Measurement of Improved Performance for Incompressible Navier-Stokes Example

P. Colella
D. F. Martin
N. D. Keen

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

April 30, 2003

In a previous report, the performance of the baseline AMR Navier-Stokes code was documented. A description of the problem was presented including the input and techniques used to measure wall-clock times along with serial and parallel performance results of the baseline code on the 'halem' machine at GSFC. We also measured the approximate peak memory usage for the two benchmark problems in serial.

In this report, we present timing measurements that show an improvement in performance after various code optimizations. There were errors discovered in the original baseline code, which resulted in changes to the code. These changes were propagated to the baseline code and new performance results were generated from this modified baseline code. We also provide a measurement for quantifying the differences between the solutions of the modified baseline code and the current code. The current code and the modified baseline code are provided with this report and are available on the web at http://davis.lbl.gov/NASA.

The summary of the outcome of our performance improvements is as follows. We reduced the wall clock time by factors ranging from 1.76 to 7.93 (tables 4 and 7), depending on the size of the problem and the number of processors. The largest speedups correspond to the largest problems. Also, these are fixed-size speedups, with all of the performance improvements coming from algorithm and code improvements, as opposed to increasing the number of processors. We were also able to reduce the memory required, but by a smaller amount, i.e., 5%-20%. However, memory appears not to be a constraining resource for this class of algorithms. Because of the speedups obtained here, and the fact that the performance improvements increased with the problem size and number of processors, we feel that we have met the requirements of Milestone F.

Baseline Code Changes and New Baseline Results

An error was discovered in the tagging criteria which resulted in incomplete tagging for refinement. When this error was corrected, more cells were tagged for refinement, which changed the solution (making it more accurate) and increased the work for the benchmark problems both in wall-clock time and peak memory usage. Also, an error was found in the reporting of certain summary values of the solution data in parallel runs; this error did not affect the solution, simply the value which was written incorrectly. We modified the baseline code to include the fixes for these errors.

We found that the code produced slightly different results when running in parallel depending on the load balance. To ensure that the code computes the exact same results regardless of load balance we modified the dot product function to gather and sort values before computing. We propagated this change to the modified baseline code. This change caused a slight performance degradation.

The only other modification to the baseline code concerns the diagnostic output printed to the screen. We added wall-clock and maximum memory usage results for three different stages of the code: setup, running all timesteps, and concluding. We added more precision

to certain values in the output. We added another diagnostic value which was the integral of kinetic energy (and the code required to compute it). These changes make it easier to compare performance results for the baseline and the current code as well as to quantify any changes in the solution.

A larger benchmark input was added with a size of 96x96x144 and a vorticity tagging factor of 0.00166666. Table 1 shows the three sizes of benchmark problems used including the respective vorticity tagging factor and total number of points updated for the run. In all of the benchmark runs, four timesteps are completed.

Problem size	Vorticity Tagging Factor	Number Points Updated for all Levels
32x32x48	0.0050	41517056
64x64x96	0.0025	226050048
96x96x144	0.00167	635633664

Table 1: Baseline Problem Data

Table 2 shows the re-computed performance results of the baseline code after modifications. We also include the maximum memory used over the entire run and all processors. This memory measurement is made using the system call **getrusage()** on the halem machine. This system call retrieves information about resources used by the current process such as the maximum resident set size. In the previous report, the memory reported was an estimate using our Linux workstations. We believe the memory measurements in these tables to be more accurate.

Prob size	Num Procs	Max Memory MB	AMR Run time (sec)	rate1 N/t	rate2 N/t*P
32x32x48	1	458	5425	7653	7653
32x32x48	4	156	1601	25932	6483
32x32x48	8	95	963	43112	5389
32x32x48	16	65	682	60875	3805
32x32x48	32	55	466	89092	2784
64x64x96	8	378	6206	36424	4553
64x64x96	16	205	3982	56768	3548
64x64x96	32	124	3029	74629	2332
64x64x96	64	86	2882	78435	1226
96x96x144	32	301	14438	44025	1376
96x96x144	64	171	13236	48023	750

Table 2: Baseline (re-computed) Results (with new tagging method)

The AMR run time is the measured wall-clock time to compute 4 timesteps and does

not include setup overhead or I/O at the end of the run. The `rate1` column is the total number of points updated divided by the wall-clock time and `rate2` is `rate1` divided by the number of processors used.

In table 3 we show unscaled speedup results for the only benchmark problem small enough to run on a single processor. All serial measurements are performed with code compiled without MPI.

Num Procs	Unscaled Speedup
4	3.4
8	5.6
16	8.0
32	11.6

Table 3: Baseline Unscaled Speedups for 32x32x48 benchmark

Summary of Performance Optimizations

Elliptic Solver

An expensive copy operation was simplified. Modifications were made to reduce the work performed when the residual becomes small. The single most expensive function in the elliptic solver is the Gauss-Siedel with Red-Black ordering (GSRB) smoothing which is implemented in fortran. We unrolled a loop over dimension which improved serial performance without modifying the solution.

Load Balance Algorithm

The load balance algorithm was improved by implementing a modified knapsack algorithm, replacing the round robin technique previously used to assign boxes to processors. We added an additional phase to the algorithm which exchanges equally sized boxes between processors to improve data locality. We also improved the efficiency of the balancing for problems in which the number of boxes to be distributed among processors was approximately equal to the number of processors.

Serial Optimizations

An expensive function in the coarse-fine interpolation function was identified and most of this function was converted into a simple fortran routine which greatly improved the serial performance. A consistency test was found to be expensive and was modified to

be performed only when the code is compiled with the debug option. Redundant work performed in the norm function was removed.

The Copier class does some initial set up to aid in certain types of data copying tasks. Improvements were made to the creation of Copier classes by implementing an alternative algorithm for determining box intersection details. This change vastly improved performance for larger problem sizes. We also improved the performance of creating Copiers for periodic problems.

The definition step for the class which enforces conservative coarse-fine matching conditions was sped up by removing work done in a critical inner loop.

The maximum box size (`maxboxsize`) is an input parameter which controls the maximum permissible box size and can be changed without altering the solution (except small changes due to accumulation of round-off errors). By changing the `maxboxsize` used in the calculations from 32 to 48 on all AMR levels except for the base level, we see a significant improvement in serial performance.

We reduced the peak memory usage by rearranging the location of data allocation in functions that were found to operate at or near the maximum memory of code execution.

Parallel Optimizations

A redundant parallel barrier was removed allowing for improved parallel performance especially as the number of processors increases. Several unnecessary parallel data exchanges were removed, which did not result in a noticeable improvement in performance.

We found a bug in the parallel data exchanges that was very slowly leaking memory. Fixing the bug provides slightly improved performance, reduces the peak memory usage, and does not alter the solution. This bug has only a small effect on the benchmark problems used in this work because all of the problems run for only four time steps.

Current Results after Improving Code Performance

The following tables show the performance results of the current code. Table 4 shows results with the `maxboxsize` equal to 32. The table includes a column that shows the improvement over the baseline code measurements. Table 5 shows the unscaled speedup for the 32x32x48 problem with `maxboxsize`=32.

When we increase the maximum permissible box size to 48, the grids slightly changed during the calculation and the total number of points updated increased slightly for the largest benchmark problem. Table 6 shows the number of points updated for each benchmark problem. Table 7 shows results with the `maxboxsize`=48. The table includes a column that shows the improvement over the baseline code measurements. Table 8 shows the unscaled speedup for the 32x32x48 problem with `maxboxsize`=48. Figure 1 plots the `rate2` computed measurements and provides a scaling comparison between baseline code performance and the current code performance (with `maxboxsize`=48).

Prob size	Num Procs	Max Memory MB	AMR Run time (sec)	improvement over baseline	rate1 N/t	rate2 N/t*P
32x32x48	1	432	3088	1.76	13445	13445
32x32x48	4	138	864	1.85	48052	12013
32x32x48	8	83	462	2.08	89864	11233
32x32x48	16	55	283	2.41	146703	9169
32x32x48	32	47	229	2.03	181297	5666
64x64x96	8	325	2824	2.20	80046	10006
64x64x96	16	176	1514	2.63	149307	9332
64x64x96	32	111	930	3.26	243065	7596
64x64x96	64	76	712	4.05	317486	4961
96x96x144	32	253	2722	5.30	233517	7297
96x96x144	64	159	1823	7.26	348675	5448

Table 4: Current Performance Results with `maxboxsize=32`

Num Procs	Unscaled Speedup
4	3.6
8	6.7
16	10.9
32	13.5

Table 5: Unscaled Parallel Speedup with `maxboxsize=32`

Perfect scaling would correspond to a horizontal line connecting all of the points. From this graph, it is clear that we have improved the scaling behavior of the method, as well as the absolute performance.

Prob size	Vort Tag Factor	N Points Updated
32x32x48	0.0050	41517056
64x64x96	0.0025	226050048
96x96x144	0.00167	635658240

Table 6: Problem Data with `maxboxsize=48`

All measurements were made on halem using the current batch system which does not guarantee dedicated use of the network between nodes. We believe that there is the potential for bandwidth competition between batch jobs which can cause inconsistent timing measurements. We found it difficult to repeat wall-clock timing results on halem,

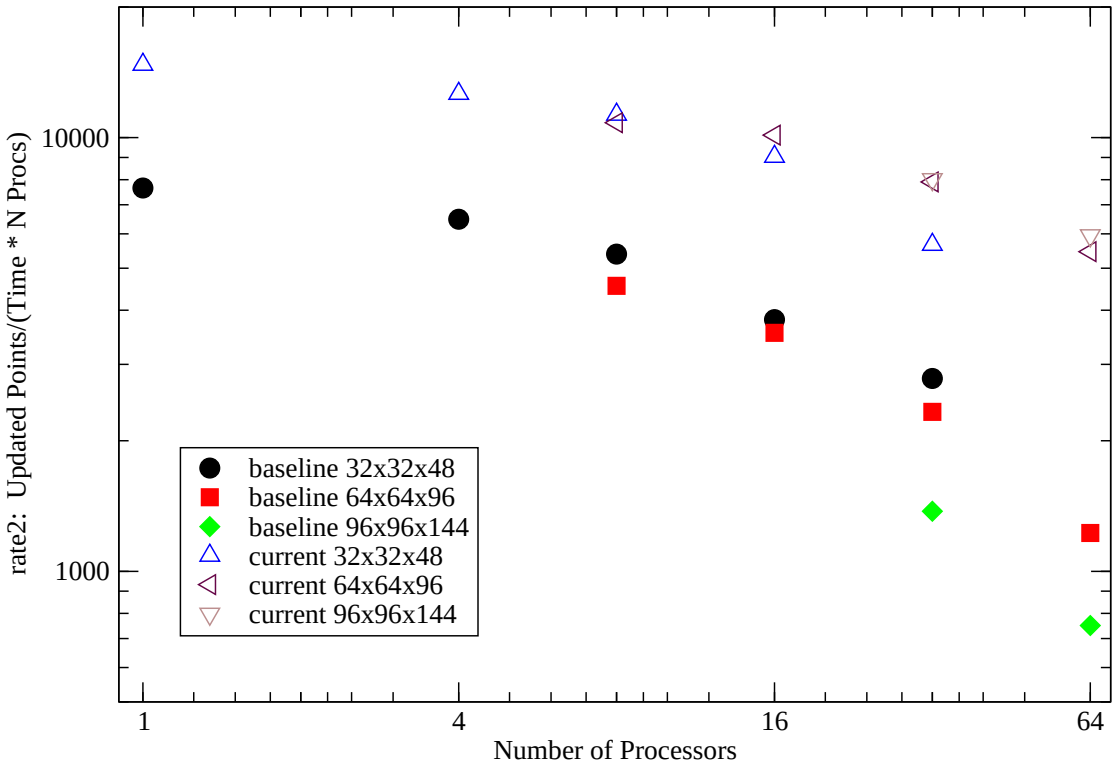

Figure 1: Scaling Results

Prob size	Num Procs	Max Memory MB	AMR Run time (sec)	improvement over baseline	rate1 N/t	rate2 N/t*P
32x32x48	1	421	2817	1.92	14738	14738
32x32x48	4	143	822	1.95	50507	12627
32x32x48	8	88	459	2.10	90451	11306
32x32x48	16	60	287	2.34	144659	9041
32x32x48	32	51	229	2.03	181297	5666
64x64x96	8	324	2610	2.34	86609	10826
64x64x96	16	176	1394	2.86	162159	10135
64x64x96	32	112	894	3.39	252852	7902
64x64x96	64	92	647	4.45	349382	5459
96x96x144	32	262	2479	5.82	256417	8013
96x96x144	64	166	1670	7.93	380634	5947

Table 7: Current Performance Results with `maxboxsize=48`

Num Procs	Unscaled Speedup
4	3.4
8	6.1
16	9.8
32	12.3

Table 8: Unscaled Parallel Speedup with `maxboxsize=48`

especially as the problem increased in size, total run time, and number of processors used. All of the timing measurements reported in this document are the minimum value of four separate measurements using the exact same code. Variations from this minimum over the four sets of runs were as high as 35%.

Quantifying Solution Differences from Performance Improvements

A measure of the changes in the solutions due to code modifications can be seen in Tables 9 and 10. Values in the table are relative differences between computed values.

In Tables 9 and 10, the maximum vorticity and integral of the kinetic energy are relative differences between the baseline benchmarks and the current results. The maximum divergence is scaled by the maximum vorticity from the baseline benchmark. We do this to make the number dimensionless by scaling it against a quantity which is not going to zero as the grids are refined.

Sum/Integral Quantity	32x32x48 serial	64x64x96 32 processors	96x96x144 64 processors
Max(Vorticity)	1.1435e-15	1.1773e-11	2.9133e-12
Integral(Kinetic Energy)	7.6126e-15	1.7124e-09	2.9375e-10
Max Divergence(u) *	2.2087e-14	3.3646e-07	4.6150e-10

Table 9: Relative Differences Between Baseline and Current Results with `maxboxsize=32`

Sum/Integral Quantity	32x32x48 serial	64x64x96 32 processors	96x96x144 64 processors
Max(Vorticity)	1.5248e-10	4.5946e-10	3.2275e-09
Integral(Kinetic Energy)	3.3393e-07	2.1788e-08	5.3447e-07
Max Divergence(u) *	1.7938e-07	3.8402e-07	2.0073e-06

Table 10: Relative Differences Between Baseline and Current Results with `maxboxsize=48`