Domain partitioning, Parallel performance and Visualisation in the ANUGA Hydrodynamic Software

Jack Kelly

April 6, 2006

1 Project Description

During my scholarship, my efforts were directed into three areas: improving the partitioning scheme for the domain, examination of the parallel version of the ANUGA software and improving the realtime visualisation component of the system.

2 Partitioning

2.1 Overview

The original partitioning scheme for distributing the domain used a simple co-ordinate based dissection. This method has been replaced by pymetis, a python wrapper around the METIS (`http://www-users.cs.umn.edu/~karypis/metis/`) partitioning library. Pymetis is currently incomplete and at the time of writing, only provided the partMeshNodal function. A new pmesh_divide function, pmesh_divide_metis, uses metis to divide the domain for parallel computation.

METIS was chosen as the partitioner based on the results in the paper: "A fast and high quality multilevel scheme for partitioning irregular graphs". (`http://www-users.cs.umn.edu/~karypis/publications/Papers/PDF/mlevel\_serial.pdf`)

2.2 Compilation

Building the Pymetis wrapper is done using Make, but there is variation depening on the host system type. Under most types of Linux, simply running `make` will work. Under x86_64 versions of Linux, the command is `make COPTIONS="-fPIC"`. Finally, under windows, the command is `make for_win32`. For a sanity check, a simple PyUnit test is provided, called test_metis.py .

2.3 Test

Running an 8-way test of ga/inundation/parallel/run_parallel_sw_merimbula_metis.py displays the following results for load distribution:

CPU	0	1	2	3	4	5	6	7
Elements	1292	1647	1505	1623	1381	1514	1605	1388
%	10.81%	13.78%	12.59%	13.58%	11.55%	12.66%	13.43%	11.61%

Total Elements: 11955, Total %: 100.01%

Which suggest that Pymetis is performing as expected.

3 Performance Analysis

Examination of the parallel performance of the ANUGA system has been done using the APAC's Linux Cluster (LC). Running MPI jobs with python through the batch system requires specific settings as follows:

1. Comment out all module load statements from .bashrc .cshrc .profile and .login
2. In the script for submission through PBS, add the lines:

   ```
   module load python
   module load lam
   ```

3. (Bash specific; translate if using csh) add export LD_ASSUME_KERNEL=2.4.1 and export PYTHONPATH=(relevant directory)
4. Instead of the call to lrun, use:

   ```
   lamboot
   mpirun -np <number of processors> -x LD_LIBRARY_PATH,PYTHONPATH
   ,LD_ASSUME_KERNEL <python filename>
   lamhalt
   ```

   The line beginning with 'mpirun' and ending with '<python filename>' should be on a single line.
5. Ensure the you have `-l other=mpi` in the PBS directives.
6. Ensure that pypar 1.9.2 is accessible. At the time of writing, it was in the SVN repository under the directory ga/inundation/pypar_dist . Move it to the directory pypar, so that this pypar directory is accessible from the PYTHONPATH.

The section of code that was profiled was the evolve function, which handles progression of the simulation in time.

The functions that have been examined were the ones that took the largest amount of time to execute. Efficiency values were calculated as follows:

$$E = \frac{T_1}{n \times T_{n,1}} \times 100\%$$

Where T_1 is the time for the single processor case, n is the number of processors and $T_{n,1}$ is the first processor of the n cpu run. Results were generated using an 8 processor test run, compared against the 1 processor test run. The test script used was run_parallel_sw_merimbula_metis.py script in svn/ga/inundation/parallel.

Function name	Parallel efficiency
evolve(shallow_water.py)	79%
update_boundary(domain.py)	52%
distribute_to_vertices_and_edges(shallow_water.py)	140.6%
evaluate(shallow_water.py)	57.8%
compute_fluxes(shallow_water.py)	192.2%

While evolve was the main function of interest in the profiling efforts, the other functions listed above displayed parallel efficiencies that warranted further attantion.

3.1 update_boundary

Whenever the domain is split for distribution, an additional layer of triangles is added along the divide. These 'ghost' triangles hold read only data from another processor's section of the domain. The use of a ghost layer is common practise and is designed to reduce the communication frequency. Some edges of these ghost triangles will contain boundary objects which get updated during the call to update_boundary. These are redundant calculations as information from the update will be overwritten during the next communication call. Increasing the number of processors increases the ratio of such ghost triangles and the amount of time wasted updating the boundaries of the ghost triangles. The low parallel efficiency of this function has resulted in efforts towards optimisation, by tagging ghost nodes so they are never updated by update_boundary calls.

3.2 distribute_to_vertices_and_edges

Initially, the unusual speedup was thought to be a load balance issue, but examination of the domain decomposition demonstrated that this was not the case. All processors were showing this unusual speedup. The profiler was then suspected, due to the fact that C code cannot be properly profiled by the python profiler. The speedup was confirmed by timing using the hotshot profiler and calls to time.time().

The main function called by distribute_to_vertices_and_edges is balance_deep_and_shallow_c and was profiled in the same way, showing the same speedup.

The call to balance_deep_and_shallow_c was replaced by a call to the equivalent python function, and timed using calls to time.time(). The total CPU time elapsed increased slightly as processors increased, which is expected. The algorithm part of the C version of balance_deep_and_shallow was then timed using calls to gettimeofday(), with the following results:

Number of processors	Total CPU time
1	0.482695999788
2	0.409433000023
4	0.4197840002601
8	0.4275599997492

For two processors and above, the arrays used by balance_deep_and_shallow fit into the L2 cache of an LC compute node. The cache effect is therefore responsible for the speed increase from one to two processors.

3.2.1 h_limiter

h_limiter is a function called by balance_deep_and_shallow. Timing the internals of the C version of balance_deep_and_shallow results in the call to h_limiter not being correctly timed. Timing h_limiter from the python side resulted in unusual (>100%) parallel efficiency, and timing the algorithm part of the C version of h_limiter (i.e. not boilerplate code required for the Python-C interface such as calls to PyArg_ParseTuple) displayed expected slowdowns. Therefore, it is presumed that this unusual speedup is due to how python passes information to C functions.

3.3 evaluate

Evaluate is an overridable method for different boundary classes (transmissive, reflective, etc). It contains relatively simple code, which has limited scope for optimisation. The low parallel efficiency of evaluate is due to it being called from update_boundary and the number of boundary objects is increasing as the number of CPUs increases. Therefore, the parallel efficiency here should be improved when update_boundary is corrected.

3.4 compute_fluxes

Like balance_deep_and_shallow and h_limiter, compute_fluxes calls a C function. Therefore, the unusual parallel efficiency has the same cause as balance_deep_and_shallow and h_limiter - presumed to be the way python calls code in C extensions.

3.5 Conclusions

The unusual speedup exhibited by compute_fluxes, balance_deep_and_shallow and h_limiter has been traced to how the Python runtime passes information to C routines. In future work we plan to investigate this further to verify our conclusions.

4 Visualisation

The real-time visualisation component of the ANUGA system originally used VPython (`http://www.vpython.org`). This implementation is being superceded by a VTK-based (`http://www.vtk.org/`) visualiser which is faster, capable of handling larger data sets and more configurable. At the time of writing, the Domain class that uses the VTK visualiser resides in ga/inundation/pyvolution/shallow_water_vtk.py. It has been tested to work under Linux and Windows.

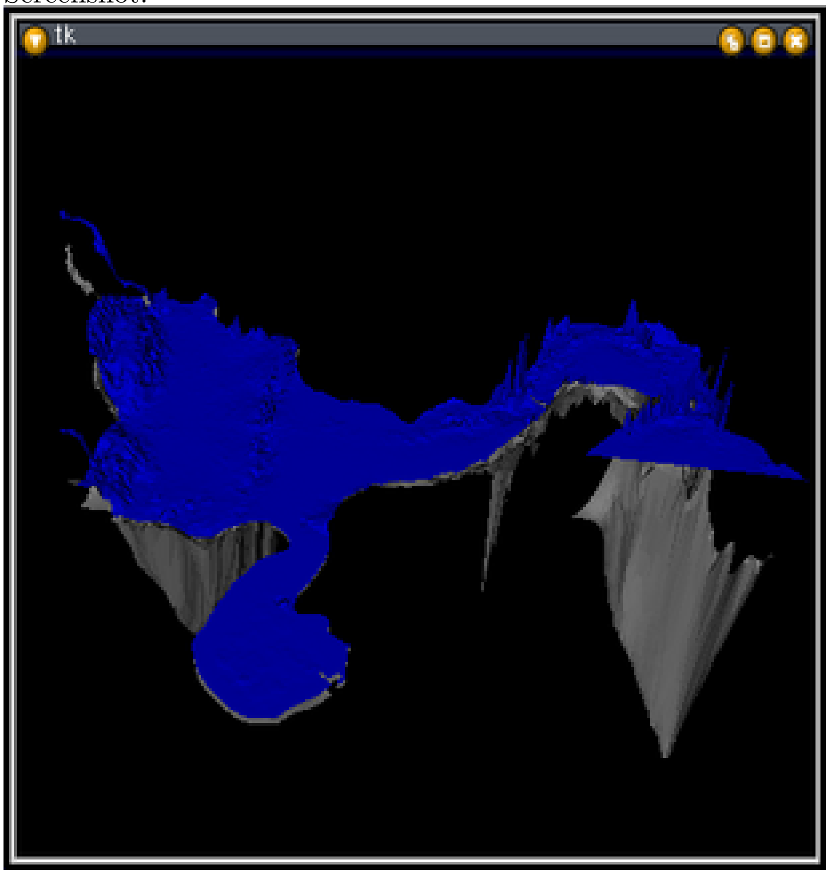
The visualiser class itself resides in ga/inundation/pyvolution/vtk_realtime_visualiser.py.

Customisation options for the visualiser are members of the visualiser class and their functions are as follows:

- setup: Dictionary mapping quantity name $\rightarrow$ boolean. if setup[q] is true, draw the quantity when the visualiser starts.

- updating: Dictionary mapping quantity name → boolean. if updating[q] is true, update the rendering of this quantity each timestep.
- qcolor: Dictionary mapping quantity name → (float, float, float). If the name of a quantity is found in qcolor, the colour specified (in (r, g, b) from 0.0 to 1.0) is used for display of that quantity instead of (0.5, 0.5, 0.5) (the default)
- scale_z: Dictionary mapping quantity name → float. Override the z scaling of this quantity with the float specified.
- default_scale_z: float. Multiply all z coordinates by this amount unless an overriding value exists in scale_z.

Screenshot:



Unlike the old VPython visualiser, the behaviour of the VTK interaction classes completely tie up the thread that they are running in. The solution was to use the TK interactor, and build a TKinter gui around the render window.

Even using TK around the VTK interactor, this still ties up the thread for TK's main loop. This was worked around by making the visualiser use its own thread for rendering. This worked fine under Linux, but under Windows the visualiser is 'starved' by the CPU-bound thread that contains the call to evolve(). To rectify this, the visualiser and the main thread are explicitly synchronised: The visualiser waits on a condition variable to signify that the evolve thread has finished computation for its yieldstep. The visualiser then updates the VTK data structures, while the evolving thread waits on another condition variable for the visualiser to finish. The visualiser then informs the evolve thread that it is finished and returns to waiting. While this does reduce the potential for concurrency, it still maintains an acceptable level of interactivity.

Currently, the rendering method is only suitable for quantities for which the concept of height makes sense (e.g. stage and elevation in the Shallow Water domain). Adding support for alternate rendering methods would be a welcome enhancement to this visualiser.