ELSEVIER

Comput. Methods Appl. Mech. Engrg. 163 (1998) 95–109

Computer methods in applied mechanics and engineering

Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines

Rainald Löhner

GMU/CSI, The George Mason University, Fairfax, VA 22030-4444, USA

Received 30 July 1996; revised 18 June 1997

Abstract

Two renumbering strategies for field solvers based on unstructured grids that operate on shared-memory, cache-based parallel machines are described. Special attention is paid to the avoidance of cache-line overwrite, which can lead to drastic performance degradation on this type of machines. Both renumbering techniques avoid cache-misses and cache-line overwrite while allowing pipelining, leading to optimal coding for this type of hardware. © 1998 Elsevier Science S.A. All rights reserved.

1. Introduction

There can hardly be any doubt that the 1990s are the decade of parallelism. Even though machines with several powerful vector-processors were installed at many places in the 1980s, the operating system or the system administration hardly allowed for the use of several processors during the same run. Moreover, in many instances the compilers were not mature, leading to meaningless gains in performance. With the advent of massively parallel machines, i.e. machines in excess of 500 nodes, the exploitation of parallelism in solvers has become a major focus of attention. According to Amdahl's Law, the speed-up s obtained by parallelizing a portion α of all operations required is given by

$$s = \frac{1}{\alpha \cdot \frac{R_s}{R_p} + (1 - \alpha)}, \tag{1}$$

where R_s, R_p denote the scalar and parallel processing rates (speeds), respectively. Table 1 shows the speed-ups obtained for different percentages of parallelization and different numbers of processors.

Note that even on a traditional shared-memory, multiprocessor vector machine, such as the CRAY T-90 with 16 processors, the maximum achievable speed-up between scalar code and parallel vector code is a staggering $R_p/R_s = 240$. What is important to note is that as we migrate to higher numbers of processors, only the

Table 1
Speed-ups obtainable (Amdahl's Law)

R_p/R_s	50%	90%	99%	99.9%
10	1.81	5.26	9.17	9.91
100	1.98	9.90	50.25	90.99
1000	2.00	9.91	90.99	500.25

0045-7825/98/$19.00 © 1998 Elsevier Science S.A. All rights reserved.
PII: S0045-7825(98)00005-X

embarrassingly parallel codes will survive. Most of the applications ported successfully to parallel machines to date have followed the Single Program Multiple Data (SPMD) paradigm. For grid-based solvers, a spatial subdomain was stored and updated in each processor. For particle solvers, groups of particles were stored and updated in each processor. For obvious reasons, load balancing [1–4] has been a major focus of activity.

Despite the striking successes reported to date, only the simplest of all solvers: explicit timestepping or implicit iterative schemes, perhaps with multigrid added on, have been ported without major changes and/or problems to massively parallel machines with distributed memory. Many code options that are essential for realistic simulations are not easy to parallelize on this type of machine. Among these, we mention local remeshing [5], repeated h-refinement, such as required for transient problems [6], contact detection and force evaluation [7], some preconditioners [8], applications where particles, flow, and chemistry interact, and applications with rapidly varying load imbalances. Even if 99% of all operations required by these codes can be parallelized, the maximum achievable gain will be restricted to 1:100. If we accept as a fact that for most large-scale codes we may not be able to parallelize more than 99% of all operations, the shared memory paradigm, discarded for a while as non-scalable, will make a comeback. It is far easier to parallelize some of the more complex algorithms, as well as cases with large load imbalance, on a shared memory machine. And it is within present technological reach to achieve a 100 processor, shared memory machine. Such an alternative, i.e. having less expensive RISC chips linked via shared memory, is currently being explored by a number of vendors. One example of such machines is the SGI Power Challenge, which at the time of writing allows up to 18 processors to work in shared memory mode on a problem, with upgrades to 92 processors planned within the next two years. In order to obtain proper performance from such a machine, the codes must be written in such a way as to avoid:

(a) Cache-misses (in order to perform well on each processor);
(b) Cache overwrite (in order to perform well in parallel); and
(c) Memory contention (in order to allow pipelining).

Thus, although in principle a good compromise, shared memory, RISC-based parallel machines actually require a fair degree of knowledge and reprogramming for codes to run optimally.

The present paper describes renumbering techniques for field solvers operating on unstructured grids that have proven useful for machines of this kind. A summary of the remainder of the paper follows. Sections 2 and 3 recall some previously described renumbering techniques to minimize cache-misses and achieve pipelining. Section 4 treats cache overwrite, a new, and previously not accounted-for design requirement for renumbering strategies. In Section 5, implementational issues are considered. Section 6 reports several scalability studies obtained on SGI Power Challenge and SGI Origin systems. Finally, some conclusions are drawn in Section 7.

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 [9–14]. Similar loops are obtained for element- or face-based solvers, and what follows is equally applicable to them. A right-hand side (RHS), or residual, is formed at the edge-level by gathering information from a vector of unknowns. This edge-RHS is then added to a global point-RHS. The operations are shown schematically in Fig. 1, and a typical FORTRAN implementation would be the following:

Loop 1

```
      do 1600 iedge=1,nedge
      ipoi1=lneod(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
```

If cache-misses are a concern, then it is clear that the storage locations for the required point information stored

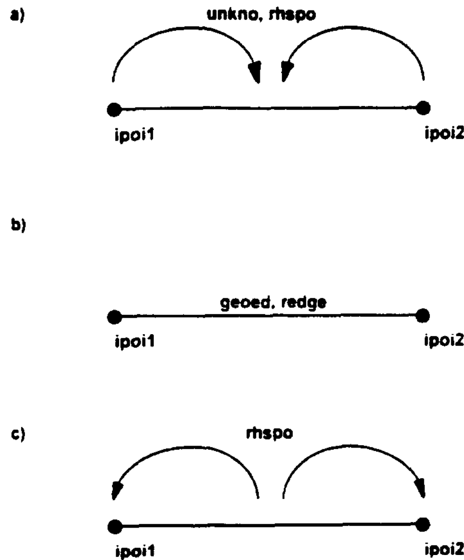


Fig. 1. Information flow for edge-based loop.

in the arrays `unkno` and `rhspo` should be as close as possible in 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 and McKee [15], wavefront [16], recursive bisection), and subsequently renumbering the edges according to the minimum point number on each edge [16]. All of these algorithms are of complexity $O(N)$ or at most $O(N \log N)$, and are well worth the effort.

3. Avoidance of memory contention

Pipelining or vectorization offers the possibility of substantial performance gain 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 RISC chips, such as the MIPS R8000 and R10000. For the latter chip, so-called software pipelining is invoked by the compiler for certain optimization options. In order to achieve pipelining or vectorization, memory contention issues must be avoided. The enforcement or 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

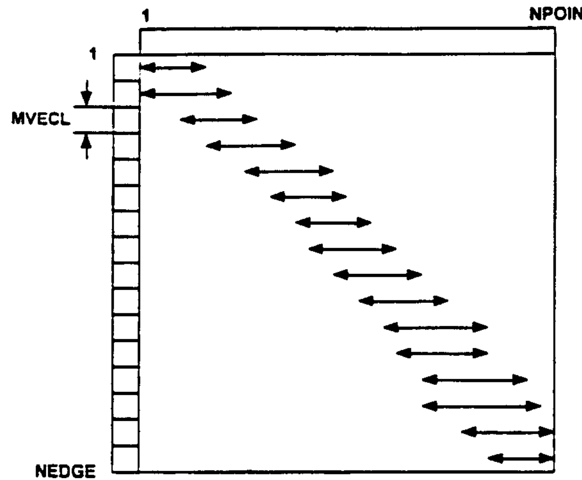


Fig. 2. Point-range covered by each group of edges; 1-Processor machine; renumbering to minimize cache-misses and avoid memory contention.

corresponding points may be accessed more than once. Given that in order to achieve good pipelining performance on current RISC-chips a relatively short vector length of 16 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 long and avoid memory contention [16]. As before, this renumbering is of complexity $O(N)$. The resulting loop is shown schematically in Fig. 2.

4. Cache line overwrite

The next stage is to port Loop 2 to a parallel, shared memory machine. If the loop is left untouched, the auto-parallelizing compiler will simply split the inner loop across processors. It would then appear that increasing the vector-length to a sufficiently large value would offer a satisfactory solution. However, this is not advisable for the following reasons:

(a) Every time a parallel `do-loop` is launched, a start-up time penalty, equivalent to several hundred Flops is incurred. This implies that if scalability to even 16 processors is to be achieved, the vector loop lengths would have to be $16*1000$. For typical tetrahedral grids we encounter approximately 22 maximum vector-length groups, indicating that we would need at least $22*16*1000 = 352\,000$ edges to run efficiently.

(b) Because the range of points in each group increases at least linearly with vector length, so do cache misses. This implies that even though one may gain parallelism, the individual processor performance would degrade. The end result is a very limited, non-scalable gain in performance.

(c) Because the points in a split group access a large portion of the edge-array, different processors may be accessing the same cache-line. When a 'dirty cache-line' overwrite occurs, all processors must update this line, leading to a large increase of interprocessor communication, severe performance degradation and non-scalability. Experiments on an 8-processor SGI Power Challenge showed a maximum speed-up of only 1:2.5 when using this option. This limited speed-up was attributed, to a large extent, to cache-line overwrites.

In view of these consequences, additional renumbering strategies have to be implemented. In the following, we discuss two edge-group agglomeration techniques that minimize cache misses, allow for pipelining on each processor, and avoid cache overwrite across processors. Both techniques operate on the premise that the points accessed within each parallel inner edge-loop (`1600 loop`) do not overlap.

Before going on, we define `edmin(1:npass)`, `edmax(1:npass)` to be the minimum and maximum point accessed within each group, `nproc` the number of processors, and the point-range of each group `ipass` as `[edmin(ipass), edmax(ipass)]`.

4.1. Local agglomeration

The first way of achieving pipelining and parallelization is by processing, in parallel, nproc independent vector-groups whose individual point-range does not overlap. The idea is to renumber the edges in such a way that nproc groups are joined together where, for each one of these groups, the point ranges do not overlap (see Fig. 3). As each one of the sub-groups has the same number of edges, the load is balanced across the processors. The actual loop is given by

Loop 3

```
      do 1400 impass=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
```

Note that the number of edges in each pass, i.e. the difference nedg1 − nedg0 + 1 is now nproc times as large as in the original Loop 2. As one can see, this type of renumbering entails no code modifications, making it very attractive for large production codes. However, a start-up cost is incurred whenever a loop across processors is launched. This would indicate that long vector-lengths should be favoured. However, cache-misses increase with vector-length, so that this strategy only yields a limited speed-up. This leads us to the second renumbering strategy.

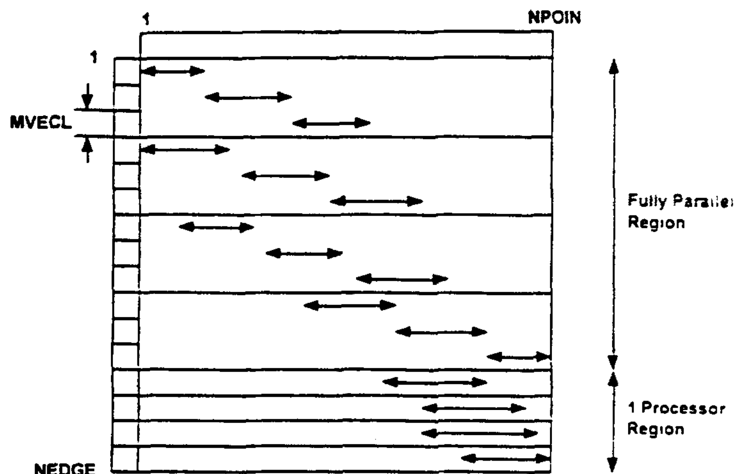


Fig. 3. Point-range covered by each group of edges; 3-Processor machine; renumbering to minimize cache-misses and avoid memory contention.

4.2. Global agglomeration

A second way of achieving pipelining and parallelization is by processing all the individual vector-groups in parallel at a higher level (see Fig. 4). In this way, short vector lengths can be kept and low start-up costs are achieved. As before, the point-range between macro-groups must not overlap. This renumbering of edges is similar to domain-splitting [1,4,17–19], but does not require an explicit message passing or actual identification of domains. Rather, all the operations are kept at the (algebraic) array level. The number of sub-groups, as well as the total number of edges to be processed in each macro-group, is not the same. However, the imbalance is small, and does not affect performance significantly. The actual loop is given by

Loop 4

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

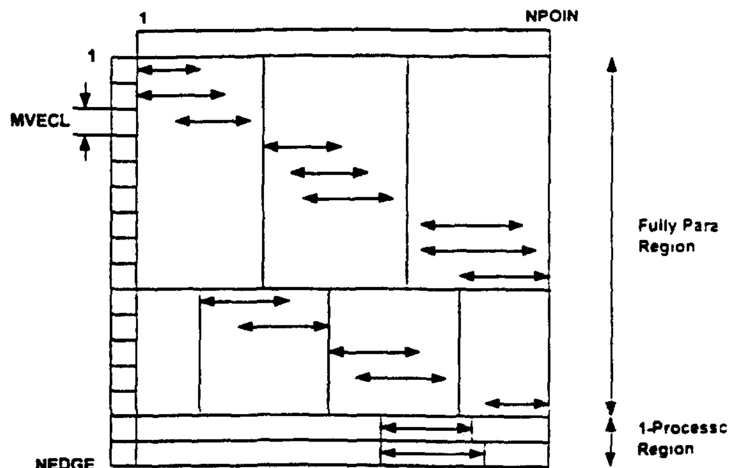


Fig. 4. Point-range covered by each group of edges; 3-processor machine; renumbering to minimize cache-misses, Avoid memory contention and minimize start-up costs.

As one can see, this type of renumbering entails two outer loops, implying that a certain amount of code rewrite is required. On the other hand, the original code can easily be retrieved by setting `edpag(1)=0, edpag(2)=npas, npasg=1` for conventional uniprocessor machines.

If the length of the cache-line is known, one may relax the restriction of non-overlapping point ranges to non-overlapping cache-line ranges. This allows for more flexibility, and leads to almost perfect load balance for all cases tested to date.

A simple edge-renumbering scheme that we have found effective is the following multipass algorithm:

S.1. *Pass 1:* Agglomerate in groups of edges with point-range `npoin/nproc`, setting the maximum number of groups in each macro-group `naggl` to the number of groups obtained for the range `1:npoin/nproc;`

S.2. *Passes 2ff:*
- For the remaining groups: determine the point-range;
- Estimate the range of the first next macro-group from the point range and the number of processors;
- Agglomerate the groups in this range to obtain the first next macro-group, and determine the number of groups for each macro-group in this pass `naggl;`
- March though the remaining groups of edges, attempting to obtain macro-groups of length `naggl`.

Although not optimal, this simple strategy yields balanced macro-groups, as can be seen from the examples below. Obviously, other renumbering or load balancing algorithms are possible, as evidenced by a large body of literature (see e.g. [1,4,17–19]).

5. Implementational issues

For large-scale codes, having to re-write and test several hundred subroutines can be an onerous burden. To make matters worse, present compilers force the user to declare explicitly the local and shared variables. This is easily done for simple loops such as the one described above, but can become involved for the complex loops with many scalar temporaries that characterize advanced CFD schemes written for optimal cache reuse. We have found that in some cases, compilers may refuse to parallelize code that has all variables declared properly. A technique that has always worked, and that reduces the amount of variables to be declared, is to write sub-subroutines. For Loop 4, this translates into:

Master Loop 4

```
         do 1000 imacg=1,npasg, proc
         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
```

Loop 2p becomes a subroutine of the form:

```
        subroutine loop2p(ipasg)
        npas0=edpag(ipasg)+1
        npasi=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
```

6. Example timings

The renumbering strategies described were coded into FEFLO97, an adaptive, edge-based finite element code for the solution of compressible and incompressible flows [20]. The compressible solver incorporates, among other options, van Leer's flux-vector and Roe's flux-difference splitting techniques. The incompressible solver is based on a projection technique, implying that the bulk of the CPU time is spent in a Laplacian loop of precisely the type discussed above. The first two cases were run on an SGI Power Challenge with 6 R8000 processors, 4 Mbytes of cache and 512 Mbytes of memory, whereas the third case was run on an SGI Origin 2000 system with 8 R10000 processors, 4 Mbytes of cache and 4 Gbytes of memory.

6.1. F-117

The surface mesh, as well as the (unconverged) solution after 50 timesteps are shown in Fig. 5(a,b). The mesh had approximately 280 Ktetra, 52 Kpts, 8.6 Kboundary points and 340 Kedges. After renumbering, the

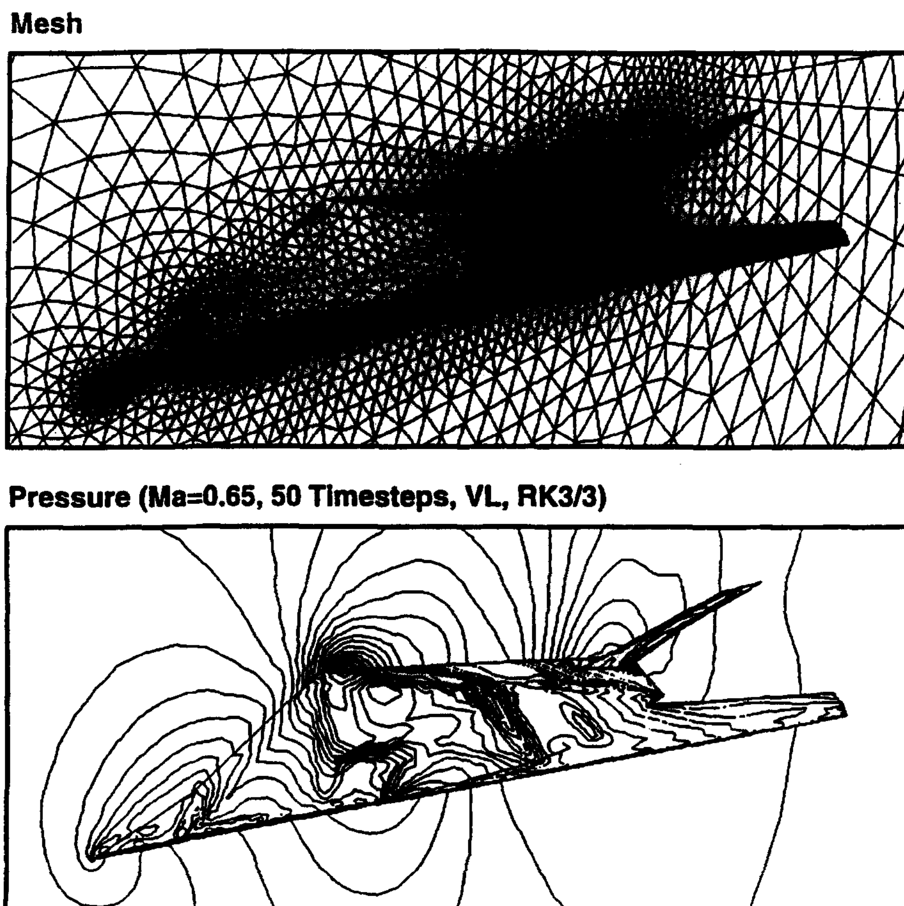


Fig. 5. F-117 (transonic). (a) Surface mesh; (b) pressure ($Ma_\infty = 0.65$).

Table 2

ipasg	npas0	npas1	loopl	ipmin	ipmax
(a) F-117 problem on 2 processors					
1	1	8705	139280	1	23588
2	8706	17410	139280	23588	46686
3	17411	18936	24416	19980	27264
4	18937	20462	24416	45465	50160
5	20463	20760	4768	49486	50832
6	20761	21058	4767	50875	51874
7	21059	21263	3280	50319	51320
(b) F-117 problem on 4 processors					
1	1	3791	60656	1	10704
2	3792	7582	60656	10709	23020
3	7583	11373	60656	23029	35623
4	11374	15164	60656	35631	46685
5	15165	16054	14240	8665	13101
6	16055	16944	14240	19470	25301
7	16945	17834	14240	32241	37526
8	17835	18724	14240	45458	48692
9	18725	19231	8112	19430	26447
10	19232	19738	8112	34507	48782
11	19739	20245	8112	48784	50727
12	20246	20590	5519	50729	51874
13	20591	20724	2144	22706	48803
14	20725	20858	2144	50168	51041
17	20859	21263	6480	47928	51226
(c) F-117 problem on 6 processors					
1	1	2160	34560	1	6282
2	2161	4320	34560	6413	13918
3	4321	6480	34560	14021	22627
4	6481	8640	34560	22668	31484
5	8641	10800	34560	31487	39596
6	10801	12960	34560	39751	46585
7	12961	13555	9520	4858	7897
8	13556	14150	9520	12696	17070
9	14151	14745	9520	20533	25686
10	14746	15340	9520	28369	33391
11	15341	15935	9520	36851	40814
12	15936	16530	9520	45325	47920
13	16531	17161	10096	6247	15424
14	17162	17792	10096	19030	24188
15	17793	18423	10096	27860	34282
16	18424	19054	10096	38363	48191
17	19055	19685	10096	48192	50496
18	19686	20121	6975	50497	51874
19	20122	20273	2432	12674	25973
20	20274	20425	2432	30955	34624
21	20426	20577	2432	47184	48525
22	20578	20729	2432	49873	50832
25	20730	20857	2048	22256	26268
26	20858	20985	2048	31320	48657
27	20986	21052	1072	50319	51007
31	21053	21180	2048	22573	48853
37	21181	21263	1328	47985	49065

(d) F-117: Actual vs. optimal edge-allocation

nproc	actual %	optimal %	loss %
2	50.482	50.00	1.0
4	26.934	25.00	7.7
6	18.234	16.67	9.4

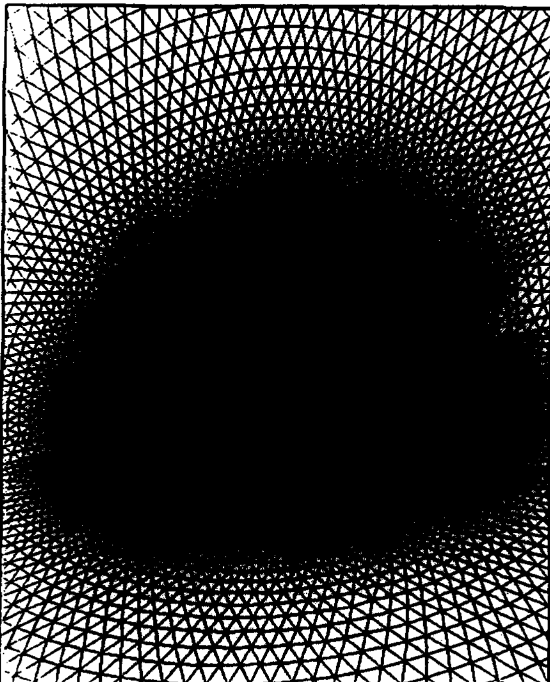
(e) Timings for F-117 problem

nproc	time (s)	CPU (s)	Speedup
1	919	897.0	1.00
2	485	942.5	1.89
4	281	1087.5	3.27
6	220	1235.1	4.17

maximum and average bandwidths were 3797 and 2510. The vector-loop length was set to 16, which was found to be sufficient for good performance on the SGI Power Challenge. The grouping of edges according to the number of processors is given in Table 2(a–c). The corresponding percentage of edges processed by the processor with the maximum number of edges, as well as the theoretical loss of performance due to imbalance (ratio of actual work carried out by this processor vs. the minimum possible work) is shown in Table 2(d). Table 2(e) summarizes the clock time and total CPU time, as well as the speed-ups obtained for a run of 50 timesteps, including i/o, renumbering, etc. As one can see, the performance degrades with the number of processors. This is to be expected, as the increasing number of passes results in higher relative loop costs, and portions of the code (i/o, renumbering, indirect data structures, some residual sums, etc.) are still running in uni-processor mode. Given that the machine used only had 6 processors, timings obtained for the 6 processor case may be higher than expected.

6.2. Sphere

The surface mesh, as well as the solution after 50 timesteps are shown in Fig. 6(a,b). The mesh had approximately 332 Ktetra, 61 Kpts, 8.8 Kboundary points and 402 Kedges. After renumbering, the maximum and average bandwidths were 1993 and 1411. The vector-loop length was again set to 16. The grouping of edges according to the number of processors is given in Table 3(a–c). The corresponding percentage of edges processed by the processor with the highest number of edges, as well as the theoretical loss of performance due to imbalance is shown in Table 3(d). Table 3(e) summarizes the speed-ups obtained for a run of 50 timesteps, including i/o, renumbering, etc. Note that the speed-up is superlinear up to four processors. A convincing explanation of this phenomenon is still elusive, but we speculate on reduced cache-misses for the multiprocessor runs. As before, the machine used had a total of 6 processors, so that the timings for the 6 processor case have to be qualified. The bulk of the work for this incompressible flow case is performed in a Laplacian-like inner loop over edges, showing that the data structures discussed perform well.



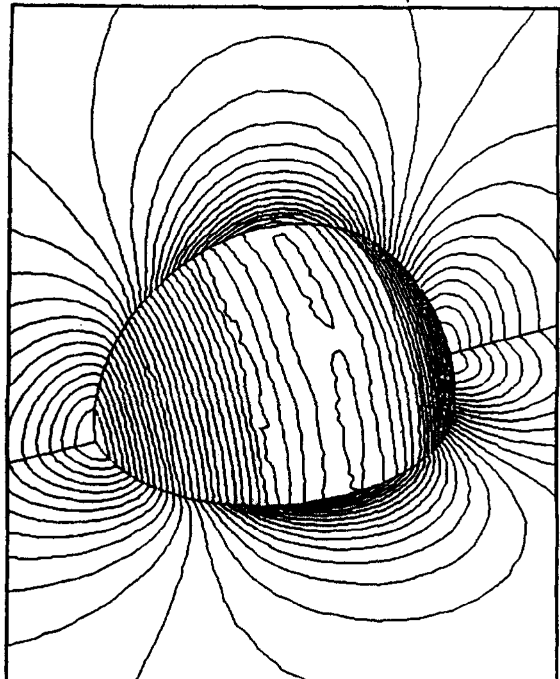


Fig. 6. Sphere (incompressible). (a) Surface mesh; (b) pressure.

Table 3

ipasg	npas0	npas1	loopl	ipmin	ipmax
(a) Sphere problem on 2 processors					
1	1	10753	172048	1	27373
2	10754	21506	172048	27377	54935
3	21507	22282	12416	25525	29268
4	22283	23058	12416	53374	56594
5	23059	23937	14064	55413	58529
6	23938	24816	14062	58530	61039
7	24817	35118	4832	57736	59207
(b) Sphere problem on 4 processors					
1	1	5023	80368	1	13171
2	5024	10046	80368	13172	26969
3	10047	15069	80368	26970	40962
4	15070	20092	80368	40964	54771
5	20093	21097	16080	11892	28113
6	21098	22102	16080	39029	55218
7	22103	23107	16080	55219	58651
8	23108	23940	13326	58654	61039
9	23941	24175	3760	26232	28688
10	24176	24410	3760	53705	55676
11	24411	24645	3760	57869	59179
13	24646	24773	2048	26788	55787
14	24774	24836	1008	58484	59308
17	24837	25118	4512	54486	56437
(c) Sphere problem on 6 processors					
1	1	3349	53584	1	8960
2	3350	6698	53584	8962	18476
3	6699	10047	53584	18478	28352
4	10048	13396	53584	28354	38356
5	13397	16745	53584	38357	48329
6	16746	18797	32832	48330	54935
7	18798	19639	13472	7924	19449
8	19640	20481	13472	26448	38468
9	20482	21323	13472	46448	55065
10	21324	22165	13472	55066	58148
11	22166	23007	13472	58149	60703
12	23008	23094	1390	60705	61039
13	23095	23363	4304	17870	38498
14	23364	23632	4304	53557	55636
15	23633	23901	4304	57295	58737
16	23902	24001	1600	60427	60914
19	24002	24149	2368	36539	38829
20	24150	24297	2368	54246	55940
21	24298	24354	912	57991	58859
25	24355	24482	2048	36906	39125
26	24483	24610	2048	54651	56231
31	24611	24738	2048	37201	39439
32	24739	24765	432	54986	56291
37	24766	25118	5648	37506	40314

(d) Sphere: Actual vs. optimal edge-allocation

nproc	actual %	optimal %	loss %
2	50.601	50.00	1.2
4	26.567	25.00	6.2
6	20.770	16.67	24.6

(e) Timings for sphere problem

nproc	time (s)	CPU (s)	Speedup
1	2259	2204.0	1.00
2	1050	2043.2	2.15
4	532	2065.0	4.25
6	430	2493.3	5.25

Table 4

ipasg	min(loopl)	max(loopl)
(a) Inlet problem on 2 processors		
1–2	237776	237776
3–4	71344	71344
5–6	21392	21392
7–8	6416	6416
9–10	1770	1936
11–12	112	1024
13–14	0	576
(b) Inlet problem on 4 processors		
1–4	118880	118880
5–8	35664	35664
9–12	10704	10704
13–16	3216	3216
17–20	266	1024
21–24	800	1024
25–28	0	256
(c) Inlet problem on 6 processors		
1–6	79248	79248
7–12	23776	23776
13–18	7136	7136
19–24	0	2144
25–30	1024	1024
31–36	0	1024
37–42	0	336
(d) Inlet problem on 8 processors		
1–8	59440	59440
9–16	17824	17824
17–24	5360	5360
25–32	650	1600
33–40	0	1024
41–48	0	1024
49–56	0	288

(e) Sphere: Actual vs. optimal edge-allocation

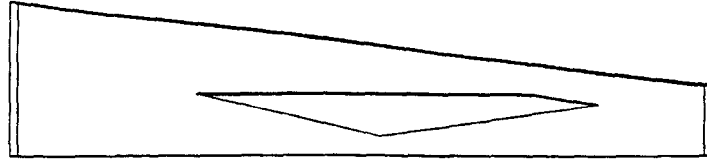
nproc	actual %	optimal %	loss %
2	50.122	50.00	0.24
4	25.140	25.00	0.56
6	16.884	16.67	1.01
8	12.743	12.50	1.94

(4f): Speedups for inlet problem

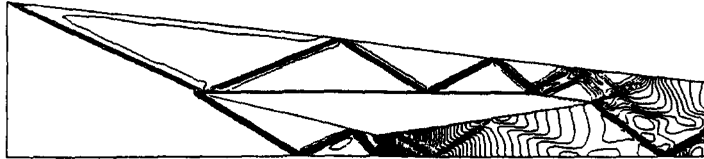
nproc	Speedup(Shared)	Speedup(PVM)
2	1.81	1.83
4	3.18	3.50
6	4.31	5.10
8	5.28	—

6.3. Supersonic inlet

The surface definition, as well as the solution after 800 timesteps are shown in Fig. 7(a,b). The mesh had approximately 540 Ktetra, 106 Kpts, 30 Kboundary points and 680 Kedges. After renumbering, the maximum and average bandwidths were 1057 and 468. The vector-loop length was again set to 16. Instead of showing detailed tables as before, only the minimum and maximum number of edges processed for each pass over the



Surface Definition



Density

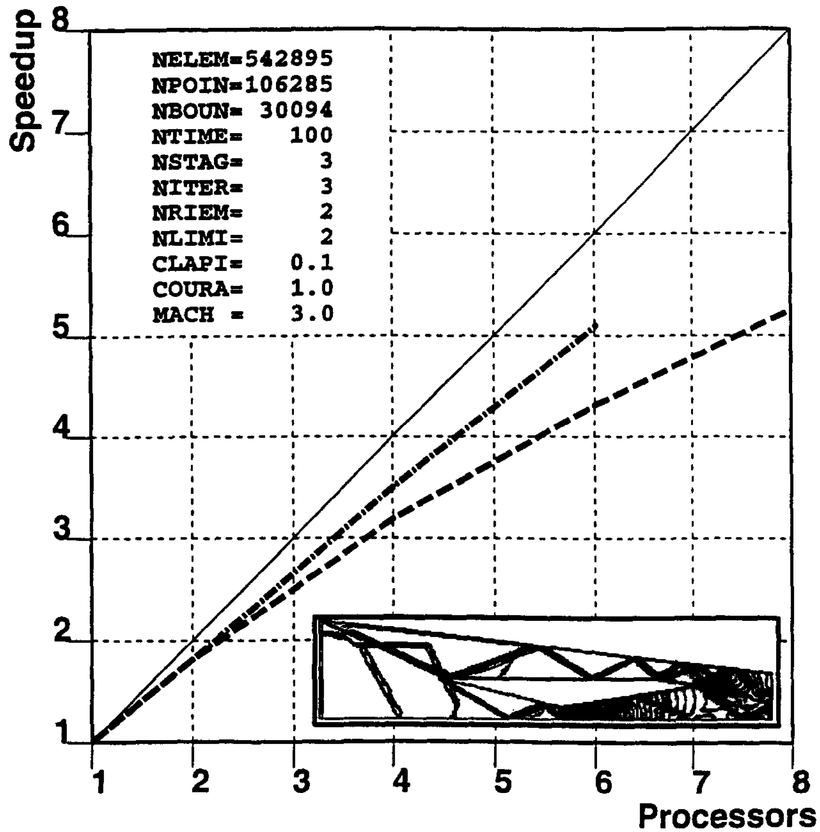


Fig. 7. Inlet problem (supersonic). (a) Surface definition; (b) density; (c) speedups.

processors is given in Table 4(a–d). The corresponding percentage of edges processed by the first processor, as well as the theoretical loss of performance due to imbalance is shown in Table 4(e). As compared to the previous two examples, a near optimal load balance is achieved. Two reasons may be given for this improvement. First, this example was run with a newer version of the renumbering techniques. Second, the constraint of monotonicity in the range of points covered by each macro-group of edges was relaxed to a non-overlap at the cache-line level, which for the SGI Power Challenge is about 15 reals. Table 4(f) and Fig. 7(c) summarize the speed-ups obtained for a run of 100 timesteps, including i/o, renumbering, etc. for an SGI Origin 2000 machine. Although this machine is not a true shared-memory machine, it exhibits very fast inter-processor transfer rates, making it possible to achieve reasonable speedups in shared memory mode. The same run was repeated using domain decomposition and message passing under PVM. Observe that although the PVM-run achieves better speedup, the shared memory run is still competitive.

7. Conclusions

Two renumbering strategies for field solvers based on unstructured grids that operate on shared-memory, cache-based parallel machines have been described. Special attention was given to the avoidance of cache-line overwrite, a hitherto not considered design requirement, which, if not taken into account, can lead to drastic performance degradation on this type of machine. Both renumbering techniques avoid cache-misses and cache-line overwrite while allowing pipelining, leading to optimal coding. While the first technique requires no code rewrite of the field solver, its scalability is expected to degrade for a large number of processors. The second technique requires a moderate rewrite of traditional field solvers, but offers the potential of near-linear scalability for a large number of processors and problem sizes. Numerical experiments indicate that with these renumbering techniques, the number of passes over the processors is always below the theoretical minimum number of passes one would require for maximum-length loops, which for tetrahedral meshes is 7. This implies that these techniques are also applicable to static memory machines like the CRAY-T90, reducing loop start-up costs and improving performance as compared to straightforward inner loop autotasking. As with any other technique, improvements and variations are possible. The techniques described will, however, work on any shared-memory, cache-based machine, and are in this sense general.

Acknowledgments

This work was partially supported by AFOSR, with Dr. Leonidas Sakell as the technical monitor. The author would also like to acknowledge the many fruitful discussions with Drs. Jan Clinkemaille (ESI Group, Paris, France), Jeffrey D. McDonald (SGI, Mountain View, CA), Jack Perry (SGI, Boston, MA), as well as Wayne Odachowsky (SGI, Bethesda, MD), who was instrumental in coordinating the collaboration that led to the techniques discussed.

References

[1] D. Williams, Performance of dynamic load balancing algorithms for unstructured grid calculations, CalTech Rep. C3P913 (1990).
[2] H. Simon, Partitioning of unstructured problems for parallel processing, NASA Ames Tech. Rep. RNR-91-008 (1991).
[3] P. Mehrota, J. Saltz and R. Voigt, eds., Unstructured Scientific Computation on Scalable Multiprocessors (MIT Press, 1992).
[4] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan, A parallel load balancing algorithm for 3-D adaptive unstructured grids, AIAA-93-3313-CP (1993).
[5] R. Löhner, Three-dimensional fluid–structure interaction using a finite element solver and adaptive remeshing, Computer Syst. Engrg. (2–4) (1990) 257–272.
[6] R. Löhner and J.D. Baum, Adaptive H-refinement on 3-D unstructured grids for transient problems, Int. J. Numer. Methods Fluids 14 (1992) 1407–1419.
[7] E. Haug, H. Charlier, J. Clinckemaillie, E. DiPasquale, O. Fort, D. Lasry, G. Milcent, X. Ni, A.K. Pickett and R. Hoffmann, Recent trends and developments of crashworthiness simulation methodologies and their integration into the industrial vehicle design cycle, Proc. Third European Cars/Trucks Simulation Symposium (ASIMUTH), Oct. 28–30, 1991.

[8] R. Ramamurti and R. Löhner, Simulation of flow past complex geometries using a parallel implicit incompressible flow solver, Proc. 11th AIAA CFD Conf., Orlando, FL (July 1993) 1049, 1050.
[9] T. Barth, A 3-D Upwind Euler solver for unstructured meshes, AIAA-91-1548-CP, 1991.
[10] D. Mavriplis, Three-dimensional unstructured multigrid for the Euler equations, AIAA -91-1549-CP (1991).
[11] A. Jameson, The AIRPLANE Code, Private communication, January 1992.
[12] J. Peraire, J. Peiro and K. Morgan, A three-dimensional finite element multigrid solver for the Euler equations, AIAA-92-0449 (1992).
[13] H. Luo, J.D. Baum, R. Löhner and J. Cabello, Adaptive edge-based finite element schemes for the Euler and Navier–Stokes equations, AIAAA-93-0336 (1993).
[14] N.P. Weatherill, O. Hassan and D.L. Marcum, Calculation of steady compressible flowfields with the finite element method, AIAA-93-0341 (1993).
[15] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, Proc. ACM Nat. Conf., New York (1969) 157–172.
[16] R. Löhner, Some useful renumbering strategies for unstructured grids, Int. J. Numer. Methods Engrg. 36 (1993) 3259–3270.
[17] N. Satofuka, J. Periaux and A. Ecer, eds., Parallel Computational Fluid Dynamics (North-Holland, 1995).
[18] V. Venkatakrishnan, H.D. Simon and T.J. Barth, A MIMD implementation of a parallel Euler solver for unstructured grids, NASA Ames Tech. Rep. RNR-91-024 (1991).
[19] R. Löhner and R. Ramamurti, A load balancing algorithm for unstructured grids, Comput. Fluid Dyn. 5 (1995) 39–58.
[20] R. Löhner, FEFLO97 Theoretical Manual; GMU-CSI-CFD Lab. Report, 1996.