Homework 2
CS 5220

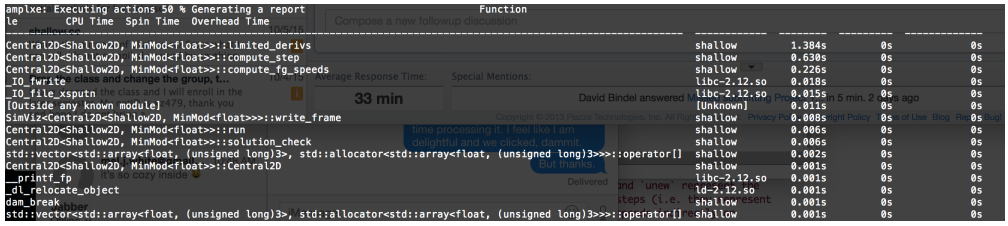
Lara Backer, Xiang Long, Saul Toscano

October 19, 2015

Initial Profiling - Vtune

An initial profiling of the shallow wave code on the Totient cluster was done using Vtune, to identify which regions of the code took the longest time to run. This was done in order to target parallelization of the functions that took the longest time to complete.

Figure 1 displays the Vtune assessment of the overall program. The top three most time consuming functions are limited_derivs, compute_step, and compute_fg_speeds.



Function	Module	CPU Time	Spin Time	Overhead Time
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	1.384s	0s	0s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	0.630s	0s	0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	0.226s	0s	0s
_IO_fwrite	libc-2.12.so	0.018s	0s	0s
_IO_file_xsputn	libc-2.12.so	0.015s	0s	0s
[Outside any known module]	[Unknown]	0.011s	0s	0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame	shallow	0.008s	0s	0s
Central2D<Shallow2D, MinMod<float>>::run	shallow	0.006s	0s	0s
Central2D<Shallow2D, MinMod<float>>::solution_check	shallow	0.006s	0s	0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]	shallow	0.002s	0s	0s
Central2D<Shallow2D, MinMod<float>>::Central2D	shallow	0.001s	0s	0s
__printf_fp	libc-2.12.so	0.001s	0s	0s
_dl_relocate_object	ld-2.12.so	0.001s	0s	0s
dam_break	shallow	0.001s	0s	0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]	shallow	0.001s	0s	0s

Figure 1: Most time consuming functions for shallow wave code on Totient cluster

The function limited_derivs is used to limit the estimation of the derivative of the fluxes by calling the limdiff function. Compute_fg_speeds is used to evaluate the fluxes at the cell centers. The compute_step function is the seciton of the code that evaluates both the predictor and corrector calculations for the timestep. To see the time spent in each step using Vtune, the specific functions must be referenced. While this is not overly useful for the limited_derivs function, which only performs the two limdiff calls, the in-depth timings for the other two most expensive functions are shown in Figures 2 and 3.

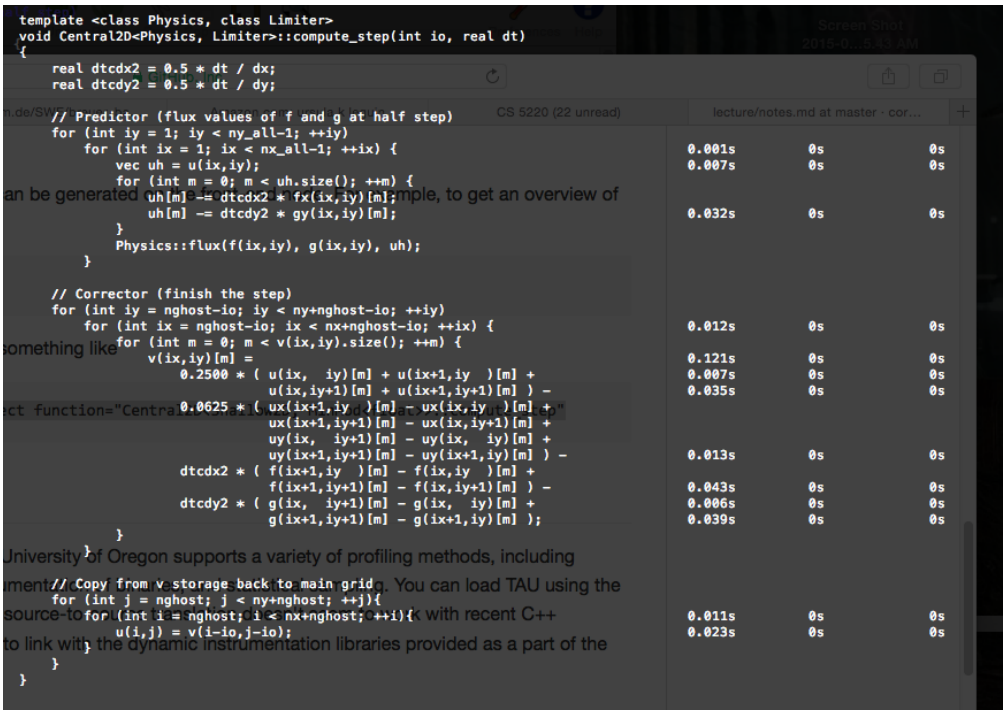
Vectorization

One method of speeding up the code is to apply vectorization, applying operations to arrays instead of array sub elements, reducing the number of required computations. To apply vectorization to our code for instance, we grouped the fluxes into single arrays, specifying pointers to the locations of each flux, and indicating the strides required for each component.

The .optreport file output shows the sections that could be vectorized to improve efficiency.

openMP Parallelization

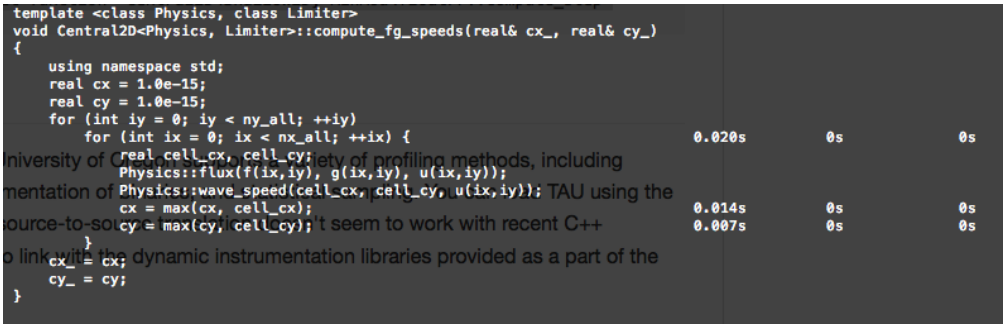
OpenMP can be used to further improve performance by multithreading, where each thread executes a section of the code independently. In C++, OpenMP uses #pragmas to signify the parallelization of code. The pragma omp parallel, for instance, is used to fork threads to carry out the enclosed work. Additional pragmas can be used to specify sections of work, or splitting up loop iterations.



```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_step(int io, real dt)
{
    real dtcdx2 = 0.5 * dt / dx;
    real dtcdy2 = 0.5 * dt / dy;

    // Predictor (flux values of f and g at half step)
    for (int iy = 1; iy < ny_all-1; ++iy)
        for (int ix = 1; ix < nx_all-1; ++ix) {
            vec uh = u(ix,iy);
            for (int m = 0; m < uh.size(); ++m) {
                uh[m] -= dtcdx2 * fx(ix,iy)[m];
                uh[m] -= dtcdy2 * gy(ix,iy)[m];
            }
            Physics::flux(f(ix,iy), g(ix,iy), uh);
        }

    // Corrector (finish the step)
    for (int iy = nghost-io; iy < ny+nghost-io; ++iy)
        for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
            for (int m = 0; m < v(ix,iy).size(); ++m) {
                v(ix,iy)[m] =
                    0.2500 * ( u(ix,  iy)[m] + u(ix+1,iy  )[m] +
                               u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -
                    0.0625 * ( ux(ix+1,iy  )[m] - ux(ix,iy  )[m] +
                               ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
                               uy(ix,  iy+1)[m] - uy(ix,  iy)[m] +
                               uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m] ) -
                    dtcdx2 * ( f(ix+1,iy  )[m] - f(ix,iy  )[m] +
                               f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -
                    dtcdy2 * ( g(ix,  iy+1)[m] - g(ix,  iy)[m] +
                               g(ix+1,iy+1)[m] - g(ix+1,iy)[m] );
            }
        }

    // Copy from v storage back to main grid
    for (int j = nghost; j < ny+nghost; ++j)
        for (int i = nghost; i < nx+nghost; ++i)
            u(i,j) = v(i-io,j-io);
    }
}
```

0.001s	0s	0s
0.007s	0s	0s
0.032s	0s	0s
0.012s	0s	0s
0.121s	0s	0s
0.007s	0s	0s
0.035s	0s	0s
0.013s	0s	0s
0.043s	0s	0s
0.006s	0s	0s
0.039s	0s	0s
0.011s	0s	0s
0.023s	0s	0s

Figure 2: compute step function timings



```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_fg_speeds(real& cx_, real& cy_)
{
    using namespace std;
    real cx = 1.0e-15;
    real cy = 1.0e-15;
    for (int iy = 0; iy < ny_all; ++iy)
        for (int ix = 0; ix < nx_all; ++ix) {
            real cell_cx, cell_cy;
            Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));
            Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));
            cx = max(cx, cell_cx);
            cy = max(cy, cell_cy);
        }
    cx_ = cx;
    cy_ = cy;
}
```

0.020s	0s	0s
0.014s	0s	0s
0.007s	0s	0s

Figure 3: fgspeeds function timings

Domain Decomposition

The method best suited to parallelize the shallow water problem is domain decomposition. For this method, the overall domain is split and solved for on separate processors. Communications between processors are needed to solve for 'ghost cells' that overlap neighboring domains. However, communications can overtake the domain computations in cost, depending on the number of processors and communications compared to the domain size that each processor has to solve, so scaling trials must be employed to determine the optimal problem size per processor.