Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR TP 97329

Partitioning and parallel development of an unstructured, adaptive flow solver on the NEC SX-4

H. van der Ven and J.J.W. van der Vegt

DOCUMENT CONTROL SHEET

	ORIGINATOR'S REF. TP 97329 U		SECURITY CLASS. Unclassiefied

ORIGINATOR
National Aerospace Laboratory NLR, Amsterdam, The Netherlands

TITLE
Partitioning and parallel development of an unstructured, adaptive flow solver on the NEC SX-4

PRESENTED AT
the Parallel Computational Fluid Dynamics '97 Conference, Manchester, England, May 19-21, 1997. The work described in this report is partially supported by the Dutch Foundation HPCN in the project NICE under contract no. 96009.

AUTHORS H. van der Ven and J.J.W. van der Vegt	DATE 970626	pp ref 12 5

DESCRIPTORS

Adaptation	Galerkin method
Algorithms	Grid generation (mathematics)
Compressible flow	Parallel processing (computers)
Computational fluid dynamics	Partitions (mathematics)
Computer systems performance	Unsteady flow
Finite element method	Unstructured grids (mathematics)

ABSTRACT
NLR is developing a parallel, unstructured, adaptive flow solver based on hexahedrons. Development is directed toboth the numerical algorithm and parallel efficiency. After an earlier do-loop parallelization of the flow solver on a NEC SX-4/16, a parallelization strategy for the adaptation algorithm based on mesh partitioning will be described and executed. Parallelization results will show that shared memory machines are excellent platforms to develop domain decomposition methods for a flow solver which is both under numerical development and used in production.

Summary

NLR is developing a parallel, unstructured, adaptive flow solver based on hexahedrons. Development is directed to both the numerical algorithm and parallel efficiency. After an earlier do-loop parallelization of the flow solver on a NEC SX-4/16, a parallelization strategy for the adaptation algorithm based on mesh partitioning will be described and executed. Parallelization results will show that shared memory machines are excellent platforms to develop domain decomposition methods for a flow solver which is both under numerical development and used in production.

Contents

(12 pages in total)

1 Introduction

The National Aerospace Laboratory NLR is developing a parallel, adaptive, unstructured flow solver based on a discontinuous Galerkin finite element discretization of the Euler/Navier-Stokes equations for compressible flow. The main incentive for the development of this flow solver is time-accurate flow simulation. Because of the computational complexity of the applications, the underlying algorithm is already parallelized during the algorithm development stage. This requires frequent mutual updates, and step-by-step development of new functionality and performance increases. Since the flow solver is also used for production runs, at each stage the functionality should not degrade. In this paper it will be shown that shared memory machines are excellent platforms to accomplish this.

The two main components of the flow solver are the solution of the unsteady Euler equations of compressible gas dynamics and a grid adaptation algorithm to improve capturing of local flow phenomena. In previous work the flow solution part of the flow solver has been parallelized on an NEC SX-4/16 with very satisfactory results using do-loop parallelization and some special features of the discretization method (Van der Ven et al, Ref. 4: a speedup of 11.5 on 14 processors reaching a speed of 8.5 Gflop/s). Nonetheless, for a more efficient parallelization of the flow solution algorithm and parallelization of the grid adaptation algorithm a mesh partitioning is required. A dynamic load balancing tool based on a local migration technique has been developed and integrated with the flow solver. During the step-by-step parallel development the mesh partition will be used to parallelize increasingly more parts of the flow solver. Further developments will be motivated by expected performance increases.

The mesh partition will be first applied to parallelize the adaptation algorithm. This algorithm cannot be efficiently parallelized by low level do-loop parallelization and for time-accurate simulations with frequent grid adaptations, the serial adaptation algorithm would consume too much computing time compared to the parallel flow solver.

The present paper is organized as follows. In the next chapter the algorithms used in the flow solver will be described; emphasis will be laid on the adaptation algorithm. In Chapter 3 the dynamic load balancing algorithm will be described. In Chapter 4 the parallelization of the adaptation algorithm and in Chapter 5 the performance results will be discussed. In Chapter 6 issues for parallel development of parallel software will be addressed. In the last chapter conclusions will be drawn.

2 Description of the algorithm

NLR is developing a new, discontinuous Galerkin finite element algorithm for the Euler/Navier-Stokes equations, Ref. 3. The discontinuous Galerkin finite element method can be considered as a mixture of an upwind finite volume method and a finite element method. A combination of local grid refinement and the discontinuous Galerkin finite element method is applied in the flow solver Hexadap. This combination is capable of efficiently resolving local phenomena such as shocks and vortical structures.

The grid structure consist of hexahedronal cells, which may be connected in an arbitrary way. Each cell can have an unlimited number of neighbouring cells. The starting grid is a structured, boundary conforming mesh, which is adapted during the flow simulation using anisotropic grid refinement and coarsening, resulting in a completely unstructured mesh.

The cell structure is stored in a data structure which is a forest consisting of trees rooting in the cells of the original mesh. Therefore, the cells of the original mesh are called 'root' cells. The trees consists of branches to the kid cells into which the (root) cell is divided. Only the leaves of the trees are cells which are used in the flow computations. They are called 'active cells', and constitute the mesh on which the flow is solved. The other cells in the tree are necessary to described the data structure. A more complete description of the data structure can be found in Van der Vegt et al, Ref. 3.

The initial mesh is frozen during the computations, that is, the data structure for this mesh is not changed. It is used to compute the face-cell connections of the adapted mesh, which are necessary for the update of the cell residuals by the face fluxes.

The anisotropic grid refinement is applied to the active cells. In each active cell c a difference $\nabla_{\vec{\mathrm{d}}}(c)$ of all flow variables is computed based on the neighbouring states in a given direction $\vec{\mathrm{d}}$, where $\vec{\mathrm{d}}$ is one of $\vec{\imath}$, $\vec{\jmath}$ or $\vec{k}$. This difference is multiplied with a power of the diameter $ds_{\vec{\mathrm{d}}}(c)$ of the cell in direction $\vec{\mathrm{d}}$ to obtain the value of the sensor function:

$$f_{\vec{\mathrm{d}}}(c) = \nabla_{\vec{\mathrm{d}}}(c)(ds_{\vec{\mathrm{d}}}(c))^2,$$

The cells are then ordered with respect to their sensor function values. Only the first N_1 cells in this order are coarsened and the last N_2 are refined. The numbers N_1 and N_2 are computed as user-defined percentages of the total number of cells. In this way the user has control over the number of cells that are deleted or created during an adaptation. Methods which adapt those cells for which the sensor function is above or below a certain threshold value are unpredictable in this respect.

3 Dynamic load balancing

Adaptive flow solvers require dynamic load balancing algorithms to repartition the mesh after a grid adaptation. This is also required when, as in the present case, the partition is only used to parallelize the adaptation algorithm, and not the flow solution part. Even though the workload during an adaptation step is much harder to predict than the workload during a flow solution step, no rebalancing of the mesh after an adaptation step would lead to large load imbalance in subsequent adaptations. Also, the rebalancing reduces memory use, since all data is stored per part in static arrays.

In this paper a dynamic load balancing tool based on the local migration technique of Özturan et al, Ref. 2, is used. Minor adaptations for shared memory machines are made. The algorithm iteratively migrates cells between neighbouring parts in the mesh. Load requests between parts determine the movement of cells. All parts request load from the most heavily loaded part. If they are not connected to that part they will try the next part. Cells to be transferred are assigned using a two dimensional greedy algorithm which repeatedly strips layers of cells which connect to both parts. Through the iterative process load may traverse from one part through another into the next part. This eventually allows load transfer between disconnected parts, and ensures connectivity.

The initial partition is obtained using the three dimensional greedy algorithm of Farhat, Ref. 1. The partitioning algorithm is applied to the root cells, in order to decrease the problem size, and hence improve efficiency. The root cells are weighted with the number of active cells contained in the root cell.

The above described algorithm efficiently produces well-balanced partitions. Moreover, the size of the boundaries of the parts remains roughly constant during the migration process. In Chapter 5 timings of the partitioning algorithm can be found. More details on the dynamic load balancing algorithm and on the results can be found in Van der Ven, Ref. 5.

4 Parallelization of the adaptation

The serial adaptation algorithm has the following structure:

```
do for all three directions
    compute sensor function for all cells
    order cells with respect to sensor function values
    coarsen/refine cells (*update data structures for cells and grid points*)
enddo
update data structures for faces
```

All but one step in this algorithm can be parallelized using the partition in a straightforward way. The ordering of the cells with respect to their sensor function values is global in nature, and would therefore imply a serial section. This serial section becomes prohibitively large for large meshes and is about 5% of the adaptation time for the tests of the next section. By Amdahl's Law this serial section is an impediment for an efficient parallelization and this part of the algorithm should be changed.

The user control over the number of cells which are added or deleted during an adaptation should be maintained. The decision therefore was made to decrease the problem size by not ordering the complete set of cells but to collect the cells in buckets and order the buckets, which results in an approximation of the sensor function by a piecewise constant function. A typical sensor function is displayed in Figure 1: a large number of cells has a sensor function value close to zero (and are candidates for deletion) and a smaller number have large function values (and are candidates for refinement). The approximation of the sensor function is chosen such that the relevant features are preserved as much as possible. The bucket sizes vary quadratically, and the first buckets contain in the order of one element. The total number of buckets is 1% of the number of cells, thereby reducing the problem size by a factor 100. In Figure 1 examples of the approximation of the sensor function are shown for both coarsening and refinement.

After each adaptation of the mesh in a certain direction the mesh is repartitioned.

5 Performance results

The parallel performance is measured for two different mesh sizes around an ONERA M6 wing. Both meshes have the same root mesh consisting of 16,384 root cells. Both meshes are adapted three times, between which 33 time steps are solved. In the first test the mesh size is increased from 16,384 to 126,040 cells. In the second test the mesh size is increased from 189,049 to 300,652 cells.

In Table 1 elapsed timings are presented for the two tests. Timings are given for the complete adaptation algorithm, and for its three components: the partitioning algorithm, the adaptation

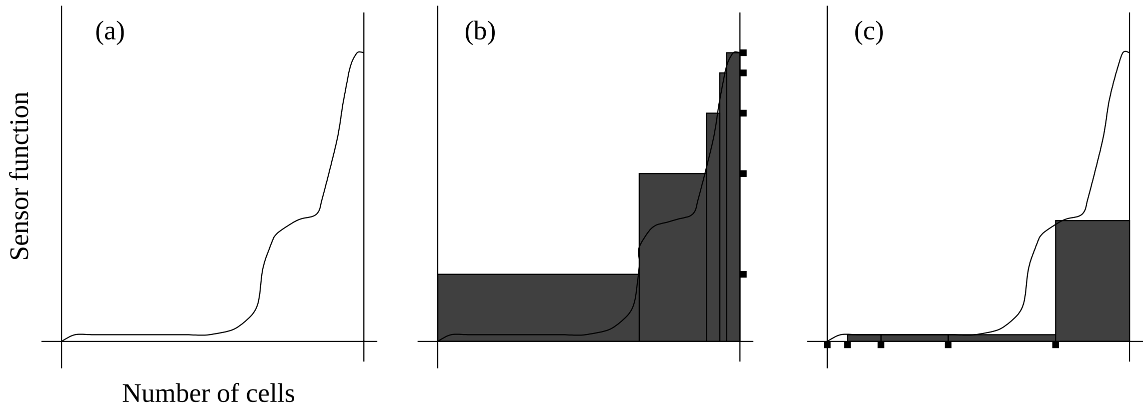

Fig. 1 Typical sensor function and approximations. (a) Sensor function. (b) approximation of sensor function for refinement, the ticks at the vertical axis determine the buckets and are placed at increasing distances from the peak value. (c) approximation of sensor function for coarsening, the ticks at the horizontal axis determine the buckets and are placed at increasing distances from zero.

Table 1 Elapsed timings [s] of the adaptation algorithm for both tests, left Test 1, right Test 2. The adaptation algorithm (Adapt) is split into three parts: partitioning (Part.), the cell part (Cells) and the face part (Faces).

procs	parts	Adapt	Part.	Cells	Faces
1	1	105	-	39.0	66.0
2	2	65.5	5.8	22.5	37.2
2	4	63.6	7.8	20.6	35.2
2	8	64.4	10.2	19.7	34.5
4	4	42.8	7.8	13.5	21.5
4	8	41.9	10.2	12.3	19.4
4	16	42.0	12.7	11.2	18.1
8	8	32.9	10.2	9.9	12.8
8	16	32.3	11.2	8.4	12.7
8	32	35.1	16.8	8.0	10.3

procs	parts	Adapt	Part.	Cells	Faces
1	1	441.	-	162.	279.
2	2	231.	5.66	86.3	139.
2	4	233.	13.6	82.4	137.
2	8	233.	17.8	78.7	137.
4	4	138.	13.7	50.7	73.6
4	8	135.	17.8	45.4	71.8
4	16	142.	25.0	45.0	72.0
7	7	95.1	16.4	33.6	45.6
7	14	102.	23.4	31.3	47.3
7	28	97.6	26.9	28.2	42.5

of the cells (including the computation of the sensor function), and the update of the face data structures.

Figure 2 shows the speedups for the cell and face parts of the adaptation algorithm. The speedups are computed based on the serial code which is obtained by compiling the parallel source ignoring the parallelization directives. The speedups are satisfactory, but not excellent. For the face part of the algorithm this can be explained by the fact that in the present implementation the work load of a part cannot be predicted beforehand. So it may happen that one of the last parts processed constitutes too much work, leading to load imbalance. This is not the case for the cell part of the algorithm. Here, once the sensor function has been calculated, the work load per part can be computed. The parts are then ordered with respect to the workload, and the most heavily loaded parts are processed first. If there are more parts than processors, quite a good load balance is achieved, as can be seen in Figure 2: the best speedups (shown as dashed lines) are obtained when more parts than processors are used. The reason for the decreasing efficiency for the cell part of the algorithm is caused by the parallel overhead for the serial sections. These serial sections are required for both the computation of the sensor function (if a cell neighbour lies on a different part) and the ordering of the bucket values. The effect of the parallel overhead is most clearly seen in the results of the smaller Test 1.

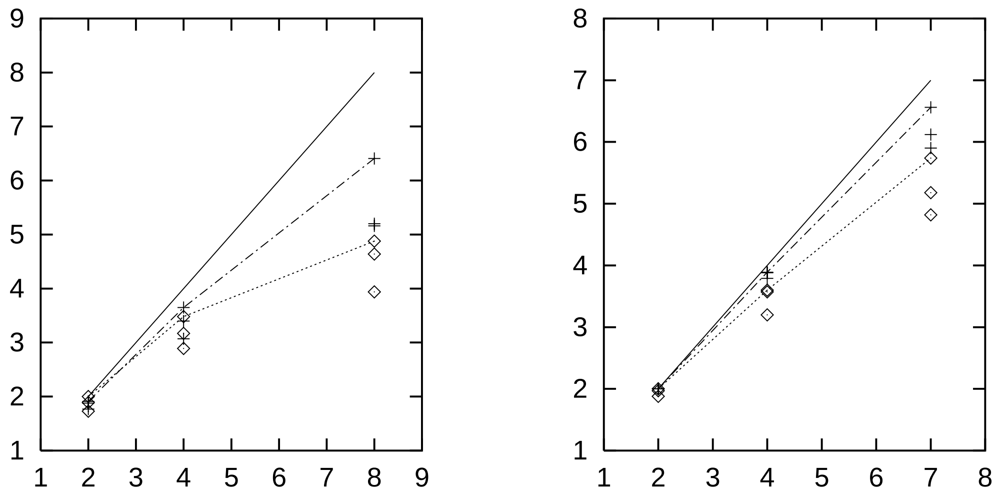

Fig. 2 *Speedups for the cell and face parts of the adaptation algorithm, left Test 1, right Test 2.* $\diamond$ *cells;* $+$ *faces; — perfect speedup; · · · maximum speedup cells; - · - maximum speedup faces. On the* x*-axis the number of processors, on the* y*-axis the speedup.*

For Test 1 the partitioning algorithm eventually takes as much time as the other two parts of the adaptation algorithm. Hence a parallelization of the partitioning algorithm is advised. This is feasible since the chosen local migration technique was selected for its parallelizability. When comparing the execution times for the partitioning of the two tests, it is clear that the strategy of reducing the problem size by partitioning the root cells instead of the active cells is a good one. Differences in the timings are caused by the fact that for the larger mesh more active cells have to be migrated from one part to another. The partitioning of the root cells takes the same amount of time, but the update of the part data structures takes more time.

6 Parallel development

The flow solution part of the flow solver presently is parallelized using do-loop parallelization without explicitly partitioning the mesh. A mesh partition allows for a different parallelization of the entire flow solver. However, application of a mesh partition requires a significant change in the data structures. Since the flow solver still is in development and also used for production simulations, it is not feasible to perform the required modifications for the partition in one step. Therefore it has been decided to perform the modifications step by step.

This step by step development of a parallelization strategy based on the mesh partition is feasible on shared memory machines. Since all the data is located on a shared memory the flow solver may *forget* the partition in certain parts of the algorithm, where the do-loop parallelization of Van der Ven et al, Ref. 4, still is used.

This strategy has proven to be feasible. In three steps the adaptation algorithm has been parallelized while keeping the functionality of the flow solver intact at each step. Moreover, during these steps other parts of the flow solver have been modified by other developers.

It is important to note that in this way two parallelization strategies are applied side-by-side in the same code.

7 Conclusions

In this paper the grid adaptation algorithm of an unstructured, adaptive flow solver has been parallelized using a dynamically obtained partition. Parallelization results are satisfying, but not perfect due to load imbalance and parallel overhead. The adaptation algorithm is, however, no longer the main limitation for improved parallel performance of the flow solver.

The partition is not used to parallelize the flow solver, as yet. The approach that two or more parallelization strategies co-exist in one code is feasible on shared memory machines. Moreover, shared memory machines allow for gradual development of a parallel algorithm, while maintaining code functionality.

Further parallelization of the algorithm using the partition will only be performed if a performance model predicts sufficient performance increase to make the effort worthwhile.

8 References

1. C. Farhat and M. Lesoinne, Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Intern. J. Num. Methods in Engrg.* **36** (1993) 745–764.
2. C. Özturan, H.L. deCougny, M.S. Shephard, and FJ.E. Flaherty, Parallel adaptive mesh refinement and redistribution on distributed memory computers. *Comp. Methods Appl. Mech. Eng.* **119** (1994) 123 – 137.
3. J.J.W. van der Vegt and H. van der Ven, Discontinuous Galerkin finite element method with anisotropic local grid refinement for inviscid compressible flows, submitted to *J. Comp. Physics*, (1997), also available as NLR TP 97421.
4. H. van der Ven and J.J.W. van der Vegt, Experiences with advanced CFD algorithms on NEC SX-4, in Laginha M. Palma and J. Dongarra, eds., *VECPAR '96 selected papers,* Lecture Notes of Computer Science, **1215**, Springer, Berlin, 1997.
5. H. van der Ven, A partitioning method for an adaptive unstructured flow solver on shared memory machines. NLR TR 97029 L, National Aerospace Laboratory, Amsterdam, 1997.