain}(p) + \texttt{gains}(q) - \texttt{gains}(p) - \texttt{loss} * (p - m) > \texttt{max\_gain}$$

must hold; that is, the inequality

$$m > p - (\texttt{net\_gain}(p) + \texttt{gains}(q) - \texttt{gains}(p) - \texttt{max\_gain})/\texttt{loss}$$

```
      p = steps(i-1) + 1
      q = steps(i) + 1
! Search between p-1 and q
      do
        next = p - (net_gain+gains(q)-gains(p)-max_gain)/loss
        if (next >= p) exit outer
        if (next <= q) exit
        q = next
      end do
! Simple search between p-1 and q
      do p = p-1,q,-1
        net_gain = net_gain + num_first(inv_perm(p)) - loss
        if (net_gain > max_gain) then
          l = p
          max_gain = net_gain
        end if
        if (net_gain+gains(q)-gains(p)-loss <= max_gain) exit
     end do
```

Fig. 4. Looking in an interval for a better l.

must hold. This inequality may tell us immediately that there is no advantageous row in the interval. Otherwise, it gives us a new bound

$$q' = \texttt{int}(p + (\texttt{net\_gain}(p) + \texttt{gains}(q) - \texttt{gains}(p) - \texttt{max\_gain})/\texttt{loss})$$

beyond which we do not need to test. If $q' > q$, we apply the same test with q' replacing q; otherwise, we test the remaining interval directly.

Our code has an outer loop that we call `outer` within which we test a single interval as shown in Figure 4. We treat the rows in which `loss` is zero specially (they are always advantageous) and apply this algorithm to each interval in turn. The effect of using this code is dramatic; this is illustrated in Section 4.

4. NUMERICAL EXPERIMENTS

In this section, we present numerical results. Our test problems, kindly provided by Kumfert and Pothen, are listed in Table I. This set was used by Kumfert and Pothen [1997] and represents a range of application areas: structural analysis, fluid dynamics, and linear programs from stochastic optimization and multicommodity flows. All our experiments were performed on a Compaq DS20, using the Compaq Fortran 90 compiler V5.4A-1472 with the -O option. The statistic used in our tables of results is the **normalized profile** defined by

$$\frac{P}{n},$$

where n is the order of the matrix and P the profile. All CPU times are given in seconds.

In Table II, we present the normalized profiles for the original matrix, for the enhanced Sloan algorithm as implemented in the HSL code `MC60`, for the

Table I. The Test Suite

Identifier	Order	Entries	Comment
barth	6691	19748	2D CFD problem
barth4	6019	17473	2D CFD problem
barth5	15606	45878	2D CFD problem
bcsstk30	28924	1007284	3D stiffness matrix
commanche_dual	7920	11880	3D CFD problem
copter1	17222	96921	3D structural problem
copter2	55476	352238	3D structural problem
finance256	37376	130560	Linear program problem
finance512	74752	261120	Linear program problem
ford1	18728	41424	3D structural problem
ford2	100196	222246	3D structural problem
nasasrb	54870	1311227	3D structural problem
onera_dual	85567	166817	3D CFD problem
pds10	16558	66550	Linear program problem
shuttle_eddy	10429	46585	3D structural problem
skirt	45361	1268164	3D structural problem
tandem_dual	94069	183212	3D CFD problem
tandem_vtx	18454	117448	3D CFD problem

Table II. Normalized Profiles for the Original Matrix, the MC60, Hybrid, and Multilevel Orderings, and for These Orderings Followed by Five Applications of the Hager Down/Up Exchanges (+Hager). The Best Profile and Any Within 1% of It Are Highlighted in Bold

		MC60		Hybrid		Multi.	
Identifier	Original		+Hager		+Hager		+Hager
barth	2370.6	70.8	67.7	59.3	**56.8**	62.1	59.6
barth4	359.7	54.5	51.2	47.9	**45.3**	49.3	46.6
barth5	261.0	91.8	84.9	82.5	**77.6**	92.7	81.3
bcsstk30	543.4	543.4	534.2	**272.5**	**272.1**	**273.6**	**273.0**
commanche_dual	2089.8	42.3	39.3	43.8	41.9	35.1	**33.3**
copter1	1103.3	346.2	**331.3**	354.7	**331.5**	351.6	335.5
copter2	19541.8	685.2	650.9	590.7	**562.9**	698.1	654.5
finance256	6459.7	169.4	167.4	172.3	168.7	127.4	**124.4**
finance512	12859.2	158.2	156.9	156.8	152.1	**114.0**	**113.2**
ford1	1880.4	126.3	99.9	104.3	88.8	101.9	**85.2**
ford2	3715.6	407.9	334.2	358.6	299.0	304.8	**260.0**
nasasrb	371.2	346.9	344.3	351.7	347.1	**323.1**	**321.2**
onera_dual	8287.8	1025.2	933.7	545.1	**520.9**	656.0	628.8
pds10	1090.3	559.0	535.0	532.0	**513.5**	632.3	559.8
shuttle_eddy	1118.7	59.4	59.2	**57.0**	**57.0**	**57.2**	**57.0**
skirt	1011.1	808.0	807.9	**615.7**	**615.6**	657.7	657.2
tandem_dual	5183.0	701.3	626.2	448.5	432.7	420.6	**407.9**
tandem_vtx	4390.7	329.1	309.8	282.6	271.8	276.4	**268.4**

hybrid algorithm [Kumfert and Pothen 1997] and for the multilevel algorithm [Hu and Scott 2001]. For each ordering we also give the normalized profiles after applying the down exchange algorithm followed by the up exchange algorithm five times. For each problem, we highlight in bold the best profile and any others that are within 1% of the best. We see that the largest reductions in the profile result from the initial reordering; using the Hager exchange algorithms results in further more modest reductions. The size of these reductions is very problem

Table III. Normalized Profiles and CPU Times for MC60 and for Applying the Hager Exchange Algorithm to the MC60 Orderings. Numbers in Parentheses are Number of Times the Down/Up Exchange Algorithms Are Applied; Inf. Indicates No Limit. Best Profile and Any Within 1% of It Are Highlighted in Bold

	MC60		+Hager Down/Up (1)		+Hager Down/Up (5)		+Hager Down/Up (inf.)	
Identifier	Profile	Time	Profile	Time	Profile	Time	Profile	Time
barth	70.8	0.05	**68.1**	0.02	**67.7**	0.09	**67.6**	0.16
barth4	54.5	0.03	**51.3**	0.02	**51.2**	0.07	**50.8**	0.32
barth5	91.8	0.10	85.4	0.08	84.9	0.22	**82.7**	0.94
bcsstk30	543.3	0.96	**534.2**	0.41	**534.2**	1.23	**534.2**	1.23
commanche_dual	42.3	0.03	39.6	0.02	**39.3**	0.09	**39.1**	0.18
copter1	346.2	0.13	**331.3**	0.20	**331.3**	0.31	**331.3**	0.31
copter2	685.2	0.60	**653.7**	0.70	**650.9**	1.88	**650.0**	7.72
finance256	169.4	0.26	**167.4**	0.10	**167.4**	0.38	**167.4**	0.38
finance512	158.2	0.55	**156.9**	0.22	**156.9**	0.81	**156.0**	0.82
ford1	126.3	0.09	105.5	0.14	99.9	0.36	**97.5**	1.41
ford2	407.9	0.71	349.3	2.71	334.2	5.14	**328.9**	15.31
nasasrb	**346.0**	1.29	**344.3**	0.55	**344.3**	2.80	**344.3**	2.80
onera_dual	1025.2	0.65	955.0	3.39	933.7	6.62	**910.0**	28.13
pds10	559.0	0.13	**536.6**	0.10	**535.5**	0.35	**535.0**	0.48
shuttle_eddy	**59.4**	0.06	**59.2**	0.02	**59.2**	0.11	**59.2**	0.37
skirt	**808.0**	1.28	**807.8**	0.56	**807.9**	1.69	**807.9**	1.69
tandem_dual	701.3	0.66	650.1	2.11	626.2	4.90	**611.0**	17.34
tandem_vtx	329.1	0.15	315.7	0.11	309.8	0.44	**305.5**	2.29

dependent. For a number of problems, including bcsstk30, shuttle_eddy, and skirt, the improvements are less than 1%. However, the reductions can be more significant. For example, for ford1 and ford2, the MC60 profiles are improved by 21% and 18%, respectively. We remark that it is important to have a good initial reordering before the application of Hager's exchange algorithms. For our test cases, the smallest profiles were obtained after applying Hager's algorithm to the best initial reordering.

In Table III, we present the normalized profiles and the CPU times for applying the exchange algorithms to the MC60 orderings. Results are given for a single application of Hager down/up, for repeating the down/up exchanges five times, and for repeating the down/up exchanges without limit until there is no further reduction in the profile. For each problem, we again use bold for the best profile and any others that are within 1% of the best. The greatest reductions result from the first application of the exchange algorithm, although for a number of problems, including ford2 and tandem_dual, useful further reductions are achieved by repeatedly applying the exchange algorithm. On the basis of these results, by default we limit the number of down/up exchanges to five.

Our final implementation of the up exchange algorithm has had a huge effect on the CPU time, and we illustrate this in Table IV. We denote by Simple the implementation based on the Figure 3 code; Simple(1000) denotes combining this with Hager's proposed limit of 1000 on the length of each cylic permutation; New denotes our final implementation that uses the Figure 4 code. In each case, the Hager down exchange followed by the up exchange is applied five times to the ordering obtained from MC60. For comparison, timings are also given for MC60.

Table IV. CPU Times for MC60 and for Five Applications of the Hager Down/Up Exchanges. Simple is Based on the Figure 3 Code; Simple(1000) Uses the Figure 3 Code with a Limit of 1000 on the Length of Each Cyclic Permutation; New Uses the Figure 4 Code

Identifier	MC60	Simple	Simple(1000)	New
barth	0.05	0.78	0.71	0.09
barth4	0.03	0.57	0.55	0.07
barth5	0.10	4.32	1.82	0.22
bcsstk30	0.96	2.11	3.33	0.09
commanche_dual	0.03	1.52	0.81	0.09
copter1	0.13	3.24	1.53	0.31
copter2	0.60	40.31	7.86	1.88
finance256	0.26	19.26	3.27	0.38
finance512	0.55	78.64	6.73	0.81
ford1	0.09	7.38	2.13	0.36
ford2	0.71	256.07	15.04	5.14
nasasrb	1.29	10.33	7.46	2.80
onera_dual	0.65	237.05	15.98	6.62
pds10	0.13	6.12	2.12	0.35
shuttle_eddy	0.06	1.23	1.03	0.11
skirt	1.28	4.97	3.71	1.69
tandem_dual	0.66	247.55	14.78	4.90
tandem_vtx	0.15	3.68	2.16	0.44

We observe that, for the larger problems, Simple(1000) is much faster than Simple. The savings for problems ford2, nasasrb, and tandem_dual are particularly impressive. However, limiting the length of the cyclic permutation does not necessarily result in a reduction in the CPU time when the Hager down/up exchanges are applied more than once; this is illustrated by problem bcsstk30. Furthermore, we found that it may result in a slight loss of quality. For example, for problem ford2, Simple reduces the MC60 profile by 18% whereas the reduction with Simple(1000) is 16%. Simple and New produce the same profiles but the performance of New is a dramatic improvement over that of Simple. We did experiment with limiting the length of the cyclic permutation within New but decided against incorporating such a limit because of the possibility of larger profiles and, with the efficiency of New, the CPU time savings were small.

Table V presents results for three problems in greater detail. For each pass, we record the total reduction in the normalized profile achieved so far, the percentage of the first reduction that the pass achieved, and the CPU time taken by the pass. The problems were chosen to illustrate the different behaviors we observed. In each case, the first application of the down exchange gave the largest profile reduction. For pds10, applying the up exchange gave a further improvement of less than 2% of the first reduction and stagnation was quickly reached. For commanche_dual, the first up exchange gave a 3% reduction and most of the reduction resulting from repeated applications of the Hager algorithms came from the down exchanges. The first up exchange for ford2 gave a reduction of about 36%. Thereafter the improvements rapidly declined to less than 1% but a total of 30 iterations was required until there was no improvement at all. Regarding CPU times, after the first down/up exchange, the CPU times

Table V. Reductions in Normalized Profiles Following MC60 with Percentages of the First Reduction and CPU Times Taken for Each Hager Pass

		commanche_dual			ford2			pds10		
		Reduction			Reduction			Reduction		
Iteration		Total	%	Time	Total	%	Time	Total	%	Time
1	down	2.593	96.7	0.013	37.272	63.6	1.357	22.064	98.8	0.060
	up	2.682	3.3	0.011	58.606	36.4	1.444	22.326	1.2	0.041
2	down	2.820	5.1	0.006	63.464	8.3	0.472	23.761	6.5	0.025
	up	2.846	1.0	0.010	65.226	3.0	0.327	23.842	0.4	0.040
3	down	2.962	4.3	0.006	67.794	4.4	0.302	23.906	0.3	0.024
	up	2.978	0.6	0.011	69.377	2.7	0.316	23.908	0.0	0.040
4	down	3.046	2.5	0.005	70.981	2.7	0.285	23.910	0.0	0.023
	up	3.050	0.1	0.011	71.866	1.5	0.269	23.910	0.0	0.040
5	down	3.060	0.4	0.006	72.900	1.8	0.231	23.910	0.0	0.023
	up	3.065	0.2	0.011	73.679	1.3	0.262	23.910	0.0	0.040
6	down	3.072	0.3	0.005	74.513	1.4	0.229	23.910	0.0	0.023
	up	3.074	0.1	0.011	75.136	1.1	0.227	23.911	0.0	0.040
7	down	3.131	2.1	0.006	75.739	1.0	0.251	23.911	0.0	0.023
	up	3.141	0.4	0.011	76.182	0.8	0.233	23.911	0.0	0.037
8	down	3.147	0.2	0.005	76.669	0.8	0.198			
	up	3.147	0.0	0.011	76.958	0.5	0.231			
9	down	3.155	0.3	0.005	77.366	0.7	0.206			
	up	3.156	0.0	0.011	77.629	0.4	0.225			
10	down	3.160	0.1	0.006	77.785	0.3	0.193			
	up	3.160	0.0	0.011	78.047	0.4	0.238			
11	down	3.160	0.0	0.005	78.219	0.3	0.186			
	up	3.160	0.0	0.011	78.295	0.1	0.213			
12	down				78.330	0.1	0.183			
	up				78.496	0.3	0.211			
13	down				78.591	0.2	0.183			
	up				78.609	0.0	0.208			
14	down				78.640	0.1	0.183			
	up				78.672	0.1	0.212			
15	down				78.724	0.1	0.188			
	up				78.737	0.0	0.211			
20	down				78.869	0.2	0.180			
	up				78.883	0.0	0.220			
25	down				78.884	0.0	0.178			
	up				78.885	0.0	0.207			
30	down				78.912	0.0	0.180			
	up				78.912	0.0	0.207			

for each exchange quickly became constant, with each up exchange generally taking less than twice as long as the corresponding down exchange.

Our aim in this article is to discuss the implementation of Hager's methods per se, but the referees have asked us to place our timings in the context of likely savings during actual factorization. We have therefore run the HSL code HSL_MA55 on the problems in Table III for which the normalized profile was less than 500 and the profile gain with up to five exchanges was at least 2% over MC60. We used the same computer and the same optimization level for the compilation. HSL_MA55 is a variable-band code that groups the rows into blocks in order to use optimized level-3 BLAS [Dongarra et al. 1990] and works out of memory, if necessary. The blocking is necessary for performance, but means

Table VI. HSL_MA55 Performance on Seven of the Test Problems. Figures in Parentheses Are Factorization Times Using Ordering Produced by Five Hager Down/Up Exchanges

	MC60		+Hager Down/Up (5)		HSL_MA55 nb=1	HSL_MA55 Best nb	
Identifier	Profile	Time	Profile	Time	Time saved	Time saved	nb
barth	70.8	0.05	67.7	0.09	0.21 (3.29)	0.10 (1.06)	4
barth4	54.5	0.03	51.2	0.07	0.14 (2.06)	0.04 (0.51)	8
barth5	91.8	0.10	84.9	0.22	0.97 (9.99)	0.28 (3.14)	4
copter1	346.2	0.13	331.3	0.31	5.62 (86.5)	18.0 (29.7)	16
ford1	126.3	0.09	99.9	0.36	4.15 (9.94)	0.23 (6.31)	4
ford2	407.9	0.71	334.2	5.14	201 (550)	−78.7 (396)	8
tandem_vtx	329.1	0.15	309.8	0.44	7.72 (63.8)	3.17 (35.2)	8

that some zeros that lie outside the profile are stored and operated upon. As the block size nb is increased, the number of flops (floating-point operations) increases but the flop rate also increases. We tried block sizes of 1, 2, 4, 8, and 16, and show in Table VI the savings in HSL_MA55 factorize times that the Hager exchanges give for nb=1 and the nb value for which the least factorize time was found. It may be seen that for all of these problems there is a saving with nb=1 that is greater than the Hager ordering time, sometimes substantially so. The saving is less for the best choice of nb and there is a substantial loss in one case (ford2). This loss is associated with a normalized block profile that increased from 954.8 to 1058.5. Nevertheless, for two problems (copter1 and tandex_vtx), the saving with the best nb is substantially greater than the Hager ordering time. Of course, it would be desirable to have a means of minimizing the block profile instead of the profile itself. We plan to look at this in the future.

Note that where many matrices with the same pattern are to be solved (quite a usual situation in applications), the balance is quite different since the reordering needs to be done only once. After the first factorization, each saving is available at no cost.

5. CONCLUDING REMARKS

We have examined in detail how to efficiently implement the exchange algorithms of Hager. In particular, we have described how to implement the up exchanges so that their cost is no longer prohibitive. Our implementations have turned what are interesting profile reduction algorithms into practical algorithms that can be used to refine existing orderings. Based on our findings, we plan to include our codes in HSL as routine MC67. MC67 will allow the user to apply the Hager down/up exchanges to any given ordering; in particular, the user interface will be designed so that it will be straightforward for the user to run the exchange algorithms to refine the ordering produced by the HSL code MC60. The limit on the number of iterations will be a parameter under the user's control. We will also allow the user to specify that the code should be left if the profile reduction in an iteration is less than a chosen percentage of the reduction in the first iteration (see the long convergence tails in Table V).

ACKNOWLEDGMENTS

We would like to thank the three anonymous referees for their constructive comments, which have led to several improvements to the article.

REFERENCES

BARNARD, S., POTHEN, A., AND SIMON, H. 1995. A spectral algorithm for envelope reduction of sparse matrices. *Numer. Linear Algebra Appl. 2*, 317–198.

CUTHILL, E. AND MCKEE, J. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM*. Brandon Systems, New York.

DEMMEL, J., GILBERT, J., AND LI, S. 1999. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. Appl. 20*, 915–952.

DONGARRA, J., DUCROZ, J., DUFF, I., AND HAMMARLING, S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1–17.

DUFF, I. AND REID, J. 1983. The multifrontal solution of unsymmetric sets of linear equations. *ACM Trans. Math. Softw. 9*, 302–325.

DUFF, I., REID, J., AND SCOTT, J. 1989. The use of profile reduction algorithms with a frontal code. *Int. J. Numer. Meth. Eng. 28*, 2555–2568.

GEORGE, A. AND LIU, J.-H. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, N.J.

GIBBS, N. 1976. A hybrid profile reduction algorithm. *ACM Trans. Math. Softw. 2*, 378–387.

HAGER, W. 2000. Minimizing the profile of a matrix. Department of Mathematics, University of Florida (www.math.ufl.edu/∼hager/). *SIAM J. Sci. Comput*. (to appear).

HU, Y. AND SCOTT, J. 2001. A multilevel algorithm for wavefront reduction. *SIAM J. Sci. Comput. 23*, 1352–1375.

KUMFERT, G. AND POTHEN, A. 1997. Two improved algorithms for envelope and wavefront reduction. *BIT 18*, 559–590.

LEWIS, J. 1982. Implementation of the Gibbs–Poole–Stockmeyer and Gibbs–King algorithms. *ACM Trans. Math. Softw. 8*, 180–189.

PAULINO, G., MENEZES, I., GATTASS, M., AND MUKHERJEE, S. 1994a. Node and element resequencing using the Laplacian of a finite element graph: Part I—General concepts and algorithm and numerical results. *Int. J. Numer. Meth. Eng. 37*, 1531–1555.

PAULINO, G., MENEZES, I., GATTASS, M., AND MUKHERJEE, S. 1994b. Node and element resequencing using the Laplacian of a finite element graph: Part II—Implementation and numerical results. *Int. J. Numer. Meth. Eng. 37*, 1531–1555.

REID, J. AND SCOTT, J. 1999. Ordering symmetric sparse matrices for small profile and wavefront. *Int. J. Numer. Meth. Eng. 45*, 1737–1755.

SLOAN, S. 1986. An algorithm for profile and wavefront reduction of sparse matrices. *Int. J. Numer. Meth. Eng. 23*, 1315–1324.

SLOAN, S. 1989. A FORTRAN program for profile and wavefront reduction. *Int. J. Numer. Meth. Eng. 28*, 2651–2679.

Received November 2001; revised April 2002; accepted July 2002