ELSEVIER

0965-9978(96)00039-7

Advances in Engineering Software 28 (1997) 189-201
© 1997 Elsevier Science Limited
Printed in Great Britain. All rights reserved
0965-9978/97/$17.00

Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines

D. A. Burgess* & M. B. Giles

Numerical Analysis Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1, 3QD, UK

(Received 28 June 1995; accepted 29 February 1996)

The performance of unstructured grid codes on workstations and distributed memory parallel computers is substantially affected by the efficiency of the memory hierarchy. This efficiency essentially depends on the order of computation and numbering of the grid. Most grid generators do not take into account the effect of the memory hierarchy when producing grids so application programmers must renumber grids to improve the performance of their codes. To design a good renumbering scheme a detailed runtime analysis of the data movement in an application code is needed. Thus, a memory hierarchy simulator has been developed to analyse the effect of existing renumbering schemes such as bandwidth reduction, the Greedy method, colouring, random numbering and the original numbering produced by the grid generator. The renumbering is applied to either vertices, edges, faces or cells and two algorithms are proposed to consistently renumber the other entities used in the solver. The simulated and actual timings show that bandwidth reduction and Greedy methods give the best performance on IBM RS/6000, SGI Indy, SGI Indigo and SGI Power Challenge machines for three-dimensional Poissons's, Maxwell's and the Euler equations solvers. The improvement in performance is over a factor of two for applications with large grids and a high ratio of memory-accesses to computation. This factor is even higher for memory hierarchies with small caches. © 1997 Elsevier Science Limited. All rights reserved.

1 INTRODUCTION

The trend in computing is towards workstations and distributed memory parallel machines. These computers are designed with memory hierarchies that supply the processor with data. The processor is usually faster than the speed at which the largest, slowest memory level can provide data, so it is essential to utilise the memory hierarchy effectively. Significant improvements in the rate of computation can often be gained by simple changes to the code or to the order of accessing data. Memory hierarchy performance is going to be a limiting constraint on the performance of future generation processors[1] so it is essential to optimise memory usage.

This paper concentrates on data ordering to optimise the speed of unstructured grid codes on hierarchical memory machines. Unstructured grid generators usually create numbers for vertices and cells as they produce them. For a frontal grid generator[2] the vertices are often numbered in a spiral fashion whereas a Delaunay generator[3] or a grid adaption generator[4] effectively has random numbering. Many grid generators do not renumber the vertices and cells before handing the grid to the user. Therefore, grid renumbering algorithms that improve locality of data to optimise code performance on all types of hierarchical memory machines need to be used.

Ineffective use of the memory hierarchy in parallel computing can often be observed if super-linear speedup occurs. This happens when the serial computation spends relatively more time accessing the whole grid from memory compared to the distributed parallel processors that access smaller grid partitions from local memories. This paper shows, by observing simulations, that renumbering produces the same rate of accessing data from memory for all grid sizes.

*To whom all correspondence should be addressed at: SCCM, CS Department, Gates Building 2B, Stanford University, Stanford, CA 94305-9025, USA.

Therefore, a more informative speedup graph can be obtained by renumbering the whole grid in the serial case and the individual grid partitions generated for the parallel cases.

This paper concentrates on two renumbering strategies that improve the locality of data. One is based on a bandwidth reduction algorithm developed by Cuthill–McKee[5] and the other is a variant of the Greedy Method.[6] To find the optimal renumbering strategy the behaviour of the memory hierarchy needs to be analysed. A memory hierarchy simulator can supply detailed information on a code's performance for classes of machines. Therefore, a memory hierarchy simulator has been developed to follow the exact movement of data between memory and the processor's registers, and predict execution times. The memory hierarchy of workstations is described in Section 2, the simulator in Section 3 and the two renumbering strategies are discussed in Section 4. These renumbering strategies are evaluated in three three-dimensional test cases: a Jacobi solver for solving Poisson's equation; a conjugate gradient solver used to solve Maxwell's equation;[7] and an edge-based algorithm to solve the Euler equations.[8] The first two test cases are simulated for an IBM RS/6000 model 350, and all cases are timed on IBM RS/6000 model 350, SGI Indy, SGI Indigo and SGI Power Challenge machines. The results are presented in Section 5 and show that the performance gain is dependent on the size of the memory and the grid. Conclusions are drawn in Section 6.

2 MEMORY HIERARCHY

In this section we briefly describe the memory hierarchy and the principles used by manufacturers to design the smaller memory levels in this hierarchy. Hardware design optimisations to larger memory levels in the hierarchy also exist but these do not significantly affect the renumbering strategy so will not be discussed.

A memory hierarchy comprises several layers of memory. The data contained in the first level, the primary cache, is a subset of that in the second level, the secondary cache, and so on. The last level is the main memory which usually holds all the data. Registers are within the processor and hold data that is currently being worked on by the processing units. The advantage of having this structure is that the processor can access the caches, small amounts of memory, much faster than main memory. For example, the IBM RS/6000 processor has a two level memory hierarchy, depicted in Fig. 1.

The renumbering strategies proposed in Section 4 try to exploit two main principles of the memory hierarchy:

— **principle of spatial locality** forecasts that data surrounding the datum currently required by the processor will probably be accessed soon;

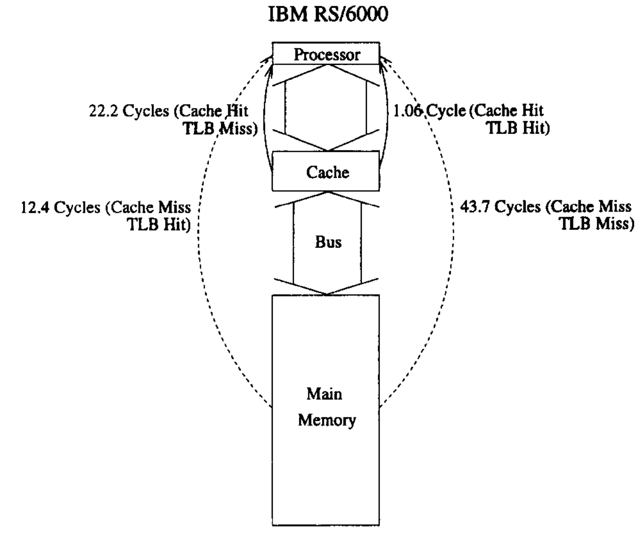
IBM RS/6000
Processor
22.2 Cycles (Cache Hit TLB Miss)
1.06 Cycle (Cache Hit TLB Hit)
Cache
12.4 Cycles (Cache Miss TLB Hit)
43.7 Cycles (Cache Miss TLB Miss)
Bus
Main Memory

Fig. 1. IBM RS/6000 Model 350: memory hierarchical structure and line access times. The cache holds a subset of the data held in main memory and is faster to access than main memory.

— **principle of temporal locality** forecasts that data that has been accessed by the processor will probably be accessed again soon.

The memory system ensures spatial locality by placing a datum into a *line* with other data that are contiguous in memory. The line is then the smallest unit of data transferred between memory levels. Temporal locality is gained by continually replacing *least recently used* (*LRU*) lines in the cache with lines currently required by the processor. However, lines can only be placed in specific positions in cache and these positions are determined by the cache design:

— **direct-mapped cache** maps a line into only one position in cache;
— **fully-associative cache** maps a line into any position in cache;
— **2-way, 3-way or 4-way set associative cache** maps a line into only two, three or four positions in cache, respectively.

The direct-mapped cache has the fastest placement strategy but does not have good temporal locality properties. The fully-associative cache has optimal temporal locality but it is expensive to find the line to replace during a memory access. The set associative cache is 'divided' into *sets* that hold 2, 3 or 4 lines and each line can map into any position within a specific set. Most manufactures opt for this later strategy as it is both cost effective and has reasonable temporal locality. Figure 2 shows the IBM RS/6000 model 350 four-way set associative cache structure.[9] It has 128 sets and therefore can store 512 lines. The line size is 64 bytes so eight double precision numbers are brought into the cache in one instruction.

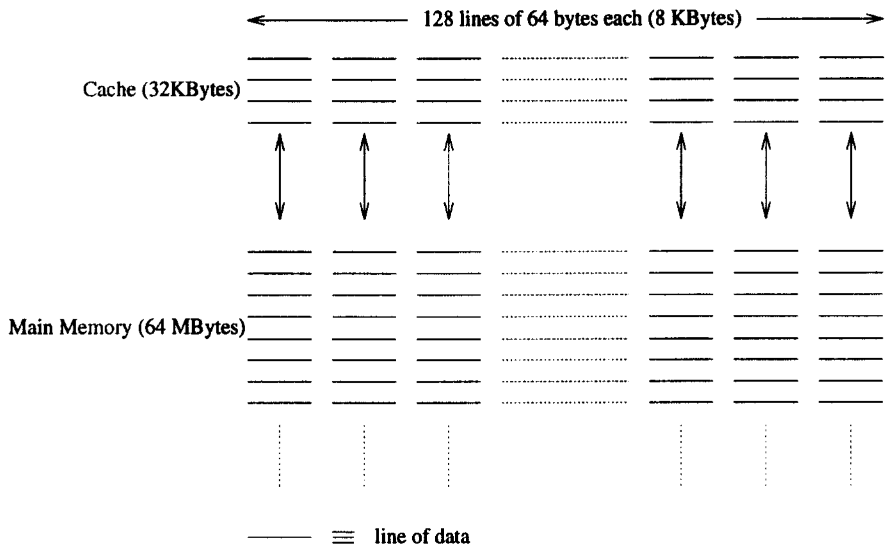

Fig. 2. Four-way set associative cache structure of IBM RS/6000 Model 350. Each line in main memory can map into only four locations in the four-way set associative cache. The cache holds the most recently accessed lines of data.

Computer codes generate virtual addresses for variables. A block, *page*, of virtual addresses is placed into a block of physical memory. Thus, a *page address mapping* between the code's virtual page address and the memory's physical page address is required to fetch data from memory to the processor's registers. These page address mappings are stored in main memory in a *page table*. However, to speed up the access of a page address mapping a subset called the *translation lookaside buffer* (*TLB*) is held in the processor unit. The TLB also has a set-associativity design and usually replaces LRU page address mappings with current mappings. Figure 3 depicts the two-way set associative TLB design of the IBM RS/6000 model 350. It has 64 sets and therefore contains 128 page address mappings. Each page is 4096 bytes which is equivalent to 64 lines of memory.

The page address mapping informs the processor where a datum's page is stored in memory. If the page address mapping is in the TLB, a *TLB hit*, and the datum's line is in the cache, a *cache hit*, then the datum will be placed in a register extremely quickly (one cycle on the IBM RS/6000). However, if the line is not in the cache, a *cache miss*, or if the page address mapping is not in the TLB, a *TLB miss*, then a significant number of cycles are required to place the datum in a register. See Fig. 1 for the *hit/miss parameter costs* on the IBM RS/6000 model 350.

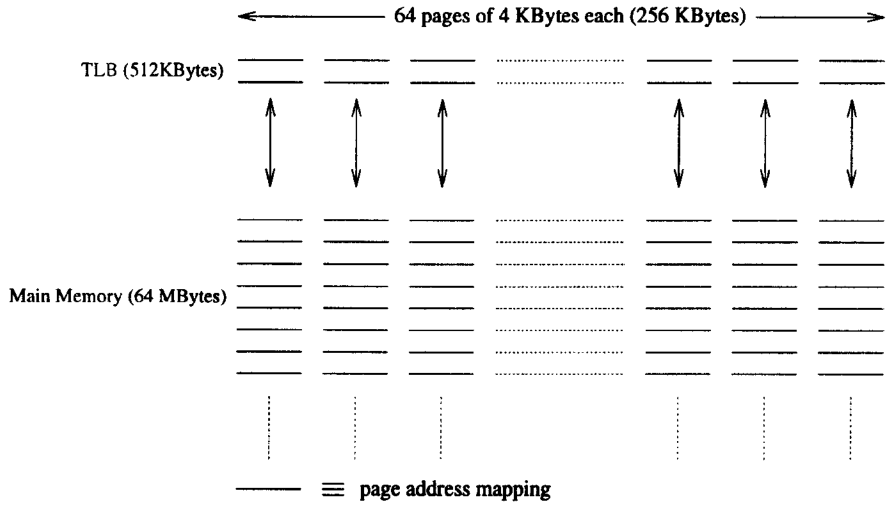

Fig. 3. Two-way set associative TLB structure of IBM RS/6000 Model 350. Each page address mapping in main memory can map into only two locations in the two-way set associative TLB. The TLB holds a subset of the page address mappings stored in main memory and is faster to access than main memory.

A fuller overview of the memory hierarchy is given in Patterson and Hennessey's book.[10] The basic properties of the five machines used in this paper are given in Table 1. Note that the IBM RS/6000 model 350,[9] SGI Indy R4600[11] have a two-level memory hierarchy, whereas the SGI Indigo R4000 and SGI R8000 have three levels of memory. The SGI R8000 is part of the shared memory SGI Power Challenge machine.

3 MEMORY HIERARCHY SIMULATOR

This section describes the general memory hierarchy simulator[12] used in this paper. It enables straightforward analysis of the data movement within the memory hierarchy for individual application codes on specific machines. It produces exact numbers for cache and TLB hits and misses for each array variable in a code. The amount of data reuse can also be evaluated from the proportion of hits to misses. Thus, the cost of various renumbering algorithms can be analysed precisely. Temam and Jalby[13] developed a model to estimate the amount of data that is reused in a two level hierarchical memory machine but this model focuses on a bandwidth reduction numbering and cannot evaluate other renumbering methods or analyse more than two levels of memory. An exact approach with a general memory hierarchy simulator is used here.

The simulator needs to know the associativity and sizes of the cache(s) and TLB of the machine being simulated. The simulator obtains the addresses being accessed during the execution of an application program. From these addresses, it can determine the positions in the cache and TLB that the corresponding lines and page address mappings could occupy. It can then calculate whether cache and TLB hits and/or misses have occurred, update the LRU addresses and increment the total number of hit/miss values. The hit/miss values of specific variables within the application program can also be incremented.

To estimate the total amount of time a code takes in memory accessing, the simulator also needs the hit/miss parameter costs. For the IBM RS/6000 model 350, approximate values for hit/miss parameters were determined by timing a FORTRAN 77 DO-loop that repeatedly accessed 200 components of a REAL*8 vector with a fixed stride length. Using stride lengths of 1, 32, 513 and 512 the average number of cycles for a cache hit–TLB hit, cache miss–TLB hit, cache hit–TLB miss and cache miss–TLB miss were found, respectively. This simplistic approach provides approximate values for hit/miss load parameters, see Fig. 1. However, when the time for other strides is predicted by the simulator using these parameter values there are occasional errors. Actual timings of the RS/6000 model 350 were taken to determine the average number of cycles per

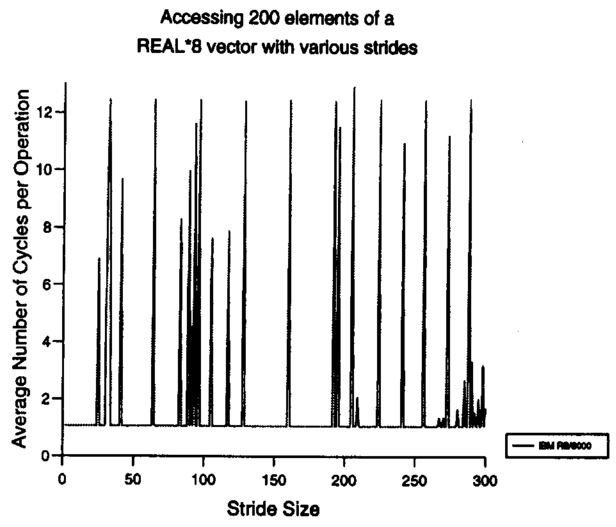

Fig. 4. Actual timings from an IBM RS/6000.

operation for various strides in Fig. 4 and the error between the actual and simulator's predicted timings are displayed in Fig. 5. The errors occur because the parameter value calculations are based upon consistent hits or consistent misses for the cache and/or TLB whereas some stride values have a variety of hits and misses, and machines are designed with various hardware features to optimise different cases. Therefore, cache–TLB hit/miss parameter values are not fixed quantities and are partly dependent on the lines that have been accessed directly preceding the current access, as shown in Warren's results.[14] Nevertheless, fixed values are a usual approximation in simulations. The hit/misses parameter values determined for the RS/6000 model 350 using the above approach are used in this paper to predict the memory access times.

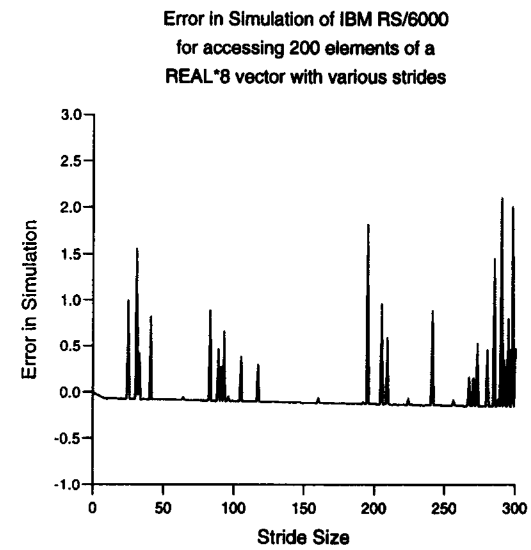

Fig. 5. Error in simulated timings (actual–simulator's predicted average number of cycles per operation) corresponding to Fig. 4 data.

To predict the total execution time of a code the amount of non-memory operations, such as arithmetic operations, needs to be calculated and added to the simulator's predicted time for memory accessing. The number of cycles taken to perform arithmetic operations can vary with machine and a RISC processor can often overlap an arithmetic operation with a load and/or store memory operation. Thus, a comprehensive knowledge of the machine is required to predict the total execution time. The low level details of processors is beyond the scope of this paper. The interested reader is referred to the manufacturer's processor guides.

In summary, the hit/miss parameter values can vary depending on the data access patterns of an application code. Thus errors will occur when predicting memory access times, as shown in Fig. 5. However, the number of hits and misses determined by the simulator is exact. Hence, the renumbering methods will be evaluated based on the information produced by the simulator and the actual execution times. Actual execution times are recorded using profilers and are run in isolation from the simulator. Simulation results are only presented for the IBM RS/6000 model 350 workstation.

4 RENUMBERING

This introduction to renumbering first describes the difference in coding structured and unstructured grid applications. The performance of both these categories of codes can be improved by using the memory hierarchy effectively and by reducing the amount of operations that are required. This paper concentrates solely on the memory hierarchy performance. The results show that the performance of unstructured grid codes can be significantly improved by simply renumbering the input grid data, and without altering the program.

There is a plethora of papers and books that describe ways that codes can be optimised for structured mesh computations on hierarchical memory machines. A structured grid solver uses an indexing system to determine neighbouring grid members with loops over index dimensions of the grid. Loop transformation techniques such as loop-interchanging and loop-blocking[15] are needed to improve data locality in dense[9] and structured sparse matrix applications. These transformations can either be hand coded or performed by a preprocessing compiler.

Unstructured grid solvers are coded differently from structured grid solvers and use a mapping approach to determine neighbours. In an unstructured grid code there can be several *sets* (for example vertices, edges, faces and cells) and *mappings* between sets (for example, the cell to vertex mapping). Most of the computational work involves looping over a set and accessing data belonging to other sets using the mappings. In many applications, the members of a set can be executed in any order without affecting the final result. The execution set can therefore be renumbered (or reordered) to improve temporal locality of 'mapped to' set members. Furthermore, renumbering the 'mapped to' set members in a consistent manner can improve their spatial locality. Together these can give significant improvements over the numbering arising from most unstructured grid generators which do not renumber the set members to optimise cache performance.

There are a variety of data structures used in unstructured sparse matrix calculations.[16] The choice of data structure is usually based upon the algorithm, the language and memory limitations. Other considerations such as whether a RISC processor will be used can also significantly affect this choice. However, data locality optimisation can be applied to all data structures and will improve the rate of computation on hierarchical memory machines.

Loop transformations for unstructured grid computations have been devised by Knijnenburg and Wijshoff[17] but these rely on taking advantage of repetitive patterns throughout the matrix and rewriting code. However, specific patterns of the unstructured sparse matrix can only be determined at runtime so the performance improvement of this approach is grid dependent. This paper deals with general unstructured sparse matrices and shows that the rate of computation can be substantially improved by simply renumbering grids.

4.1 Renumbering strategy

To renumber the sets, the mappings between sets are used. These mappings provide information about the connectivity of a grid and this can be exploited to place neighbouring set members close together in memory. The following strategy is used:

- a mapping is chosen to initially renumber one set (described in Section 4.2);
- once one set has been renumbered it is then possible to *consistently* renumber all other connected sets based on this first one (described in Section 4.3).

4.2 Renumbering the first set

There are various algorithms that renumber unstructured grids. Duff and Meurant[18] compared 17 different orderings to determine the best ILU preconditioner for the Conjugate Gradients algorithm.[19] The main idea behind renumbering for improving cache performance is to place neighbouring members of a set close together in memory. Thus, two main algorithms have been chosen with this aim in mind. One is a bandwidth reduction method and the other is a variant of the Greedy method. Both methods require a mapping between the same set, such as a list of connected vertex pairs. If a mapping of this form is not explicitly available in the grid then an artificial one can be constructed from a mapping between different sets. To determine an artificial set B$\mapsto$B mapping based on a set A$\mapsto$B mapping, loop over set A members and for each set A member define a mapping between all pairs of B members that it maps to.

4.2.1 Bandwidth reduction

Bandwidth reduction is a family of renumbering methods that place non-zeros of a sparse matrix close to the main diagonal. These methods have been used successfully in several fields such as matrix preconditioning[18] and mesh partitioning.[20,21] Das *et al.*[22] also used a bandwidth reduction method for renumbering a grid to double the computational rate of an Euler solver on an iPSC/860 processor.

There are a number of popular bandwidth reduction algorithms such as the Reverse Cuthill–McKee (RCM)[5,23] and Gibbs–Poole–Stockmeyer[24] algorithms. These methods can be viewed physically as splitting the mesh into a number of layers (surfaces in three dimensions). The members within each layer are then numbered contiguously in memory. Figure 6(b) shows the sparsity pattern of a 4913 vertex matrix using the RCM algorithm. This originated from the $17 \times 17 \times 17$ structured cube mesh in Fig. 6(a). Duff and Meurant[18] showed that the RCM algorithm produces a good numbering for ILU preconditioning and sparse LU solvers. All bandwidth reduction results in this paper use the RCM algorithm.

Random numbering within each layer of the bandwidth reduction (RCMLR) algorithm is also tested to see whether the numbering within layers is important for cache performance, Fig. 6(c).

4.2.2 Greedy

The Greedy method described by Farhat[6] places neighbouring members of a set into blocks. The members within each block are numbered contiguously in memory. The Greedy method originated from ideas on partitioning grids. We expect that the Greedy ordering with small block sizes would not be as good as the bandwidth reduced ordering for preventing fill-in of sparse LU solvers or producing good ILU preconditioners. Some tests revealed that the Greedy ordering with small block sizes created worse ILU preconditioners than the bandwidth reduced ordering. The reason for choosing this method is based on experiences gained in structured solver loops.

As described previously, in structured grid codes the members of a set are numbered based on their index positions in the mesh. Thus, an array is referenced by two (three) indices in two (three) dimensions and the loops are generated over index directions. This can induce spatial locality of data but if the grid is large then temporal locality is lost as the cache cannot retain data associated with a point (line) of the grid by the time it has completed computations on a line (plane) of the grid. To increase temporal locality, and thus enhance the performance of the code on hierarchical memory machines, loop-blocking[15] is frequently used. This enables one physical block of the grid to be worked on at a time. The members of the set within a block are brought into the cache and reused before moving onto the next block. This loop-blocking

Table 1. Properties of the IBM RS/6000, SGI Indy, SGI Indigo and SGI R8000

Computer	IBM RS/6000	SGI Indy	SGI Indigo	SGI R8000
Processor	Model 350	R4600	R4000	R8000
Frequency	41·7 MHz	100 MHz	100 MHz	75 MHz
Peak performance	83·4 Mflops	33 Mflops	33 Mflops	300 Mflops
Instruction cache	32 KBytes	16 Kbytes	8 KBytes	16 KBytes
Data cache	32 KBytes	16 Kbytes	8 KBytes	16 KBytes
Cache organisation	Four-way	Two-way	Direct	Direct
Line size	64 Bytes	32 Bytes	16 Bytes	32 Bytes
Secondary cache	N/A	N/A	1 MBytes	4 MBytes
Cache organisation	N/A	N/A	Direct	Four-way
Line size	N/A	N/A	128 Bytes	512 Bytes
TLB organisation	Two-way	Fully	Fully	Three-way
# TLB entries	128	96	96	384
Page size	4 KBytes	4 KBytes	4 KBytes	16 KBytes
Main memory	64 MBytes	32 Mbytes	64 MBytes	512 MBytes

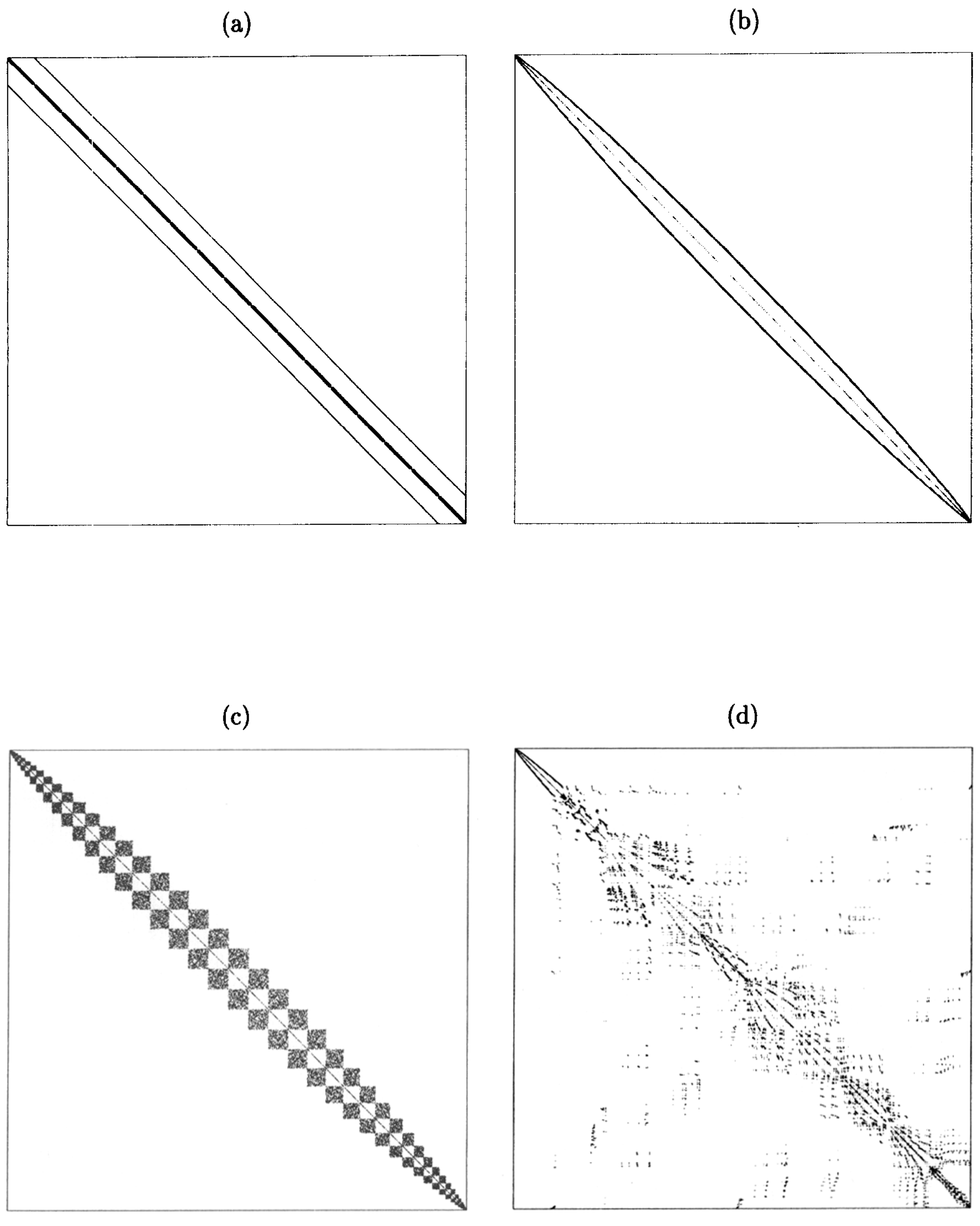

Fig. 6. Sparse matrix originating from 17 × 17 × 17 cube mesh with (a) lexicographical ordering, (b) Reverse Cuthill–McKee ordering, (c) Reverse Cuthill–McKee with random ordering within each layer, and (d) Greedy orderings with a block size of 500.

concept can be 'applied' implicitly to the unstructured solver's loops by renumbering the input data into blocks. This is the result of applying the Greedy method.

A variant of the Greedy algorithm that uses a mapping between the same set is described here. Essentially, the algorithm implicitly creates blocks in the grid when assigning a new number (number permutation) to each member of the set. To start the Greedy algorithm, a member on the boundary of the grid is chosen (step 1). The neighbouring information, specified by the set mapping to itself, is then used to fill blocks of a given block size (step 4). Each block originates from the interface of a previous block to ensure data locality between blocks whenever possible (step 6). Once the algorithm runs out of neighbouring

Table 2. Timing (in ms) a Jacobi smoother on a 65 × 65 × 65 structured cube grid with various vertiex numberings

Algorithm	IBM RS/6000	SGI Indigo	SGI Indy	SGI R8000
Generator	267	593	530	201
Completely random	1 633	2 494	1 664	247
Greedy (10)	296	671	573	218
Greedy (100)	281	643	552	214
Greedy (1000)	284	636	579	210
Greedy (10 000)	314	648	611	209
RCM	270		546	202
RCMLR	402		893	202

members, it tries to jump to a previous block–interface location (step 7). This produces a disconnected block within the same connected grid component. However, if all previous block–interface members have been assigned a number permutation then all the members within the connected grid component have a number permutation. In this case, the algorithm jumps to a new disconnected grid component (step 8). The precise algorithm to find a number permutation from old to new numbers for N_B set B members is as follows:

Greedy algorithm

```
1. input $M_B$ = block size, and choose a member $b_k \in B$
2. initialise: $counter = 0$, number permutation $P_B(1:N_B) = 0$ and list $I = \emptyset$
3. initialise list $L$ to be $b_k$'s neighbours with no number permutation
4. while ($L \neq \emptyset$ and $mod(counter, M_B) \neq 0$ and $counter < N_B$)
      $b_i$ = first member in list $L$
      $counter = counter + 1$
      $P_B(b_i) = counter$
      remove $b_i$ from $L$ and $I$
      add $b_i$'s neighbours with no number permutation to end of list $L$
5. if ($counter = N_B$) FINISHED
6. if ($mod(counter, M_B) = 0$) start a new block with $b_k \in L$
7. if ($L = \emptyset$ and $I \neq \emptyset$) find a previous block–interface location with $b_k \in I$
8. if ($L = \emptyset$ and $I = \emptyset$) find a $b_k \in B$ with no number permutation
9. add $L$ to interface list $I$ and goto 3.
```

The Greedy algorithm is $O(N_B)$. Figure 6(d) shows the sparsity pattern of a cube mesh with a block size of 500 to illustrate the effect of the Greedy method.

4.3 Consistent renumbering of all sets based on the first set

Once the first set has been renumbered it is important to renumber all the other sets consistently based on this first set. To describe the concept of consistent renumbering of all sets a FORTRAN 77 example sparse matrix–vector product in *compressed sparse row* format[16] is analysed below:

```
  SUBROUTINE AX(NROW,IROW,NE,NCOL,A,X,Y)
C—Sparse matrix–vector product Y=AX in Compressed
C—Sparse Row format
  INTEGER NROW,IROW(NROW+1),NE
  INTEGER(NE),IE,I,J
  REAL*8 A(NE),X(NROW),Y(NROW),SUM
  DO IE=1,NROWS
    SUM=0.0D0
    DO I=IROW(IE),IROW(IE+1)-1
C... Find column position of Ith non-zero value of A
      J=NCOL(I)
C...Perform matrix–vector product
      SUM=SUM+A(I)*X(J)
    ENDDO
    Y(IE)=SUM
  ENDDO
```

The double precision vector A represents an $N \times N$ matrix, where N is the number of vertices in the mesh, with the non-zero values a_{ij} of the sparse matrix stored row by row; i and j are the global vertex numbers in the mesh and a_{ij} represents an edge joining these vertices. Integer vector NCOL contains the column indices of the elements a_{ij} stored in A and integer vector IROW contains the pointers to the beginning of each row in A and NCOL. A and NCOL are accessed sequentially in the DO-loop and will have spatial locality. Thus on an IBM RS/6000 with a line size of 64 bytes there will be one cache miss for every eight loads of A when working in double precision and one cache miss for every 16 loads of NCOL. However, the number of cache misses that occur for vector X will depend crucially on the order of the columns and interferences between other lines of data.[13] To increase the spatial locality of X the vertices should be renumbered. If the vertices are renumbered then edges will have to be reordered so that the rows will be worked on in ascending order. This will also increase the temporal locality of X. Thus vertices and edges should be renumbered consistently with each other.

Before consistently renumbering other sets the number permutation for set B, calculated using one of the methods in Section 4.2, should be applied to:

— reorder all data associated with set B
— renumber all mappings that map to set B
— reorder all mappings that map from set B

Table 3. Simulation of one Jacobi smoother iteration on a 65 × 65 × 65 cube grid with various vertex numberings on an IBM RS/6000 model 350

Quantity	Generator	Random	Greedy (100)	RCM	RCMLR
LOADS into registers					
# cache hits and TLB hits	4 328 173	2 778 080	4 303 262	4 328 173	3 867 043
# cache hits and TLB misses	0	14 926	1 145	0	0
# cache misses and TLB hits	336 157	633 581	356 104	336 157	797 287
# cache misses and TLB misses	4 295	1 242 038	8 114	4 295	4 295
STORES into memory					
# cache hits and TLB hits	514 921	513 827	514 783	514 921	514 921
# cache hits and TLB misses	0	1 094	138	0	0
# cache misses and TLB hits	33 791	33 658	33 774	33 792	33 792
# cache misses and TLB misses	538	671	555	537	537
Cycle predictions					
# cycles to load A	697 201	726 744	699 089	697 201	697 201
# cycles to load P	2 964 858	2 972 175	2 967 567	2 964 858	2 964 858
# cycles to load and store Q	4 178 585	60 577 078	4 584 913	4 178 585	9 407 799
# cycles to load F	697 201	727 534	699 522	697 201	697 201
# cycles to load and store DQ	1 394 360	1 421 682	1 397 841	1 394 360	1 394 360
Total	9 932 206	66 425 212	10 348 933	9 932 206	15 161 420
Average # cycles per memory operation	1·907	12·73	1·983	1·903	2·906
Total # cycles + 7*NVERT	11 854 581	68 347 587	12 271 308	11 854 581	17 083 795
Total time (ms)	284	1 639	294	284	410
Actual time	267	1 633	281	270	402

All other sets that are connected to set B can now be consistently renumbered. For mappings where set B$\mapsto$C, a number permutation from old to new set C numbers can be found using the new ordering of set B members in the mapping. All set C members will be assigned a new number if they all have mappings from set B members. The algorithm is as follows:

Consistent renumbering of set C using B $\mapsto$ C mapping

```
1. initialise $counter = 0$ and number permutation
   $P_C(1 : N_C) = 0$
2. loop over set B members $b_k$ and for each neighbouring $c_i$
   with no number permutation:
       $counter = counter + 1$
       $P_C(c_i) = counter$
```

Table 4. Timings (in ms) a matrix–vector product with 439 542 matrix non-zeros and 27 720 edges with various vertex numberings

Algorithm	IBM RS/6000	SGI Indigo	SGI Indy	SGI R8000
Generator	126·8	193·4	201·2	64·0
Completely random	185·6	211·4	280·0	63·9
Greedy (10)	89·7	172·5	150·5	63·6
Greedy (100)	89·9	172·4	151·0	63·4
Greedy (1 000)	90·7	172·6	152·1	62·9
RCM	90·5	172·2	151·1	63·1
RCMLR	96·5	176·3	162·4	63·4

Table 5. Timings (in ms) a matrix–vector product with 2 706 709 matrix non-zeros and 168 403 edges with various vertex numberings

Algorithm	IBM RS/6000	SGI Indigo	SGI Indy	SGI R8000
Generator	1 207	2 013	5 073	530
Completely random	2 395	3 302	6 373	537
Greedy (10)	554	1 166	3 315	476
Greedy (100)	558	1 134	3 279	471
Greedy (1 000)	567	1 166	3 278	471
Greedy (10 000)	567	1 138	3 248	471
RCM	563	1 186	3 239	470
RCMLR	627	1 182	3 449	474

Table 6. Simulation of a matrix–vector product with 439 542 matrix non-zeros and 27 720 edges on a IBM RS/6000 model 350

Quantity	Generator	Random	Greedy (100)	RCM	RCMLR
LOADS into registers					
# cache hits and TLB hits	1 140 003	922 835	1 274 675	1 272 050	1 250 360
# cache hits and TLB misses	927	901	491	278	578
# cache misses and TLB hits	231 400	448 237	97 372	100 344	121 706
# cache misses and TLB misses	1 736	2 093	1 528	1 394	121 706
STORES into memory					
# cache hits and TLB hits	23 905	23 882	24 170	24 224	24 227
# cache hits and TLB misses	349	372	84	30	27
# cache misses and TLB hits	3 357	3 350	3 399	3 407	3 407
# cache misses and TLB misses	109	116	67	59	59
Cycle predictions					
# cycles to load irow	87 974	90 047	81 674	81 146	81 970
# cycles to load ncol	794 916	794 011	794 505	791 826	793 896
# cycles to load x	2 172 801	4 644 731	641 349	669 435	914 853
# cycles to load A	1 118 515	1 119 020	1 118 714	1 117 322	1 118 790
# cycles to store y	79 477	80 183	72 560	71 169	71 105
Total	4 253 683	6 727 993	2 708 803	2 730 897	2 980 614
Average # cycles per memory operation	2·38	3·77	1·51	1·53	1·67
Total # cycles + #matrix non-zeros + 10 * #edges	4 970 425	7 444 735	3 425 545	3 447 639	3 697 356
Total time (ms)	119·2	178·5	82·1	82·7	88·7
Actual time	126·8	185·6	89·9	90·5	96·5

For mappings where set D maps to set B, a number permutation from old to new set D numbers can also be determined based on the new set B numbers in the mapping. One way of approaching this is to first create the inverse mapping of set B to set D generated by a linked list. The set D number permutation can then be found by applying the above algorithm. This will work provided each set D member maps to at least one set B member in the D$\mapsto$B mapping.

Now all sets can be assigned a number permutation using the two consistent renumbering algorithms. Once the data and mappings have been converted to

Table 7. Simulation of a matrix–vector product with 2 706 709 matrix non-zeros and 168 403 edges on a IBM RS/6000 model 350

Quantity	Generator	Random	Greedy (100)	RCM	RCMLR
LOADS into registers					
# cache hits and TLB hits	6 891 273	5 397 666	7 828 500	7 804 827	7 574 402
# cache hits and TLB misses	17 576	31 701	5 191	5 948	6 523
# cache misses and TLB hits	1 016 529	1 452 410	612 577	636 577	866 515
# cache misses and TLB misses	531 555	1 575 156	10 665	9 581	9 493
STORES into memory					
# cache hits and TLB hits	145 402	142 858	147 068	146 566	146 286
# cache hits and TLB misses	1 949	4 487	283	785	1 065
# cache misses and TLB hits	20 457	20 132	20 679	20 607	20 563
# cache misses and TLB misses	595	926	373	445	489
Cycle predictions					
# cycles to load irow	533 094	597 934	494 650	499 428	504 544
# cycles to load ncol	4 891 474	4 898 215	4 903 092	4 908 678	4 907 866
# cycles to load x	31 211 084	80 877 570	4 173 516	4 403 105	7 013 131
# cycles to load A	6 893 198	6 895 771	6 904 207	6 906 195	6 907 766
# cycles to store y	477 062	541 144	434 894	447 760	455 057
Total	44 005 912	93 810 633	16 910 360	17 165 167	19 788 363
Average # cycles per memory operation	4·00	8·53	1·54	1·56	1·80
Total # cycles + #matrix non-zeros + 10 * #edges	48 396 650	98 201 372	21 301 099	21 555 906	24 179 102
Total time (ms)	1 161	2 355	511	517	580
Actual time	1 207	2 395	558	564	627

Table 8. Timing (in s) 10 time steps of an edge-based Euler calculation on a grid of 586 920 edges and 84 734 vertices with various vertex numberings

Algorithm	IBM RS/6000	SGI Indigo	SGI Indy	SGI R8000
Generator	512	653	821	154
Completely random	760	1 107	1094	332
Colouring	577	844	948	224
Greedy (10)	414	485	682	139
Greedy (100)	421	483	681	138
Greedy (1 000)	423	485	686	138
Greedy (10 000)	431	486	684	139
RCM	418	486	686	138
RCMLR	482	568	810	140

the new numberings, the grid is ready to be operated on.

5 RESULTS

Timings have been carried out for three unstructured grid solvers on IBM RS/6000 model 350, SGI Indy, SGI Indigo and SGI Power Challenge R8000 machines described in Table 1. All real arrays have been stored and operated on in double precision for the Jacobi and electromagnetic solvers, and in single precision for the Euler solver. The following numbering schemes have been analysed:

— original numbering from grid generator;
— bandwidth reduction using RCM;[5,23]
— RCM with random numbering within each level (RCMLR);
— Greedy method (vertex block sizes are given in brackets);
— completely random numbering;
— vector colouring for the Euler solver.

The first four renumbering schemes have been applied to the first set and all other sets have been consistently renumbered based on the new numbering of the first set. The purpose of testing bandwidth reduction with random numbering within each level is to find out if the numbering within a level is important for data locality. Completely random numbering of each set is chosen to show the effect of a numbering method that does not take into account data locality. Colouring is used in codes executed on vector processors to avoid data dependencies and is also included to show the effect on performance when codes are directly ported from vector processors to workstations.

The first solver tested is a Jacobi smoother[19] to solve Poisson's equation using unstructured grid information on a structured cube grid. The smoother loops over vertices and accesses neighbouring vertex values via a vertices$\mapsto$vertices mapping to update the variables at the vertices. In this example vertices are the only set that require renumbering. The times per iteration on different sized structured cube meshes are shown in Table 2. The Jacobi smoother has the following FORTRAN 77 implementation:

```
      SUBROUTINE JACOBI(NS,P,A,F,Q,DQ,NITER)
C—Apply several Jacobi smoothing iterations
      INTEGER NS,P(6,NS),I,ITER,NITER
      REAL*8  A(NS),F(NS),Q(NS),DQ(NS)
      DO ITER=1, NITER
C...Calculate residual DQ
        DO I=1, NS
          DQ(I)=A(I) * (Q(P(1,I))+Q(P(2,I))+Q(P(3,I))
     &                 +Q(P(4,I))+Q(P(5,I))+Q(P(6,I))
     &                 -6·0*Q(I)+F(I))
        ENDDO
C...Update Q
        DO I=1, NS
          Q(I)=Q(I)+0·1*DQ(I)
        ENDDO
      ENDDO
      RETURN
      END
```

The subroutine calculates the residual, DQ, and updates the NS unknowns, Q, using information gathered by the mapping, P(6,:), that maps to the six neighbouring vertices in the 7-point stencil. F is a forcing function and A are weights that impose Dirichlet boundary conditions when the vertex is on the boundary.

Table 3 displays the simulator's results of IBM RS/6000 model 350 memory accessing for the Jacobi smoother with different grid numberings. The simulator gives precise information on the type of memory accesses required for one iteration. It also predicts the total number of cycles per variable, the average number of cycles per memory access and the execution time. In order to calculate the execution time we use the formula:

$$\text{estimated execution time} = \frac{\#\text{memory accessing cycles} + \#\text{non-memory cycles}}{\text{frequency of clock cycle}}. \quad (1)$$

The simulator predicts the total number of cycles required for memory accessing using the hit/miss parameter costs determined in Section 3. An extra

Table 9. Time (in s) taken to renumber an unstructured grid consistently for the electromagnetic solver on a IBM RS/6000 model 350

Grid sizes				Grid renumbering times (s)	
Cells	Faces	Edges	Vertices	Greedy (100)	RCM
22 499	46 138	27 720	4 083	19·0	19·3
139 367	283 684	168 403	24 088	122·3	131·4

seven non-memory cycles per vertex per iteration are also required according to the assembly code produced by the IBM RS/6000 compiler. The clock cycle frequency for the IBM RS/6000 model 350 is 41·7 MHz. The last two rows in Table 3 give the predicted and actual RS/6000 execution times.

The second code is a three-dimensional electromagnetic solver for determining the solution of Maxwell's equations. The reader is referred to Monk *et al.*[7] for details of the discretisation and implementation of the solver. Essentially, the main body of work in the solver uses the method of conjugate gradients[19] to find the solution of a linear system of equations. Therefore, most of the expense is in computing a matrix–vector product. This has been implemented in compressed sparse row format, as in Section 4.3. The discretisation leads to a symmetric positive definite matrix of size $N_E \times N_E$, where N_E is the number of edges in the grid, and A is of length $(2 * \#\text{ edge pairs} + \#\text{ edges})$. Timings of the matrix–vector product are presented here for two cone–sphere meshes: 27 720 edges with 439 542 matrix non-zeros in Table 4 and 168 403 edges with 2 706 709 matrix non-zeros in Table 5. The original grids were created using a frontal grid generator similar to that described in Ref. 2. It should be noted that the relatively large times taken by the SGI Indy in the larger grid calculation in Table 5 were due to many page faults caused by a small main memory.

Again, the number of cache and TLB hits/misses are determined by the simulator for an IBM RS/6000 model 350, see Tables 6 and 7. An estimate of the execution time is calculated using eqn (1). The simulator predicts the total number of cycles for memory accessing, and from the RS/6000 compiler's assembly code the number of non-memory cycles is 1 in the inner loop and 10 in the outer loop. The estimated and actual times for the various renumbering strategies are also shown in Tables 6 and 7.

The third code is a fluids application for solving the Euler equations in three dimensions. The discretisation and edge-based implementation is given in Peraire *et al.*[8] The timings shown in Table 8 include reading in the grid of 151 158 vertices and 980 891 edges and weights from file, and taking 10 time steps. The original grid was obtained from Weatherill's Delaunay grid generator.[3]

The cost of renumbering is now evaluated. The time taken on an IBM RS/6000 model 350 to renumber the electromagnetic grids for each renumbering technique is displayed in Table 9. Vertices were the first set to be renumbered using the cells$\mapsto$vertices mapping. The cells were renumbered second also using the cells$\mapsto$vertices mapping, the edges third using the cells$\mapsto$edges mapping and finally the faces using the cells$\mapsto$faces mapping. These mappings along with vertex coordinates and boundary flags were stored in the input file, and the edge-pair's matrix is generated in the electromagnetic code. If the edge numbering has good data locality then the edge-pair numbers will preserve data locality when rearranged into compressed sparse row format.

6 CONCLUSIONS

Timings from all the unstructured grid codes show that renumbering with a bandwidth reduction or Greedy method, combined with consistent renumbering significantly improves the performance of all codes on memory hierarchy machines. These renumbering methods optimise data locality. As expected, renumbering has a greater influence on machines with smaller caches, such as the SGI Indy machine in this paper. However, all hierarchical memory machines benefit from the small initial cost of renumbering the input grid; the larger the grid the greater the performance benefit.

Code timings with grids renumbered by the Greedy algorithm appear to be independent of block size. This means that there is a flexibility in the choice of block size for the Greedy algorithm. This is contrary to the structured grid solvers where the block size in loop-blocking is important and closely related to the size of the cache.

For the larger grid test cases the numbering within each level of the bandwidth reduced matrix affects the timings. A random numbering within each level gave slower times than numbering produced by the RCM algorithm. This indicates that the choice of bandwidth reduction method could be important.

The simulator gives realistic predictions for timings and confirms all of the above results. The simulator also shows that the average number of cycles per memory operation for bandwidth reduced and Greedy grid numberings remains constant with grid size, see Tables 5 and 6. However, the average number of cycles per memory operation grows for the original, random numbered and coloured grids. The simulator also confirms that the number of cycles to access variables that are sequentially loaded is independent of renumbering while renumbering of indirectly accessed variables can significantly affect the number of cycles. This means that renumbering of the grid will be more effective on codes with a high proportion of indirect addressing compared to other operations.

Many grid generators have an option to renumber the grid with a colouring algorithm for vector processing. Inevitably, a renumbering strategy for memory hierarchy machines will also be built into grid generators. A renumbered grid will optimise memory accessing of codes run on workstations, distributed memory and shared memory machines (all have memory hierarchies). We recommend that grid generators incorporate the RCM algorithm as it optimises memory hierarchy performance, and produces a good ordering for LU solvers and ILU preconditioners.

In the Jacobi timings it is noted that the original structured grid with lexicographical ordering gave better performance than any other numberings. This is not the case though with unstructured grids generated from Delauney or frontal methods. With grids originating from these methods, renumbering can improve the performance by over a factor of two.

ACKNOWLEDGEMENTS

This work was performed within Oxford Parallel with financial support from Rolls-Royce plc, DTI and EPSRC. The authors would like to thank: Ron Bell (IBM Corp.) for his helpful suggestions on the simulator; Nigel Weatherill (Swansea University) for providing grids for the Euler computation; Paul Wesson and Kevin Parrott (Oxford University) for their electromagnetic code; Rolls-Royce plc and Jamie Peraire for their Euler code; and Paul Crumpton (Oxford University) and Mike Rudgyard (CERFACS) for their stimulating discussions.

REFERENCES

1. Baskett, F. & Hennessy, J. L., Microprocessors: from desktops to supercomputers. *Science*, 1993, **261**, 864–71.
2. Müller, J. D., Roe, P. L. & Deconinck, H., A frontal approach for node generation in Delaunay triangulation. AGARD Report 878, pp. 9.1–9.7, May 1992.
3. Weatherill, N. P. & Hassan, O., Efficient three-dimensional grid generation using the Delaunay triangulation. *Computational Fluid Dynamics '92*, eds Ch. Hirch, J. Périaux & W. Kordulla, vol. 2, pp. 961–968. Elsevier, Oxford, 1992.
4. Löhner, R., Finite element methods in CFD: grid generation, adaptivity and parallelisation. AGARD Report 787, pp. 8.1–8.58, May 1992.
5. Cuthill, E. & McKee, J., Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM Nat. Conf.*, pp. 157–172, 1969.
6. Farhat, C., A simple and efficient automatic FEM domain decomposer. *Comput. Struct.*, 1988, **28**, 579–602.
7. Monk, P. B., Parrott, A. K. & Wesson, P. J., A parallel electromagnetic scattering code. *COMPEL*, 1994, **13**, Supplement A:237–242.
8. Peraire, J., Peiró, J. & Morgan, K., A 3d finite element multigrid solver for Euler equations. *AIAA Paper 92-0449*, 1992.
9. Bell, R., IBM RISC system/6000 NIC tuning guide for fortran and C. Technical Report GG24-3611-01, IBM International Technical Support Center, Poughkeepsie, New York 12602, July 1991.
10. Patterson, D. A. & Hennessy, J. L., *Computer Architecture: A Qualitative Approach*, Morgan Kaufmann, 1990.
11. Simha, S., *R4400 Microprocessor Product Information.* MIPS Technologies Inc, 2011 North Shoreline Blvd, Mountain View, CA 94039, September 1993.
12. Bell, R., IT Consultant (NIC), IBM UK Ltd, February 1994. Personnal Communication.
13. Teman, O. & Jalby, W., Characterizing the behavior of sparse algorithms on caches. In *Supercomputing '92*, Minneapolis, December 1992.
14. Warren, H. S., Predicting execution time on the IBM RISC System/6000. Technical Report GG24-3711, IBM International Technical Support Center, Poughkeepsie, NY 12602, July 1991.
15. Bacon, D. F., Graham, S. L. & Sharp, O. J., Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-891, Computer Science Division, University of California, Berkley, CA 94720, 1993.
16. Saad, Y., *SPARSKIT: a Basic Tool Kit for Sparse Matrix Computations.* Computer Science Department, University of Minnesota, Minneapolis, MN 55455, version 2 edition, June 1994.
17. Knijnenburg, P. M. W. & Wijshoff, H. A. G., On improving data locality in sparse matrix computations. Technical Report 94-15, Department of Computer Science, Leiden University, 1994.
18. Duff, I. S. & Meurant, G. A., The effect of ordering on preconditioned conjugate gradients. *BIT*, 1989, **29**, 635–57.
19. Golub, G. H. & Van Loan, C. F., *Matrix Computations*, 2nd edn, John Hopkins, Baltimore, MD, 1989.
20. Malone, J. G., Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers. *Computer Meth. Appl. Mech. Engng.*, 1988, **70**, 27–58.
21. Simon, H. D., Partitioning of unstructured problems for parallel processing. *Comput. Systems Engng.*, 1991, **2**, 135–48.
22. Das, R., Mavriplis, D. J., Saltz, J., Gupta, S. & Ponnusamy, R., The design and implementation of a parallel unstructured Euler solver using software primitives. ICASE Report 92-12, March 1992.
23. George, A., Computer implementation of the finite element method. Technical Report STAN-CS-71-208, Computer Science Department, Stanford University, California, 1971.
24. Gibbs, N. E., Poole Jr, W. G. & Stockmeyer, P. K., An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.*, 1976, **13**, 236–50.