LSG/RISC: A Version of the Hamburg LSG Model Optimized for RISC Workstations

David W. Pierce
Scripps Institution of Oceanography
January 31, 1994

1. Summary

The standard version of the Hamburg LSG model (developed by E. Maier-Reimer and U. Mikolajewicz at the Max Planck Institute for Meteorology) is optimized for performance on a vectorizing supercomputer. However, we do not use the model in such an environment; rather, we run it on RISC-based desktop workstations. A version of the LSG model with several key routines recoded and hand-tuned to take advantage of the RISC architecture typically runs 45-65% faster then the original version.

2. Timing Comparisons

The aspect of the LSG model which will be shown here is *net CPU time* per timestep. This is the amount of wall clock time it takes to compute one timestep of the LSG model, when the model is occasionally saving data and restart files (as it typically does in the runs we actually care about). This is the best indicator of how rapidly we would be able to get analyzable LSG results from the computer in question.

Methodology

For the tests, the models were configured to dump diagnostic information every six months, full model data every 25 years (with the data averaged over one year prior to dumping), and restart files every 50 years. The models were run for 100 years; in total, then, 1200 model timesteps were taken, with 200 diagnostic dumps, 4 full data dumps, and two restart files saved, for each run. This rate of dumping

model data is somewhat higher than is typical (which might be once every 50 or 100 years); since the increases in performance come primarily from reductions in the CPU time per timestep, while the time it takes to dump the model output was not changed, the results shown here are a conservative estimate of the speedup factor. Timing was done with the UNIX /bin/time command.

The command lines used to compile the programs were as follows:

- Tecate (HP Apollo 9000/735):

```
fort77 +U77 +O3 -K +e +es myflush.o lsg.F
```

  The routine `myflush` is a FORTRAN stub routine which adds the C flush routine to the FORTRAN code.

- Magik (DEC 3000/800 AXP):

```
f77 -O4 -align dcommons -assume noaccuracy_sensitive lsg.F
```

- Hank (IBM Power2/590 66 MHz):

```
xlf -qextend -qmaxmem=32000 -qdpc -O3 -Pv -Wp,-ea478 lsg.F
```

Results

Times shown in the table below are given as average time per timestep, in seconds. This lumps together the time to dump model output into the time per timestep, and so should be considered to be a representative time it would take to accomplish useful work on the system, rather than a measure of CPU performance.

The rows marked "WC" are the wall clock time per timestep; the rows marked "CPU" are the CPU time per timestep. On the DEC 3000/800, paging caused efficiency to drop and underestimated the benefit of using LSG/RISC, looking at wall clock times. With the increase of memory from 64 to 128 Megabytes this problem should disappear. On the IBM Power2/590, another job was being run part of the time, again underestimating the benefit of using the RISC version. Also, it is important to note that on the IBM Power2/590, an old version of the FORTRAN compiler was used; I estimate that the new version will reduce overall running times by 60-80%.

Table 1: Timings for the Standard LSG model and LSG/RISC. Values are average seconds per model timestep.

Machine		Standard LSG (sec/ts)	LSG/RISC (sec/ts)	Speedup
HP Apollo 9000/735	WC	7.19	4.43	62%
(Tecate)	CPU	7.17	4.41	63%
DEC 3000/800	WC	5.38	3.96	36%
(Magik)	CPU	5.07	3.24	56%
IBM Power2/590 66 MHz	WC	4.08	2.45	67%
(Hank)	CPU	3.18	2.19	45%

3. Backwards Compatibility

It is important to know that LSG/RISC will run *slower* on a vectorizing machine than the original code will. Because of this, selecting between the Standard and RISC versions of the LSG model is now a compile-time option; only one Makefile flag has to be set, either to `-DARCH_VECTOR` for the vectorizing architecture, or to `-DARCH_RISC` for the RISC-based architecture. This ensures that the optimum version of the code can be used in all circumstances.

4. Additional Features

The LSG/RISC code is distributed under the RCS Revision Control System. RCS is a freely-redistributable product, and is shipped already installed on DEC Alpha systems. Using RCS allows additional features to be included, such as: built-in support for the OSF/1, HP-UX, A/IX, and Ultrix operating systems; the option of using the full, linearized, or quadratic equation of state; and simple, run-time specification of the fresh water flux weighing factor, and model save, dump, and restart intervals. It will soon incorporate the K-Y Kim's Energy Balance Model, so that the choice between using the EBM, or using a specified surface temperature field to relax to, will simply be a matter of specifying a Makefile flag.

5. Types of Optimizations Performed

Figure 1 shows the timing profile of the Standard LSG code and LSG/RISC. The majority of the improvement is from chanes to the `GEOSTR` and `ADVECT` routines, with an important contribution from the density routine `RHOF1`.

Three basic types of optimization yielded almost all the performance gain. They will be described below, ranked according to the overall importance in running time

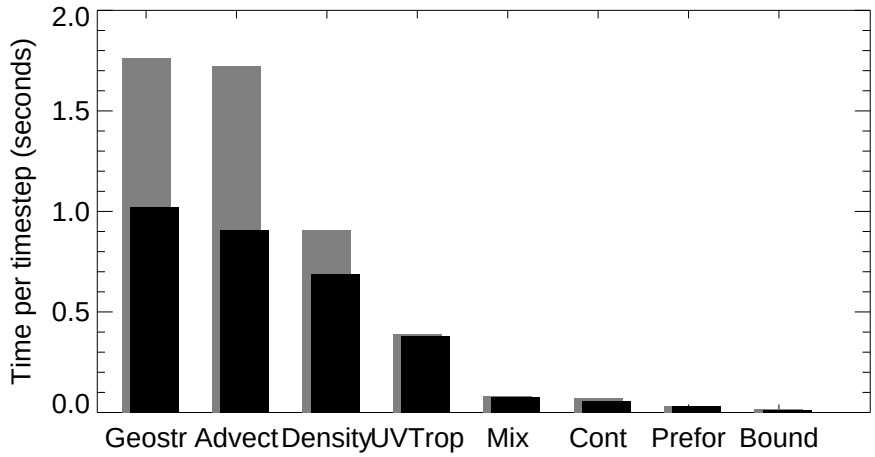

Figure 1: Time taken in key routines of the LSG code. Shaded bars are in the Standard version of the LSG; solid bars are for LSG/RISC. Measurements were made on a DEC 3000/800 AXP (Magik).

improvement.

1) *Removing the "nearest neighbor" vectors.* On vector machines, it is a classic technique to improve performance by constructing seperate auxilliary arrays for the left and right neighbors of a cell, for use when performing finite-difference calculations. This yields a double performance hit on RISC architectures, because a) there is no speed advantage to vectorized code, so the cycles taken to copy the data into the auxilliary arrays are wasted; b) the arrays are so large in the LSG model, RISC architectures already have trouble fitting data into the cache. Having more arrays with redundant, nearest-cell data in them decreases the efficiency of cache useage, slowing performance. This is not a consideration on vector supercomputers, which have no cache. This optimization accounts for most of the performance gain in routines `GEOSTR` and `ADVECT`.

2) *Inserting* `IF` *clauses into* `DO` *loops.* It is a maxim of programming for vector supercomputers to avoid putting `IF` clauses into `DO` loops, as they tend to inhibit vectorization of the code. This restriction is far less severe in RISC architectures; in fact, if an `IF` clause can keep the program from performing a number of floating point operations, it can be worth putting it into the loop. The performance gains in the equation-of-state routine, `RHOF1` (which appear under "Density" in Figure 1),

come from adding an `IF` clause to avoid calculating the water density at land points.

3) *Moving Invariant Code out of Loops*. The advection and baroclinic velocity calculations are iterative in nature. There are some quantities which do not change over the number of iterations, yet are calculated once per iteration. Moving these calculations to outside the iterative loop yields modest performance increases in routines `GEOSTR` and `ADVECT`. This is not strictly a RISC optimzation; the vector code would benefit from having this done also. A sufficiently clever compiler would theoretically be able to make this optimization automatically; however, the calculations involve a network of `EQUIVALANCE`'d matrices inside commonblocks, and FORTRAN-77 does not rule out the possibility that arrays manipulated inside the iteration loop are logically the same as the invariant quantities. Because of this, virtually all compilers will be unable to make this optimization on their own.