Swept 1-D Scheme Description

Daniel Magee

August 2016

1 Introduction

The swept scheme is an algorithm for solving PDEs (partial differential equations) with explicit methods. The purpose of the algorithm is to reduce the solver program's run-time by advancing in time as far as possible in a node's private memory space. A node can be any processing element with a private memory space that provides faster access to stored values than the program's global memory space such as a single thread on a CPU, a single CPU in a cluster, or, in this case, a block of threads on a GPU. By performing as many computations as possible on values local to the node, the number of global memory accesses is kept to a minimum and each memory access transfers more of the values that are required to continue the computation. On most modern machines the memory transfer bandwidth is high so the difference between transferring large arrays (1000 values) and a few values is negligible. This scheme reduces the impact of latency, the fixed cost of memory transfer, on the overall program performance.

2 First order

The simplest application of the swept scheme is in solving a linear PDE of order two or less, such as the 1D heat conduction equation, with a first order explicit method. Figure 1a shows the progression of nodal computations from the initial conditions, $u(x, 0)$, in a node with $n = 16$ spatial points. In GPU code each node must have a multiple of 32 spatial points since there are 32 threads in a warp. In the code, this procedure is performed by the kernel function upTriangle. With a first order method, each sub-timestep is a full timestep.

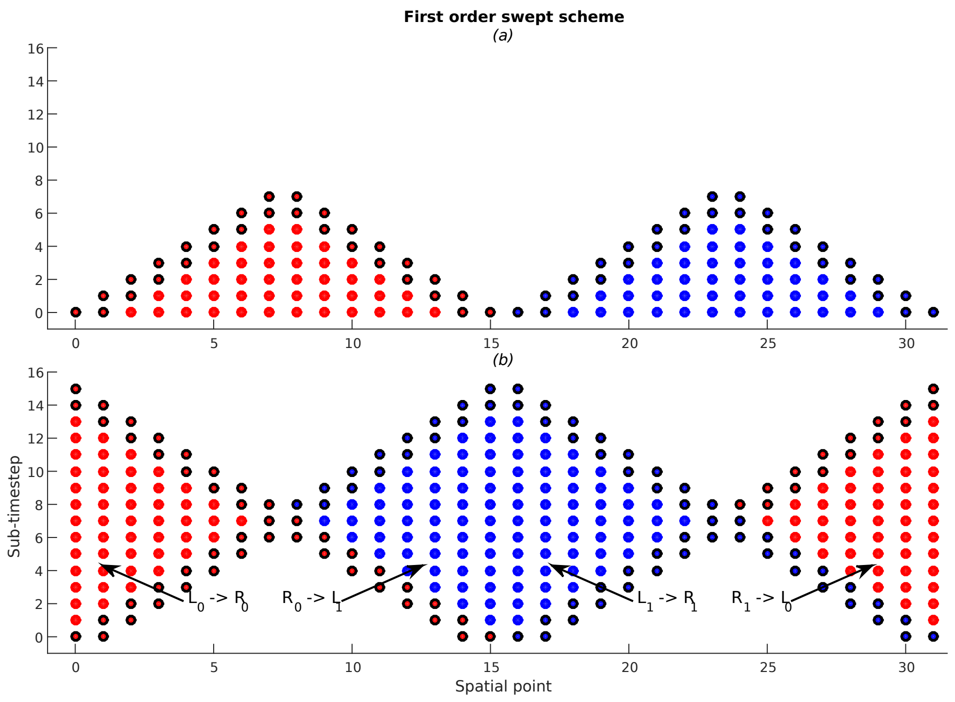

Figure 1: Swept rule procedure for first order numerical method. Red dots are values computed by node 0, blue by node 1. Dots with black edges are the values that must be stored for the solution to proceed.

The progression in Figure 1a is a triangle. Since only the edges of each tier have unknown neighbor values, which are uncomputed or known only to another node, these values are ignored and only the ones that are dependent only on local data are computed. The edges, which have not been used as the basis for a timestep at a spatial point, must be saved and passed to the neighboring node in a single global memory access. The saved values must be kept according to their individual edges and share the top tier values.

Figure 1b shows why the two edges are stored individually. After the first step, the edges are either passed to the neighboring node or traded within the node to populate the values necessary to proceed with the computation. For example, the node 0 moves it's left edge to the right edge and receives the right edge of node 1 as it's left edge and applies the spatial boundary condition at it's center. With these values, the computation can proceed locally by filling in the complimentary triangle and advancing another triangle forming a diamond. This way the computation can advance by n timesteps with only two rather than n global memory accesses. The procedure advances further by passing values in the opposite direction and zig-zagging in this fashion until the simulation is complete. Since the diamonds do not store values at a single timestep, the simulation can only output values when a complimentary triangle is computed and the final n length local tier is returned. In the code, this is performed by the kernel function downTriangle. This can only be called after the values are passed to the right, the direction shown in Figure 1b.

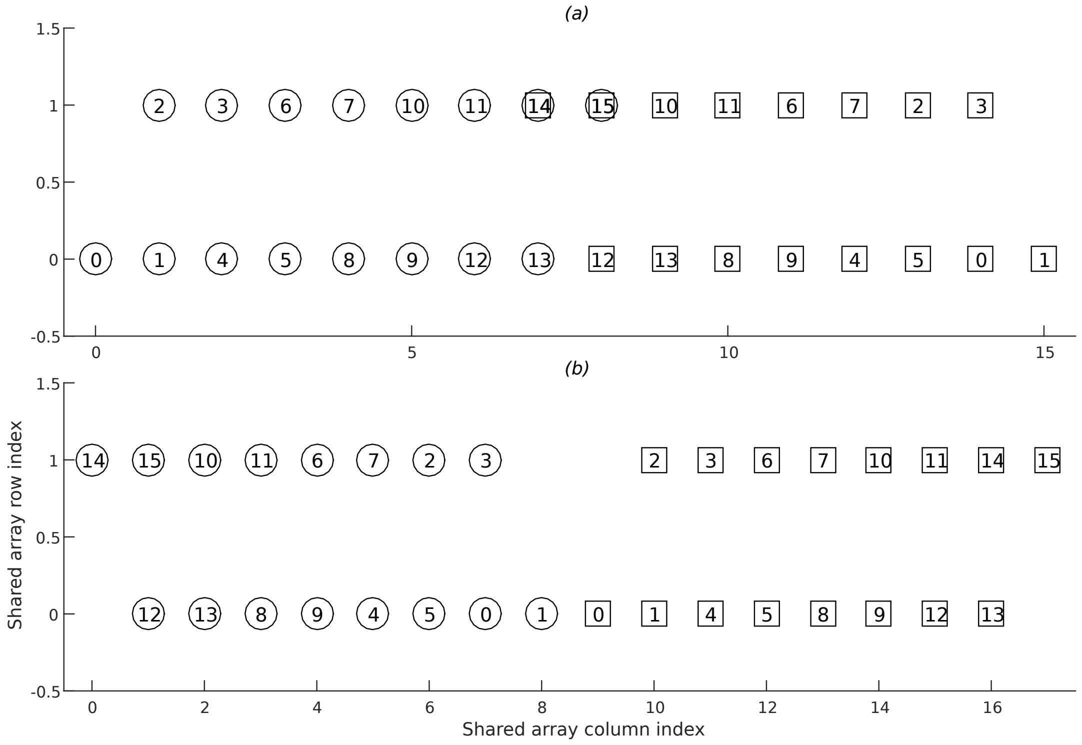

Figure 2: (a) The left *(circles)* and right edges *(squares)* of the locally computed triangle as stored in the local matrix are passed to their respective global arrays with the indices shown. (b) The edges are reinserted to seed the complimentary triangle after being swapped using the indices shown.

It can be recognized from Figure 1a that the interior of the triangle is only needed to progress to the next timestep, and the even and odd tier edges respectively do not interfere with each other in the spatial domain. Since the nodes' private memory space is by definition limited, it is important to minimize its usage. The interior nodes only need to be stored for the next timestep so the triangle may be stored as a matrix with two rows where the first row contains even tier values and the second row contains odd values. This way, the interior values are overwritten once they are used and the only values that remain are the edges. Figure 2a shows the result of a local computation with this method. The axes show their position within the matrix and the values in the figure refer to their index in the global left or right array, the last two values, the tip of the triangle, are copied into both arrays.

To proceed in this fashion, the left and right arrays, after being traded, must be reinserted into a new two row matrix as shown in Figure 2b. Now, the edges of the previous first row are moved to the center of the matrix x dimension, and the x dimension is two units longer since the row after the complimentary triangle is complete requires $n + 2$ values. The computation proceeds by filling the empty two indices on the top row and then overwriting the bottom row's middle four indices and so on.

3 Second order with flux

While the first order method shows the basis of the swept scheme, complications arise when higher order methods or equations are considered, such as a second order Runge-Kutta method applied to the Kuramoto-Sivashinsky equation. The equation and method are discussed in the 1-D Swept Equations document in this repository.

Since the flux is required at neighboring spatial points to calculate the new timestep or predictor value, there are four sub-timesteps per timestep as shown in Figure 3a. Since the timestep value is used to calculate the next timestep, it must be saved to be used four tiers after it is computed. In the previous method, several of these interior values would be overwritten on the next even tier sub-timestep. These forgotten but required values are marked with x in Figure 3b.

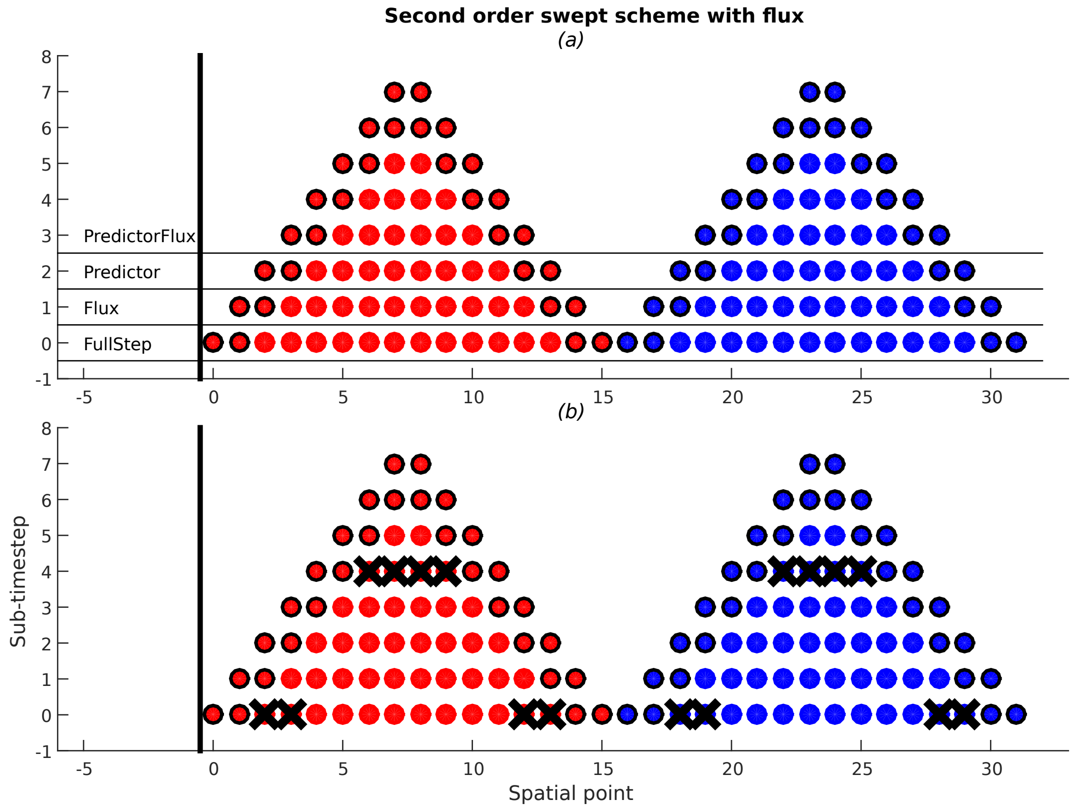

Figure 3: (a) Second order scheme with flux sub-timestep in the same form as first order scheme. (b) The problem with this procedure is that values marked with x are overwritten before they are needed.

Saving four values per tier would fix the problem, but would require a larger matrix in private memory and most of the values stored there would be unnecessary. The solution is shown in Figure 4a. Here the flux computation is included in each sub-timestep so there are only two sub-timesteps per timestep (a predictor and a final value), and spatial neighbors of neighbors are used in each timestep. This flattens the triangle or diamond and requires saving four values per sub-timestep, but the two row matrix may be used as described in section 2 with some minor, but notable, differences. First, the indices shown in Figure 2 are not the same, but they are similar, instead of the two open slots in (b) there are four. Also, the matrix in (b) has four more rows than (a) rather than two. Conveniently, the predictor corrector method ensures that all odd tiers, the second matrix row, will contain predictor values, and the bottom row will contain the final values.

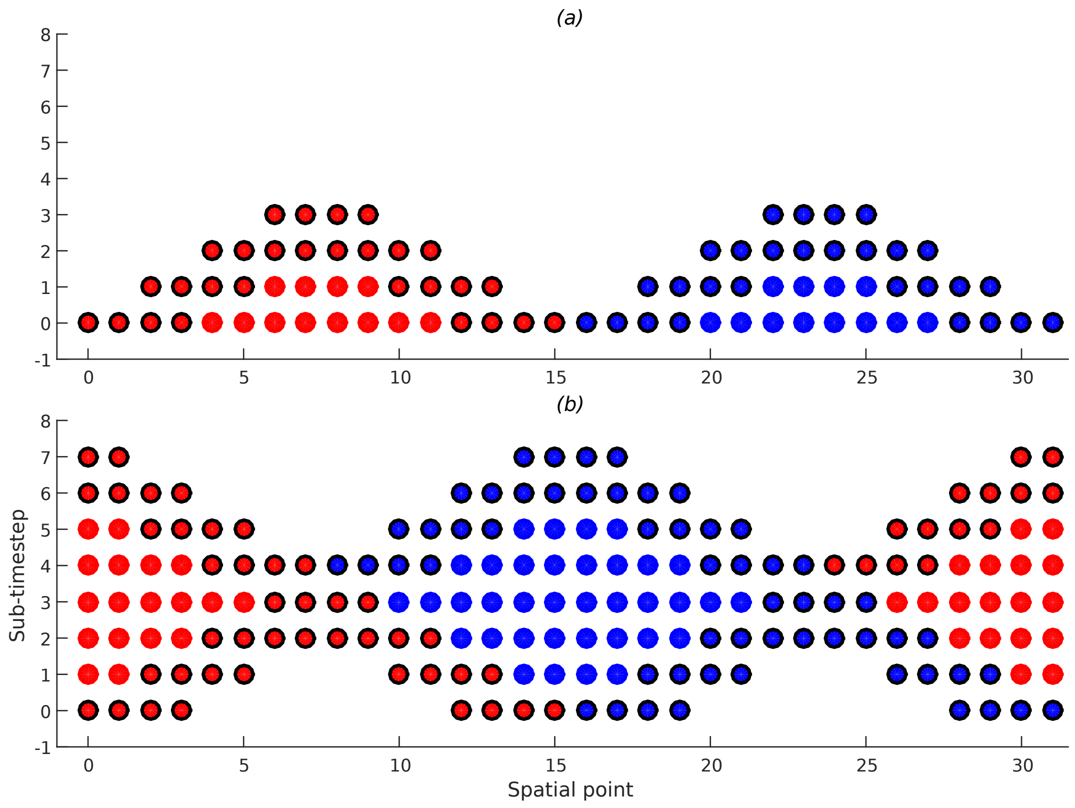

Figure 4: Flattened swept scheme includes flux calculation in sub-timestep. Values are passed between nodes as shown in Figure 1b.