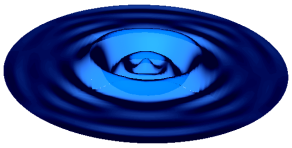
Spectral Element Libraries in Fortran (SELF)

Shallow Water Equations



PCSRI Projects

Joseph Schoonover

Contents

Abstract

1 Background and Motivation

1.1 The Conservative Shallow Water Equations

The inviscid shallow water equations are a system of coupled partial differential equations that is hyperbolic. The Nodal Discontinuous Galerkin Spectral Element Method (NDGSEM) is well suited for hyperbolic systems. It is highly accurate through its use of arbitrarily high order piecewise polynomial representation of the approximated solution and geometry. It is well known for exhibiting minimal numerical dispersion and dissipation errors for linear problems. It additionally allows for a flexible treatment of the geometry through the use of a structured or unstructured arrangement of *spectral elements* that can represent boundaries through high order polynomial interpolants. The discretization yields dense and local matrix-vector multiplies for computing derivatives and only requires nearest neighbor communication to calculate exchanges between elements. This last property makes the NDGSEM trivial to parallelize. For these reasons, the NDGSEM is applied to the shallow water equation systems (1.1), (1.2), and (1.3) to demonstrate its accuracy and scalability for a system of equations that hold many of the important dynamics in geophysics.

1.2 Flavors of the Shallow Water Equations

The nonlinear shallow water equations can be written in either "conservative" (Eqs. 1.1) or "skew-symmetric" (Eqs. 1.2) form :

$$U_t + \nabla \cdot \left(\frac{\vec{U}U}{h+\eta} + g\left(h + \frac{\eta}{2}\right)\eta\hat{x} \right) - fV = g\eta h_x + q_U \tag{1.1a}$$

$$V_t + \nabla \cdot \left(\frac{\vec{U}V}{h+\eta} + g\left(h + \frac{\eta}{2}\right)\eta\hat{y} \right) + fU = g\eta h_y + q_V \tag{1.1b}$$

$$\eta_t + \nabla \cdot \vec{U} = q_\eta \tag{1.1c}$$

$$u_t + \nabla \cdot ((e+p)\hat{x}) - (f+\zeta)v = q_u \tag{1.2a}$$

$$v_t + \nabla \cdot ((e+p)\hat{y}) + (f+\zeta)u = q_v \tag{1.2b}$$

$$p_t + \nabla \cdot ((gh+p)\vec{u}) = q_p \tag{1.2c}$$

For either system, and throughout the rest of this documentation the symbols have the following meaning :

x, y : Cartesian coordinate positions ($Length$)

t : Time

U, V : The x, y components of the the fluid *transport* ($\frac{Length^2}{Time}$)

H : The total fluid thickness, bathymetric plus free surface ($Length$)

h : Resting fluid depth, the fluid bathymetry ($Length$)

g : Local acceleration of gravity ($\frac{Length}{Time^2}$)

f : Coriolis parameter ($Time^{-1}$)

u, v : The x, y components of the *depth average* fluid velocity ($\frac{Length}{Time}$)

e : The kinetic energy of the depth averaged velocity field ($\frac{Length^2}{Time^2}$)

p : Barotropic pressure, the product of gravitational acceleration and free surface anomaly ($\frac{Length^2}{Time^2}$)

ζ : The relative vorticity of the depth averaged velocity field ($Time^{-1}$)

q_ϕ : Any additional non-conservative source for the variable ϕ ($\frac{\phi}{Time}$)

The linear shallow water equations (1.3) are given as

$$u_t + \nabla \cdot (p\hat{x}) - fv = q_u \tag{1.3a}$$

$$v_t + \nabla \cdot (p\hat{y}) + fu = q_v \tag{1.3b}$$

$$p_t + \nabla \cdot (gh\vec{u}) = q_p \tag{1.3c}$$

Any flavor of the shallow water equations arise from a conservation law, and as such, (1.1), (1.2), or (1.3) can be written in the compact form

$$\vec{s} + \nabla \cdot \vec{f} = \vec{q}. \tag{1.4}$$

The solution vector is represented by $\vec{s}$, $\vec{f}$ is a conservative flux tensor, and $\vec{q}$ is a vector of non-conservative source terms. The derivation of the discrete equations is presented through a discretization of (1.4).

In the applications discussed in this document, we are primarily focused on the Conservative Form of the Shallow Water equations. In all of the benchmarks, we use the `dipole` example with a fixed unstructured mesh.

1.3 Nodal Discontinuous Galerkin Spatial Discretization

The NDGSEM discretizes (1.4) in its weak form. The physical space over which (1.4) is solved is denoted by Ω. Solutions to (1.4) are real functions that are piecewise continuous on Ω. This space of functions is denoted $\mathbb{C}_0(\Omega)$, and is defined mathematically

$$\mathbb{C}_0(\Omega) \equiv \left\{ \phi(\vec{x}) \in \mathbb{R} : \int_\Omega \phi^2 d\Omega < \infty \right\}, \tag{1.5}$$

" the set of all functions ϕ that are square integrable over the physical space Ω ".

$\mathbb{C}_0(\Omega)$ has an inner product associated with it, defined by

$$(u, v) = \int_\Omega uv \, d\Omega. \tag{1.6}$$

From the definition of the inner product, the *induced norm* of a function that is in $\mathbb{C}_0(\Omega)$ is defined as

$$||u|| = \sqrt{(u, u)}. \tag{1.7}$$

Similar to vectors, the inner-product (1.6) is a measure of the "common information" between u and v. Two functions are said to be *orthogonal* if

$$(u, v) = 0. \tag{1.8}$$

Any piecewise continuous function can be represented as a linear combination of a *basis functions* that span $\mathbb{C}_0(\Omega)$. The weak form of (1.4) is obtained by taking the inner product of (1.4) with each basis function of $\mathbb{C}_0(\Omega)$,

$$(\vec{s}_t, \phi) + (\nabla \cdot \vec{f}, \phi) = (\vec{s}, \phi), \quad \forall \phi \in \mathbb{C}_0(\Omega) \tag{1.9}$$

In Eqs. (1.1), (1.2), and (1.3), the conservative flux depends on the solution. In (1.9), the divergence of the flux requires that the solution is at least once differentiable. This differentiability constraint is relaxed in the weak form by performing integration by parts, which moves the differentiability criteria onto the basis function (ϕ). Performing integrations by parts gives

$$(\vec{s}_t, \phi) + \oint_{\partial\Omega} \phi \vec{f} \cdot \hat{n} \, dS - (\vec{f}, \nabla\phi) = (\vec{s}, \phi), \quad \forall \phi \in \mathbb{C}_0(\Omega) \tag{1.10}$$

To develop a discrete set of equations, all of $\mathbb{C}_0(\Omega)$ is approximated by piecewise continuous polynomials of degree N, a space we denote $\mathbb{P}^N$. The solution, conservative flux, and the non-conservative source term are approximated by a linear combination of a suitable basis for $\mathbb{P}^N$. Since the solution is piecewise continuous, the domain Ω, is decomposed into a set of non-overlapping elements Ω_l. The integration is broken into integration over each of the elements,

$$\sum_{\kappa=0}^{nEl} \left\{ \int_{\Omega_\kappa} (\vec{s}_t + \vec{q})\phi_\kappa \, d\Omega_\kappa - \int_{\Omega_l} \vec{f} \cdot \nabla\phi_\kappa \, d\Omega_\kappa + \oint_{\partial\Omega^\kappa} \phi_\kappa \vec{f} \cdot \hat{n} \, dA_\kappa = 0 \right\}, \quad \forall \phi_\kappa \in \mathbb{P}^n. \tag{1.11}$$

In the *Discontinuous* Galerkin approximation, the ϕ_l are identically zero outside of the l^{th} element, so that the contributions from each element are decoupled (except for the element boundary integration). This feature of DG is called *compact support*. Again, the solution, flux, and source are approximated by a polynomial of degree N. We choose to write each term as a linear combination of polynomial basis function; here the Lagrange interpolating polynomials are used,

$$\vec{s} \approx I^N[\vec{s}] = \vec{S} = \sum_{i,j=0}^{N} \vec{S}_{i,j} l_i(\xi^1) l_j(\xi^2) \tag{1.12a}$$

$$\vec{q} \approx I^N[\vec{q}] = \vec{Q} = \sum_{i,j=0}^{N} \vec{Q}_{i,j} l_i(\xi^1) l_j(\xi^2) \tag{1.12b}$$

$$\vec{f} \approx I^N[\vec{f}] = \vec{F} = \sum_{i,j=0}^{N} \vec{F}_{i,j} l_i(\xi^1) l_j(\xi^2) \tag{1.12c}$$

The basis functions for $\mathbb{P}^N$ are chosen as each of the Lagrange interpolating polynomials,

$$\phi_{m,n} = l_m(\xi^1) l_n(\xi^2) \tag{1.13}$$

With this, equation (1.11) becomes

$$\int_{\Omega^\xi} (\vec{S}_t^\kappa + \vec{Q}^\kappa)\phi_{m,n}\, d\Omega^\xi - \int_{\Omega^\xi} \vec{F}^\kappa \cdot \nabla_{\vec{\xi}} \phi_{m,n}\, d\Omega^\xi + \oint_{\partial\Omega^\xi} \phi_{m,n} \vec{F}^{\kappa,*} \cdot \hat{n}\, dA^\xi = 0, \quad m,n = 0,1,...,N \tag{1.14}$$

The final step is to replace the integrals in (1.14) with discrete quadrature. For this we use the Legendre-Gauss quadrature, which yields exact integration for each term in (1.14). Additionally, the interpolation nodes are specified as the Legendre-Gauss nodes, which simplifies the integration.

$$\begin{aligned} \left(J_{m,n}\vec{S}_{m,n}\right)_t = &- \left[\sum_{i=0}^{N} \hat{D}_{m,i}^{(\xi^1)} \tilde{F}_{i,n}^{(\xi^1)} + \left(\frac{l_m(1)}{w_m^{(\xi^1)}} \tilde{F}^*(1,\xi_n^2) - \frac{l_m(-1)}{w_m^{(\xi^1)}} \tilde{F}^*(-1,\xi_n^2)\right) \cdot \hat{\xi^1}\right] \\ &- \left[\sum_{j=0}^{N} \hat{D}_{n,j}^{(\xi^2)} \tilde{F}_{m,j}^{(\xi^2)} + \left(\frac{l_n(1)}{w_n^{(\xi^2)}} \tilde{F}^*(\xi_m^1,1) - \frac{l_n(-1)}{w_n^{(\xi^2)}} \tilde{F}^*(\xi_m^1,-1)\right) \cdot \hat{\xi^2}\right] \\ &+ J_{m,n}\vec{Q}_{m,n}; \quad m,n = 0,1,...,N \end{aligned} \tag{1.15}$$

Each element can support curvilinear geometry. Define the mapping from physical space $\vec{x}$ to computational space $\vec{\xi}$ using

$$\vec{x} = \vec{x}(\vec{\xi}). \tag{1.16}$$

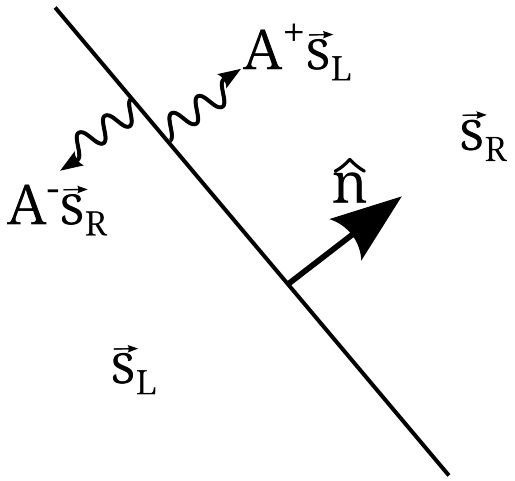


Figure 1.1: A depiction of the setup for computing the flux across an edge. The flux is split by upwinding the characteristic variables of the Jacobian matrix.

In this case, the solution and the source term are weighted by the Jacobian of the mapping and the flux is rotated to the *contravariant* flux. Introducing mappings that are also approximated by polynomial interpolants introduces additional (spectrally accurate) *aliasing* errors. Appendix **??** provides the details on the metric terms that are introduced along with the form of the divergence, gradient, and curl under such a mapping.

Computing the divergence of the conservative flux in this framework can be viewed as calculating a sequence of derivatives in each computational direction. Two steps are required to compute the derivative in each direction. The first is an internal matrix-vector multiply, and the second is computing the weighted Riemann fluxes at the element boundaries. The latter is the only step which requires element-to-element communication. The approximate fluxes are now presented for the systems (1.1), (1.2), and (1.3).

Riemann Flux

This section needs some attention!

Regardless of which system we are solving, the DG approximation requires that we compute an estimate of the flux across an element's edge given the solution on either side of the edge. In general, the solution is discontinuous across the edges. Let $\vec{s}_L$ and $\vec{s}_R$ denote the solution to the "left" and to the "right" of the edge as depicted in Fig. 1.1. The goal is to compute

the flux across an edge given the left and right states. The conservation law, (1.4), can be written

$$\vec{s}_t + \frac{\partial f^n}{\partial \vec{s}} \frac{\partial \vec{s}}{\partial n} = 0, \tag{1.17}$$

where, for the sake of exposition, the source term has been dropped. The flux in the edge-normal direction is $f^n = \vec{f} \cdot \hat{n}$ and the directional derivative of the solution is $\frac{\partial \vec{s}}{\partial n}$. For a short period of time, Δt,

$$\vec{s}_t + \frac{\partial f}{\partial \vec{s}}|_{t=t_0} \frac{\partial \vec{s}}{\partial n} = \mathcal{O}(\Delta t). \tag{1.18}$$

In (1.18), the *Jacobian* of the flux, $\frac{\partial f^n}{\partial \vec{s}}$, is evaluated at the fixed time $t = t_0$. For hyperbolic problems, like the shallow water equations, the Jacobian has real eigenvalues and can be diagonalized. Let

$$\frac{\partial f}{\partial \vec{s}}|_{t=t_0} = \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1}, \tag{1.19}$$

define the diagonalization, where $\boldsymbol{P}$ is a matrix whose columns are the eigenvectors and $\boldsymbol{D}$ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues of the Jacobian. Substituting (1.19) into (1.18) and multiplying on the left by $\boldsymbol{P}^{-1}$ gives

$$\vec{w}_t + \boldsymbol{D}\boldsymbol{P}^{-1} \frac{\partial \vec{s}}{\partial n} = \mathcal{O}(\Delta t), \tag{1.20}$$

where $\vec{w} = P^{-1}\vec{s}$ are the *characteristic* variables. Equation (1.20) can be rewritten, approximately as

$$\vec{w}_t + \boldsymbol{D} \frac{\partial \vec{w}}{\partial n} \approx \mathcal{O}(\Delta t). \tag{1.21}$$

where variations in the eigenvectors with n have been ignored. Equation (1.21) has approximate solutions

$$w^i = w_0^i(n - \lambda_i(t - t_0)) \tag{1.22}$$

where w^i and λ_i are the i^{th} eigenvector and eigenvalue, w_0^i is the characteristic variable at time $t = t_0$, and n is the physical distance normal to the edge. To evaluate the flux at the edge, we need to know the solution at the edge ($n = 0$). At time $t_0 + \Delta t$,

$$w^i(0, \Delta t) = w_0^i(-\lambda_i \Delta t) \tag{1.23}$$

so that if $\lambda_i > 0$, the solution depends on the state to the left of the edge, and if $\lambda_i < 0$, the solution depends on the initial state to the right of the edge. Because of this, we split the

diagonalization into two components,

$$\frac{\partial f}{\partial \vec{s}}|_{t=t_0} = \boldsymbol{P}\boldsymbol{D}^{+}\boldsymbol{P}^{-1} + \boldsymbol{P}\boldsymbol{D}^{-}\boldsymbol{P}^{-1} = \boldsymbol{A}^{+} + \boldsymbol{A}^{-}, \tag{1.24}$$

where $\boldsymbol{D}^{+}$ is the diagonal matrix with only positive eigenvalues and $\boldsymbol{D}^{-}$ is the diagonal matrix with only negative eigenvalues. The compact notation $\boldsymbol{A}^{+}$ and $\boldsymbol{A}^{-}$ is used for the Jacobian matrix associated with the splitting of the positive and negative eigenvalues.

Under similar assumptions used to obtain (1.22), the flux at the boundary can be approximated

$$\vec{f}^{n} \approx \vec{f}^{n,*} = A^{+}\vec{s}_L + A^{-}\vec{s}_R \tag{1.25}$$

It can be shown that

$$A^{+} = \frac{A + |A|}{2} \tag{1.26a}$$

$$A^{-} = \frac{A - |A|}{2} \tag{1.26b}$$

so that (1.25) can be written

$$\vec{f}^{n,*} = \frac{1}{2}\left(\vec{f}^{n}(\vec{s}_L) + \vec{f}^{n}(\vec{s}_R) - |A|(\vec{s}_L - \vec{s}_R)\right) \tag{1.27}$$

Equation (1.27) is the approximate Riemann flux. The choice of approximation for $|A|$ yields different linear flux schemes. In the shallow water software, we use the Lax-Friedrich's flux where $|A|$ is approximated by the maximum eigenvalue using either the left or the right state,

$$|A| = max(\lambda_i(\vec{s}_L), \lambda_i(\vec{s}_R)). \tag{1.28}$$

1.4 Algorithm and Implementation

This section reports on the original algorithm design, originally based on the work of Kopriva (2009).

To forward-step the discrete system in Eq. (1.15), the flux divergence and the source terms need to be calculated. The first two lines of (1.15) indicate that we need to calculate the flux at each of the interior nodes (for each element) and to estimate the Riemann flux at each of the element boundaries. (Kopriva, 2009) showed that using the Legendre-Gauss

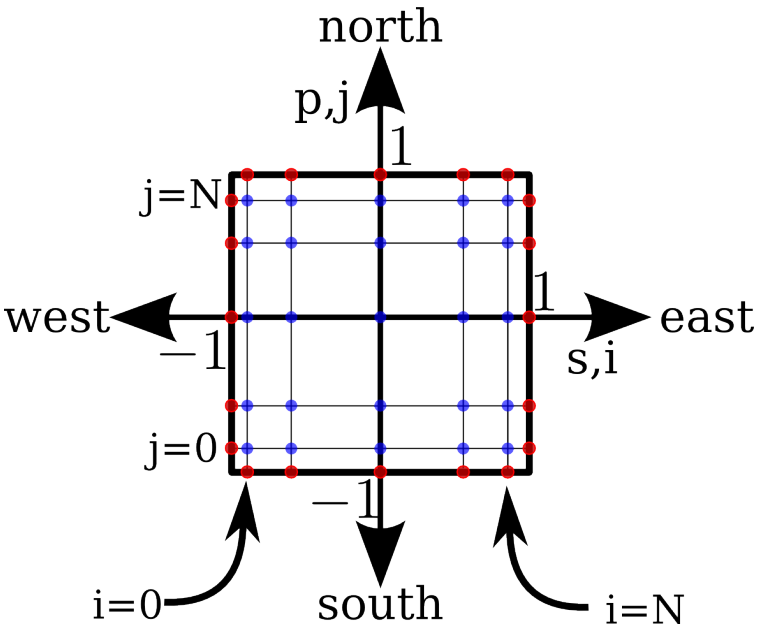


Figure 1.2: This schematic depicts where the solution is known (at the blue nodes) within an element; these are the Legendre-Gauss points. The computational coordinate are s and p. In this document, the integer i increments the solution in the s direction and j increments the solutions in the p direction. To compute the Riemann Flux, we need to calculate the solution at the element boundaries (at the red nodes). The schematic also indicates how the SELF software references each of the sides of an element. "South" refers to the side where $p = -1$, "North" $p = 1$, "West" $s = -1$, and "East" $s = 1$.

quadrature satisfies the discrete metric identities and therefore does not introduce non-conservative spurious errors due to coordinate mapping. For this reason, and the higher accuracy associated with Gauss Quadrature (as opposed to Gauss-Lobatto), we employ the Legendre-Gauss quadrature. The main drawback with the Legendre-Gauss points is that they do not include the element edges. To calculate the Riemann flux then, we must first interpolate the solution to the element boundaries.

Interpolation to the Element Boundaries

Recall that the solution is represented as

$$\vec{S} = \sum_{i,j=0}^{N} \vec{S}_{i,j} l_i(s) l_j(p). \tag{1.29}$$

As an example, suppose we want to calculate the solution on the eastern boundary, where $s = 1$, $p = p_n$. Note that the p_n are the interpolation nodes for the Lagrange interpolating polynomials, so that $l_j(p_n) = \delta_{j,n}$, where $\delta_{j,n}$ is the Kronecker Delta function. Then

$$\vec{S}_{east} = \sum_{i=0}^{N} \vec{S}_{i,n} l_i(1) \vec{S}_{west} = \sum_{i=0}^{N} \vec{S}_{i,n} l_i(-1) \tag{1.30a}$$

$$\vec{S}_{south} = \sum_{j=0}^{N} \vec{S}_{m,j} l_j(-1) \tag{1.30b}$$

$$\vec{S}_{north} = \sum_{j=0}^{N} \vec{S}_{m,j} l_j(1) \tag{1.30c}$$

Computing the Flux

Once the solution is known at the boundaries of each of the elements, the boundary flux can be computed using a Riemman flux. To do this, we cycle over the edges in the mesh and calculate the flux across each edge, exchanging the flux with the elements that share common edges. Once we have the flux at the boundary of the elements, we can then calculate the flux divergence using the first two lines of (1.15).

Computing the Flux-Divergence

1.5 Callgraph of S/R GlobalTimeDerivative

The subroutine `GlobalTimeDerivative` is the workhorse of the DGSEM implemented by the SELF. It is comprised of three main steps

1. Loop over the elements and interpolate the solution to the element's boundaries.
2. Loop over the edges, compute the Riemann Flux, and copy the flux to the abutting elements.
3. Loop over the elements and compute the flux divergence according to (1.15)

To begin optimizing the DGSEM-SELF module (for example, with the shallowwater solver) we first make a call-graph of `GlobalTimeDerivative`, which will indicate where

most of the time is spent. The call-graph will be generated here using `valgrind` with the `callgrind` tool. A program is written that calls only the `GlobalTimeDerivative` subroutine after the shallowwater data-structure is constructed. The modules and program are compiled with debugging information turned on using the `-g` compile-time flag. The program is run underneath valgrind on a Dell-Latitude E6410 Laptop (see Appendix A.1 for architecture details). The parameters for the shallowwater configuration give 10^{th} degree polynomials within the elements of an unstructured mesh. The unstructured mesh is generated from `SpecMesh+3D` using the control file

`SELF/examples/shallowwater/dipole/run/box.control`.

The mesh is stored in the SpecMesh file

`SELF/examples/shallowwater/dipole/run/box.mesh`.

This mesh consists of 489 corner-nodes, 448 elements, and 936 edges and is shown in Fig. 1.3.

The program `GTD_Test` is run underneath valgrind with the callgrind tool enabled using the call,

```
valgrind --tool=callgrind ./GTD_Test
```

The output of callgrind is visualized using `kcachegrind`, and is shown in Fig. 1.4. This call-graph indicates that, of the time spent in `GlobalTimeDerivative`, 85.68% of the time is spent in `MappedTimeDerivative` (even though it is called fewer times than `EdgeFlux`). Inside of `MappedTimeDerivative`, most of the time is spent gathering the solution (8.45%), bathymetry(7.72%), and covariant metric terms (6.87%) from memory, totalling 23.04% of the time spent in `MappedTimeDerivative`. A significant portion of the time (19.77%) is spent within `DGSystemDerivative` which computes the flux gradient along a single computational direction.

This call graph clearly indicates that `MappedTimeDerivative` is the most expensive routine, with most of its time spent performing data-gathers from memory. Although spectral element methods have the potential to be computationally intensive, this particular

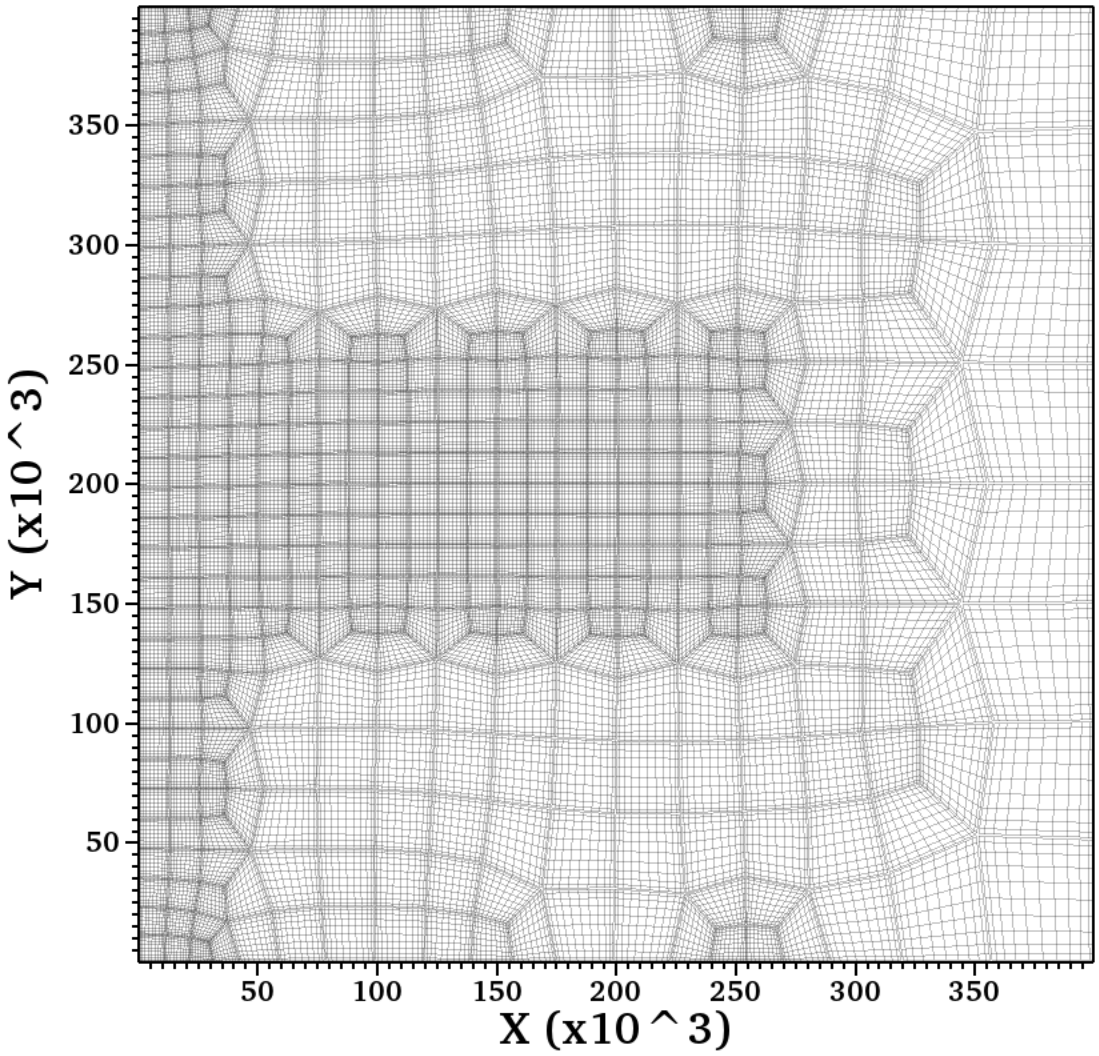


Figure 1.3: The unstructured mesh that is used to calculate the flux-divergence for the examples discussed here.

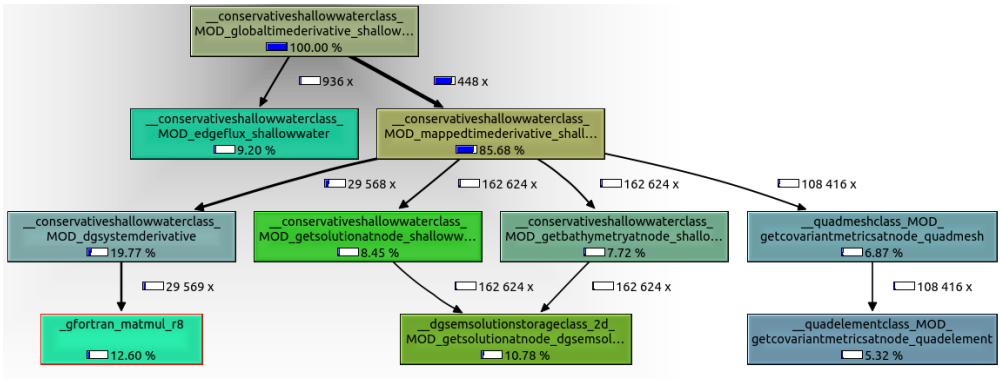


Figure 1.4: The call-graph generated from callgrind and visualized with kcachegrind for the original version of S/R GlobalTimeDerivative for the SELF-shallowwater solver. Immediately, we can see that most of the execution time is spent within the MappedTimeDerivative subroutine.

implementation is not meeting this potential. Instead it is bogged down retrieving data from memory. A sampling profile of the integration program (`integrate_sw`) obtained from Allinea-MAP is shown in Fig. 1.5. For reference, the integration program was run here for 1000 time-steps so that the initialization (manual constructor) did not occupy the majority of the program cost. On average, 68.6% percent of the program is spent executing memory instructions and only 19.6% is spent executing floating point operations. For the dense computations that the discretization (1.15) implies, we would expect a larger floating-point to memory load *(Can we estimate this ratio ??)*.

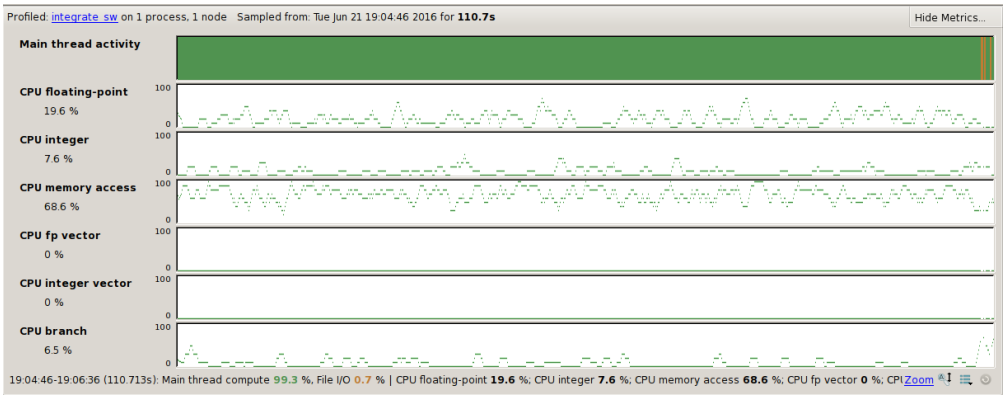


Figure 1.5: A sampling profile obtained with Allinea-MAP profiler. The `integrate_sw` executable (for this profile) is generated with no optimizations and no OpenMP. The profile gives a breakdown of the percent time spent, on average, executing various CPU instructions. Currently, the execution time is dominated by memory instructions.

1.6 Suggested Algorithm Improvements

Calculating the Mapped Time Derivative

The multiplication between the internal flux and the derivative matrix require $\mathcal{O}(N^3)$ operations. The internal flux can be represented as a matrix, so that the flux divergence can be computed by a single matrix-matrix multiply.

```
DO j = 0, N
   DO i = 0, N

      Fx = Xflux( solution(i,j) )
      Fy = Yflux( solution(i,j) )
      GetMetricTerms
      [F1(i,j), F2(j,i)] = ContravariantFlux( Fx, Fy, MetricTerms )

   ENDDO
ENDDO

tend = MATMUL( dMat, F1 ) + MATMUL( dMat, F2 )

DO j = 0, N
   DO i = 0, N
      tend(i,j) = tend(i,j)  + ( least(i)*F1east(j) + &
                                 lwest(i)*F1west(j) )/weight(i) + &
                               ( lnorth(j)*F2north(i) + &
                                 lsouth(j)*F1south(i) )/weight(j)
   ENDDO
ENDDO
```

Boundary Interpolation

When interpolating the solutions within each element to the element boundaries, we currently follow Algorithm ..., where we loop over each computational direction separately, pull a single dimension array of the solution from memory, and perform a vector-vector product. This procedure follows logically from the 1-D problem and made it trivial to extend the 1-D DGSEM software to 2-D and 3-D. However, in 2-D, this algorithm results in many, very

small $\mathcal{O}(80 - 200\,bytes)$ memory accesses that are interrupted by vector-vector products. This sort of low memory access coupled with small floating point operations is pervasive throughout the SELF and can be improved by having a single, larger memory access and a single matrix-vector product.

Recall that the boundary interpolation procedure is mathematically described by (1.30c). If the 2-D array $\vec{S}_{i,j}$ is seen as a matrix, call it $\boldsymbol{S}$, and if we write the Lagrange interpolating polynomials interpolants as vectors, then,

$$\vec{S}_{east} = \boldsymbol{S}^T \vec{l}_{east} \tag{1.31a}$$

$$\vec{S}_{west} = \boldsymbol{S}^T \vec{l}_{west} \tag{1.31b}$$

$$\vec{S}_{south} = \boldsymbol{S} \vec{l}_{south} \tag{1.31c}$$

$$\vec{S}_{north} = \boldsymbol{S} \vec{l}_{north} \tag{1.31d}$$

The boundary interpolation can be condensed further as two matrix-matrix products by combining east and west interpolations, and south and north interpolations,

$$\begin{pmatrix} \vec{S}_{east} & \vec{S}_{west} \end{pmatrix} = \boldsymbol{S}^T \begin{pmatrix} \vec{l}_{east} & \vec{l}_{west} \end{pmatrix} \tag{1.32a}$$

$$\begin{pmatrix} \vec{S}_{south} & \vec{S}_{north} \end{pmatrix} = \boldsymbol{S} \begin{pmatrix} \vec{l}_{south} & \vec{l}_{north} \end{pmatrix} \tag{1.32b}$$

A matrix vector product would require a single solution access per element where we retrieve $Mem = 8 * (N+1)^2$ bytes from memory; there are $(N+1)^2$ solution nodal values in each element and we multiply by 8 assuming that the nodal values are double precision floating point values. Typical values of N, the polynomial degree, range from $N = 7$ up to $N = 15$, so that $Mem = 512 \rightarrow 2048$ bytes. Eqs. (1.31d) requires four separate passes of the solution matrix and four separate passes of each boundary interpolant array to a matrix-vector product routine. Eqs. (1.32b), on the other hand, require only two passes of the solution matrix and two passes of pairs interpolant array to a matrix-matrix product routine. Since all of the arrays fit into typical cache memory, an algorithm based off of (1.32b) will have a larger floating point - to - memory access ratio given the fewer passes of the solution and will likely execute faster.

Current CPU architectures have cache sizes on the order of MB. The memory fetch ($\mathcal{O}(kB)$) for each boundary interpolation is far smaller than current CPU cache sizes ($\mathcal{O}(MB)$).

Mapping solutions to a plotting mesh

2 Projects

2.1 Algorithm parallelization through OpenMP

2.2 Algorithm parallelization through MPI

2.3 Forcing Vectorization

2.4 Mixed OpenMP and MPI

2.5 Matrix Multiplication on the GPU

2.6 Data Structure Mods for Reducing Memory Access Costs

2.7 Parallel File I/O

A Computing Platforms

This section of the appendix lists the details of the computing platforms that the SELF implementation of the shallowwater solver has been tested on.

A.1 Dell Latitude E6410 (Quad-Core laptop)

Below is the output from *cat /proc/cpuinfo* for one of the four cores

```
vendor_id : GenuineIntel
cpu family : 6
model : 37
model name : Intel(R) Core(TM) i7 CPU       M 640  @ 2.80GHz
cpu MHz : 1199.000
cache size : 4096 KB
physical id : 0
siblings : 4
core id : 2
cpu cores : 2
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
```

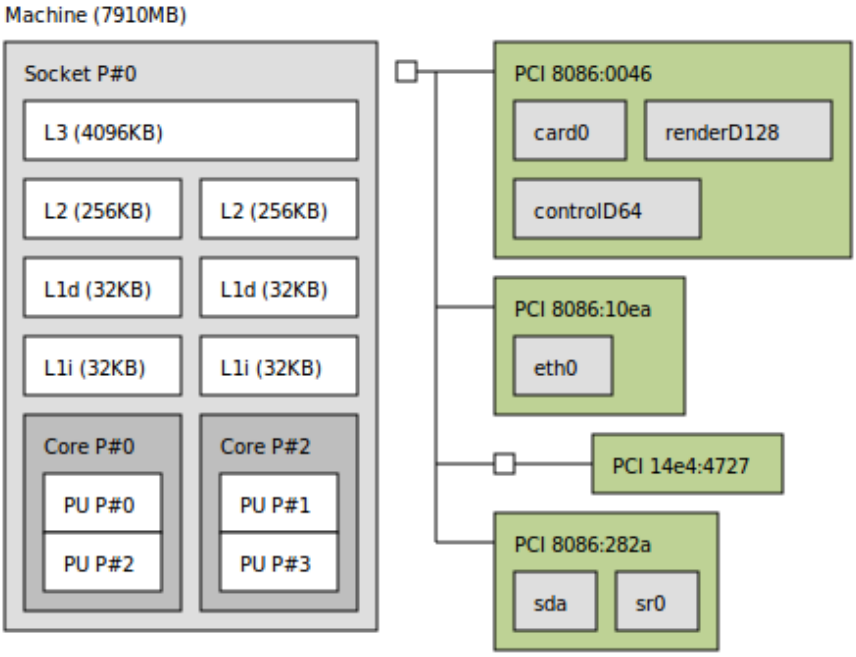


Figure A.1: A graphical representation of the Dell-Latitude E6410 hardware, from `lstopo`.

Bibliography

D. Kopriva. *Implementing spectral methods for partial differential equations*, volume 33. Springer, 2009.