Automatic Determination of Matrix-Blocks

Lapack Working note 151,
University of Tennessee Computer Science Report ut-cs-01-458

Victor Eijkhout[*]

16 April 2001

Abstract

Many sparse matrices have a natural block structure, for instance arising from the discretisation of a physical domain. We give an algorithm for finding this block structure from the matrix sparsity pattern. This algorithm can be used for instance in iterative solver libraries in cases where the user does not or can not pass this block structure information to the software. The block structure can then be used as a basis for domain-decomposition based preconditioners.

1 Introduction

Sparse matrices can often be described as having a limited bandwidth and a limited number of nonzeros per row. However, this description does not do justice to a structure that is visible to the naked eye. Many sparse matrices come from discretised partial differential equations on a physical domain in two or three space dimensions. From the way the variable numbering traverses the problem domain, in a natural way a block structure arises. In a plot of the matrix sparsity pattern, blocks corresponding to lines or planes in the domain, or whole substructures, can be easily discerned.

Direct matrix solvers often ignore such a matrix structure. Indeed, succesful solvers are based on renumbering the matrix, regardless the original ordering. Examples are the Cuthill-McKee ordering [3] which reduces the bandwidth of the matrix, and the multiple minimum degree ordering [4] which more directy aims to minimise fill-in. This approach succeeds by virtue of the fact that such direct solvers are purely based on the structure of the matrix, and disregard the numerical entries. Time to solution is fully a property of the structure and independent of the numerics.

For iterative solvers such an approach is less desirable. The time to solution is strongly dependent on numerical properties, and only to a lesser degree on structural properties. This issue is only exaccerbated by the incorporation of a preconditioner in the iterative scheme. It would then make sense – and we will show with an example how serious this issue is – to take structure information into account in the construction of a preconditioner. In particular, for preconditioners that are based on partitioning of the domain, such as Schur complement methods and Schwarz methods, one would aim to let the domains chosen correspond to domains arising naturally from the application.

In cases where the user writes the full application and the iterative solver, our story would now end on the above note of recommendation. However, in practical cases, users may

[*] This work was supported in part by the National Science Foundation, grant number ACI-9876895.

rely on an iterative solver library, and be limited to the interface it provides for supplying structural information in addition to the bare matrix entries. Looking at this problem from the side of the library developer, we can not always assume that a user has the opportunity, sophistication, or time to supply such annotations to the matrix.

We conclude that there is a legitimate opportunity for software that automatically determines a matrix structure. Such software could be incorporated into existing iterative solver libraries, where it would retrieve information that, because of a fixed user interface, simply can not be provided by the user. Another application for this software would be the NetSolve package [2]. We have proposed such a structural partitioner as part of a more general intelligent black-box linear equation solver [1].

In the next two sections we describe two partitioning algorithms, one for regular matrices, and one for general matrices. We conclude by giving a practical example showing the efficacy, and indeed necessity, or our partitioning approach.

2 Regular matrix partitioner

If a matrix derives from a discretized PDE on a ‘brick’ domain, it has a structure where all blocks are of equal size. Facilitating the analysis is the fact that all nonzero diagonals are parallel to the main diagonal. For this regular case we develop a partitioner that finds all possible block structures. The fact that there can be more than one block structure is due to the physical nature of the problem: blocks can correspond to for instance lines or planes in a three-dimensional domain. Our algorithm proceeds by successively discarding outer diagonals, which would correspond to the connections between blocks, and finding any block-diagonal structure in the remainder. We always start by symmetrising the matrix, so that we need test only in, say, the lower triangle.

```
For $i = 2 \ldots n$
    if the subblock $A(i : n, 1 : i - 1)$ is zero,
      mark $i$ as a split point
```

Figure 1: Find starting points of block-diagonal blocks (algorithm outline)

Finding whether a matrix subblock is zero is a computationally expensive routine; the practical implementation would test consecutive rows and abort once a zero element has been found.

```
For $i = 2 \ldots n$
    test all row segments $A(j, 1 : i - 1)$ for $j = i \ldots n$ in succession,
        if any is nonzero, $i$ is not a split point;
          continue with the next (outer) $i$ iteration
        if all segments are zero, mark $i$ as a split point
```

Figure 2: Find starting points of block-diagonal blocks (practical implementation)

The algorithm for finding the block structure spit points is then enclosed in a loop that finds all values p such that the p-th diagonal of A is nonzero and the $p + 1$-st is zero. For such values, we apply algorithm 1 to the $2p + 1$ bandpart of A.

Matlab code implementing the whole algorithm can be found in appendix A.

3 General matrix partitioner

The algorithm above relied on the fact that the nonzero off-diagonals are parallel to the main diagonal to discard the connections between blocks. For matrices from irreguar domains, or regular domains that have already been subjected to a Cuthill-McKee ordering, we can make no such assumption. What is more, the connecting blocks can be arbitrarily close to the main diagonal, since the diagonal blocks can be of any size, especially with the Cuthill-McKee ordering, there are guaranteed to be both large and small blocks.

Thus we need a different test for whether a point i can be the start of a block. The test we used is the following:

> If i is the start of a block, then $j > i$ is the start of a block, if $A_{ij} \neq 0$, $A_{ij-1} = 0$, $A_{i-1j-1} \neq 0$ and $A_{i-1j} = 0$.

We start off the algorithm by declaring that 1 is the start of a block.

This simple test formalises the common sense criterium that subsequent blocks correspond to subsequent slices out of the domain, and that their respective beginnings are connected, as are their endings, and no beginning of one block can be connected to the end of another. Occasionally this test will be too stringent, so we keep track separately of those points for which only the conditions on A_{ij} and A_{ij-1} are satisfied; we can use those points to restart the process if needed. If there are several choices of possible next split points, we choose one that gives a block not too different in size from what we have encountered so far; we use deviation from the average size as a measure.

As a first refinement of this test, we observe that testing on single matrix elements may often not give the right results. Instead we test on whether a small subblock is zero. The subblocks have the indicated matrix elements as a corner point. We have to choose the size of the subblock; right now we use $(j - i)/10$ as a crude heuristic, but more sophisticated estimates are possible.

The above process will occasionally give blocks of disparate sizes; in a post-processing step we merge small blocks with adjoining large blocks.

Matlab code implementing the whole algorithm can be found in appendix B.

4 Practical application and further research

As a practical application we used the Bi-Conjugate Gradient algorithm with an alternating Schwarz preconditioner on a two-material problem with large differences in material coefficients; figure 3. The is almost regular in structure, but the last diagonal block is smaller than the rest, so an even distribution will not cut the block boundaries. Additionally, because of the way boundary conditions between the materials are discretised, the off-diagonal nonzero structure has gaps and a few outlying diagonals.

We do not plot the results of the regular splitting algorithm of section 2, since it gives precisely the structure as desired and expected. We give two plots of the output of the general split algorithm (section 3): once with all splits found indicated (figure 4), and once after consolidation of the small blocks (figure 5). We see that the general algorithm finds all the large blocks, and is only minimally confused by the gaps in the off-diagonal sparsity.

We tested two matrices of the same sparsity domain, one small of size 1641, and one of medium size 5655; we simulated 8 processors throughout. In the first case (table 1) we see

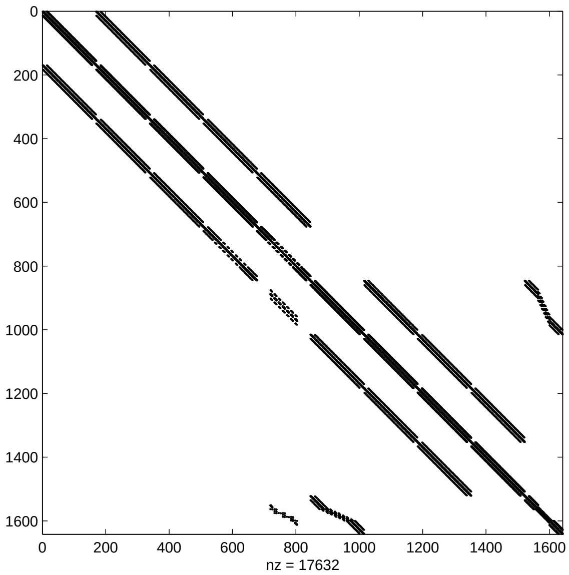

Figure 3: Matrix of a two-material problem

that the general split algorithm gives the same number of iterations as the optimal split, generated by the regular algorithm. The penalty for using an even splitting is a factor of almost 4 in iterations. By comparison, we give the number of iterations for the Jacobi method. In the case of the larger matrix we see that through fortuitous circumstances the general splitting performs marginally better than the ‘optimal’ one. Again there is a large penalty for choosing incorrect blocks as the even splitting does.

optimal splitting	73
general splitting	73
even splitting	261
jacobi preconditioner	494

Table 1: Iteration counts for differently split Schwarz preconditioners on a small matrix problem

optimal splitting	145
general splitting	138
even splitting	465
jacobi preconditioner	1044

Table 2: Iteration counts for differently split Schwarz preconditioners on a medium size matrix problem

There are some opportunities for refinement of the algorithms developed here. In our algorithms we used the ‘fact’ that the upper right corner of a block in the upper triangle of a matrix is zero. This fact does not hold if the differential equation has periodic boundary conditions. We aim to develop heuristics that can detect this case.

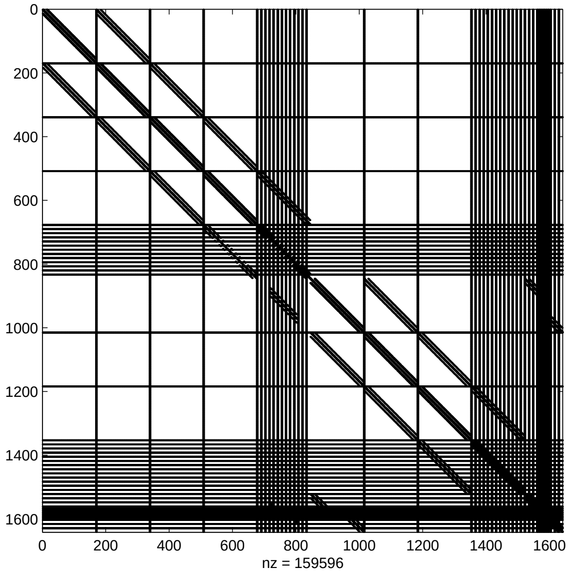

Figure 4: Same matrix as figure 3, with all split points found indicated

References

[1] D. C. Arnold, S. Blackford, J. Dongarra, V. Eijkhout, and T. Xu. Seamless access to adaptive solver algorithms. In M. Bubak, J. Moscinski, and M. Noga, editors, *SGI Users' Conference*, pages 23–30. Academic Computer Center CYFRONET, October 2000.

[2] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[3] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM proceedings of the 24th National Conference*, 1969.

[4] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matices.* Clarendon Press, Oxford, 1986.

A Software: Regular matrix partitioner

```
function [sinfo,structs] = blockstructures(A)
% function [sinfo,structs] = blockstructures(A)
%
% compute all possible block structures of a matrix
% sinfo(i,:) is i’th band and length of corresponding splits array;
%   (number of blocks is one less than length of splits)
% structs(i,:) is i’th splits, padded with zeros.

b = bands(A);

sinfo = []; structs = []; ol = 0;
for i=1:size(b,2),
  splits = blockstructure(bandpart(A,b(i)));
  if size(splits,2)==2,
```

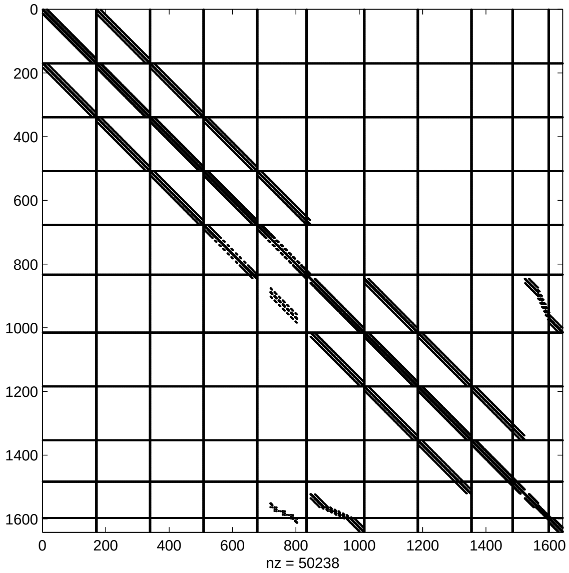

Figure 5: Same matrix as figure 4, after consolidation of small blocks

```
    fprintf(’no useful partition for b=%d\n’,b(i)); break; end;
  l = size(splits,2); [m,n] = size(structs);
  if l==ol,
    if ol>0, new = setdiff(splits,osplits);
      if size(new,1)>0,
        fprintf(’ERROR non-nested splits at band %d of %d\n’,i,b(i)); break;
      end; end;
    fprintf(’band %d of %d supersedes previous.\n’,i,b(i));
    sinfo(m,:) = [b(i),l]; structs(m,:) = [splits,zeros(1,n-l)];
  else,
    fprintf(’band %d of %d gives %d blocks.\n’,i,b(i),l-1);
    sinfo = [sinfo’,[b(i),l]’]’; structs(m+1,1:l) = splits;
  end;
  ol = l; osplits = splits;
end;

fprintf(’Number of partitions found: %d\n’,size(structs,1));
function [splits] = blockstructure(A);
% function [splits] = blockstructure(A)
%
% make vector of splits points of the block structure
% last split is n+1.
%

[m,n]=size(A);
splits = [1];
for i=2:m,
  J = find(A(i,1:i-1));
  if size(J,2)==0,
    [I,J]=find(A(i:m,1:i-1));
    if size(I,2)==0,
      splits = [splits, i];
    end;
  end;
end;
```

```
splits = [splits, m+1];
function [right,left] = bands(A)
% function [right,left] = bands(A)
%
% find all outer bands in the matrix, that is, diagonals
% such that the diagonal beyond it is entirely zero.
%
% right,left are arrays of positive numbers, sorted up.
% if only one output is request, the union of left & right is returned.

[m,n] = size(A);

right = []; bp = n;
for b=n-1:-1:1,
  d = diag(A,b); [I,J] = find(d);
  if size(I,1)>0,
    if b==bp-1, right = [b,right]; end;
  else, bp = b; end;
end;

left = []; bp = n;
for b=n-1:-1:1,
  d = diag(A,-b); [I,J] = find(d);
  if size(I,1)>0,
    if b==bp-1, left = [b,left]; end;
  else, bp = b; end;
end;

if nargout==1, right = union(right,left); end;
function [M] = bandpart(A,n);
% function [M] = bandpart(A,n)
%
% n scalar: take the 2n+1 inner bands of A.
% n vector: take part inside split points.
%           see: block_make
%

if size(n,2)==1,
  if n>size(A,1)-1,
    M = A;
  else,
    M=triu(A,-n)-triu(A,n+1);
  end;
else,
  M = block_make(A,n);
end;
```

B Software: General matrix partitioner

```
function [pp,ppt] = find_cm(A);
% function [pp,ppt] = find_cm(A);

trace = 1;

%
% initially, get all points on the first
% sub/super diagonal that allows a split
%
A = A+A’;
splits = blockstructure(bandpart(A,1));
```

```
if trace>0, fprintf(’a priori block structure: %s\n’,vec2str(splits)); end;

%
% Find all blocks
%
[pp,rr] = all_blocks(A,splits);

%
% Now find a string of blocks that looks like CM;
% set limits on growth
%
pp = string_of_blocks(pp,rr,splits,trace);
fprintf(’block structure prelim: %s\n’,vec2str(pp));

%
% post-processing to eliminate small blocks
%
ppt = pp; pp = compact_blocks(pp,0);
fprintf(’block structure final: %s\n’,vec2str(pp));

%%
%% end of main function
%%

%
% main function 1: all_blocks
%
function [pp,rr] = all_blocks(A,splits);

[m,n] = size(A);
nsplits = size(splits,2);
pp = sparse(nsplits,nsplits); rr = pp;

% Loop over all split points, and assume that they are the start of a block;
% find all points that can be the end of that block.
%
for first=1:nsplits-1,
  this_split = splits(first);
  % init; this also covers the case of the last block, for which the
  %       following loop is not executed
  p = [first]; r = [first];
  for next=first+1:nsplits-1,
    next_split = splits(next); test_size = next_split-this_split;
    add = 0; d = floor(test_size/10);
    % first_split is first point of block of current block
    % next_split is tentative first point of next block.
    % Now test whether connected:
    % 1/ first point not to last point
    % 2/ first point to next first point
    % 3/ previous last point to next last point
    % 4/ previous last point not to next first point
    t1 = empty_corner(A,this_split,next_split-1,d,-d);
    t2 = empty_corner(A,this_split,next_split,d,d);
    t3 = empty_corner(A,this_split-1,next_split-1,-d,-d);
    t4 = empty_corner(A,this_split-1,next_split,-d,d);
    if t1==0 & t2>0,
      if this_split==1 | (t3>0 & t4==0), p = [p,next];
      % if only conditions 1 and 2 are met, mark this as a restart point
      else r = [r, next]; end; end;
  end;
  fprintf(’%d: blocks from %d: %s, restarts: %s\n’,...
          first,this_split,vec2str(splits(p)),vec2str(splits(r)));
  pp(first,1:size(p,2)) = p; rr(first,1:size(r,2)) = r;
end;

[I,J] = find(pp);
pp = full(pp(1:max(max(I)),1:max(max(J))));
[I,J] = find(rr);
rr = full(rr(1:max(max(I)),1:max(max(J))));

%
% auxiliary function empty_corner
%
% test whether a corner of matrix (coordinate plus i/j offset) is empty
% positive result: elements nonzero, zero result: empty
%
function res = empty_corner(A,i,j,di,dj)
[m,n] = size(A);
if i<1 | i>m, res = 0;
else,
  if di<0, i0=max(1,i+di); i1=i; else, i0=i; i1=min(m,i+di); end;
  if dj<0, j0=max(1,j+dj); j1=j; else, j0=j; j1=min(n,j+dj); end;
  res = norm(A(i0:i1,j0:j1),inf);
end;

%
% main function 2 string_of_blocks
%
function pp = string_of_blocks(pp,rr,splits,trace);

% control parameters
growth = 1.5;
% setup
nsplits = size(splits,2);
start = 1; p = [start]; big_block = 0;

% we loop maximally to the number of splits, in practice much less,
% see the break command at the end of the loop
for seq=1:nsplits-1,
  % look at all possible blocks from this point
  [I,J] = find(pp(start,:)); last_p = max(max(J));
  [I,J] = find(rr(start,:)); last_r = max(max(J));
  ps = pp(start,2:last_p); rs = rr(start,2:last_r);
  if trace>0, fprintf(’@%d => %d : choices are %s %s\n’,...
        start,splits(start),vec2str(splits(ps)),vec2str(splits(rs))); end;

  % tough case: there is no continuation;
  % use a restart block, which satisfies a less stringent test
  if last_p==1,
    % first see if we can restart the process
    if last_r>1,
      if trace>0, fprintf(’.. stuck; restarting\n’); end;
      % find a block that doesn’t grow too fast
      next = decent_block(rs,start,splits,growth,big_block,trace);
    % if we cannot even restart the process, just take the next block
    else, next = start+1;
      if trace>0, fprintf(’.. really stuck; taking the next block\n’); end;end;

  % in the regular case, look at all possibilities for the end point;

  % if we are only starting the process, just take the biggest jump
  elseif big_block==0, next = pp(start,last_p);
    if trace>0, fprintf(’.. starting out; just take the biggest.\n’); end;
  % otherwise, limit growth
  else,
    next = decent_block(ps,start,splits,growth,big_block,trace);
  end;

  % update parameters
  p = [p,next]; this_size = splits(next)-splits(start);
  big_block = max(big_block,this_size);

  % if we have exhausted the matrix, quit.
  if next>=nsplits, break; else, start = next; end;
end;

pp = splits(p); % convert to real numbering.

%
% auxiliary function decent_block
%
% find continuation canditate that doesn’t grow too fast
% result next is index in array nexts
%
function next = decent_block(nexts,start,splits,growth,big_block,trace)
last = size(nexts,2);
for pos=last:-1:1,
  next = nexts(pos);
  next_size = splits(next)-splits(start);
  if next_size<=growth*big_block,
    if trace>0,
      fprintf(’.. based on growth taking %d=#%d out of %d\n’,...
              next,pos,last); end;
    break; end;
end;

%
% main function 3 compact_blocks
%
function pp = compact_blocks(pp,trace);

ppt = pp; np = size(ppt,2); pp = []; last_size = 0; avg_size = 0; nb = 0;
for p=1:np-1,
   this_size = ppt(1,p+1)-ppt(1,p);
   if p>1, % general block: test whether too small
     if this_size<avg_size/6, % if it’s small, accumulate it
       if r>0, % if we are already accumulating,
         if p==np-1, add = r; % if last block, flush
         else, % in general, add and flush if big enough
           cum = cum+this_size;
           if cum>5*avg_size/6; add = r; else, add = 0; end; end;
       else, % if we are not accumulating, start doing it now
         if p==np-1, add = p; % if last block, flush
         % in general, remember where we started
         else add = 0; r = p; cum = this_size; end;
       end;
     else, add = p; end; % if the block is not too small, just accept
   else, % first block: add
     add = 1; end;
   if add>0,
     if trace>0, fprintf(’accepting block %d: %d\n’,p,ppt(1,p)); end;
     last_size = this_size; avg_size = nb*avg_size+this_size;
     pp = [pp, ppt(1,add)]; nb = nb+1; avg_size = avg_size/nb;
     r = 0;
   else, if trace>0, fprintf(’merging block %d: %d\n’,p,ppt(1,p)); end;
   end;
end;

pp = [pp, ppt(1,np)];
```