RAL-TR-1999-030

On algorithms for permuting large entries to the diagonal of a sparse matrix

1

Iain S. Du(cid:11)

and Jacko Koster

2

3

ABSTRACT

We consider bipartite matching algorithms for computing permutations of a sparse matrix so that the diagonal of the permuted matrix has entries of large absolute value. We discuss various strategies for this and consider their implementation as computer codes. We also consider scaling techniques to further increase the relative values of the diagonal entries. Numerical experiments show the e(cid:11)ect of the reorderings and the scaling on the solution of sparse equations by a direct method and by an iterative technique. The e(cid:11)ect on preconditioning for iterative methods is also discussed.

Keywords: sparse matrices, bipartite weighted matching, Dijkstra's algorithm, direct methods, iterative methods, preconditioning.

AMS(MOS) sub ject classi(cid:12)cations: 65F05, 65F50.

1

Current

reports available by anonymous

ftp to ftp.numerical.rl.ac.uk in directory

pub/reports. This report is in (cid:12)le dukoRAL99030.ps.gz. Report also available through

URL http://www.numerical.rl.ac.uk/reports/reports.html. Also published as Technical Report

TR/PA/99/13 from CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France.

I.Du(cid:11)@rl.ac.uk

2

3

J.Koster@rl.ac.uk

Computational Science and Engineering Department

Atlas Centre

Rutherford Appleton Laboratory

Oxon OX11 0QX

April 19, 1999.

Contents

1 Introduction

We say that an n (cid:2) n matrix A has a large diagonal if the absolute value of each diagonal entry is large relative to the absolute values of the o(cid:11)-diagonal entries in its row and column. Permuting large nonzero entries onto the diagonal of a sparse matrix can be

useful in several ways. If we wish to solve the system

$$Ax = b; \tag{1.1}$$

where A is a nonsingular square matrix of order n and x and b are vectors of length n, then a preordering of this kind can be useful whether direct or iterative methods are used for solution (see Olschowka and Neumaier (1996) and Du(cid:11) and Koster (1997)).

The work in this report is a continuation of the work reported by Du(cid:11) and Koster (1997) who presented an algorithm that maximizes the smallest entry on the diagonal and relies on repeated applications of the depth (cid:12)rst search algorithm MC21 (Du(cid:11) 1981) in the Harwell Subroutine Library (HSL 1996). In this report, we will be concerned with other bipartite matching algorithms for permuting the rows and columns of the matrix so that the diagonal of the permuted matrix is large. The algorithm that is central to this report computes a matching that corresponds to a permutation of a sparse matrix such that the product (or sum) of the diagonal entries is maximized. This algorithm is already mentioned and used in Du(cid:11) and Koster (1997), but is not fully described. In this report, we describe the algorithm in more detail. We also consider a modi(cid:12)ed version of this algorithm to compute a permutation of the matrix that maximizes the smallest diagonal entry. We compare the performance of this algorithm with that of Du(cid:11) and Koster (1997). We also investigate the in(cid:13)uence of scaling of the matrix. Scaling can be used before or after computation of the matching to make the diagonal entries even larger relative to the o(cid:11)-diagonals. In particular, we look at a sparse variant of a bipartite matching and

scaling algorithm of Olschowka and Neumaier (1996) that (cid:12)rst maximizes the product of the diagonal entries and then scales the matrix so that these entries are one and all other entries are no greater than one.

The rest of this report is organized as follows. In Section 2, we describe some concepts of bipartite matching that we need for the description of the algorithms. In Section 3, we review the basic properties of algorithm MC21. MC21 is a relatively simple algorithm that computes a matching that corresponds to a permutation of the matrix that puts as many entries as possible onto the diagonal without considering their numerical values. The algorithm that maximizes the product of the diagonal entries is described in Section 4. In Section 5, we consider the modi(cid:12)ed version of this algorithm that maximizes the smallest diagonal entry of the permuted matrix. In Section 6, we consider the scaling of the reordered matrix. Computational experience for the algorithms applied to some practical problems and the e(cid:11)ect of the reorderings and scaling on direct and iterative methods of

1

solution are presented in Sections 7 to 7.2. The e(cid:11)ect on preconditioning is also discussed. Finally, we consider some of the implications of this current work in Section 8.

2 Bipartite matching

Let A = (a

) be a general n (cid:2) n sparse matrix. With matrix A, we associate a bipartite

ij

graph G

= (V

; V

; E) that consists of two disjoint node sets V

and V

and an edge set

A

r

c

r

c

E , where (u; v) 2 E implies that u 2 V

, v 2 V

. The sets V

and V

have cardinality n and

r

c

r

c

correspond to the rows and columns of A respectively. Edge (i; j) 2 E if and only if a

6= 0.

ij

We de(cid:12)ne the sets ROW (i) = fj j(i; j) 2 Eg, for i 2 V

, and COL(j) = fij(i; j) 2 Eg, for

r

j 2 V

. These sets correspond to the positions of the entries in row i and column j of the

c

sparse matrix respectively. We use j : : : j both to denote the absolute value and to signify

the number of entries in a set, sequence, or matrix. The meaning should always be clear

from the context.

A subset M (cid:18) E is called a matching (or assignment) if no two edges of M are incident

to the same node. A matching containing the largest number of edges possible is called a

maximum cardinality matching (or simply maximum matching). A maximum matching

is a perfect matching if every node is incident to a matching edge. Obviously, not every

bipartite graph allows a perfect matching. However, if the matrix A is nonsingular, then

there exists a perfect matching for G

. A perfect matching M has cardinality n and

A

de(cid:12)nes an n (cid:2) n permutation matrix P = (p

) with

ij

(

p

= 1;

for (i; j) 2 M ;

j i

p

= 0; otherwise;

j i

so that both P A and AP are matrices with the matching entries on the (zero-free) diagonal.

Bipartite matching problems can be viewed as a special case of network (cid:13)ow problems (see,

for example, Ford Jr. and Fulkerson (1962)).

The more e(cid:14)cient algorithms for (cid:12)nding maximum matchings in bipartite graphs make

use of augmenting paths. Let M be a matching in G

. A node v is matched if it is incident

A

to an edge in M . A path P in G

is de(cid:12)ned as an ordered set of edges in which successive

A

edges are incident to the same node. A path P is called an M -alternating path if the

edges of P are alternately in M and not in M . An M -alternating path P is called an M -

augmenting path if it connects an unmatched row node with an unmatched column node.

In the bipartite graph in Figure 2.1, there exists an M -augmenting path from column node

8 to row node 8. The matching M (of cardinality 7) is represented by the thick edges. The

black entries in the accompanying matrix correspond to the matching and the connected

matrix entries to the M -augmenting path. If it is clear from the context which matching

2

M is associated with the M -alternating and M -augmenting paths, then we will simply

refer to them as alternating and augmenting paths.

Let M and P be subsets of E . We de(cid:12)ne

M (cid:8) P := (M n P) [(P n M):

If M is a matching and P is an M -augmenting path, then M (cid:8) P is again a matching,

and jM (cid:8) P j = jM j + 1. If P is an M -alternating cyclic path, i.e., an alternating path

whose (cid:12)rst and last edge are incident to the same node, then M (cid:8) P is also a matching

and jM (cid:8) P j = jM j.

Figure 2.1: Augmenting path

Vr

Vc

1

2

3

4

5

9

1

2

3

4

5

6

7

8

9

In the sequel, a matching M will often be represented by a pointer array m : V

[V

!

r

c

V

[V

[fnullg with

r

c

(

m

= j and m

= i;

for (i; j) 2 M ;

i

j

m

= null;

for i unmatched:

i

Augmenting paths in a bipartite graph G can be found by constructing alternating

trees. An alternating tree T = (T

; T

; E

) is a subgraph of G rooted at a row or column

r

c

T

node and each path in T is an M -alternating path. An alternating tree rooted at a

column node j

can be grown in the following way. We start with the initial alternating

0

tree (;; fj

g; ;) and consider all the column nodes j 2 T

in turn. Initially j = j

. For

0

c

0

each node j , we check the row nodes i 2 COL(j) for which an alternating path from i

to j

does not yet exist. If node i is already matched, we add row node i, column node

0

3

m

, and edges (i; j) and (i; m

) to T . If i is not matched, we extend T by row node i and

i

i

edge (i; j) (and the path in T from node i to the root forms an augmenting path). A key

observation for the construction of a maximum or perfect matching is that a matching M

is maximum if and only if there is no augmenting path relative to M .

Alternating trees can be implemented using a pointer array p : V

! V

such that,

c

c

given an edge (i; j) 2 E

n M , node j is either the root node of the tree, or the edges

T

(i; j), (m

; j), and (m

; p

) are consecutive edges in an alternating path towards the root.

j

j

j

Augmenting paths in an alternating tree (provided they exist) can thus easily be obtained

from p and m.

Alternating trees are not unique.

In general, one can construct several alternating

trees starting from the same root node that have equal node sets, but di(cid:11)erent edge

sets. Di(cid:11)erent alternating trees in general will contain di(cid:11)erent augmenting paths. The

matching algorithms that we describe in the next sections impose di(cid:11)erent criteria on the order in which the paths in the alternating trees are grown in order to obtain augmenting paths and maximum matchings with special properties.

3 Matching

The asymptotically fastest currently known algorithm for (cid:12)nding a maximum matching is

$$p$$

by Hopcroft and Karp (1973). It has a worst-case complexity of $O(\sqrt{n}(cid:28))$, where $(cid:28) = jE j$ is the number of entries in the sparse matrix. An e(cid:14)cient implementation of this algorithm can be found in Du(cid:11) and Wiberg (1988). The algorithm MC21 implemented by Du(cid:11) (1981) has a theoretically worst-case behaviour of $O(n(cid:28))$, but in practice it behaves more like $O(n + (cid:28))$. Because this latter algorithm is simpler, we concentrate on this in the following although we note that it is relatively straightforward to use the algorithm of Hopcroft and Karp (1973) in a similar way to how we will use MC21 in later sections.

MC21 is a depth-(cid:12)rst search algorithm with look-ahead. It starts o(cid:11) with an empty matching M, and hence all column nodes are unmatched initially. See Figure 3.1. For each unmatched column node j in turn, an alternating tree is grown until an augmenting

$$0$$

path with respect to the current matching M is found (provided one exists). A set B is used to mark all the matched row nodes that have been visited so far. Initially, $B = ;$.

First, the row nodes in COL(j

) are searched (look-ahead) for an unmatched node i

.

0

0

If one is found, the singleton path P = f(i

; j

)g is an M -augmenting path. If there is

0

0

no such unmatched node, then an unmarked matched node i

2 COL(j

) is chosen, i

0

0

0

is marked, the nodes i

and j

, j

= m

, and the edges (i

; j

), (i

; j

) are added to

0

1

1

i

0

0

0

1

0

the alternating tree (by setting p

= j

). The search then continues with column node

j

0

1

j

. For node j

, the row nodes in COL(j

) are (cid:12)rst checked for an unmatched node.

1

1

1

If one exists, say i

, then the path P = f(i

; j

); (i

; j

); (i

; j

)g forms an augmenting

1

0

0

0

1

1

1

4

path. If there is no such unmatched node, a remaining unmarked node i

is picked from

1

COL(j

), i

is marked, p

is set to j

, j

= m

, and the search moves to node j

.

1

1

j

2

1

2

i

2

1

This continues in a similar (depth-(cid:12)rst search) fashion until either an augmenting path

P = f(i

; j

); (i

; j

); (i

; j

); : : : ; (i

; j

)g is found (with nodes j

and i

unmatched) or

0

0

0

1

1

1

k

k

0

k

until for some k > 0, COL(j

) does not contain an unmarked node. In the latter case,

k

MC21 backtracks by resuming the search at the previously visited column node j

for

k(cid:0)1

some remaining unmarked node i

2 COL(j

). Backtracking for k = 0 is not possible;

k(cid:0)1

k(cid:0)1

0

if MC21 resumes the search at column node j

and COL(j

) does not contain an unmarked

0

0

node, then an M -augmenting path starting at node j

does not exist. In this case, MC21

0

continues with the construction of a new alternating tree starting at the next unmatched

column node. (The (cid:12)nal maximum matching will have cardinality at most n (cid:0) 1 and hence

will not be perfect.)

Figure 3.1: Outline of MC21.

for j

2 V

do

0

c

j := j

; p

:= null; iap := null;

0

j

B := ;;

repeat

if there exists i 2 COL(j) and i is unmatched then

iap := i;

else

if there exists i 2 COL(j) n B then

B := B + fig;

p

:= j ;

m

i

j := m

;

i

else

j := p

;

j

end if;

end if;

until iap 6= null or j = null;

if iap 6= null then augment along path from node iap to node j

;

0

end for

5

4 Weighted matching

In this section, we describe an algorithm that computes a matching for permuting a sparse

matrix A such that the product of the diagonal entries of the permuted matrix is maximum

in absolute value. That is, the algorithm determines a matching that corresponds to a

permutation (cid:27) that maximizes

n

Y

i=1

ja

j:

(4.1)

i(cid:27)

i

This maximization multiplicative problem can be translated into a minimization

additive problem by de(cid:12)ning matrix C = (c

) as

ij

(

c

=

ij

log a

(cid:0) log ja

j; a

6= 0;

j

ij

ij

1;

otherwise;

where a

= max

ja

j is the maximum absolute value in column j of matrix A. Maximizing

j

i

ij

(4.1) is equal to minimizing

Q

Q

n

n

n

n

n

X

X

X

i=1

i=1

i

i

(cid:27)

a

a

log

Q

= log

=

log a

(cid:0)

log ja

j =

(log a

(cid:0) log ja

j) =

Q

n

n

(cid:27)

i

i(cid:27)

(cid:27)

i(cid:27)

i

i

i

i=1

i

i=1

i

i(cid:27)

i(cid:27)

ja

j

ja

j

i=1

i=1

i=1

n

X

i=1

c

:

(4.2)

i(cid:27)

i

Minimizing (4.2) is equivalent to (cid:12)nding a minimum weight perfect matching in an edge weighted bipartite graph. This is known in literature as the bipartite weighted matching problem or (linear sum) assignment problem in linear programming and combinatorial optimization. Numerous algorithms have been proposed for computing minimum weight perfect matchings, see for example Burkard and Derigs (1980), Carpaneto and Toth (1980), Carraresi and Sodini (1986), Derigs and Metz (1986), Jonker and Volgenant (1987), and Kuhn (1955). A practical example of an assignment problem is the allotment of tasks to people; entry c

in the cost matrix C represents the cost or bene(cid:12)t of assigning person i

ij

to task j .

Let C = (c

) be a real-valued n (cid:2) n matrix, c

(cid:21) 0. Let G

= (V

; V

; E) be the

ij

ij

C

r

c

corresponding bipartite graph each of whose edges (i; j) 2 E has weight c

. The weight

ij

of a matching M in G

, denoted by c(M), is de(cid:12)ned by the sum of its edge weights, i.e.,

C

X

c(M) =

c

:

ij

(i;j)2M

A perfect matching M is said to be a minimum weight perfect matching if it has smallest

possible weight, i.e., c(M) (cid:20) c(M

), for all possible maximum matchings M

.

0

0

6

The key concept for (cid:12)nding a minimum weight perfect matching is the so-called shortest

augmenting path. An M -augmenting path P starting at an unmatched column node j is

called shortest if c(M (cid:8) P) (cid:20) c(M (cid:8) P

), for all other possible M -augmenting paths P

0

0

starting at node j . We de(cid:12)ne

l(P) := c(M (cid:8) P) (cid:0) c(M) = c(P n M) (cid:0) c(M \ P)

as the length of alternating path P . A matching M is called extreme if and only if it does

not allow any alternating cyclic path with negative length.

The following two relations hold. First, a perfect matching has minimum weight if it

is extreme. Second, if matching M is extreme and P is a shortest M -augmenting path,

then M (cid:8) P is extreme also. The proof for this goes roughly as follows. Suppose M (cid:8) P is

not extreme. Then there exists an alternating cyclic path Q such that c((M (cid:8) P) (cid:8) Q) <

c(M (cid:8) P). Since (M (cid:8) P) (cid:8) Q = M (cid:8) (P (cid:8) Q) and M is extreme, there must exist a

subset P

(cid:18) P (cid:8) Q that forms an M -augmenting path and is shorter than P . Hence, P is

0

not a shortest M -augmenting path. This contradicts the supposition.

These two relations form the basis for many algorithms for solving the bipartite

weighted matching problem: start from any (possibly empty) extreme matching M and

successively augment M along shortest augmenting paths until M is maximum (or perfect).

In the literature, the problem of (cid:12)nding a minimum weight perfect matching is often

stated as the following linear programming problem. Find matrix X = (x

) 2 R

,

ij

n(cid:2)n

minimizing

sub ject to

X

c

x

ij

ij

(i;j)2E

X

j2V

c

X

i2V

r

x

= 1;

for i 2 V

;

ij

r

x

= 1;

for j 2 V

;

ij

c

x

(cid:21) 0;

for (i; j) 2 E ;

ij

x

= 0;

for (i; j) 62 E :

ij

If there is a solution to this linear program, there is one for which x

2 f0; 1g and there

ij

exists a permutation matrix X such that M = f(i; j)jx

= 1g is a minimum weight perfect

ij

matching (Edmonds and Karp 1972, Kuhn 1955). Furthermore, M has minimum weight

if and only if there exist dual variables u

and v

with

i

j

(

u

+ v

(cid:20) c

;

for (i; j) 2 E ;

i

j

ij

u

+ v

= c

;

for (i; j) 2 M :

i

j

ij

(4.3)

7

Using the reduced weight matrix C = (c

), with

ij

c

:= c

(cid:0) u

(cid:0) v

(cid:21) 0;

ij

ij

i

j

the reduced weight c(M) of matching M equals

c(M) = 0;

the reduced length l(P) of any M -alternating path P equals

X

l(P) =

c

(cid:21) 0;

ij

(i;j)2P nM

and if M (cid:8) P is a matching, the reduced weight of M (cid:8) P equals

c(M (cid:8) P) = l(P):

Thus, (cid:12)nding a shortest augmenting path in graph G

is equivalent to (cid:12)nding an

C

augmenting path in graph G

, with minimum reduced length. Since c

= 0 for every

C

ij

edge (i; j) 2 M and graph G

contains no alternating paths P with negative length,

C

0

0

l(P

) (cid:20) l(P) for every principal leading subpath P

of P .

Shortest augmenting paths in a weighted bipartite graph G = (V

; V

; E) can be

r

c

obtained by means of a shortest alternating path tree. A shortest alternating path tree T

is an alternating tree each of whose paths is a shortest path in G. For any node i 2 V

[V

,

r

c

we de(cid:12)ne d

as the length of the shortest path in T from node i to the root node (d

= 1

i

i

if no such path exists). T is a shortest alternating path tree if and only if d

+ c

(cid:21) d

,

i

ij

j

for every edge (i; j) 2 E and tree nodes i, j ,

An outline of an algorithm for constructing a shortest alternating path tree rooted at

column node j

is given in Figure 4.1. Because the reduced weights c

are non-negative,

0

ij

and graph G

contains no alternating paths with negative length, we can use a sparse

C

variant of Dijkstra's algorithm (Dijkstra 1959). The set of row nodes is partitioned into

three sets B , Q, and W . B is the set of (marked) nodes whose shortest alternating paths

and distances to node j

are known. Q is the set of nodes for which an alternating path

0

to the root is known that is not necessarily the shortest possible. W is the set of nodes for

which an alternating path does not exist or is not known yet. (Note that since W is de(cid:12)ned

implicitly as V

n (B [Q), it is not actually used in Figure 4.1.) The algorithm starts with

r

(;; fj

g; ;) as initial shortest alternating tree and extends the tree until an augmenting

0

path is found that is guaranteed to be a shortest augmenting path with respect to the

current matching M . Initially, the length of the shortest augmenting path lsap in the tree

is set to in(cid:12)nity, and the length of the shortest alternating path lsp from the root to any

node in Q is set to zero. On each pass through the main loop, another column node j is

chosen that is closest to the root j

. Initially j = j

.

0

0

8

Each row node i 2 COL(j) whose shortest alternating path to the root is not known

yet (i 62 B), is considered. If P

, the shortest alternating path from the root node

j

!j!i

0

j

to node j (with length lsp) extended by edge (i; j) from node j to node i (with length

0

c

), is longer than the tentative shortest augmenting path in the tree (with length lsap),

ij

then there is no need to modify the tree. If P

has length smaller than lsap, and i

j

!j!i

0

is unmatched, then a new shorter augmenting path has been found and lsap is updated.

If i is matched and P

is also shorter than the current shortest alternating path to

j

!j!i

0

i (with length d

), then a shorter alternating path to node i has been found and the tree

i

is updated, d

is updated, and if node i has not been visited previously, i is moved to Q.

i

Next, if Q is not empty, a node i 2 Q is determined that is closest to the root. Since all

weights c

in the bipartite graph are non-negative, there cannot be any other alternating

ij

path to node i that is shorter than the current one. Node i is marked (by adding it to

B), and the search continues with column node j

= m

. This continues until there are

i

0

no more column nodes to be searched (Q = ;), or until no new augmenting path can be

found whose length is smaller than the current shortest one (line lsap (cid:20) lsp).

The original Dijkstra algorithm (intended for dense graphs) has O(n

) complexity.

2

For sparse problems, the complexity can be reduced to O((cid:28) log n) by implementing the

set Q as a k-heap in which the nodes i are sorted by increasing distance d

from the root

i

(see for example Tarjan (1983) and Gallo and Pallottino (1988)). The running time of

the algorithm is dominated by the operations on the heap Q of which there are O(n)

delete operations, O(n) insert operations, and O((cid:28)) modi(cid:12)cation operations (these are

necessary each time a distance d

is updated). Each insert and modi(cid:12)cation operation

i

runs in O(log

n) time, a delete operation runs in O(k log

n) time. Consequently, the

k

k

algorithm for (cid:12)nding a shortest augmenting path in a sparse bipartite graph has run time

O(((cid:28) + kn) log

n) and the total run time for the sparse bipartite weighted algorithm is

k

O(n((cid:28) + kn) log

n). If we choose k = 2, the algorithm uses binary heaps and we obtain a

k

time bound of O(n((cid:28) + n) log

n). If we choose k = d(cid:28) =ne (and k (cid:21) 2), we obtain a bound

2

of O(n(cid:28) log

n).

(cid:28) =n

The implementation of the heap Q is similar to the implementation proposed in Derigs

and Metz (1986). Q is a pair (Q

; Q

) where Q

is an array that contains all the row

1

2

1

nodes for which the distance to the root is shortest (lsp), and Q

= Q n Q

is a 2-heap.

2

1

By separating the nodes in Q that are closest to the root, we may reduce the number

of operations on the heap, especially in those situations where the cost matrix C has

only few di(cid:11)erent numerical values and many alternating paths have the same length.

Deleting a node from Q for which d

is smallest (see Figure 4.1), now consists of choosing

i

an (arbitrary) element from Q

. If Q

is empty, then we (cid:12)rst move all the nodes in Q

1

1

2

that are closest to the root to Q

.

1

After the augmentation, the reduced weights c

have to be updated to ensure that

ij

alternating paths in the new G have non-negative length. This is done by modifying the

9

Figure 4.1: Construction of a shortest augmenting path.

B := ;; Q := ;;

for i 2 V

do d

:= 1;

r

i

lsp := 0;

lsap := 1;

j := j

; p

:= null;

0

j

while true do

for i 2 COL(j) n B do

dnew := lsp + c

;

ij

if dnew < lsap then

if i unmatched then

lsap := dnew; isap := i;

else

if dnew < d

then

i

d

:= dnew; p

:= j ;

i

m

i

if i 62 Q then Q := Q + fig;

end if;

end if;

end if;

end for;

if Q = ; then exit while-loop;

choose i 2 Q with minimal d

;

i

lsp := d

;

i

if lsap (cid:20) lsp then exit while-loop;

Q := Q (cid:0) fig; B := B + fig;

j := m

;

i

end while;

if lsap 6= 1 then augment along path from node isap to node j

;

0

10

dual vectors u and v . If T = (T

; T

; E

) is the shortest alternating path tree that was

r

c

T

constructed until the shortest augmenting path was found, then u

and v

are updated as

i

j

follows:

(

u

:= u

+ d

(cid:0) lsap;

for i 2 T

;

i

i

i

r

v

:= c

(cid:0) u

;

for j 2 T

:

j

ij

i

c

The updated dual variables u and v satisfy (4.3) and the new reduced weights c

are

ij

non-negative.

The running time of the weighted matching algorithm can be decreased considerably

by means of a cheap heuristic that determines a large initial extreme matching M . We

use the strategy proposed by Carpaneto and Toth (1980). We calculate

u

:= min

c

;

for i 2 V

;

i

ij

r

j2ROW (i)

v

:= min

(c

(cid:0) u

);

for j 2 V

:

j

ij

i

c

i2COL(j)

Inspecting the sets COL(j) for each column node j in turn, we determine a large initial

matching M of edges for which c

(cid:0) u

(cid:0) v

= 0. Then, for each remaining unmatched

ij

i

j

column node j , every node i 2 COL(j) is considered for which c

(cid:0) u

(cid:0) v

= 0 and that is

ij

i

j

matched to a column node other than j , say j

. So (i; j

) 2 M . If an unmatched row node

1

1

i

2 COL(j

) can be found for which c

(cid:0) u

(cid:0) v

= 0, then (i; j

) in M is replaced

1

1

i

j

i

j

1

1

1

1

1

by (i; j) and (i

; j

). After having repeated this for all unmatched columns, the search for

1

1

shortest augmenting paths starts with respect to the current matching.

Finally, we note that the above weighted matching algorithm can also be used for

maximizing the sum of the diagonal entries of matrix A (instead of maximizing the product

of the diagonal entries). To do this, we again minimize (4.2), but we rede(cid:12)ne matrix C as

(

c

=

ij

a

(cid:0) ja

j; a

6= 0;

j

ij

ij

0;

otherwise:

Maximizing the sum of the diagonal entries is equal to minimizing (4.2), since

n

n

n

n

X

X

X

X

a

(cid:0)

ja

j =

(a

(cid:0) ja

j) =

c

:

(cid:27)

i(cid:27)

(cid:27)

i(cid:27)

i(cid:27)

i

i

i

i

i

i=1

i=1

i=1

i=1

5 Bottleneck matching

We describe a modi(cid:12)cation of the weighted bipartite matching algorithm from the previous

section for permuting rows and columns of a sparse matrix A such that the smallest ratio

11

between the absolute value of a diagonal entry and the maximum absolute value in its

column is maximized. That is, the modi(cid:12)cation computes a permutation (cid:27) that maximizes

ja

j

i(cid:27)

i

min

(5.1)

1(cid:20)i(cid:20)n

a

(cid:27)

i

where a

is the maximum absolute value in column j of the matrix A. Similarly to the

j

previous section, we transform this into a minimization problem. We de(cid:12)ne the matrix

C = (c

) as

ij

8

<

ja

j

ij

1 (cid:0)

; a

6= 0;

ij

c

=

ij

a

j

:

1;

otherwise:

Then maximizing (5.1) is equal to minimizing

a

(cid:0) ja

j

(cid:27)

i(cid:27)

i

i

max

= max

c

:

i(cid:27)

i

1(cid:20)i(cid:20)n

1(cid:20)i(cid:20)n

a

(cid:27)

i

Given a matching M in the bipartite graph G

= (V

; V

; E), the bottleneck value of

C

r

c

M is de(cid:12)ned as

c(M) = max

c

:

ij

(i;j)2M

The problem is to (cid:12)nd a perfect (or maximum) bottleneck matching M for which c(M) is

minimal, i.e. c(M) (cid:20) c(M

), for all possible maximum matchings M

. A matching M is

0

0

called extreme if it does not allow any alternating cyclic path P for which c(M (cid:8)P) < c(M).

The bottleneck algorithm starts o(cid:11) with any extreme matching M . The initial

bottleneck value b is set to c(M). Each pass through the main loop, an alternating tree

is constructed until an augmenting path P is found for which either c(M (cid:8) P) = c(M)

or c(M (cid:8) P) (cid:0) c(M) > 0 is as small as possible. The initializations and the main loop

for constructing such an augmenting path are those of Figure 4.1. Figure 5.1 shows the

inner-loop of the weighted matching algorithm of Figure 4.1 modi(cid:12)ed to the case of the

bottleneck ob jective function. The main di(cid:11)erences are that the sum operation on the path

lengths in Figure 4.1 is replaced by the \max" operation and, as soon as an augmenting

path P is found whose length lsap is less than or equal to the current bottleneck value

b, the main loop is exited, P is used to augment M , and b is set to max(b; lsap). The

bottleneck algorithm does not modify the edge weights c

.

ij

Similarly to the implementation discussed in Section 4, the set Q is implemented as a

pair (Q

; Q

), but now the array Q

contains all the nodes whose distance to the root is less

1

2

1

than or equal to the tentative bottleneck value b. Q

contains the nodes whose distance

2

to the root is larger than the bottleneck value but not in(cid:12)nity. Q

is again implemented

2

as a heap.

12

Figure 5.1: Modi(cid:12)ed inner loop of Figure 4.1 for the construction of a bottleneck

augmenting path.

for i 2 COL(j) n B do

dnew := max(lsp; c

);

ij

if dnew < lsap then

if i unmatched then

lsap := dnew; isap := i;

if lsap (cid:20) b then exit while-loop;

else

if dnew < d

then

i

d

:= dnew; p

:= j ;

i

m

i

if i 62 Q then Q := Q + fig;

end if;

end if;

end if;

end for;

A large initial extreme matching can be found in the following way. We de(cid:12)ne

r

:= min

c

;

for i 2 V

;

i

ij

r

j2ROW (i)

s

:= min

c

;

for j 2 V

;

j

ij

c

i2COL(j)

as the smallest entry in row i and column j , respectively. A lower bound b

for the

0

bottleneck value is

b

:= maxfmax

r

; max

s

g:

0

i

j

i

j

An extreme matching M can be obtained from the edges (i; j) for which c

(cid:20) b

; we scan

ij

0

all nodes j 2 V

in turn and for each node i 2 COL(j) that is unmatched and for which

c

c

(cid:20) b

, edge (i; j) is added to M . Then, for each remaining unmatched column node j ,

ij

0

every node i 2 COL(j) is considered for which c

(cid:20) b and that is matched to a column

ij

node other than j , say j

. So (i; j

) 2 M . If an unmatched row node i

2 COL(j

) can be

1

1

1

1

found for which c

(cid:20) b, then (i; j

) in M is replaced by (i; j) and (i

; j

). After having

i

j

1

1

1

1

1

done this for all unmatched columns, the search for shortest augmenting paths starts with

respect to the current matching.

Other initialization procedures can be found in the literature. For example, a slightly

more complicated initialization strategy is used by Finke and Smith (1978) in the context

13

of solving transportation problems. For every i 2 V

, j 2 V

, they use

r

c

g

:= jfc

jk 2 ROW (i) and c

(cid:20) b

gj;

i

ik

ik

0

h

:= jfc

jk 2 COL(j) and c

(cid:20) b

gj;

j

kj

kj

0

as the number of admissible edges incident to row node i and column node j respectively.

The idea behind using g

and h

is that once an admissible edge (i; j) is added to M , all

i

j

the other admissible edges that are incident to nodes i and j are no longer candidates

to be added to M . Therefore, the method tries to pick admissible edges such that the

number of admissible edges that become unusable is minimal. First, a row node i with

minimal g

is determined. From the set ROW (i) an admissible entry (i; j) (provided one

i

exists) is chosen for which h

is minimal and (i; j) is added to M . After deleting the edges

j

(i; k), k 2 ROW (i), and the edges (k ; j), k 2 COL(j), the method repeats the same for

another row node i

with minimal g

. This continues until all admissible edges are deleted

0

i

0

from the graph.

Finally, we note that instead of maximizing (5.1) we also could have maximized the

smallest absolute value on the diagonal. That is, we maximize

and de(cid:12)ne the matrix C as

min

ja

j;

i(cid:27)

i

1(cid:20)i(cid:20)n

(

c

=

ij

a

(cid:0) ja

j; a

6= 0;

j

ij

ij

1;

otherwise:

Note that this problem is rather sensitive to the scaling of the matrix A. Suppose for example that the matrix A has a column containing only one nonzero entry whose absolute value v is the smallest absolute value present in A. Then, after applying the bottleneck algorithm, the bottleneck value b will be equal to this small value. The smallest entry on the diagonal of the permuted matrix is maximized, but the algorithm did not have any in(cid:13)uence on the values of the other diagonal values. Scaling the matrix prior to applying the bottleneck algorithm avoids this.

In Du(cid:11) and Koster (1997), a di(cid:11)erent approach is taken to obtain a bottleneck

matching. Let A

denote the matrix that is obtained by setting to zero in A all entries

(cid:15)

a

for which ja

j < (cid:15) (thus A

= A) and M

denote the matching obtained by removing

ij

ij

0

(cid:15)

from matching M all the entries (i; j) for which ja

j < (cid:15) (thus M

= M). Throughout the

ij

0

algorithm, (cid:15)max and (cid:15)min are such that a maximum matching of size jM j does not exist

for A

but does exist for A

. At each step, (cid:15) is chosen in the interval ((cid:15)min; (cid:15)max),

(cid:15)max

(cid:15)min

and a maximum matching for the matrix A

is computed using a variant of MC21. If

(cid:15)

this matching has size jM j, then (cid:15)min is set to (cid:15), otherwise (cid:15)max is set to (cid:15). Hence,

14

the size of the interval decreases at each step and (cid:15) will converge to the bottleneck value.

After termination of the algorithm, M

is the computed bottleneck matching and (cid:15) the

0

corresponding bottleneck value.

6 Scaling

Olschowka and Neumaier (1996) use the dual solution produced by the weighted matching

algorithm to scale the matrix. Let u and v be such that they satisfy relation (4.3). If we

de(cid:12)ne

then we have

D

= diag(d

; d

; : : : ; d

); d

= exp(u

);

1

i

1

2

n

i

1

1

1

1

D

= diag(d

; d

; : : : ; d

); d

= exp(v

)=a

;

2

j

j

1

2

n

j

2

2

2

2

(6.1)

1

2

d

ja

jd

=

i

j

ij

exp(u

+ log(ja

j) + v

(cid:0) log(a

)) =

i

ij

j

j

exp(u

+ v

(cid:0) (log(a

) (cid:0) log(ja

j))) =

i

j

j

ij

exp(u

+ v

(cid:0) c

) (cid:20) 1

i

j

ij

Equality holds when u

+ v

= c

, that is (i; j) 2 M . In words, D

AD

is a matrix whose

i

j

ij

1

2

diagonal entries are one in absolute value and whose o(cid:11)-diagonal entries are all less than or equal to one. Olschowka and Neumaier (1996) call such a matrix an I -matrix and use this in the context of dense Gaussian elimination to reduce the amount of pivoting that is needed for numerical stability. The more dominant the diagonal of a matrix, the higher the chance that diagonal entries are stable enough to serve as pivots for elimination.

For iterative methods, the transformation of a matrix to an I -matrix is also of interest.

For example, from Gershgorin's theorem we know that the union of all discs

8

9

<

=

X

K

=

(cid:22) 2 C j j(cid:22) (cid:0) a

j (cid:20)

ja

j

i

ii

ik

:

;

k 6=i

contains all eigenvalues of the n (cid:2) n matrix A. Disc K

has center at a

and radius that is

i

ii

equal to the sum of the absolute o(cid:11)-diagonal values in row i. Since the diagonal entries of an I -matrix are all one, all the n disks have center at 1. The estimate of the eigenvalues will be sharper as A deviates less from a diagonal matrix. That is, the smaller the radii of the discs, the better we know where the eigenvalues are situated. If we are able to reduce the radii of the discs of an I -matrix, i.e. reduce the o(cid:11)-diagonal values, then we tend to cluster the eigenvalues more around one. In the ideal case, all the discs of an I -matrix have a radius smaller than one, in which case the matrix is strictly row-wise diagonally

15

dominant. This guarantees that many types of iterative methods will converge (in exact arithmetic), even simple ones like the Jacobi and Gauss-Seidel method. However, if at least one disc remains with radius larger than or close to one, zero eigenvalues or small eigenvalues are possible.

A straightforward (but expensive) attempt to decrease large o(cid:11)-diagonal entries of a matrix is by row and column equalization (Olschowka and Neumaier 1996). Let A be an I -matrix. We de(cid:12)ne matrix C = (c

) as c

= log ja

j. (For simplicity we assume

ij

ij

ij

that A contains no zero entries.) Equalization consists of repeatedly equalizing the largest

absolute value in row i and the largest absolute values in column i:

p := 0;

for k := 1; 2; : : : do

for j := 1 to n do

y

:= maxfc

+ p

(cid:0) p

jr 6= j; c

6= 0g;

1

j r

j

r

j r

y

:= maxfc

+ p

(cid:0) p

jr 6= j; c

6= 0g;

2

rj

r

j

rj

p

:= p

+ (y

(cid:0) y

)=2;

j

j

2

1

end;

end;

For k = 1, this algorithm minimizes

maxfc

+ p

(cid:0) p

ji 6= j; c

6= 0g

ij

i

j

ij

and thus, if we de(cid:12)ne d

:= exp(p

) and d

:= 1= exp(p

), the algorithm minimizes the

i

j

i

j

1

2

largest o(cid:11)-diagonal absolute value in matrix D

AD

. The diagonal entries do not change.

1

2

Note that the above scaling strategy does not guarantee that all o(cid:11)-diagonal entries

of an I -matrix will be smaller than one in absolute value, for example if the I -matrix A

contains two o(cid:11)-diagonal entries a

and a

, k 6= l, whose absolute values are both one.

kl

lk

7 Experimental results

In this section, we discuss several cases where the reorderings algorithms from the previous

section can be useful. These include the solution of sparse equations by a direct method

and by an iterative technique. We also consider its use in generating a preconditioner for

an iterative method.

The set of matrices that we used for our experiments are unsymmetric matrices taken

from the Harwell-Boeing Sparse Matrix Test Collection (Du(cid:11), Grimes and Lewis 1992)

and from the sparse matrix collection at the University of Florida (Davis 1997).

All matrices are initially row and column scaled. By this we mean that the matrix is

scaled so that the maximum entry in each row and in each column is one.

The computer used for the experiments is a SUN UltraSparc with 256 Mbytes of main memory. The algorithms are implemented in Fortran 77.

We use the following acronyms. MC21 is the matching algorithm from the Harwell Subroutine Library for computing a matching such that the corresponding permuted matrix has a zero free-diagonal (see Section 3). BT is the bottleneck bipartite matching algorithm from Section 5 for permuting a matrix such that the smallest ratio between the absolute value of a diagonal entry and the maximum absolute value in its column is maximized. BT' is the bottleneck bipartite matching algorithm from Du(cid:11) and Koster (1997). MPD is the weighted matching algorithm from Section 4 and computes a permutation such that the product of the diagonal entries of the permuted matrix is maximum in absolute value. MPS is equal to the MPD algorithm, but after the permutation, the matrix is scaled to an I -matrix (see Section 6).

Table 7.1 shows for some large sparse matrices the order, number of entries, and the time for the algorithms to compute a matching. The times for MPS are not listed, because they are almost identical to those for MPD. In general, MC21 needs the least time to compute a matching, except for the ONETONE and TWOTONE matrices. For these matrices, the search heuristic that is used in MC21 (a depth-(cid:12)rst search with look-ahead) does not perform well. This is probably caused by the ordering of the columns (variables) and the entries inside the columns of the matrix. A random permutation of the matrix prior to applying MC21 might lead to other results. There is not a clear winner between

the bottleneck algorithms BT and BT', although we note that BT' requires the entries inside the columns to be sorted by value. This sorting can be expensive for relatively dense matrices. MPD is in general the most expensive algorithm. This can be explained by the more selective way in which this algorithm constructs augmenting paths.

7.1 Experiments with a direct solution method

For direct methods, putting large entries on the diagonal suggests that pivoting down the diagonal might be more stable. Indeed, stability can still not be guaranteed, but if we have a solution scheme like the multifrontal method of Du(cid:11) and Reid (1983), where a symbolic phase chooses the initial pivotal sequence and the subsequent factorization phase then modi(cid:12)es this sequence for stability, it can mean that the modi(cid:12)cation required is less than if the permutation were not applied.

In the multifrontal approach of Du(cid:11) and Reid (1983), later developed by Amestoy and Du(cid:11) (1989), an analysis is performed on the structure of A + A

to obtain an ordering that

T

reduces (cid:12)ll-in under the assumption that all diagonal entries will be numerically suitable for pivoting. The numerical factorization is guided by an assembly tree. At each node of the tree, some steps of Gaussian elimination are performed on a dense submatrix whose Schur complement is then passed to the parent node in the tree where it is assembled

17

Table 7.1: Times (in seconds) for matching algorithms. Order of matrix is n and

number of entries (cid:28) .

Matrix

n

(cid:28) MC21

BT'

BT MPD

MAHINDAS

1258

7682

0.01

0.01

0.01

0.02

GEMAT11

4929

33185

0.01

0.03

0.01

0.04

ONETONE1

36057

341088

2.67

0.70

0.18

0.61

ONETONE2

36057

227628

2.63

0.53

0.14

0.42

TWOTONE

120750

1224224

60.10

6.95

2.82

2.17

GOODWIN

7320

324784

0.27

2.26

4.17

1.82

LHR01

1477

18592

0.02

0.04

0.10

0.10

LHR02

2954

37206

0.04

0.14

0.16

0.21

LHR07

7337

156508

0.04

0.58

0.24

0.82

LHR14C

14270

307858

0.28

1.13

1.12

3.32

LHR71C

70304

1528092

1.86

9.00

11.96

37.73

AV41092

41092

1683902

35.72

10.81

37.82

65.13

(or summed) with Schur complements from the other children and original entries of the matrix. If, however, numerical considerations prevent us from choosing a pivot then the algorithm can proceed, but now the Schur complement that is passed to the parent is larger and usually more work and storage will be needed to e(cid:11)ect the factorization. The logic of (cid:12)rst permuting the matrix so that there are large entries on the diagonal, before computing the ordering to reduce (cid:12)ll-in, is to try and reduce the number of pivots that are delayed in this way thereby reducing storage and work for the factorization. We show the e(cid:11)ect of this in Table 7.2 where we can see that even using MC21 can be very bene(cid:12)cial although the other algorithms can show signi(cid:12)cant further gains. In Table 7.3, we show the e(cid:11)ect of this on the number of entries in the factors. Clearly this mirrors the results in Table 7.2.

In addition to being able to select the pivots chosen by the analysis phase, the multifrontal code MA41 will do better on matrices whose structure is symmetric or nearly so. Here, we de(cid:12)ne the structural symmetry for a matrix A as the number of entries a for

ij

which a

is also an entry, divided by the total number of entries. The structural symmetry

j i

after the permutations is shown in Table 7.4. The matching orderings in some cases

increase the symmetry of the resulting reordered matrix, which is particularly apparent

when we have a very sparse system with many zeros on the diagonal. In that case, the

reduction in number of o(cid:11)-diagonal entries in the reordered matrix has an in(cid:13)uence on

the symmetry. Notice that, in this respect, the more sophisticated matching algorithms

may actually cause problems since they could reorder a symmetrically structured matrix

with a zero-free diagonal, whereas MC21 will leave it unchanged.

18

Table 7.2: Number of delayed pivots in factorization from MA41. An \-" indicates

that MA41 needed more than 200 MBytes of memory.

Matrix

Matching algorithm used

None MC21

BT MPD MPS

GEMAT11

-

76

0

0

0

ONETONE1

-

16261

298

100

0

ONETONE2

40916

8310

411

100

0

GOODWIN

536

1622

427

53

41

LHR01

1378

171

42

18

0

LHR02

3432

388

143

56

0

LHR14C

-

7608

1042

169

274

LHR71C

-

35354

7424

2643

3190

AV41092

-

10151

2141

1730

1722

Table 7.3: Number of entries (10

) in the factors from MA41.

3

Matrix

Matching algorithm used

None MC21

BT

MPD MPS

GEMAT11

-

128

79

78

78

ONETONE1

-

10,359

7,329

4,715

4,713

ONETONE2

14,083

2,876

2,298

2,170

2,168

GOODWIN

1,263

2,673

2,058

1,282

1,281

LHR01

997

137

210

113

111

LHR02

2,299

333

374

235

230

LHR14C

-

3,111

2,676

2,164

2,165

LHR71C

-

18,787

17,528

11,600

11,630

AV41092

-

16,226

14,968

14,110

14,111

Table 7.4: Structural symmetry after permutation. (1.00 = symmetric)

Matrix

Matching algorithm used

None MC21

BT MPD/MPS

GEMAT11

0.002

0.530

0.947

0.957

ONETONE1

0.990

0.368

0.427

0.434

ONETONE2

0.148

0.461

0.564

0.574

GOODWIN

0.642

0.288

0.365

0.583

LHR01

0.009

0.302

0.133

0.168

LHR02

0.009

0.302

0.141

0.168

LHR14C

0.007

0.336

0.125

0.150

LHR71C

0.002

0.384

0.182

0.207

AV41092

0.001

0.101

0.082

0.082

19

Finally, Table 7.5 shows the e(cid:11)ect on the solution times of MA41. We sometimes observe a dramatic reduction in time for the solution when preceded by a permutation.

Table 7.5: Solution time required by MA41.

Matrix

Matching algorithm used

None MC21

BT

MPD MPS

GEMAT11

-

0.28

0.20

0.20

0.20

ONETONE1

-

225.71

95.33

44.22

42.97

ONETONE2

81.45

17.05

11.70

11.54

11.13

GOODWIN

3.64

14.63

7.98

3.56

3.56

LHR01

10.10

0.41

0.72

0.28

0.28

LHR02

24.85

1.07

1.10

0.58

0.55

LHR14C

-

12.66

10.48

5.88

5.83

LHR71C

-

148.07

127.92

43.33

42.90

AV41092

-

226.20

180.39

155.70

154.44

Our implementations of the algorithms described in this paper have been used successfully by Li and Demmel (1998) to stabilize sparse Gaussian elimination in a distributed-memory environment without the need for dynamic pivoting. Their method decomposes the matrix into an N (cid:2) N block matrix A[1 : N ; 1 : N] by using the notion of unsymmetric supernodes (Demmel, Eisenstat, Gilbert, Li and Liu 1995). The blocks are

mapped cyclically (in both row and column dimensions) onto the nodes (processors) of a two-dimensional rectangular processor grid. The mapping is such that at step k of the numerical factorization, a column of processors factorizes the block column A[k : N ; k], a row of processes participates in the triangular solves to obtain the block row U [k ; k +1 : N], and all processors participate in the corresponding multiple-rank update of the remaining matrix A[k + 1 : N ; k + 1 : N].

The numerical factorization phase in this method does not use (dynamic) partial pivoting on the block columns. This allows for the a priori computation of the nonzero structure of the factors, the distributed data structures, the communication pattern, and a good load balancing scheme, which makes the factorization more scalable on distributed-memory machines than factorizations in which the computational and communication tasks only become apparent during the elimination process. To ensure a solution that is numerically stable, the matrix is permuted and scaled before the factorization to make the diagonal entries large compared to the o(cid:11)-diagonal entries, any tiny pivots encountered during the factorization are perturbed, and a few steps of iterative re(cid:12)nement are performed during the triangular solution phase if the solution is not accurate enough. Numerical experiments demonstrate that the method (using the implementation of the MPS algorithm) is as stable as partial pivoting for a wide range of problems.

20

7.2 Experiments with iterative solution methods

For iterative methods, simple techniques like Jacobi or Gauss-Seidel converge more

quickly if the diagonal entry is large relative to the o(cid:11)-diagonals in its row or column, and techniques like block iterative methods can bene(cid:12)t if the entries in the diagonal blocks are large. Additionally, for preconditioning techniques, for example for diagonal preconditioning or incomplete LU preconditioning, it is intuitively evident that large diagonals should be bene(cid:12)cial.

7.2.1 Preconditioning by incomplete factorizations

In incomplete factorization preconditioners, pivots are often taken from the diagonal and (cid:12)ll-in is discarded if it falls outside a prescribed sparsity pattern. (See Saad (1996) for an overview.) Incomplete factorizations are used so that the resulting factors are more economical to store, to compute, and to solve with.

One of the reasons why incomplete factorizations can behave poorly is that pivots can be arbitrarily small (Benzi, Szyld and van Duin 1997, Chow and Saad 1997). Pivots may even be zero in which case the incomplete factorization fails. Small pivots allow the numerical values of the entries in the incomplete factors to become very large, which leads to unstable and therefore inaccurate factorizations. In such cases, the norm of the residual matrix R = A(cid:0)

L

U will be large. (Here,

L and

U denote the computed incomplete

^

^

^

^

factors.)

A way to improve the stability of the incomplete factorization, is to preorder the

matrix to put large entries onto the diagonal. Obviously, a successful factorization still

cannot be guaranteed, because nonzero diagonal entries may become very small (or even

zero) during the factorization, but the reordering may mean that zero or small pivots

are less likely to occur. Table 7.6 shows some results for the reorderings applied prior to

incomplete factorizations of the form ILU(0), ILU(1), and ILUT and the iterative methods

GMRES(20), BiCGSTAB, and QMR. In some cases, the method will only converge after

the permutation, in others it greatly improves the convergence.

However, we emphasize that permuting large entries to the diagonal of matrix will

not always improve the accuracy and stability of incomplete factorization. An inaccurate

factorization can also occur in the absence of small pivots, when many (especially large)

(cid:12)ll-ins are dropped from the incomplete factors. In this respect, it may be bene(cid:12)cial to

apply a symmetric permutation after the matching reordering to reduce (cid:12)ll-in. Another

kind of instability in incomplete factorizations, which can occur with and without small

pivots, is severe ill-conditioning of the triangular factors. (In this situation, jjRjj

need

F

not be very large, but jjI (cid:0) A(

L

U)

jj

will be.) This is also a common situation when

F

^

^

(cid:0)1

the coe(cid:14)cient matrix is far from diagonally dominant.

21

Table 7.6: Number of iterations required by some preconditioned iterative methods after

permutation.

Matrix and method

Matching algorithm

MC21 BT MPD MPS

IMPCOL E

ILU(0)

GMRES(20)

-

17

15

14

BiCGSTAB

123

25

11

10

QMR

101

25

17

16

ILU(1)

GMRES(20)

59

15

11

11

BiCGSTAB

98

19

8

7

QMR

72

21

12

12

ILUT

GMRES(20)

8

7

8

7

BiCGSTAB

9

4

5

4

QMR

10

7

8

8

MAHINDAS

ILU(0)

GMRES(20)

-

-

179

116

BiCGSTAB

-

-

39

38

QMR

-

-

55

55

ILU(1)

GMRES(20)

-

-

69

58

BiCGSTAB

-

-

26

21

QMR

-

-

34

34

ILUT

GMRES(20)

-

-

15

13

BiCGSTAB

-

151

11

8

QMR

-

-

17

14

WEST0497

ILU(0)

GMRES(20)

-

40

19

19

BiCGSTAB

-

71

22

16

QMR

-

48

23

21

ILU(1)

GMRES(20)

-

19

15

15

BiCGSTAB

-

26

15

11

QMR

-

30

18

16

ILUT

GMRES(20)

-

14

10

7

BiCGSTAB

-

11

7

4

QMR

-

15

12

7

22

We also performed a set of experiments in which we (cid:12)rst permuted the columns of the matrix A by using a reordering computed by one of the matching algorithms, followed by a symmetric permutation of A generated by the reverse Cuthill-McKee ordering (Cuthill and McKee 1969) applied to A + A

. The motivation behind this is that the number

T

of entries that is dropped from the factors can be reduced by applying a reordering of

the matrix that reduces (cid:12)ll-in. In the experimental results, we noticed that the additional

permutation sometimes has a positive as well as a negative e(cid:11)ect on the performance of the

iterative solvers. Table 7.7 shows some results for the three iterative methods from Table

7.6 preconditioned by ILUT on the WEST matrices from the Harwell-Boeing collection.

Table 7.7: Number of iterations required by some ILUT-preconditioned iterative methods

after the matching reordering with and without reverse Cuthill-McKee.

Matrix and method

Matching algorithm

Matching algorithm

without RCM

with RCM

MC21 BT MPD MPS MC21 BT MPD MPS

WEST0497 GMRES(20)

-

14

10

7

14

15

10

5

BiCGSTAB

-

11

7

4

60

23

10

3

QMR

-

15

12

7

-

19

10

6

WEST0655 GMRES(20)

-

-

58

17

-

-

-

-

BiCGSTAB

-

-

42

14

-

-

-

71

QMR

-

-

38

18

-

-

-

76

WEST0989 GMRES(20)

-

-

-

-

-

37

13

8

BiCGSTAB

-

-

280

-

262

35

9

5

QMR

-

-

-

-

-

36

15

8

WEST1505 GMRES(20)

-

-

-

-

-

-

-

117

17

BiCGSTAB

-

-

809

-

-

-

42

15

QMR

-

-

-

-

-

-

60

20

WEST2021 GMRES(20)

-

-

-

-

-

-

36

11

BiCGSTAB

-

-

-

-

-

-

26

7

QMR

-

-

-

-

-

-

30

12

7.2.2 Experiments with a block iterative solution method

The Jacobi method is not a particularly current or powerful method so we focussed our experiments on the block Cimmino implementation of Arioli, Du(cid:11), Noailles and Ruiz (1992), which is equivalent to using a block Jacobi algorithm on the normal equations. In this implementation, the subproblems corresponding to blocks of rows from the matrix

23

are solved by the sparse direct method MA27 (HSL 1996).

We show the e(cid:11)ect of this in Table 7.8 on the problem MAHINDAS from Table 7.6. The matching algorithm was followed by a reverse Cuthill-McKee algorithm to obtain a block tridiagonal form. The matrix was partitioned into 2, 4, 8, and 16 blocks of rows and the accelerations used were block CG algorithms with block sizes 1, 4, and 8. The block rows are chosen of equal (or nearly equal) size.

Table 7.8: Number of iterations of block Cimmino algorithm for the matrix MAHINDAS.

Acceleration +

Matching algorithm

block rows

CG(1)

None MC21 BT MPD MPS

2

324

267

298

295

105

4

489

383

438

438

141

8

622

485

532

524

167

16

660

572

574

574

175

CG(4)

2

148

112

130

133

68

4

212

190

199

194

92

8

261

235

232

233

111

16

281

245

253

253

112

CG(8)

2

80

62

72

75

54

4

117

105

109

108

71

8

140

133

127

130

84

16

151

142

137

136

90

In general, we noticed in our experiments that the block Cimmino method often was more sensitive to the scaling (in MPS) and less to the reorderings. The convergence properties of the block Cimmino method are independent of row scaling. However, the sparse direct solver MA27 (HSL 1996) used for solving the augmented systems, performs numerical pivoting during the factorizations of the augmented matrices. Row scaling might well change the choice of the pivot order and a(cid:11)ect the (cid:12)ll-in in the factors and the accuracy of the solution. Column scaling should a(cid:11)ect convergence of the method since it can be considered as a diagonal preconditioner. For more details see (Ruiz 1992).

24

8 Conclusions and future work

We have considered, in Sections 3-4, techniques for permuting a sparse matrix so that the diagonal of the permuted matrix has entries of large absolute value. We discussed various criteria for this and considered their implementation as computer codes. We also considered in Section 6 possible scaling strategies to further improve the weight of the diagonal with respect to the o(cid:11)-diagonal values.

In Section 7, we then indicated several cases where such a permutation (and scaling) can be useful. These include the solution of sparse equations by a direct method and by an iterative technique. We also considered its use in generating a preconditioner for an iterative method. The numerical experiments show that for a multifrontal solver and preconditioned iterative methods, the e(cid:11)ect of these reorderings can be dramatic. The e(cid:11)ect on the block Cimmino iterative method seems to be less dramatic. For this method, the discussed scaling tends to have a more important e(cid:11)ect.

While it is clear that reordering matrices so that the permuted matrix has a large diagonal can have a very signi(cid:12)cant e(cid:11)ect on solving sparse systems by a wide range of techniques, it is somewhat less clear that there is a universal strategy that is best in all cases. One reason for this is that increasing the size of the diagonal only is not always su(cid:14)cient to improve the performance of the method. For example, for the incomplete preconditioners that we used for the numerical experiments in Section 7, it is not only the size of the diagonal but also the amount and size of the discarded (cid:12)ll-in plays an important role. We have thus started experimenting with combining the strategies mentioned in Sections 3-4 and, particularly for generating a preconditioner and the block Cimmino

approach, with combining our unsymmetric ordering with symmetric orderings.

Another interesting extension to the discussed reorderings is a block approach to increase the size of diagonal blocks instead of only the diagonal entries and use for example a block Jacobi preconditioner on the permuted matrix. This is of particular interest for the block Cimmino method. One could also build other criteria into the weighting for obtaining a bipartite matching, for example, to incorporate a Markowitz cost so that sparsity would also be preserved by the choice of the resulting diagonal as a pivot. Such combination would make the resulting ordering suitable for a wider class of sparse direct solvers.

Finally, we notice in our experiments with MA41 that one e(cid:11)ect of the matching algorithm was to increase the structural symmetry of unsymmetric matrices. We are exploring further the use of ordering techniques that more directly attempt to increase structural symmetry.

Acknowledgments

We are grateful to Michele Benzi of Los Alamos National Laboratory and Miroslav Tuma of the Czech Academy of Sciences for their assistance on the preconditioned iterative methods and Daniel Ruiz of ENSEEIHT for his help on block iterative methods.

References

Amestoy, P. R. and Du(cid:11), I. S. (1989), `Vectorization of a multiprocessor multifrontal code', Int. J. of Supercomputer Applics. 3, 41{59.

Arioli, M., Du(cid:11), I. S., Noailles, J. and Ruiz, D. (1992), `A block pro jection method for sparse matrices', SIAM J. Sci. Stat. Comput. pp. 47{70.

Benzi, M., Szyld, D. B. and van Duin, A. (1997), Orderings for incomplete factorization preconditioning of nonsymmetric problems, Technical Report LA-UR-97-3525, Los Alamos National Laboratory, Los Alamos, NM.

Burkard, R. E. and Derigs, U. (1980), Assignment and Matching Problems: Solution Methods with FORTRAN-Programs, Springer, Berlin-Heidelberg-New York. Lecture Notes in Economics and Mathematical Systems 184.

Carpaneto, G. and Toth, P. (1980), `Solution of the assignment problem (Algorithm 548)', ACM Trans. Math. Software pp. 104{111.

Carraresi, P. and Sodini, C. (1986), `An e(cid:14)cient algorithm for the bipartite matching problem', European Journal of Operational Research 23, 86{93.

Chow, E. and Saad, Y. (1997), Experimental study of ILU preconditioners for inde(cid:12)nite matrices, Technical Report TR 97/95, Department of Computer Science, and Minnesota Supercomputer Institute, University of Minnesota, Minneapolis.

Cuthill, E. and McKee, J. (1969), Reducing the bandwidth of sparse symmetric matrices, in `Proceedings 24th National Conference of the Association for Computing Machinery', Brandon Press, New Jersey, pp. 157{172.

Davis, T. A. (1997), University of Florida sparse matrix collection, Available at http://www.cise.u(cid:13).edu/(cid:24)davis and ftp://ftp.cise.u(cid:13).edu/pub/faculty/davis.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S. and Liu, J. W. H. (1995), A

supernodal approach to sparse partial pivoting, Technical Report UCB//CSD-95-883,

Computer Science Division, University of California at Berkeley, CA. To appear in

SIAM J. Matrix Anal. Appl.

26

Derigs, U. and Metz, A. (1986),

`An e(cid:14)cient labeling technique for solving sparse

assignment problems', Computing 36, 301{311.

Dijkstra, E. W. (1959), `A note on two problems in connection with graphs', Numerische

Mathematik 1, 269{271.

Du(cid:11), I. S. (1981), `Algorithm 575. Permutations for a zero-free diagonal', ACM Trans.

Math. Software 7(3), 387{390.

Du(cid:11), I. S. and Koster, J. (1997), The design and use of algorithms for permuting

large entries to the diagonal of sparse matrices, Technical Report RAL-TR-97-059,

Rutherford Appleton Laboratory, Oxfordshire, England. Accepted for publication in

SIAM J. Matrix Anal. Appl.

Du(cid:11), I. S. and Reid, J. K. (1983), `The multifrontal solution of inde(cid:12)nite sparse symmetric

linear systems', ACM Trans. Math. Software 9, 302{325.

Du(cid:11), I. S. and Wiberg, T. (1988), `Remarks on implementations of O(n

(cid:28)) assignment

1=2

algorithms', ACM Trans. Math. Software 14(3), 267{287.

Du(cid:11), I. S., Grimes, R. G. and Lewis, J. G. (1992), Users' guide for the Harwell-Boeing sparse matrix collection (Release 1), Technical Report RAL-92-086, Rutherford Appleton Laboratory, Oxfordshire, England.

Edmonds, J. and Karp, R. M. (1972), `Theoretical improvements in algorithmic e(cid:14)ciency for network problems', Journal of the ACM 19, 248{264.

Finke, G. and Smith, P. (1978), `Primal equivalents for the threshold algorithm', Op. Res. Verf. 31, 185{198.

Ford Jr., L. R. and Fulkerson, D. R. (1962), Flows in Networks, Princeton Univ. Press, Princeton, NJ.

Gallo, G. and Pallottino, S. (1988), `Shortest path algorithms', Annals of Operations Research 13, 3{79.

Hopcroft, J. E. and Karp, R. M. (1973), `An n algorithm for maximum matchings in 5=2 bipartite graphs', SIAM J. Comput. 2, 225{231.

HSL (1996), Harwel l Subroutine Library, A Catalogue of Subroutines (Release 12), AEA Technology, Harwell Laboratory, Oxfordshire, England.

Jonker, R. and Volgenant, A. (1987), `A shortest augmenting path algorithm for dense and sparse linear assignment problems', Computing 38, 325{340.

Kuhn, H. W. (1955), `The Hungarian method for the assignment problem', Naval Research

Logistics Quarterly 2, 83{97.

Li, X. S. and Demmel, J. W. (1998), Making sparse Gaussian elimination scalable by static pivoting, in `Proceedings of Supercomputing', Orlando, Florida.

Olschowka, M. and Neumaier, A. (1996), `A new pivoting strategy for Gaussian elimination', Lin. Alg. and Its Appl. 240, 131{151.

Ruiz, D. (1992), Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment, PhD thesis, CERFACS, Toulouse, France.

Saad, Y. (1996), Iterative methods for sparse linear systems, PWS Publishing Company, Boston.

Tarjan, R. E. (1983), Data structures and network algorithms, SIAM, Philadelphia, USA. CBMS-NSF Regional conference series in applied mathematics 44.