NIC-TR-2004-018

NIC ADI Implementation

Purpose

Alternating Directions Implicit (ADI) is a memory-conserving algorithm for solving differential equations. This document will describe the algorithm and the NIC's implementation of this algorithm.

ADI

The derivation of the ADI algorithm is attached to this report, and a more thorough description will be the topic of a forthcoming technical report.

Implementation

The current C++ version of the ADI code is based on a previously developed FORTRAN implementation. Figure 1 shows the general flow of the algorithm. Information regarding the source data and the data itself are passed to a set of configuration routines that initialize parameters and allocate and initialize arrays used in the computation. Once the parameters are initialized, the algorithm iteratively solves each directional equation and, after the z-direction is computed, the error in the computation is estimated. If the error estimate exceeds a predetermined tolerance, the solution sequence is repeated. The process is shown in figure 1. Each of the directional computations has to loop through all three dimensions, for example, the x-direction computation iterates through the z-axis at the top level, the y-axis following that and then has three separate iterations on the x-axis at the lowest level.

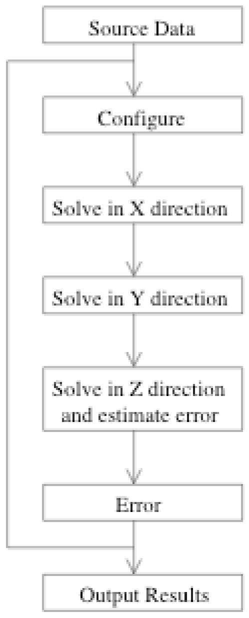

Figure 1. The ADI algorithm iteratively solves the Poisson equation in 3-dimensions by solving the equation in one direction at a time. This leads to a simple flow structure for the code, however, the directional solvers are complex.

Parallelization Issues

The computations for each direction solver depend on the computations of the previous solvers. For example, the x-direction solver depends on the values of the previously computed y-direction and z-direction solvers. These dependencies prevent us from computing the x-, y- and z-directions simultaneously and they prevent us from parallelizing anything but the inner loops.

In the shared memory environment, we can automatically setup loop unrolling with openMP. This is a simple matter of adding “pragma” statements to the existing code. The “pragmas” identify which loops can be unrolled and which values can be shared. With a judicious use of data sharing, the production of an efficient parallel implementation is a straightforward matter.

Producing an efficient parallel implementation on a distributed system is more complex than the shared memory implementation. Consider the geometry of the problem. The algorithm represents space as a regular 3-dimensional grid. The first two outer loops partition space into slices and rows (in this case a row implies a set of contiguous points in the x-, y- or z-directions). The selection of a particular slice or row is independent of any other slice or row. Each iteration of the inner loops can be executed independently of the others, but the order of the directional loops (x, y then z) must be maintained.

We can, as an initial implementation, distribute the entire problem set among the processors and assign a subset of the x, y and z rows to each processor. This approach has two key advantages: the spatial representation is consistent across processors and directional solvers. It also suffers from two key problems: it uses excessive amounts of memory and distributing the whole data space to several processors will be time consuming. However, as an initial hack at parallelizing the problem, this approach should be easy to implement.

A better approach, though somewhat more complex, is to distribute only those x, y and z rows required by the processors. This would reduce the time required to distribute the spatial data and reduce the amount of memory required by each processor. As an additional benefit, this representation would store each set of rows contiguously. We know from experience that cache misses have a significant impact on the performance of the code and storing the row elements contiguously will reduce the number of cache misses. Unfortunately, this representation also causes a miss match between the representations for the data within and between processors. Nevertheless, this approach might have a significant impact on the execution time of the code.

Validation

A sampling of 10 points were collected from the solution produced by the Fortran code, which we use a standard for correctness, and were compared to sample points produced by the C++ code. The maximum relative in the code was on the order of 10^{-4}. This result suggests that the C++ code is equivalent to the Fortran code.

Performance

We measured the execution time it took the sequential code to execute 200 iterations of the convergence loop. Next, we measured the execution time of the parallel version using between 1 and 8 processors and computed the speed up as the ratio of the parallel execution time to the sequential execution time. The speed up results are shown in Figure 2. NOTE: these are preliminary results and we expect the numbers to change, but not the trends.

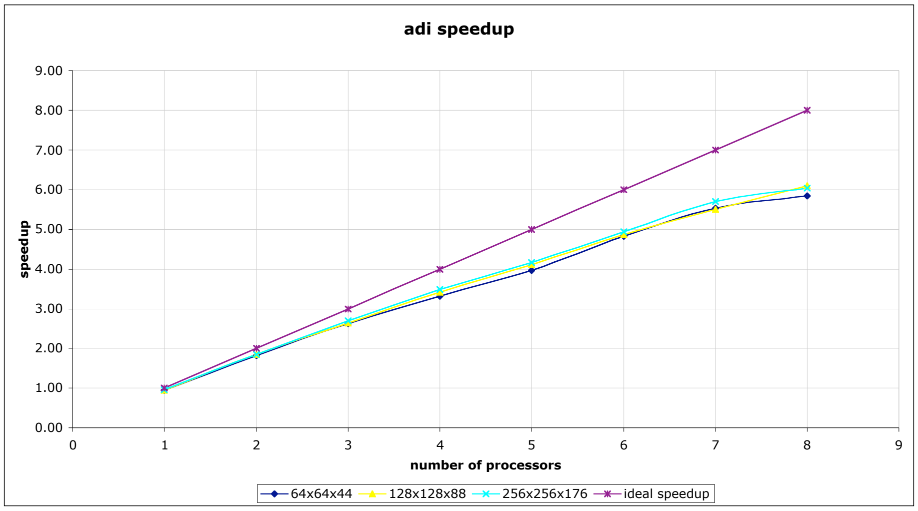

Figure 2 shows the speed up caused by the shared-memory processor.