Strategies for Approaching Parallel and Numerical Scalability in CFD Codes

Roy Williams
California Institute of Technology, Pasadena, California, USA
Jochem Häuser and Ralf Winkelmann
Center for Logistics and Expert Systems, Salzgitter, Germany

In this paper, we consider strategies for achieving the two distinct types of scalability which are the objectives for most parallel scientific software: *parallel scalability* and *numerical scalability.* Parallel scalability is the ideal condition of perfect speedup, when the execution time of the code is inversely proportional to the number of processors. Numerical scalability is another ideal: that the computational resources required to solve a problem of a given size are linearly related to the problem size. If a code is scalable in both senses, then the solution time for a problem remains constant when problem size and the number of processors used to solve it increase together in a fixed ratio. Here we adopt the number of floating-point operations (flops) as the measure of computer resources, but we note that "problem size" may not be well-defined; for example, we might ask whether the problem size for a matrix operation is proportional to the order of the matrix, to the number of matrix elements, or to the number of non-zero matrix elements.

Clearly a parallel, efficient numerical code must be to some extent scalable in both senses, but often in practice these ideal conditions are conflicting. As an example, consider the so-called *N*-body problem: integrating the motion of *N* particles according to Newtonian gravity. The simplest algorithm, computing the interaction of each particle with each other particle, uses resources of order N^2 flops to evaluate the force on each particle, and it is easy to make this algorithm very efficient due to the simple load-balancing and high ratio of computation to communication. This algorithm has excellent parallel scalability, but poor numerical scalability. In contrast, the more complex multipole methods compute forces between local groups of particles, and can achieve a count of order $N\log N$ for force evaluation at the cost of difficult load-balancing and communication problems. In this case numerical scalability is achieved at the cost of less than perfect parallel scalability.

In the following discussion of a CFD code, a large proportion of the computation time is devoted to solving linear systems of equations. We consider the problem size to be defined by the number of grid points, *N*, which is also the order of the matrix. With a direct method like LU-decomposition, the flop-count is of order N^3, which is far from numerically scalable. The convergence rate of Krylov-space methods, such as the GMRES that we use, is controlled by the condition number, which is generally inversely proportional to the square of the mesh spacing, *h*. If we assume that Nh^3 remains constant with scaling, this means that the condition number for an iterative method rises as $N^{2/3}$, so that the flop count for solution rises at least as fast as $N^{5/3}$, which is much better than the direct methods, but still not numerically stable.

In the following, we use the ParNSS code [1-4] as a workbench on which to test some ideas of how to reach for numerical and parallel scalability. ParNSS is a multiblock Navier-Stokes solver, and the computational objective is the steady state flow rather than a time-accurate unsteady computation. In particular, this paper is concerned with approximating the full linear system inherent in a full implicit step with many small linear systems, through domain decomposition.

First we discuss the code in general and its convergence strategies, then the proposed new tools, and we finish with some experimental results on a 3D grid surrounding the Huygens space probe.

ParNSS: Parallel Navier-Stokes

The objective of this paper is to consider mechanisms for achieving parallel and numerical scalability in a CFD code. We present a new tool for this objective that is now available in the ParNSS code for convergence acceleration, and some preliminary tests of its efficacy.

ParNSS is a scalable, parallel Navier-Stokes code, originally developed to simulate hypersonic flow, now extended to transonic and subsonic flow also. The message-passing is written in PVM and it has run on SGI systems at CLE in Salzgitter, Germany, Intel Paragons at Caltech in Pasadena, California, and on the IBM SP2 at ESTEC in Noordwijk, Netherlands. ParNSS is a finite-volume, hexahedral multiblock solver that utilizes various techniques to accelerate convergence to a steady state, and we briefly summarize these techniques below. The new technique is the use of sub-block and super-block implicit steps, which is a way to interpolate smoothly between an explicit and a fully implicit step.

In earlier work [1-4], we have shown that ParNSS is already close to the ideal of parallel scalability due the sequencing of explicit steps, processor-local implicit steps, and fully implicit steps.

Explicit steps

When the code begins, the flow is set to the free-stream values everywhere in the domain. We start with an explicit scheme to create the general configuration of the flow, including a rough approximation of the position of the shocks. The CFL number is initially very small, and it is increased cautiously and conservatively. Explicit schemes are highly efficient on parallel machines because of the large ratio of computation to calculation, and in simple cases this explicit iteration is sufficient for full convergence. However, in interesting cases a point is reached when the residual decreases very little even with large numbers of iterations: the computation is said to be stalled.

Block-implicit steps

An implicit scheme is used to continue iterations. The major advantage is stability even with CFL numbers considerably greater than one, and the consequent rapid convergence. The disadvantage is that a fully implicit step involves the solution of a linear system involving the mesh over the whole solution domain, and is therefore much more computationally intensive than explicit steps. In fact we do not use fully implicit steps at this stage, but rather a block-implicit scheme with a locally-determined time step on each block, meaning that the time steps are in general different on each block. We are approximately solving the system of linear equations by neglecting matrix elements which connect cells in different blocks. Since it is the blocks which are distributed among processors, this approximation means that each solve can occur within a processor with no communication to other processors. Furthermore it should be noted that the value of a step is measured according to how much it can reduce the residual, not according to how well it solves the fully-implicit linear system; we are not doing a time-accurate solve here, but only trying to drive the residual to zero by any means possible.

During the block-implicit phase, the time-step is chosen on a block-by-block basis by a minimum over the block of a local smoothness measure. Here is another deviation form exact time-stepping,

which would of course require the same time-step on all blocks.

Root polishing

Finally, when it is clear that we are close to the steady state, we use the fully implicit step with a global time step, with all the communication overhead that this implies. We allow the timestep to become very large or infinite. The justification for this is by observing that an implicit step of a differential equation system with infinite time step is equivalent to using Newton's method for the steady state of that system. Newton's method is quadratically convergent when the initial approximation to the solution is close enough to the exact solution, meaning that the magnitude of the residual is squared with each step, so that it tends to zero very quickly.

We now consider two other convergence acceleration tools: sub-block implicit steps and grid sequencing.

Sub-block and super-block implicit steps

We have demonstrated that block-implicit steps are almost as efficient as fully implicit steps, but with considerably lower costs in terms of the number of floating-point operations (flops) required, which is because the number of flops needed to solve a linear system scales faster than linearly with system size. Also the solution of multiple, independent linear systems can be parallelized with high efficiency because no communication is required between the separate systems.

Thus we are led to consider the idea of further splitting of the original blocks into yet smaller pieces, in an attempt to provide a stepping scheme that can effectively reduce the residual but with even lower computational cost. We might heuristically justify sub-block implicit steps by considering a "range of influence" argument. The CFL number in compressible flow computations is a measure of the extent to which a change of fluid data at one grid point in the computational mesh can affect things at other grid points, and it is well-known that for explicit schemes the CFL number must be order one or less. On the other hand an implicit step is not restricted in CFL number, which is because a change in fluid data at a point can, in principle, be felt by all other points in the computational mesh. This infinite range of influence in the fully implicit step is the reason why it is computationally intensive, and furthermore the parallel scalability is severely reduced because of high communication and synchronization costs. Practically, however, we do not need an infinite range of influence at each step, and we may not even need a range of influence all the way across a block of the multiblock grid.

The explicit-implicit continuum

We observe that as the sub-block implicit step works with smaller and smaller blocks, it becomes closer to an explicit scheme, at least in terms of the range of influence. In contrast, a block-implicit scheme approaches the fully implicit scheme (at least when the number of blocks is small). Thus we observe that a sub-block implicit scheme may be thought of as a flexible interpolation between an explicit and an implicit scheme.

Continuing, we agglomerate blocks into larger entities, which we might call "super-blocks", with each linear solve taking place over the super-block rather than a single block. The final super-block is the whole solution domain, where we use the fully implicit step with an infinite timestep (precisely the Newton method) to drive the residual to machine zero. At each stage, the linear solves for the implicit steps are over sub-blocks or super-blocks, and the time-step is chosen as a minimum of a local quantity over the whole sub-block or super-block.

The three steps to a solve that are expressed above may be recast as a “schedule of implicitness”: beginning with a fully explicit scheme, we move to a slightly implicit scheme with very small blocks (perhaps 3x3x3) being treated independently, then larger and larger sub-blocks until we are doing the block-implicit scheme.

The schedule of implicitness is reminiscent of the temperature schedule in optimization by simulated annealing (SA). In that case, a stochastic acceptance-rejection procedure is used to minimize a function. The procedure is parameterized by a variable analogous to temperature, which decreases slowly to zero. The difficult part of a practical SA algorithm is the decision on how fast the temperature decreases, whether it is decided in advance or dynamically. In the same way, we must decide how many steps should be done for each sub-blocking or super-blocking, or produce a decision mechanism for moving on to the next larger subset of the gridpoints for the implicit steps.

References

1. J. Häuser, R. D. Williams and W. Schröder, *Parallel Computational Fluid Dynamics in Complex Geometries*, p. 858 in *Computational Fluid Dynamics Review 1995*, eds. M. Hafez and K. Oshima, Wiley, 1995.
2. J. Häuser, R.D. Williams, H.-G. Paap, M. Spel, J. Muylaert and R. Winkelmann, *A Newton-GMRES Method for the Parallel Navier-Stokes Equations*, Proceedings of CFD95, Pasadena, CA, eds. S. Taylor et. al., Elsevier North-Holland, Amsterdam, 1995.
3. J. Häuser, J. Muylaert, M. Spel, R. D. Williams and H.-G. Paap, *Results for the Navier-Stokes Solver ParNSS on Workstation Clusters and IBM SP1 using PVM*, p. 432 in *Computational Fluid Dynamics*, Eds. S. Wagner et. al., Wiley, 1994.
4. J. Häuser and R. D. Williams, *Strategies for Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines*, Int. J. Num. Meth. Fluids, **15** (1992) 51.