INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN BIOMEDICAL ENGINEERING
Int. J. Numer. Meth. Biomed. Engng. 2010; **26**:628–636
Published online 20 August 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cnm.1160
COMMUNICATIONS IN NUMERICAL METHODS IN ENGINEERING

Cache-efficient renumbering for vectorization

Rainald Löhner*,†

CFD Center, Department of Computational and Data Science, M.S. 6A2, College of Sciences, George Mason University, Fairfax, VA 22030-4444, U.S.A.

SUMMARY

A renumbering strategy for field solvers based on unstructured grids that avoids memory contention and minimizes cache-misses is described. Compared with usual colouring techniques, the new renumbering strategy reduces the spread in point-data access for edge-based solvers by more than an order of magnitude. The technique is particularly suited for multicore, cache-based machines that allow for vectorization or pipelining. Copyright © 2008 John Wiley & Sons, Ltd.

Received 11 December 2007; Revised 21 June 2008; Accepted 21 June 2008

KEY WORDS: grid renumbering; unstructured grids; vector machines; multicore machines; cache

1. INTRODUCTION

A considerable percentage of field solvers is presently run on computers whose microchips have a cached memory and allow for vectorization (also known as pipelining). In order to enable vectorization, a point in any given edge/element/face-loop should only be accessed once. This requirement can lead to a considerable jump in memory access for points (and their associated data) when looping through edges/elements/faces, leading to an increase in cache-misses and the associated decrease in performance. Given the rise in available memory, the extensive use of shared memory and the natural urge to solve ever larger problems, these problems are expected to increase in the future. The present paper presents a renumbering technique to mitigate the increase in memory access jumps.

2. RENUMBERING TO AVOID CACHE-MISSES

Consider the following loop over edges that typifies the central loop of many field solvers based on unstructured grids [1–6], written in Fortran pseudo-code and shown schematically in Figure 1:

Loop 1

```
      do 1600 iedge=1,nedge
      ipoi1=lnoed(1,iedge)
      ipoi2=lnoed(2,iedge)
      redge=geoed( iedge)*(unkno(ipoi2)-unkno(ipoi1))
      rhspo(ipoi1)=rhspo(ipoi1)+redge
      rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600 continue
```

*Correspondence to: Rainald Löhner, CFD Center, Department of Computational and Data Science, M.S. 6A2, College of Sciences, George Mason University, Fairfax, VA 22030-4444, U.S.A.
†E-mail: rlohner@gmu.edu

A right-hand side (RHS), or residual, is formed at the edge-level (`redge`) by gathering information from a vector of point unknowns (`unkno`) and performing a series operations on them. This edge-RHS is then added to a global point-RHS (`rhspo`). The array `lnoed` stores the end-points of each edge. Similar loops are obtained for element- or face-based solvers, and what follows is equally applicable to them.

If cache-misses are a concern, then it is clear that the storage locations for the required point information stored in the arrays `unkno` and `rhspo` should be as close as possible in the memory when required by an edge. At the same time, as the loop progresses through the edges, the point information should be accessed as uniformly as possible. This may be achieved by first renumbering the points using a bandwidth-minimization technique (e.g. Reverse Cuthill McKee [7], wavefront [8], recursive bisection, space-filling curve, etc.), and subsequently renumbering the edges according to the minimum point number on each edge [8, 9]. This will result in a near-optimal access of the memory as shown in Figure 2. All of these renumbering algorithms are of complexity $O(N)$ or at most $O(N \log N)$, and are well worth the effort.

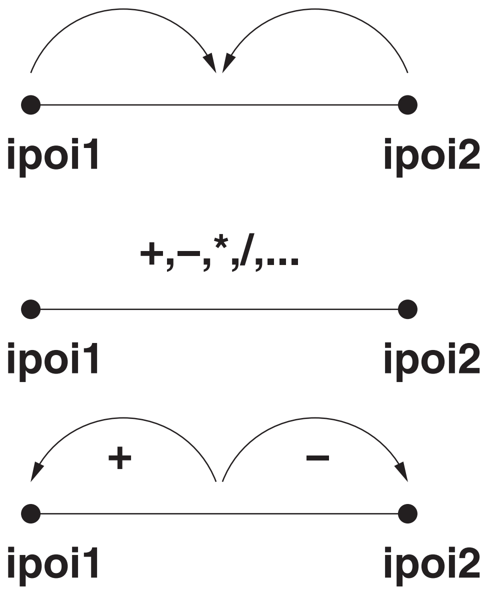

Figure 1. Basic edge-loop.

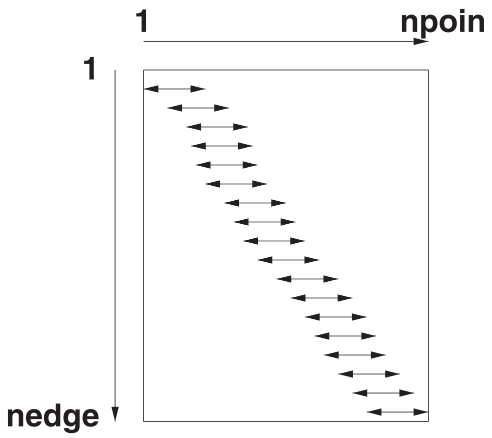

Figure 2. Point access in edge-loop.

3. AVOIDANCE OF MEMORY CONTENTION

Pipelining or vectorization offers the possibility of substantial performance gains on any kind of system. While previously restricted to so-called vector machines, such as those manufactured by CRAY, Convex, NEC, Fujitsu or Hitachi, the concept has migrated to current microchips, such as the Intel Xeon, Intel Itanium, AMD Opteron and IBM Power-series. In order to achieve pipelining or vectorization, memory contention must be avoided. The enforcement of pipelining or vectorization is then carried out using a compiler directive, as Loop 1, which becomes an inner loop, still offers the possibility of memory contention. In this case, we would have

Loop 2

```
       do 1400 ipass=1,npass
       nedg0=edpas(ipass)+1
       nedg1=edpas(ipass+1)
c$dir ivdep                                     ! Pipelining directive
         do 1600 iedge=nedg0,nedg1
         ipoi1=lnoed(1,iedge)
         ipoi2=lnoed(2,iedge)
         redge=geoed( iedge)*(unkno(ipoi2)-unkno(ipoi1))
         rhspo(ipoi1)=rhspo(ipoi1)+redge
         rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600  continue
 1400 continue
```

It is clear that in order to avoid memory contention, for each of the groups of edges (`1600 loop`), none of the corresponding points should be accessed more than once. Given that in order to achieve good pipelining performance on current microchips, a relatively short vector length of `mvecl=16` or `mvecl=32` is sufficient, one can simply start from the edge-renumbering obtained in order to minimize cache-misses, and renumber it further into groups of edges that are 16 or 32 long and avoid memory contention [8, 9]. The most straightforward renumbering or 'colouring' realization is given by the following algorithm:

```
       ledge(1:nedge)=0              ! initialize new edge number array
       lpoin(1:npoin)=0                ! initialize point marking array
       nenew=0                             ! initialize new edge counter
       npass=0                            ! initialize edge pass counter
       nedg0=1                                    ! initialize start edge
 1000 continue                               ! open loop over the passes
       npass=npass+1                                ! update pass counter
       nvecl=0                      ! initialize current vector length
       do 1200 iedge=nedg0,nedge
       if(ledge(iedge).eq.0) then                      ! edge is unmarked
           if(lpoin(lnoed(1,iedge)).ne.npass) then     ! point 1 unused
               if(lpoin(lnoed(2,iedge)).ne.npass) then ! point 2 unused
                   nenew=nenew+1                             ! store edge
                   ledge(iedge)=nenew
                   lpoin(lnoed(1,iedge))=npass              ! mark points
                   lpoin(lnoed(2,iedge))=npass
                   nvecl=nvecl+1                 ! update vector counter
                   if(nvecl.eq.mvecl) goto 1201  ! desired vector length
               endif
           endif
        endif
 1200 continue
 1201 continue
```

```
      edpas(npass)=nenew                              ! store pass counter
      do 1400 iedge=nedg0,nedge
      if(ledge(iedge).eq.0) goto 1401                   ! edge is unmarked
 1400 continue
 1401 continue
      nedg0=min(iedge,nedge)
      if(nenew.ne.nedge) goto 1000            ! unrenumbered edges left
```

As before, this renumbering is of complexity $O(N)$.

4. AVOIDANCE OF MEMORY CONTENTION AND CACHE-MISSES

The basic edge-renumbering technique outlined above has the disadvantage of not taking into account the jumps in memory access for the second index. As the edges are numbered according to ascending point numbers, the first index will lead to a monotonic indirect access of memory with limited jumps, and hence low cache-misses. However, the second index can jump considerably. In order to see how this happens, consider the portion of the 2-D mesh shown in Figure 3.

Some edges will access near-neighbour information on the current 'line', while others will access near-neighbour information from the next 'line'. For the case shown, the first vector group accesses points along the 'line' $i, i+1$. When starting the second vector group, the only edge-connection left for point i is i, j. However, the next edge in this group will use points $i+1, i+2$, producing a large jump in the second index. In 3-D the situation is even worse, as we have hyperplanes, i.e. a much higher jump in memory between neighbours. The aim is therefore to reduce the jumps as much as possible in the second index.

Of the many algorithms that may be devised to achieve this, the following offers a good compromise of simplicity and performance. The assumption is made that the edges have been ordered according to points, i.e. the first edge-point (first index) increases monotonically as one progresses through the edges. At the start of a new edge-group, the second point of the first edge of the group *must* be taken. The aim is then to find as many edges as possible that avoid memory contention, and at the same time minimize the jump in the second edge-point (second index). As the edges have been ordered according to ascending first edge-point numbers, all second edge-points for a given first edge-point can be tested immediately. The idea is then to keep the second edge-point that minimizes the jump with respect to the first second edge-point of the vector group. Interestingly, this by itself does not yield the expected reduction in jumps for the second edge-points. The reason is that in many cases the edges that lead to no memory contention have large jumps with respect to the second edge-point of the first edge in the vector group. Clearly, this depends on the topology of the mesh and the renumbering technique chosen, but experience with many cases indicates that it is quite common. The solution is to increase the search for vector-group candidates to a multiple of the desired vector length (we have used four times the desired vector length), and then select the best from these. This will lead to a larger range in the values of the first points of the vector group. However, as will be seen from the examples,

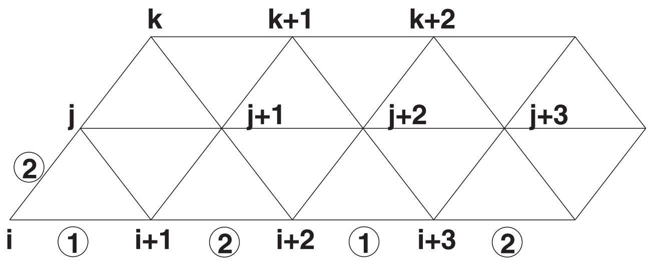

Figure 3. Edge-groups.

this increase is modest, and the reduction obtained for the second edge-point range more than compensates for this increase. Although lengthier, we include the new renumbering algorithm here for completeness.

```
      ledge(1:nedge)=0              ! initialize new edge number array
      lpoin(1:npoin)=0                ! initialize point marking array
      nenew=0                            ! initialize new edge counter
      npass=0                           ! initialize edge pass counter
      nedg0=1                                  ! initialize start edge
 1000 continue                             ! open loop over the passes
      npass=npass+1                              ! update pass counter
      nvecl=0                      ! initialize current vector length
      nedgl=0                        ! initialize local storage list
      ipoi1=lnoed(1,nedg0)                         ! add the 1st edge
      ipoi2=lnoed(2,nedg0)
      nenew=nenew+1
      nvecl=nvecl+1
      ledge(nedg0)=nenew
      lpoin(ipoi1)=npass
      lpoin(ipoi2)=npass
      if(nedg0.eq.nedge) goto 1201               ! renumbering complete
      kpoi1=ipoi1
      kvecl=  0
      do 1200 iedge=nedg0+1,nedge
      if(ledge(iedge).eq.0) then                     ! edge is unmarked
         ip1=lnoed(1,iedge)
         ip2=lnoed(2,iedge)
         if(lpoin(ip1).ne.npass) then                 ! point 1 unused
            if(lpoin(ip2).ne.npass) then              ! point 2 unused
               if(ip1.gt.kpoi1) then                   ! new 1st index
                  if(kvecl.eq.0) then                      ! new point
                     kpoi1=ip1                ! update new 1st index
                     kvecl= 1                   ! initialize counter
                     ijum1=abs(ipoi1-ip1)         ! initialize jumps
                     ijum2=abs(ipoi2-ip2)
                     iminl=iedge      ! initialize min(jump) edge
                     ijmi1=ijum1
                     ijmi2=ijum2
                     ipmi1=ip1
                     ipmi2=ip2
                  else
                     nedgl=nedgl+1                ! already compared
                     ledgl(nedgl)=iminl             ! transcribe best
                     lqual(nedgl)=ijmi2        ! transcribed quality
                     lpoin(ipmi1)=npass                ! mark points
                     lpoin(ipmi2)=npass
                     kvecl=0                  ! reset vector length
                  endif
               else
                     kvecl=kvecl+1                  ! point is known
                     ijum1=abs(ipoi1-ip1)                  ! compare
                     ijum2=abs(ipoi2-ip2)
                     if(ijum2.lt.ijmi2) then
                        iminl=iedge                 ! keep as better
```

```
                         ijmi1=ijum1
                         ijmi2=ijum2
                         ipmi1=ip1
                         ipmi2=ip2
                       endif
                  endif
               endif
            endif
         endif
         if(nedgl.ge.mxedl) goto 1201                ! group is big enough
 1200 continue
 1201 continue
         if(nedgl.eq.0) goto 1301                     ! no edges in group
         nvecl=min(mvecl,nedgl)
         call iordv(nedgl,nvecl,lqual,leorl)            ! get nvecl best
         nene0=nenew                       reset start edge for new search
         do 1300 iedgl=1,nvecl                         ! add the nvecl best
         iedge=ledgl(leorl(iedgl))
         nenew=nenew+1
         ledge(iedge)=nenew
 1300 continue
 1301 continue
         edpas(npass)=nenew                           ! store pass counter
         do 1400 iedge=nedg0,nedge
         if(ledge(iedge).eq.0) goto 1401               ! edge is unmarked
 1400 continue
 1401 continue
         nedg0=min(iedge,nedge)
         if(nenew.ne.nedge) goto 1000          ! unrenumbered edges left
```

As the local vector length is assumed small, the local ordering of edges (`call iordv(...)`) may be carried out by either using a simple O(`nedgl*nvecl`) exhaustive search, or any faster ordering technique. For the examples shown below a heap-sort technique was used.

5. EXAMPLES

The renumbering strategies described were coded into FEFLO, an adaptive, edge-based finite element code for the solution of compressible and incompressible flows [10].

5.1. NACA0012

The first example considered is the classic NACA0012 wing at $\alpha = 5^\circ$ angle of attack. This is a steady, inviscid (Euler) case. A projection-type incompressible solver with upwind edge limiting for the advective terms and 4th-order damping for the divergence constraint [9] is used. Figure 4(a) and (b) shows the surface mesh employed, as well as the surface pressures obtained. Although the mesh is rather coarse (`nelem=370,514, npoin=68,664, nedge=451,187`), it still allows for a meaningful comparison of the different renumbering schemes.

Table I summarizes the average maximum jumps for the first and second points `jump1, jump2` and total average jump between first and second points `jmp12` in each vector group, as well as the average edge-to-edge jumps for the first and second points `jum1a, jum2a` in each vector group. As expected, the jumps increase with vector group size. As compared with the simple renumbering technique, the new technique leads to a moderate increase in jumps for the first point (which is low in any case) and a drastic decrease in jumps for the second point. This case was repeated with a finer mesh (`nelem=1,450,793, npoin=259,711, nedge=1,739,575`), and the

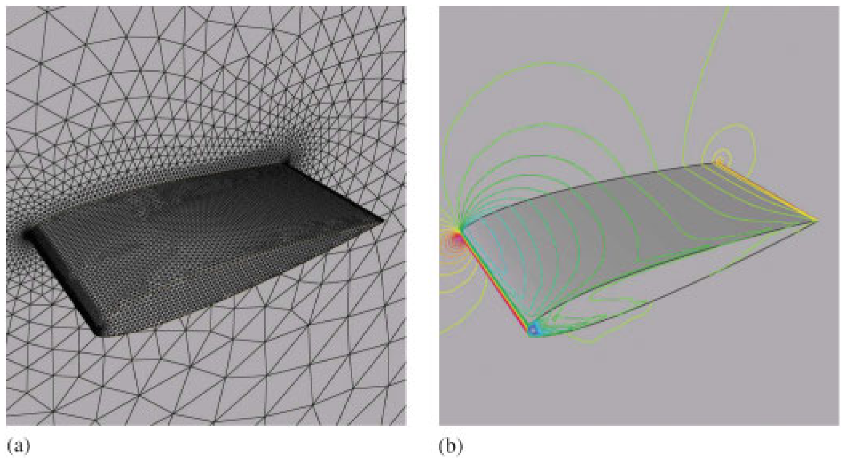

Figure 4. (a,b) NACA 0012: surface mesh and pressure.

Table I. NACA0012 wing: Coarse mesh.

Method	mvecl	jump1	jump2	jum1a	jum2a	jum12
Original	16	23	2465	1.57	597	2500
Improved	16	79	282	5.27	63	1505
Original	32	55	2498	1.77	572	2534
Improved	32	183	412	5.91	70	1625
Original	64	123	2516	1.95	562	2585
Improved	64	410	630	6.52	83	1840

Table II. NACA0012 wing: Fine mesh.

Method	mvecl	jump1	jump2	jum1a	jum2a	jum12
Original	16	23	5708	1.56	1382	5757
Improved	16	79	516	5.25	115	3351
Original	32	54	5760	1.75	1331	5796
Improved	32	180	758	5.81	133	3535
Original	64	121	5780	1.94	1332	5848
Improved	64	398	1063	6.32	147	3793

results are summarized in Table II. Note that the differences between the old and new renumbering remain almost unchanged.

5.2. Dam-break with column

This case considers a column of water collapsing close to a square column. As before, a projection-type incompressible solver with upwind edge limiting for the advective terms and fourth-order damping for the divergence constraint [9] is used. The free surface is captured using a volume of fluid approach [11]. A closeup of the surface mesh, as well as a typical solution are shown in Figure 5(a) and (b).

This was a larger mesh (`nelem=4,902,314, npoin=857,901, nedge=5,814,629`). Table III summarizes the average jumps for the first and second points in the vector groups of length `mvecl=16,32,64`.

As before, the jumps increase with vector group size, and as compared with the simple renumbering technique, the new technique leads to a moderate increase in jumps for the first point and a drastic decrease in jumps for the second point.

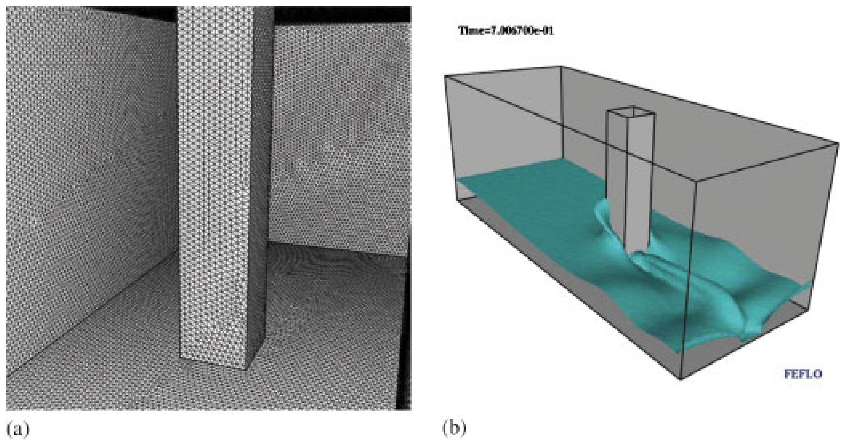

Figure 5. (a,b) Dam-break with column: surface mesh and free surface.

Table III. Dam-break with column.

Method	mvecl	jump1	jump2	jum1a	jum2a	jum12
Original	16	24	6249	1.59	1552	6296
Improved	16	79	521	5.30	116	3617
Original	32	56	6285	1.80	1498	6323
Improved	32	180	748	5.81	130	3790
Original	64	125	6304	1.99	1483	6375
Improved	64	394	1057	6.26	144	4057

Regarding the run-times, the new renumbering technique leads to a modest run-time reduction of 10% as compared with the old renumbering technique. The machine on which the problem was run had Intel Xeon quadcore processors with 2 Mbyte cache per core. It is expected that for larger problem sizes these differences will increase.

6. CONCLUSIONS

A renumbering scheme has been developed that allows vectorization while reducing significantly the jumps in memory access for edge-based solvers. The reduction should lead to a considerable decrease in cache-misses for large problems, thereby improving CPU performance.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the input and encouragement of Dr J. D. Baum from SAIC, whose push for the one-billion element flow case on shared memory machines spurred this effort.

REFERENCES

1. Barth T. A 3-D upwind Euler solver for unstructured meshes. *AIAA-91-1548-CP*, 1991.
2. Mavriplis D. Three-dimensional unstructured multigrid for the Euler equations. *AIAA-91-1549-CP*, 1991.
3. Jameson A. *The AIRPLANE Code*. Private communication, January, 1992.
4. Peraire J, Peiro J, Morgan K. A three-dimensional finite element multi-grid solver for the Euler equations. *AIAA-92-0449*, 1992.
5. Luo H, Baum JD, Löhner R, Cabello J. Adaptive edge-based finite element schemes for the Euler and Navier–Stokes equations. *AIAA-93-0336*, 1993.
6. Weatherill NP, Hassan O, Marcum DL. Calculation of steady compressible flowfields with the finite element method. *AIAA-93-0341*, 1993.

7. Cuthill E, McKee J. Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the ACM National Conference*, New York, 1969; 157–172.
8. Löhner R. Some useful renumbering strategies for unstructured grids. *International Journal for Numerical Methods in Engineering* 1993; **36**:3259–3270.
9. Lohner R. *Applied CFD Techniques* (2nd edn). Wiley: New York, 2008.
10. Löhner R. FEFLO theoretical manual. *GMU-CSI-CFD Lab. Report*, 2003.
11. Löhner R, Yang C, Oñate E. On the simulation of flows with violent free surface motion. *Computer Methods in Applied Mechanics and Engineering* 2006; **195**:5597–5620.