

Homework 2

CS 5220

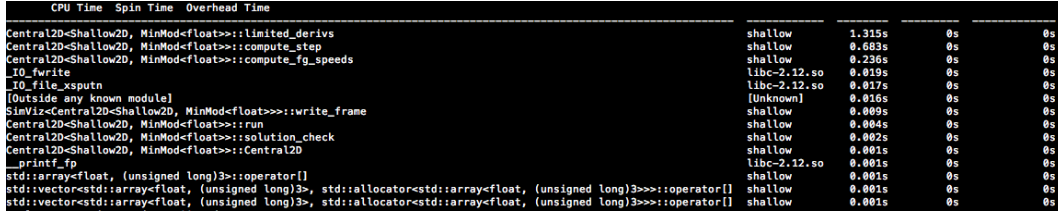
Lara Backer, Xiang Long, Saul Toscano

October 20, 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.



CPU Time	Spin Time	Overhead Time			
Central2D<Shallow2D, MinMod<float>>::limited_derivs			shallow	1.315s	0s
Central2D<Shallow2D, MinMod<float>>::compute_step			shallow	0.683s	0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds			shallow	0.236s	0s
LIO_fwrite			libc-2.12.so	0.019s	0s
LIO_fwrite			libc-2.12.so	0.017s	0s
LIO_file_xspn			(Unknown)	0.016s	0s
[Outside any known module]					
SimViz<Central2D<Shallow2D, MinMod<float>>::write_frame			shallow	0.009s	0s
Central2D<Shallow2D, MinMod<float>>::run			shallow	0.004s	0s
Central2D<Shallow2D, MinMod<float>>::solution_check			shallow	0.002s	0s
Central2D<Shallow2D, MinMod<float>>::Central2D			shallow	0.001s	0s
_printf_fp			libc-2.12.so	0.001s	0s
std::array<float, (unsigned long)3>::operator[]			shallow	0.001s	0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>::operator[]			shallow	0.001s	0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>::operator[]			shallow	0.001s	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.

There could be some performance gains in tuning the most expensive functions, however we chose to put most of our efforts into vectorization and parallelization as it is expected the gains here would be much more significant.

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.

We drew ideas from the vectorization code produced by Prof. Bindel. In our code, the ipo_out.optreport file output after compilation shows the sections that have be vectorized to improve efficiency. Our vectorization still uses the C++ standard library vector and iterator structures, which seem to have significant performance overheads. We are currently working on a version that uses pointers to reduce overheads in manipulating the data structure.

```

326     template <class Physics, class Limiter>
327     void Central2D<Physics, Limiter>::compute_step(int io, real dt)
328     {
329         real dtcdx2 = 0.5 * dt / dx;
330         real dtcdy2 = 0.5 * dt / dy;
331
332         // Predictor (flux values of f and g at half step)
333         for (int iy = 1; iy < ny_all-1; ++iy)
334             for (int ix = 1; ix < nx_all-1; ++ix) {
335                 vec uh = u(ix,iy);
336                 for (int m = 0; m < uh.size(); ++m) {
337                     uh[m] -= dtcdx2 * fx(ix,iy)[m];
338                     uh[m] -= dtcdy2 * gy(ix,iy)[m];
339                 }
340                 Physics::flux(f(ix,iy), g(ix,iy), uh);
341             }
342
343         // Corrector (finish the step)
344         for (int iy = nghost-io; iy < ny+nghost-io; ++iy)
345             for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
346                 for (int m = 0; m < v(ix,iy).size(); ++m) {
347                     v(ix,iy)[m] =
348                         0.2500 * ( u(ix, iy)[m] + u(ix+1,iy ) [m] +
349                             u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -
350                         0.0625 * ( ux(ix+1,iy ) [m] - ux(ix,iy ) [m] +
351                             ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
352                             uy(ix, iy+1)[m] - uy(ix, iy) [m] +
353                             uy(ix+1,iy+1)[m] - uy(ix+1,iy) [m] ) -
354                         dtcdx2 * ( f(ix+1,iy ) [m] - f(ix,iy ) [m] +
355                             f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -
356                         dtcdy2 * ( g(ix, iy+1)[m] - g(ix, iy) [m] +
357                             g(ix+1,iy+1)[m] - g(ix+1,iy) [m] );
358                 }
359             }
360
361         // Copy from v storage back to main grid
362         for (int j = nghost; j < ny+nghost; ++j){
363             for (int i = nghost; i < nx+nghost; ++i){
364                 u(i,j) = v(i-io,j-io);
365             }
366         }
367     }

```

Figure 2: compute step function timings

```

258     * bound on the CFL number).
259     */
260
261     template <class Physics, class Limiter>
262     void Central2D<Physics, Limiter>::compute_fg_speeds(real& cx_, real& cy_)
263     {
264         using namespace std;
265         real cx = 1.0e-15;
266         real cy = 1.0e-15;
267         for (int iy = 0; iy < ny_all; ++iy)
268             for (int ix = 0; ix < nx_all; ++ix) {
269                 real cell_cx, cell_cy;
270                 Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));
271                 Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));
272                 cx = max(cx, cell_cx);
273                 cy = max(cy, cell_cy);
274             }
275         cx_ = cx;
276         cy_ = cy;
277     }

```

Figure 3: fgspreads function timings

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.

The vectorization report was useful in identifying which loops should be targeted for parallelization. Loops that have data dependencies between iterations cannot be unrolled to work in parallel, so it would be futile to assign iterations to different threads. It was found that only the time-stepping loop in `central2d` was suitable for parallelization.

In order to test the relative performance of the original, vectorized and OpenMP-parallelized implementations, we performed a strong scaling study where the size n of the side of the field is increased and the simulation times of each implementation is recorded. It was found that our vectorized code ran slower than the original implementation, and that with OpenMP it performed slower still. This could be due to overheads in our implementation, which we are aiming to fix for the final report. In particular the slowdown in OpenMP could be due to spawning too many threads to work on short computational tasks, where overheads in thread management will dominate.

We also performed a weak scaling study where we forced OpenMP to use between 1 to 12 threads, and the run times of these configurations were tested against the original `unlimitedSetting` where it was up to OpenMP

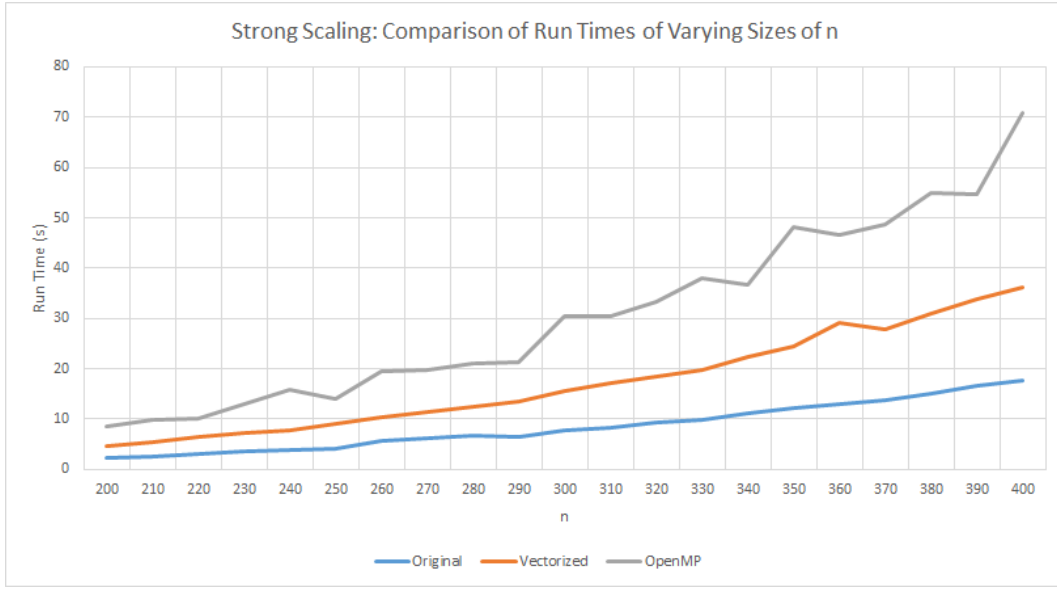


Figure 4: Strong scaling study

how many threads to spawn. We found that with fewer threads performance was actually better, perhaps due to eliminating overheads in managing many threads. However there is clearly much more room for improvement in both the vectorization and parallelization performance.

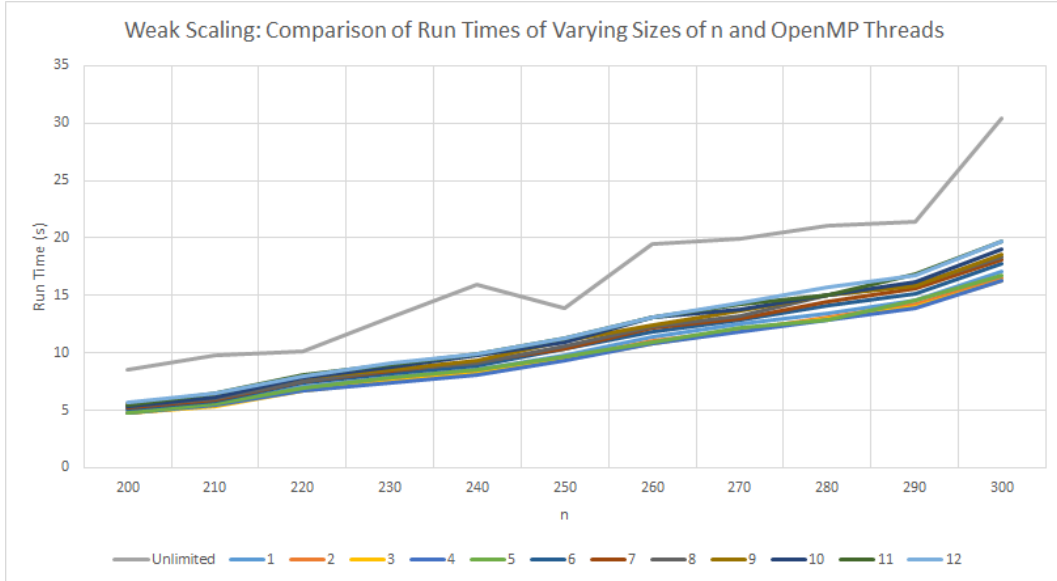


Figure 5: Weak scaling study

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. We are working on investigating this area for our final report.