EFFECTIVE USE OF HARDWARE

Why:

- Time is Insight ⇒
- Algorithms Have to Run Efficiently on Hardware
- Combination Hardware/Software Allows for Runs
  - Larger
  - More Physics
- No Perfect Hardware

CFD HARDWARE

- PCs
    - Cache
    - Small RAM
    - Graphics
- Workstations
    - Cache
- Vector Machines
    - Degradation for Scalar Ops
    - Appropriate Coding
    - No Graphics
- Parallel Machines
    - Cache
    - Degradation for Scalar Ops
    - Appropriate Coding for Message Passing

$\Rightarrow$ Nobody is Perfect

MICROPROCESSOR TRENDS

- Number of Transistors Doubles Every 18 Months
  Moore's Law
- Possible Number of Operations Increases Faster Then Memory Access
- L1, L2, L3 Caches
- Pipelining, Vectorization, SIMD
- Coding:
  - Do Many Operations on Same Data
    (Low Data Dependency)
  - Blend Instructions
  - Allow > 4 Operations at the Same Time
  - Enable Prefetching of Data
  - Predictable Branches

REDUCTION OF CACHE-MISSES (1)

When: Need to Fetch Data Beyond Close Proximity

Degradation: In Excess of 1:10 (!)

Examples:

- Particle-Particle Interaction
- Point-Data of an Element/Edge/Face
- Element-Data for a Particle
- Etc...

REDUCTION OF CACHE-MISSES (2)

How: Keep Data That Is Required Close In Memory

By:

- Proper Array Access in Loops
- Renumbering of Points
- Reordering of Element Nodes
- Renumbering of Elements and Edges

Ref: Löhner, *Int. J. Num. Meth. Eng.* 36, 3259-3270 (1993)

PROPER ARRAY ACCESS IN LOOPS (1)

FORTRAN: `A(I,J)`: Stored Column-Wise

⇒ Index `I` Moves Fastest

C, C++: `A(I,J)`: Stored Row-Wise

⇒ Index `J` Moves Fastest

In Sequel, Only Consider **FORTRAN**

PROPER ARRAY ACCESS IN LOOPS (2)

Example: Edge-Vector $\mathbf{x}_j - \mathbf{x}_i$

Flat Storage

```
do iedge=1,nedge
    ipoi1=inpoed(1,iedge)
    ipoi2=inpoed(2,iedge)
    xpoi1=coord(1,ipoi1)          !  Jump to ipoi1
    ypoi1=coord(2,ipoi1)           !  Jump by 1
    zpoi1=coord(3,ipoi1)           !  Jump by 1
    xpoi2=coord(1,ipoi2)          !  Jump to ipoi2
    ypoi2=coord(2,ipoi2)           !  Jump by 1
    zpoi2=coord(3,ipoi2)           !  Jump by 1
    vedge(1,iedge)=xpoi2-xpoi1     !  Jump by 1
    vedge(2,iedge)=ypoi2-ypoi1     !  Jump by 1
    vedge(3,iedge)=zpoi2-zpoi1     !  Jump by 1
enddo
```

PROPER ARRAY ACCESS IN LOOPS (3)

Example: Edge-Vector $\mathbf{x}_j - \mathbf{x}_i$

Vertical Storage: Form 1

```
do iedge=1,nedge
    ipoi1=inpoed(iedge,1)
    ipoi2=inpoed(iedge,2)          !  Jump by nedge
    xpoi1=coord(ipoi1,1)           !  Jump to ipoi1
    ypoi1=coord(ipoi1,2)           !  Jump by npoin
    zpoi1=coord(ipoi1,3)           !  Jump by npoin
    xpoi2=coord(ipoi2,1)           !  Jump to ipoi2
    ypoi2=coord(ipoi2,2)           !  Jump by npoin
    zpoi2=coord(ipoi2,3)           !  Jump by npoin
    vedge(iedge,1)=xpoi2-xpoi1     !  Jump by nedge
    vedge(iedge,2)=ypoi2-ypoi1     !  Jump by nedge
    vedge(iedge,3)=zpoi2-zpoi1     !  Jump by nedge
enddo
```

PROPER ARRAY ACCESS IN LOOPS (4)

Example: Edge-Vector $\mathbf{x}_j - \mathbf{x}_i$

Vertical Storage: Form 2

```
do idimn=1,ndimn
do iedge=1,nedge
    vedge(iedge,idimn)=coord(inpoed(iedge,1),idimn)
enddo
do iedge=1,nedge
    vedge(iedge,idimn)=coord(inpoed(iedge,2),idimn)
                      -vedge(iedge,idimn)
enddo
enddo
```

POINT RENUMBERING (1)

Idea:

- Reflect Closeness in Space in Storage
- Similar to Bandwidth Minimization

Options:

- Coordinate-Based (Plane-by-Plane)
- Bins
- Advancing Front (Wavefront)
- Space-Filling Curve
- Cuthill-McKee
- Recursive Bisection

POINT RENUMBERING (2)

Structured Mesh:

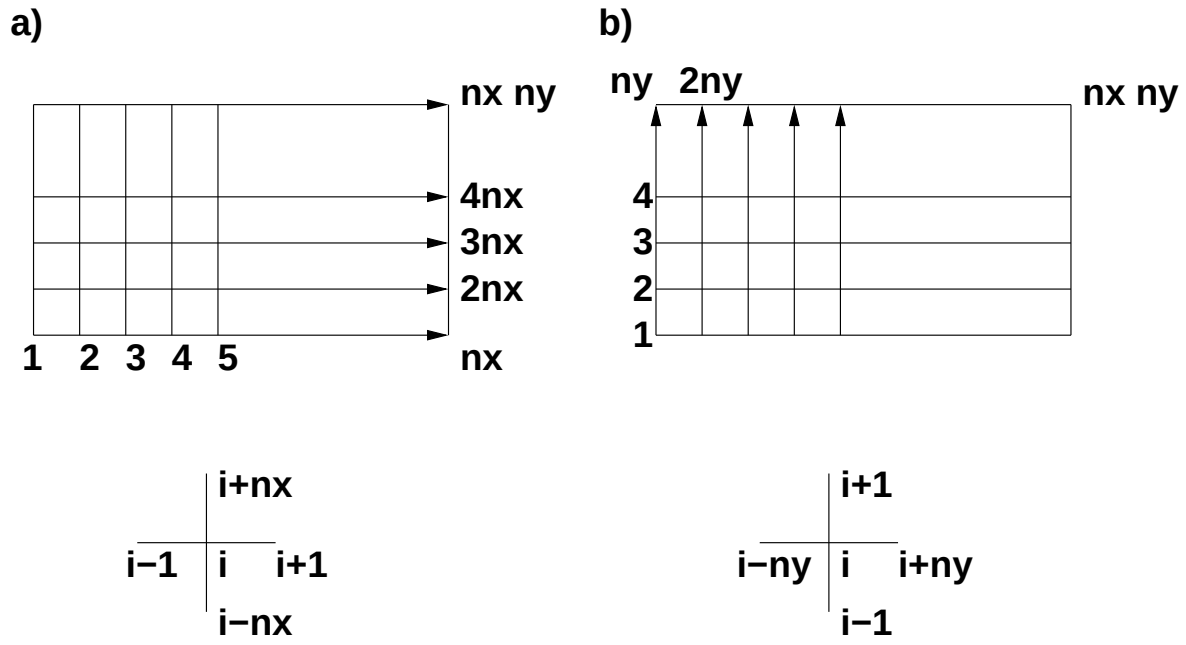
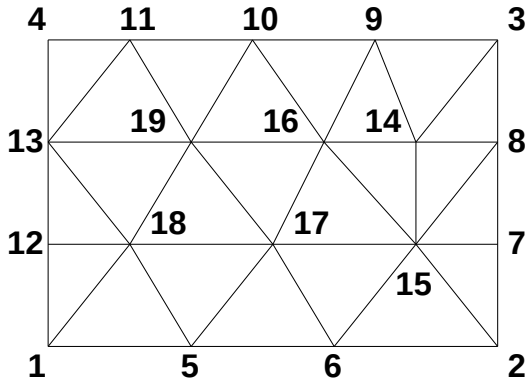
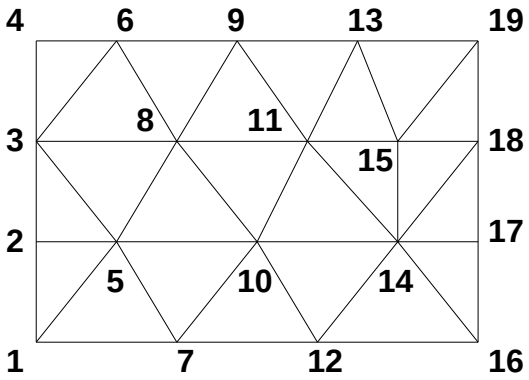


Figure 1 Ordering of Points for 2-D Mesh

POINT RENUMBERING (3)

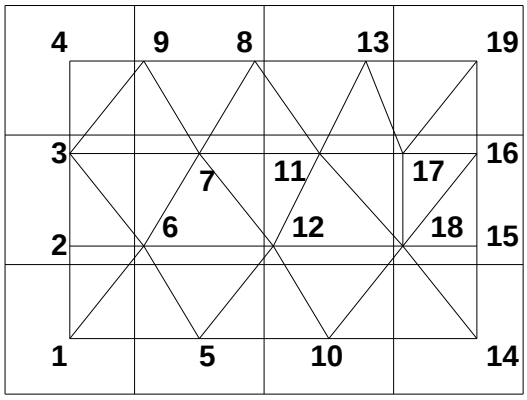


Figures 2 Original Mesh

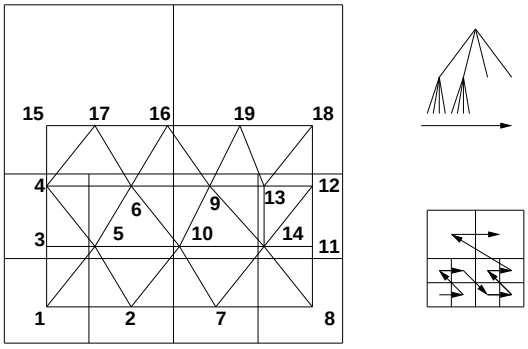


Figures 3 Directional Ordering

POINT RENUMBERING (4)

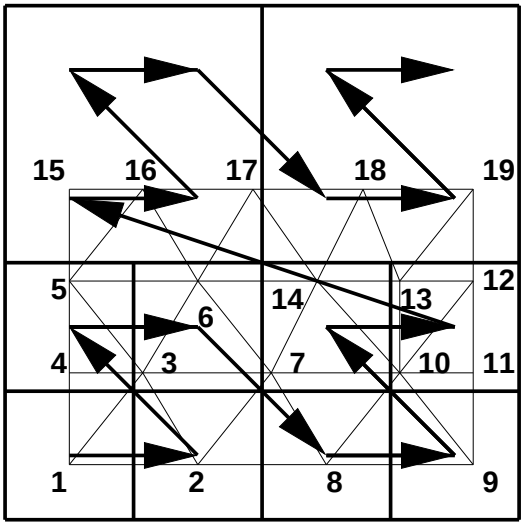


Figures 4 Renumbering Using Bins

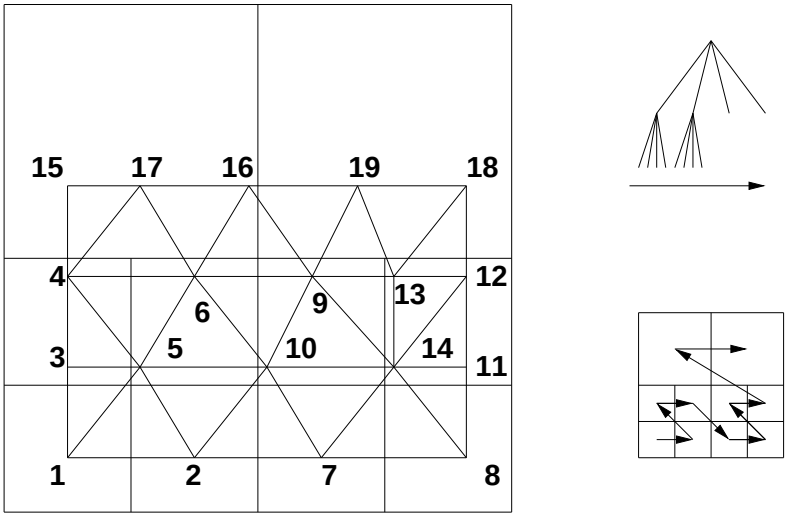


Figures 5 Renumbering Using Quad-Tree

POINT RENUMBERING (5)



Figures 6 Renumbering Using Space-Filling Curve



Figures 7 Renumbering Using Wavefront-Technique

REORDERING OF NODES WITHIN ELEMENTS (1)

Idea:

- Avoid Back/Forth Access of Data

Example:

a) Before:

```
INTMAT(1:4,IELEM)= 100, 5000, 400, 4600 .
```

b) After:

```
INTMAT(1:4,IELEM)= 100, 400, 4600, 5000 .
```

REORDERING OF NODES WITHIN ELEMENTS (2)

Table-Based Algorithm for Tetrahedra

RE1. Define Switching Tables

```
DATA LNOD2/ 0 , 3 , 4 , 2 ,
            4 , 0 , 1 , 3 ,
            2 , 4 , 0 , 1 ,
            3 , 1 , 2 , 0 /
DATA LNOD3/ 0 , 4 , 2 , 3 ,
            3 , 0 , 4 , 1 ,
            4 , 1 , 0 , 2 ,
            2 , 3 , 1 , 0 /
```

RE2. `DO`: Loop Over Elements

- Get the Nodes `INMIN, INMAX` Corresponding to the Minimum and Maximum Point in the Element;
- New Element Definition:

```
IPOI1=INTMAT(             INMIN ,IELEM)
IPOI2=INTMAT(LNOD2(INMAX,INMIN),IELEM)
IPOI3=INTMAT(LNOD3(INMAX,INMIN),IELEM)
IPOI4=INTMAT(             INMAX ,IELEM)
ENDDO
```

REORDERING OF ELEMENTS ACCORDING TO POINTS (1)

Idea:

- When Looping Over Elements: Access Point Data As Uniformly as Possible

Pointer Array Algorithm

R1. Compute the minimum point touching each element and store it in `LEMIN(1:NELEM)`;

R2. Initialize the pointer array `LPOIN(1:NPOIN)=0`;

R3. Loop over the elements, storing in `LPOIN` the number of times the minimum point in each element is accessed;

R4. Add all the entries in `LPOIN` in order to obtain the storage locations for the element reordering;

R5. Loop over the elements:

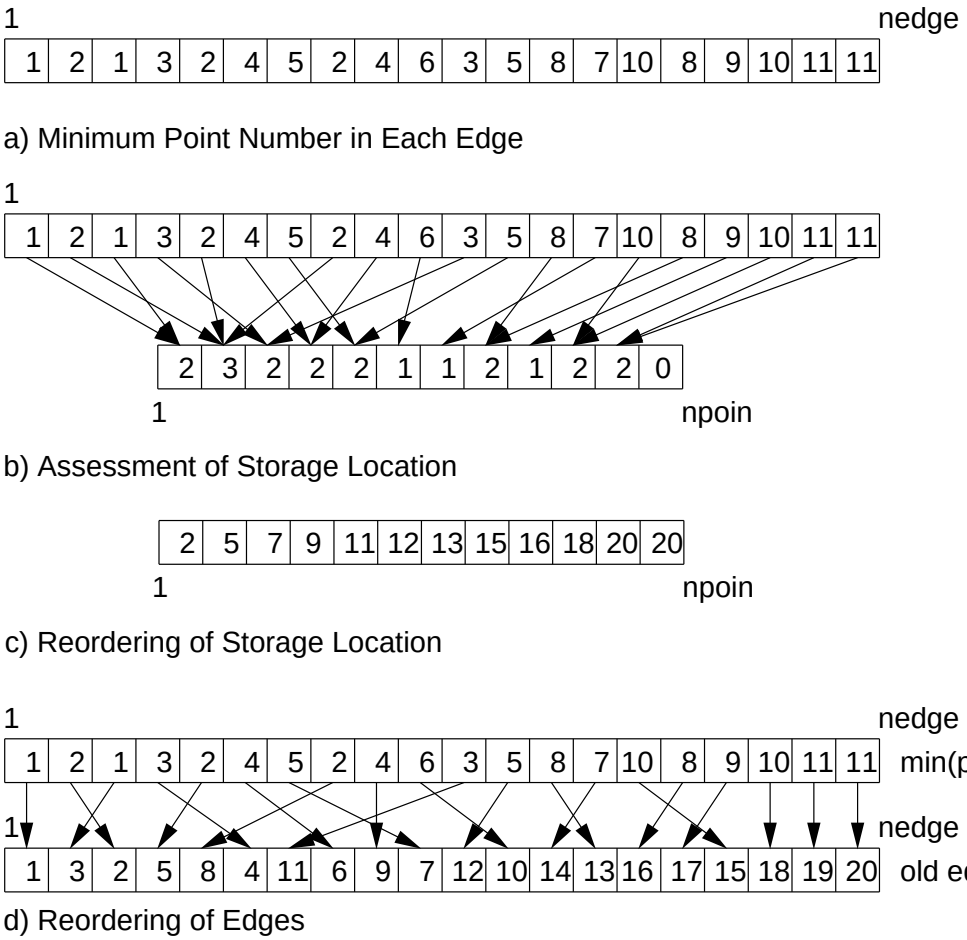
Obtain the new element location from `IPMIN=LEMIN(IELEM)` and `IENEW=LPOIN(IPMIN)`;

Store the new element location: `LENEW(IENEW)=IELEM`;

Update the storage location in `LPOIN`: `LPOIN(IPMIN)=IENEW+1`;

R6. Renumber the elements with `LENEW(1:NELEM)`.

REORDERING OF ELEMENTS ACCORDING TO POINTS (2)



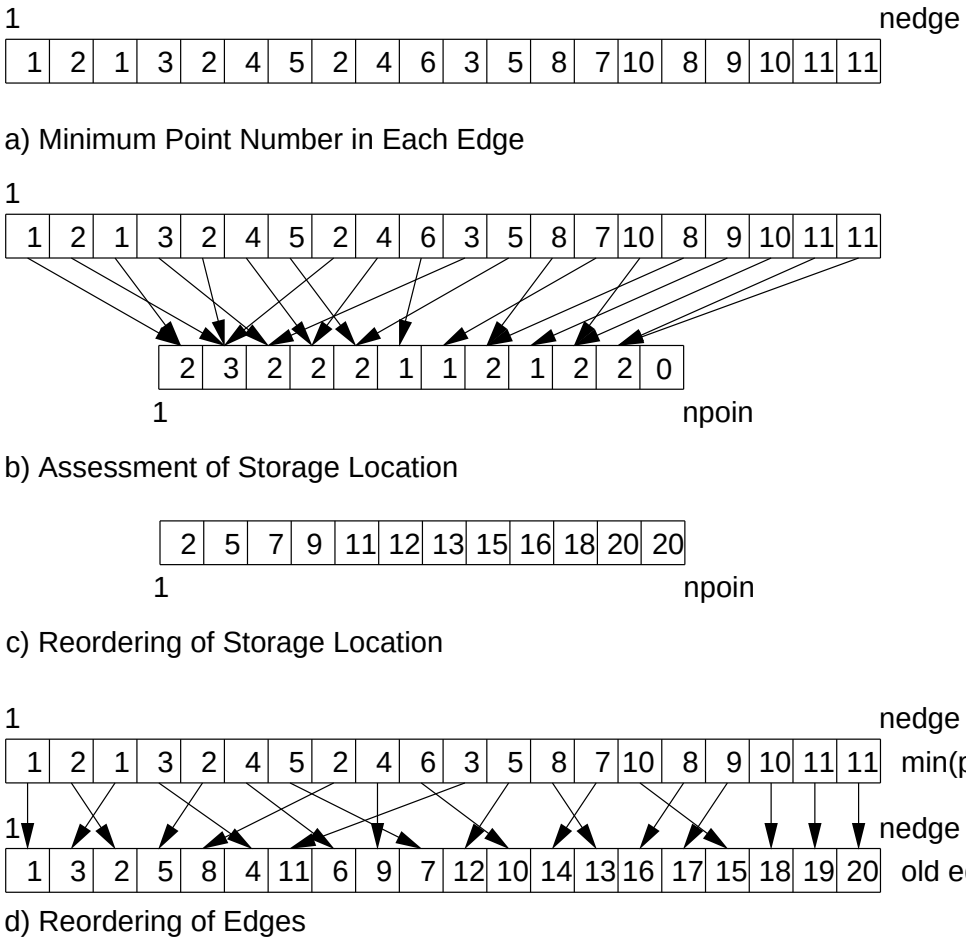
Renumbering of Edges According to Points

REORDERING OF ELEMENTS ACCORDING TO POINTS (3)

Improvement: Two-Pass Strategy

- Pass 1: Order According to Maximum Point Number
- Pass 2: Order According to Minimum Point Number

Use the Same Techniques for Edges/Faces/...



Renumbering of Edges According to Points

Table 1: Timings for Train Problem (240KTetra, 48KPt, 340KEdge)

Computer	FRNTMIN	RENUEMI	ORDNOEL	RENUEL	CPU(se
IBM-6000/530	OFF	OFF	OFF	MVMAX	958
IBM-6000/530	OFF	ON	OFF	MVMAX	882
IBM-6000/530	ON	ON	OFF	MVMAX	834
IBM-6000/530	OFF	ON	OFF	128	597
IBM-6000/530	OFF	ON	OFF	NELEM	534
IBM-6000/530	ON	ON	OFF	NELEM	524
IBM-6000/530	ON	ON	ON	NELEM	510

VECTOR/PIPELINE COMPUTERS

Basic Thought:

- Need To Do Many Times Same Operation
- Subdivide Arithmetic Operation Into Stages
- ⇒ Perform Every Stage On A Different Entry

Example: Add Two Numbers: `A(I)=B(I)+C(I) , I=1,N`

Stages:

- Fetch `B(I)`
- Fetch `C(I)`
- Compare Exponents
- Shift
- Add
- Normalize
- Store code A(I)

Re: In Reality, Even More Stages !

⇒ Gains of 1:15 Possible (CRAY)

AMDAHL'S LAW (1)

Rate or Speed: R

$$R = \frac{Ops}{t} \quad [FLOPS]$$

α: Amount of Vectorized Operations
T : Time

Scalar:

$$T_s = \frac{Ops}{R_s}$$

Scalar + Vector:

$$T = \frac{\alpha \cdot Ops}{R_v} + \frac{(1-\alpha) \cdot Ops}{R_s}$$

$\Rightarrow$ Speedup S:

$$S = \frac{T_s}{T} = \frac{1}{\alpha \cdot \frac{R_s}{R_v} + (1-\alpha)}$$

AMDAHL'S LAW (2)

Table 1 Speed-Ups Obtained

$\frac{R_v}{R_s}$	50%	90%	99%
10	1.81	5.26	9.17
20	1.90	6.87	16.80
50	1.96	9.80	33.56
100	1.98	9.90	50.25

$\Rightarrow$ Need to Vectorize as Much As Possible

Remark:

- 16-Head CRAY C-90: $\frac{R_v}{R_s} = 16 * 15 = 240$ (!)

VECTOR COMPUTERS

Vectorization Pays Off If:

- Vectors Sufficiently Long ($N > 16$ OK, but $N > 128$ Preferable)
- Loop Can be Vectorized
  - No Recurrence
  - No Complex `IF` Statements/Branching
  - Memory Access Orderly

Remark:

- Even After 20 Years, There Are No Intelligent Compilers !

VECTORIZATION: EASY

- Long or Complicated Loops
- Non-Unit Incrementing Subscripts
- Expression in Subscript
- Intrinsic Function References

VECTORIZATION: STRAIGHTFORWARD

- Scalar Temporary Variables
- Function Calls to Programmer-Supplied Functions
- Inner Products
- Logical `IF`-Statements
- Reduction Operations

VECTORIZATION: DIFFICULT

- Linear Reductions
- Some `IF`-Statements
- Complicated Subscript Expressions
- Non-Linear Indexing

VECTORIZATION: 'IMPOSSIBLE'

- Complex Branching Within Loop
- Ambiguous Subscripting
- Transfers Into a Loop
- Subroutine Calls
- Nonlinear Recursion
- Some I/O

EXAMPLES (1)

Original:

```
do I=1,1000
do J=1,5
A(I,J)=(A(I,J)+B(I,J))*A(I,J)+B(I,J)
enddo
enddo
```

Improved:

```
do J=1,5
do I=1,1000
A(I,J)=(A(I,J)+B(I,J))*A(I,J)+B(I,J)
enddo
enddo
```

EXAMPLES (2)

Original:

```
do I=1,N
do J=1,N
do K=1,N
C(I,J)=C(I,J)+A(I,K)*B(K,J)
enddo
enddo
enddo
```

Improved:

```
do J=1,N
do K=1,N
do I=1,N
C(I,J)=C(I,J)+A(I,K)*B(K,J)
enddo
enddo
enddo
```

BASIC EDGE-LOOP

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
```

Problem: Possible Access of Same Point By Edges

$\Rightarrow$ Memory Contention

AVOIDANCE OF MEMORY CONTENTION

Idea: Colouring/Grouping

- In Group: No Point Touched More Than Once

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
```

For RISC-Chips:

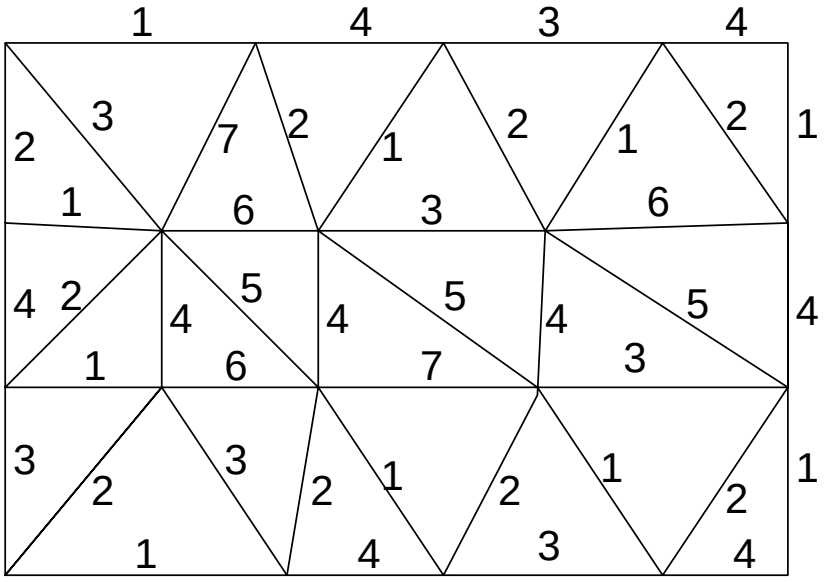
- Short Vector Length (16,32) Sufficient;
- $\Rightarrow$ Make Small Groups To Avoid Cache Misses

BASIC COLOURING ALGORITHM (1)

noi Aim: Points Accessed Once Only By Edges of a Group

C1 Set a maximum desired vector length `MVECL`;
C2 Initialize new edge number array: `LELEM=0`;
C3 Initialize point marking array: `LPOIN=0`;
C4 Set new edge counter: `NENEW=0`;
C5 Set group counter: `NGROU=0`;
C6 Update group counter: `NGROU=NGROU+1`;
C7 Initialize current vector length counter: `NVECL=0`;
C8 Loop over the edges:
If the edge `IEDGE` has not been marked before: then
If none of the points of this edge has been marked as belonging to this group: then
- Update new edge counter: `NENEW=NENEW+1`;
- Store this edge: `LENEW(NENEW)=IEDGE`;
- Set `LPOIN(IPOIN)=NGROU` for all the points of this edge;
- Update current vector length counter: `NVECL=NVECL+1`;
Endif
Endif
If `NVECL.EQ.MVECL`: exit edge loop (Goto C9);
C9 Store the group counter: `LGROU(NGROU)=NENEW`;
C10 If unmarked edge remain (`NENEW.NE.NEDGE`): Goto C6.

AVOIDANCE OF MEMORY CONTENTION



Group	# Edges
1	**10**
2	**9**
3	**7**
4	**9**
5	**3**
6	**3**
7	**2**

Figure 12 Renumbering of Edges for Vectorization

AVOIDANCE OF MEMORY CONTENTION (cont)

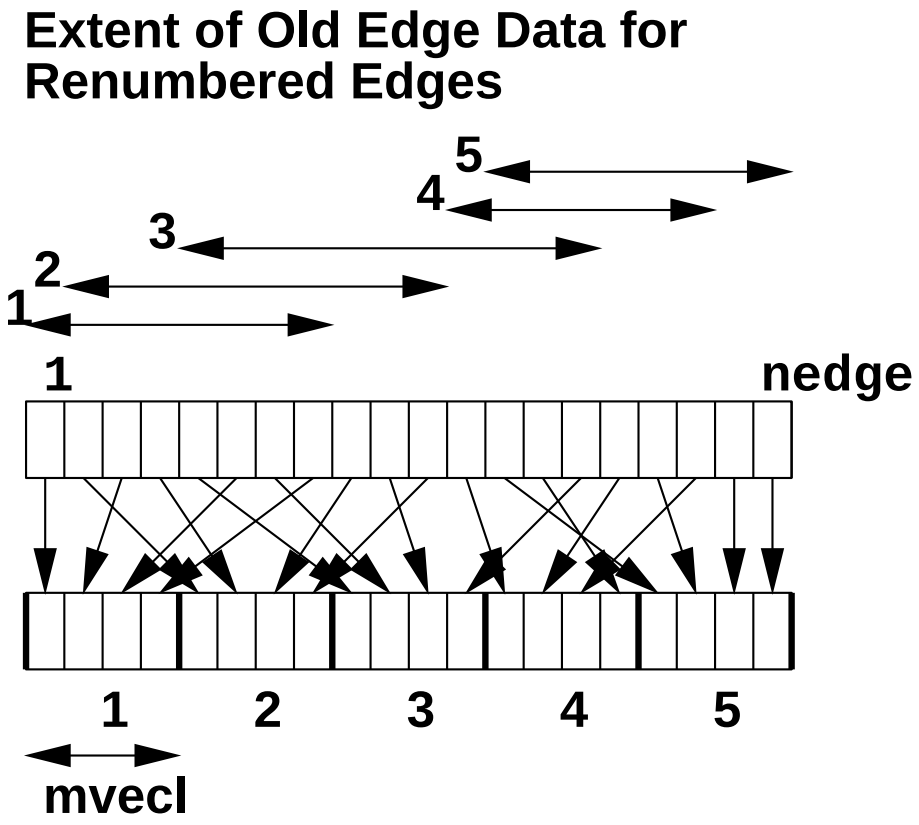


Figure 14 Renumbering for Vectorizability and Data Locality

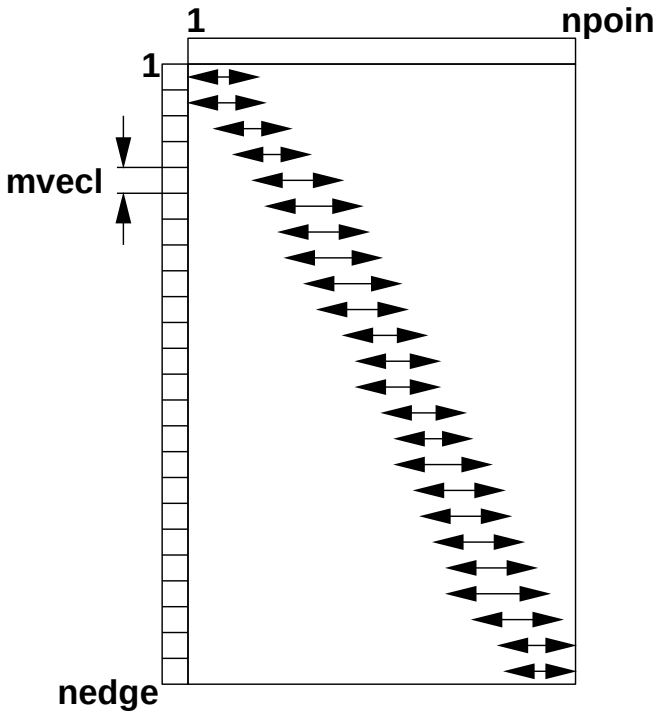


Figure 15 Point-Range Accessed by Edge-Groups

REDUCTION OF INDIRECT ADDRESSING (I/A)

LOOP 1

```
      1000 ipass=1,npass
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
c$dir ivdep
       do 1100 iedge=nedg0,nedg1
       ip1=lnoed(1,iedge)
       ip2=lnoed(2,iedge)
       redge=geoed(iedge)*(unkno(ip2)
     &                    -unkno(ip1))
       rhspo(ip1)=rhspo(ip1)+redge
       rhspo(ip2)=rhspo(ip2)-redge
1100 continue
1000 continue
```

⇒ Per Edge:

- 4 i/a Fetches
- 0 d/a Fetches
- 2 i/a Stores
- 0 d/a Stores
- 4 FLOPS (Only)
- 70% of Incompressible Projection Solver

REDUCED I/A LOOP (1)

Suppose:
- `lnoed(1,iedge) < lnoed(2,iedge)`
- First Point Advances With Stride 1

⇒ Rewrite Loop 1 As:

Loop 4

```
          do 1400 iedge=nedg0,nedg1
          ip1=kpoi0+iedge
          ip2=lnoed(2,iedge)
          redge=geoed(iedge)*(unkno(ip2)
        &                    -unkno(ip1))
          rhspo(ip1)=rhspo(ip1)+redge
          rhspo(ip2)=rhspo(ip2)-redge
    1400 continue
```

⇒ Per Edge:

- 1 i/a Fetch
- 2 d/a Fetches
- 1 i/a Store
- 2 d/a Stores
- 4 FLOPS

REDUCED I/A LOOP (2)

⇒

- Saving Factor of 2:1 in i/a
- Same Number of FLOPS

Same Loop Structure
⇒ Progressive Rewrite of Codes Possible

POINT RENUMBERING (1)

Aim:

- Edges With Uniform Stride for 1st Point
- Long Vector Lengths

⇒

- Points Ordered According to Number of Neighbours
- Enhance Probability of Free 2nd Edge-Point

Initialization:

- From the edge-connectivity array `lnoed`:
  Obtain the points that surround each point;
- Store the number of points surrounding each point: lpsup(1:npoin);
- Set `npnew=0`;

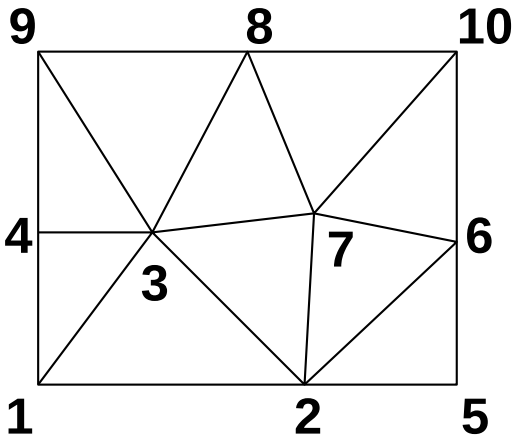
POINT RENUMBERING (2)

Point Renumbering:

- `while(npnew.ne.npoin)`:
  - Obtain the point `ipmax` with the maximum value of `lpsup(ip)`;
  - `npnew=npnew+1`
  - `lpnew(npnew)=ipmax`
  - `lpsup(ipmax)=    0`
  - `do`: for all points `jpoin` surrounding `ipmax`:
    - `lpsup(jpoin)=max(0,lpsup(jpoin)-1)`
  - `enddo`
- `endwhile`

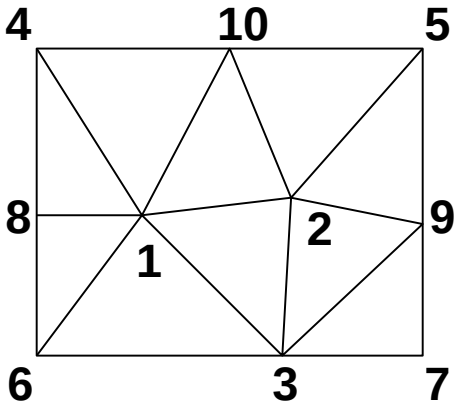
After Point Renumbering:

- Order Edges Accoring to Points
- Group Edges to Avoid Memory Contention



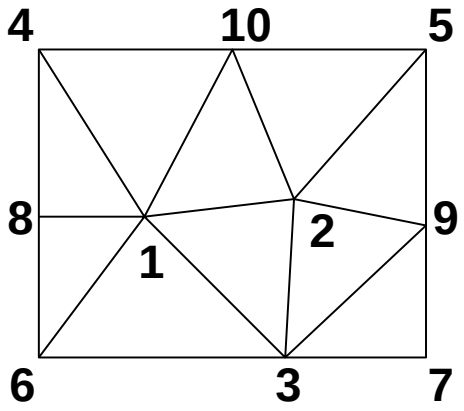
Point	Connections			
1	2	3	4	
2	3	5	6	7
3	4	7	8	9
4	9			
5	6	7		
6	7	10		
7	8	10		
8	9	10		
9				
10				

Figure 1 Original Point Renumbering




```
Point Connections
   1   2  3  4  6  8 10
   2   3  5  9 10
   3   6  7  9
   4   8 10
   5   9 10
   6   8
   7   9
   8
   9
  10
```

Figure 2 Maximal Connectivity Point Renumbering



Point	Connections					
1	10	8	6	4	2	3
2	9	10	5	3		
3	7	9	7	6		
4	8	10				
5	10	9				
6	8					
7	9					
8						
9						
10						

Figure 3 Reodering Into Vector Groups

LOOP 5: FINAL FORM

```
      1000 ipass=1,npass
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
      nedg2=nedg1-nedg0
      if(lnoed(1,nedg1).ne.
     &   lnoed(1,nedg0)+nedg2) then
         usual loop 1(nedg0:nedg1)
      else
         min i/a loop 5(nedg0:nedg1)
      endif
 1000 continue
```

AVOIDANCE OF CACHE MISSES (1)

Point Renumbering Can Lead to Very Large Jumps in Point Index

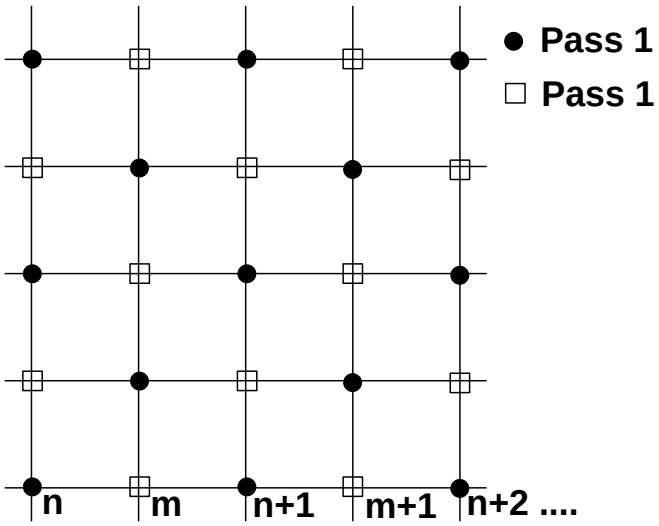


Figure 4 Point Jumps Per Edge for Stuctured Grid

Maximum Bandwith nb_{max} Depends on Maximum Point Connectivity nc_{max}

$$nb_{max} = O((1 - 2/nc_{max})N_p)$$

AVOIDANCE OF CACHE MISSES (2)

Employ 2-Step Procedure:

- Renumber Points for Minimal Cache-Misses
  - Cuthill-McKee
  - Wavefront
  - Recursive Bisection
  - Space-Filling Curve
  - Bin
  - Coordinate-Based, etc.
- Renumber Points According to Maximal Connectivity, But in Groups:
  `npoi0:npoi0+ngrou`
- `ngrou`=O(bandwidth)

LOOP 6: ALTERNATIVE RHS FORMATION

- Timings on Loop 5 Disappointing
- Source of Problem: Stores
  (Removal of 1 Store Doubled Performance)

$\Rightarrow$

```
      1000 ipass=1,npass
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
c$dir ivdep
       do 1100 iedge=nedg0,nedg1
       ip1=lnoed(1,iedge)
       ip2=lnoed(2,iedge)
       redge=geoed(iedge)*(unkno(ip2)
     &                    -unkno(ip1))
       rhsp0(ip1)=rhsp0(ip1)+redge
       rhsp1(ip2)=rhsp1(ip2)-redge
1100 continue
1000 continue
```

- Hypothesis: Compiler Cannot Exclude Case `ip1=ip2`
- RE: No Extra Storage/Initialization Required

SPHERE CLOSE TO WALL (1)

- Typical of Low Re Incompressible Flow
- `npoin= 49,574`
- `nelem=272,434`
- `nedge=328,634`

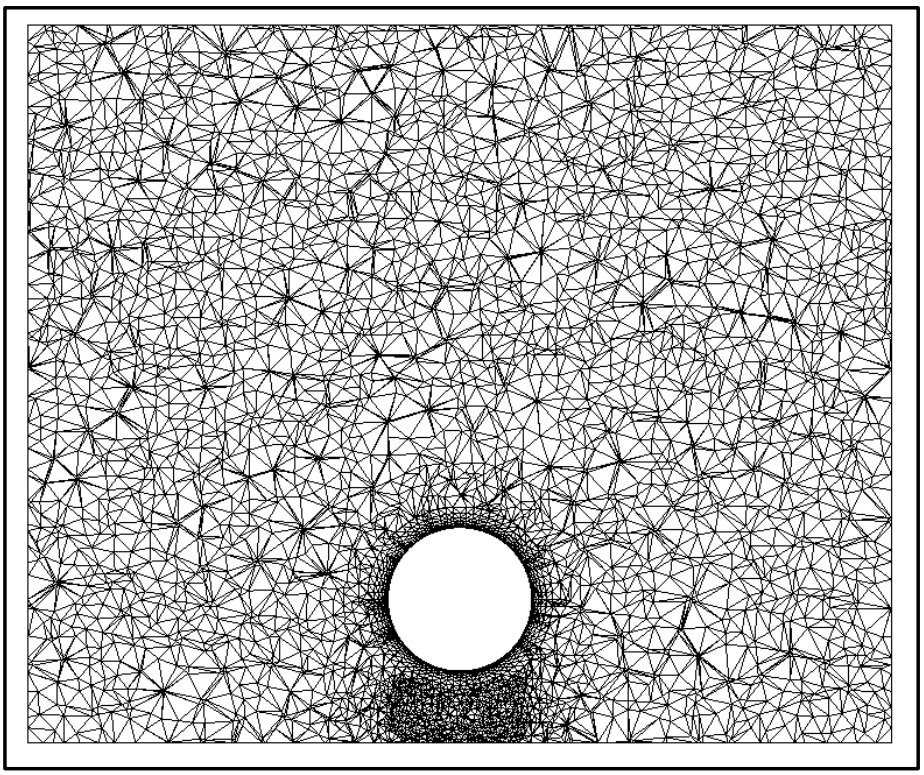


Figure 5 Sphere in Wall Proximity: Mesh in Cut Plane

SPHERE CLOSE TO WALL (2)

Table 1: Sphere Close to Wall: `nedge=328,634`

`mvecl`	% redi/a	`nvecl`	% redi/a	`nvecl`
128	89.53	126	94.88	119
256	87.23	251	94.94	218
512	84.50	490	94.69	371
1024	78.58	947	93.10	592
2048	69.85	1748	90.85	797

NACA0012 WING (1)

- Typical of Inviscid, Incompressible Flow
- `npoin= 68,585`
- `nelem=370,541`
- `nedge=451,108`

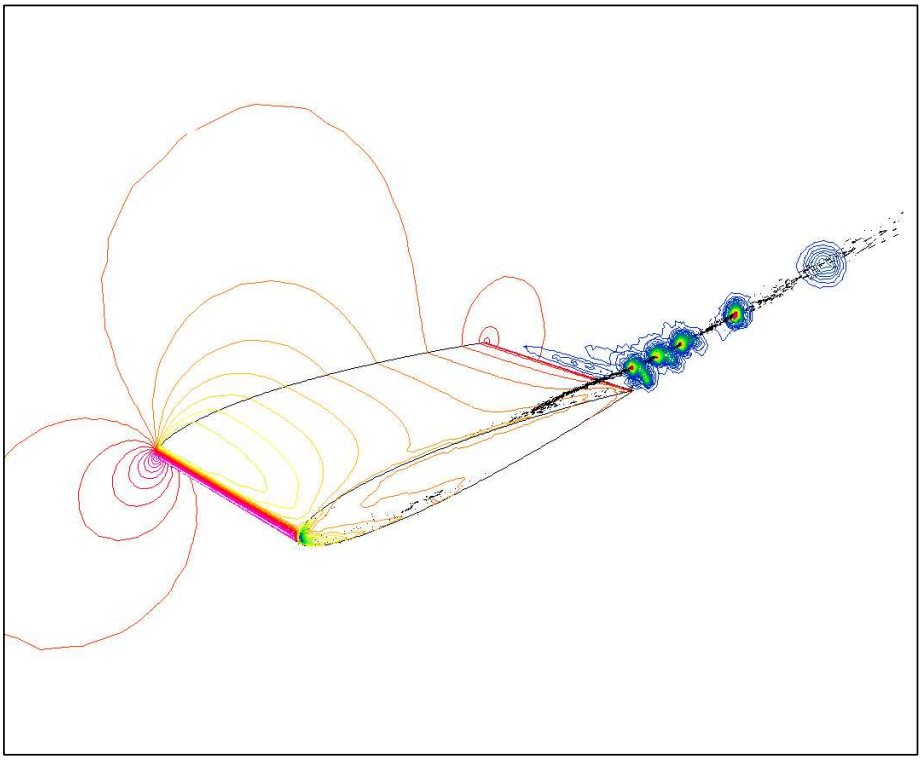


Figure 6 NACA0012 Wing

NACA0012 WING (2)

Table 2: NACA0012 Wing: `nedge=451,108`

`mvecl`	% redi/a	`nvecl`	% redi/a	`nvecl`
128	90.24	127	95.67	119
256	87.77	253	95.73	220
512	86.07	502	95.80	380
1024	82.55	993	94.90	612
2048	73.47	1919	92.90	835

F117 CONFIGURATION (1)

- Typical of Inviscid, Compressible Flow
- `npoin=  619,278`
- `nelem=3,509,926`
- `nedge=4,179,771`

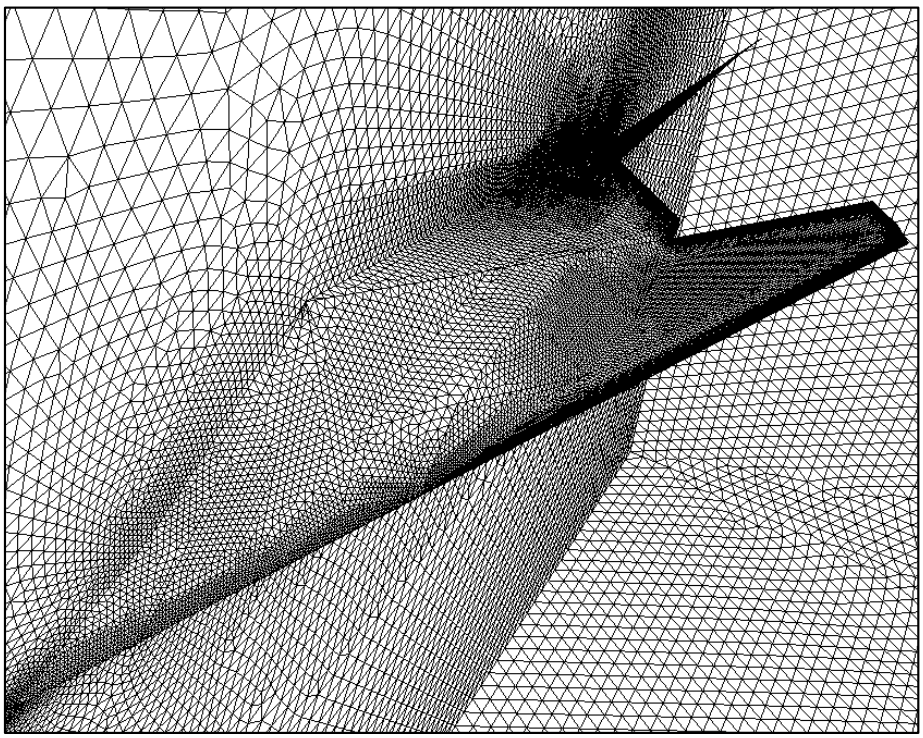


Figure 7 F117: Surface Discretization

F117 CONFIGURATION (2)

Table 3: F117 Configuration: `nedge=4,179,771`

`mvecl`	% redi/a	`nvecl`	% redi/a	`nvecl`
64	97.22	63	97.22	63
128	93.80	127	98.36	121
256	89.43	255	97.87	223
512	86.15	510	97.24	384
1024	82.87	1018	96.77	608
2048	79.30	2026	96.44	855
4096	76.31	4019	96.05	1068
8192	73.25	7856	95.35	1199
16384	67.16	15089	92.93	1371

TIMINGS FOR SPHERE CLOSE TO WALL

Table 4: Laplacian RHS Evaluation (Relative Timings)

Loop Type	O2K	SV1	SX5
Loop 1	1.0000	1.0000	1.0000
Loop 5	0.9563	1.0077	0.8362
Loop 6	0.9943	0.8901	0.7554
Loop 7	0.9484	0.8416	0.7331
Loop 8	-	-	0.7073

PARALLELIZATION

Option 1: Use Auto-Parallelizing Compiler for LOOP 2

Pros:

- Easy

Cons:

- Start-Up Cost of Parallel `do-loop` (300 Flops)
  ⇒ For `nproc=16` Need Vector Lengths of `16*1,000`
  ⇒ Need ≈ `22*16*1,000=352,000` Edges
  ⇒ Need Big Mesh
- Range of Points in Group Increases
  ⇒ Cache Misses Increase
  ⇒ Not Scalable
- Large Portion of Points Accesses in Each Group
  ⇒ Same Range of Points Accessed By Each Group
  ⇒ Dirty Cache-Line Overwrite
  ⇒ Serious Degradation (>1:10)

⇒ Need Further Renumbering Strategies

Define:

- `nproc`: Number of Processors
- `edmin(1:npass), edmax(1:npass)`: Minimum/Maximum Point Accessed by Each Group
- `[edmin(ipass),edmax(ipass)]`: Point-range of group `ipass`

LOCAL AGGLOMERATION

Idea: Process, In Parallel, `nproc` Independent Vector-Groups With Non-Overlapping Point-Range

LOOP 3

```
      do 1400 ipass=1,npass
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
c                        !  Parallelization directive
c$doacross local(iedge,ipoi1,ipoi2,redge)
c$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
```

- `nedg1-nedg0+1` Is `nproc` Times Larger Than in LOOP 2
- Minor Code Modifications
- Start-Up Cost
- ⇒ Favour Long Vector Lengths
- ⇒ Increase of Cache Misses
- ⇒ Attempt Higher Level of Parallelism

GLOBAL AGGLOMERATION (1)

Idea: Process, In Parallel, At A Higher Level

⇒ Form Macro-Groups

GLOBAL AGGLOMERATION (2)

LOOP 4

```
      do 1000 imacg=1,npasg,nprol
      imac0=           imacg
      imac1=min(npasg,imac0+nprol-1)
c                      !  Parallelization directive
c$doacross local(ipasg,ipass,npas0,npas1,iedge,
c$&              nedg0,nedg1,ipoi1,ipoi2,redge)
      do 1200 ipasg=imac0,imac1
      npas0=edpag(ipasg)+1
      npas1=edpag(ipasg+1)
       do 1400 ipass=npas0,npas1
       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
 1200 continue
 1000 continue
```

GLOBAL AGGLOMERATION (3)

RE:

- New Outer `1200-loop`
- Code Rewrite
- Original Code Recovered By Setting:
  - `edpag(1)=0`
  - `edpag(2)=npass`
  - `npasg=1`

BASIC COLOURING ALGORITHM (1)

Aim: Points Accessed Once Only By Edges of a Macro-Group

Greedy-Type Algorithm

S.1 Pass 1:

- Agglomerate in Groups With Point-Range `npoin/nproc`

S.2 Passes 2ff:

- Determine Point-Range of Remaining Groups
- Estimate Range First Macro-Group (Point Range, `nproc`);
- Agglomerate Groups in This Range ($\Rightarrow$ First Macro-Group);
- March Through the Remaining Groups ($\Rightarrow$ Macro-Groups of Length `naggl`).

Remark: Relax Non-Overlap of Point-Range to Cache-Line Range

- More Flexibility to Deal With 'Holes'
- Almost Perfect Load Balance

IMPLEMENTATIONAL ISSUES (1)

Situation:

- Multiphysics/Multinumerics
  ⇒ Hundreds of Subroutines
- Present Compiler Technology
  - Declare All Shared/Local Variables (Error-Prone)
  - Refusal to Parallelize

⇒ Write **Sub-Subroutines**

IMPLEMENTATIONAL ISSUES (2)

Master Loop 4

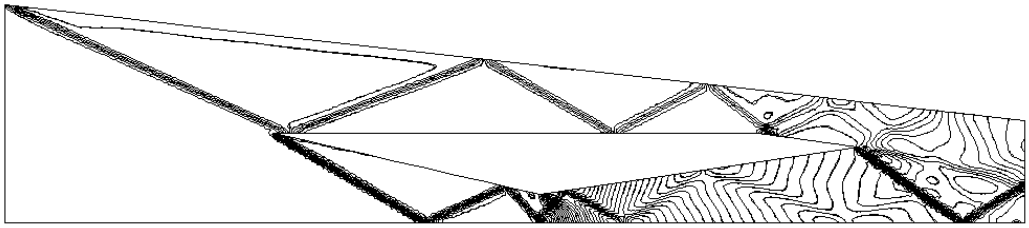
```
      do 1000 imacg=1,npasg,nproc
      imac0=           imacg
      imac1=min(npasg,imac0+nproc-1)
c                      !  Parallelization directive
c$doacross local(ipasg)
      do 1200 ipasg=imac0,imac1
      call loop2p(ipasg)
 1200 continue
 1000 continue
```

IMPLEMENTATIONAL ISSUES (3)

Sub-Subroutine Loop2p

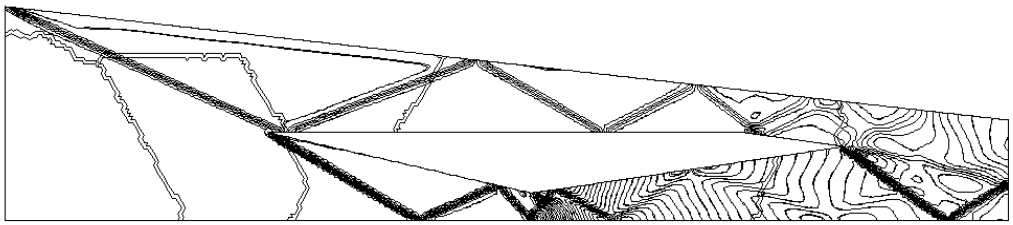
```
      subroutine loop2p(ipasg)
      ......
      npas0=edpag(ipasg)+1
      npas1=edpag(ipasg+1)
       do 1400 ipass=npas0,npas1
       nedg0=edpas(ipass)+1
       nedg1=edpas(ipass+1)
c$dir ivdep                  !  Pipelining directive
       do 1600 iedge=nedg0,nedg1
       ipoi1=lnoed(1,iedge)
       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
```

INLET PROBLEM



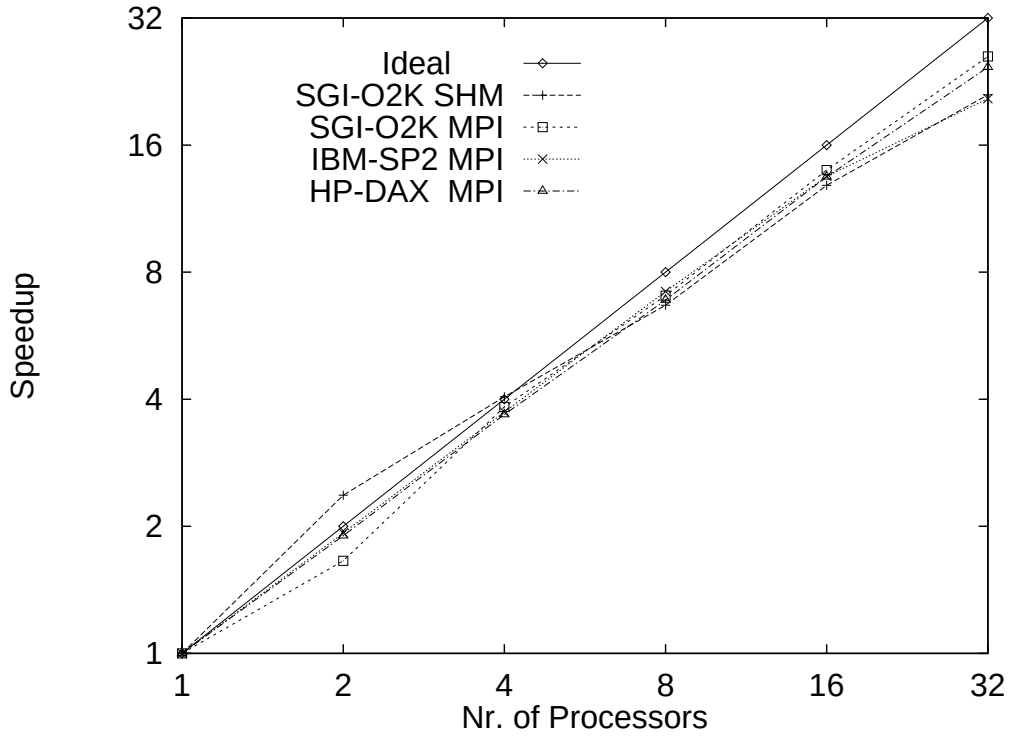
Mach−Number (min=0.825, max=3.000, incr=0.05)

Euler, 500Ktet, RK3, Roe+MUSCL



Mach−Number: Usual vs. 6−Proc Run (min=0.825, max=3.000, incr=0.05)

PARALLEL FLOW SOLVER PERFORMANCE: SMP



Euler, 500Ktet, RK3, Roe+MUSCL, SMP

Parallelizing a Typical Hydro-Code (MIMD)

- Parallel Input
- Parallel Domain Subdivision (Load Balancing)
- Loop over the Timesteps
    - Parallel Execution of Each Subdomain
    - Interdomain Info-Transfer
- Parallel Adaptation
    - Parallel Adaptive Grid Regeneration
    - Parallel H-refinement
    - Parallel Domain Subdivision
- Parallel Output

Note: Do it all in parallel, or nothing

there is no middle ground !

OPTIMIZING CODE

- Always Strive To Use Best Algorithm
  - Lowest Overall Operation Count
  - Computational Complexity Key
- Perform (Same, Standard) Benchmarks
- Use Performance Monitoring Tools of Compiler Provider
- Detect Hotspots; Work From Highest to Lowest
- Do Not Waste Time on 5%; Focus on 95%
- Optimize First for Single Processor,
  then for Multi-Processor