Speeding Up BathymetryCrosses in Julia Using Threads and Grid-Based Acceleration

This document explains the key ideas and implementation details behind accelerating the detection of crossings between bathymetric transects using a spatial grid and optional multithreading in Julia.

Overview

The `bathymetryCrosses` function detects intersection points between bathymetric transects by checking for crossings between line segments derived from bathymetry data.

Key features of this implementation include:

- Segmenting transects into sub-line segments of length `dp` pings.
- Creating a spatial index grid to reduce the number of segment comparisons.
- Using multithreading (`Base.Threads`) to parallelize processing of the segments.

Segment Struct

```
struct Segment
    transect::Int           # Index of the parent transect in `baths`
    start_idx::Int          # Starting index of the segment in the
transect
    bbox::Tuple{Float64, Float64, Float64, Float64}  # Bounding box:
(xmin, xmax, ymin, ymax)
end
```

Each `Segment` represents a line between two pings (`start_idx` and `start_idx + dp`) in a transect. Its bounding box is used to determine grid cells it overlaps.

Creating Segments

```
function create_segments(baths, dp)
```

This function creates segments for all transects:

- For each transect, it iterates over ping indices with step `dp`.
- It constructs a bounding box for each segment using min/max latitudes and longitudes.
- It stores the result in a `Segment` array.

Building the Grid Index

```
function build_grid_index(segments, cell_size)
```

This function accelerates spatial search by indexing segments into a 2D uniform grid:

- Global bounds (`all_xmin`, `all_xmax`, etc.) are computed from segment bounding boxes.
- The number of grid cells along x and y is determined based on `cell_size`.
- For each segment, grid cell coordinates it overlaps are computed and stored in `grid`, a dictionary mapping cell coordinates to lists of segment indices.

This reduces the number of pairwise segment comparisons.

Querying the Grid for Candidates

```
function query_grid(grid, xmin, ymin, cell_size, x_cells, y_cells,
bbox)
```

This function returns a list of candidate segments that overlap with a given bounding box:

- The function computes which grid cells the bounding box spans.
- It collects and returns the unique segment indices present in those grid cells.

Core Crossing Detection: `bathymetryCrosses`

```
function bathymetryCrosses(baths::Vector{Bathymetry}, dp::Int64;
cell_size=0.01)
```

This function:

1. Calls `create_segments` and `build_grid_index` to prepare data.
2. Iterates over all segments (optionally using `Threads.@threads` for parallel speedup).
3. For each segment `seg1`, it queries the grid for candidate overlapping segments `seg2`.
4. Checks if `seg1` and `seg2` intersect in 2D using dot product tests.
5. If intersection is found:
   - Interpolates intersection position and depth.
   - Stores the crossing information in dynamically resized result arrays.

Multithreading is **safe** here because no segment pair is tested more than once (only if `j > i`) and each thread only reads from shared data or writes to independent memory (with care taken if `Threads.@threads` is used).

Performance Optimization Techniques

1. **Spatial Grid Indexing**
   - Avoids O(N^2) comparisons between all segments.
   - Candidate selection is narrowed to segments in overlapping grid cells.
2. **Segment Bounding Boxes**
   - Stored in each `Segment` to avoid recomputation.
   - Used for fast pruning before costly intersection test.
3. **Efficient Intersection Test**
   - Projects coordinates and computes dot products to test for intersection.
   - Avoids full geometry libraries for performance.
4. **Dynamic Array Resizing**
   - Preallocates crossing arrays with initial size.
   - Doubles capacity as needed when storing results.
5. **Optional Multithreading**
   - Outer loop over segments can be parallelized with `Threads.@threads`.
   - Ensures thread safety by controlling write access.

Summary

The `bathymetryCrosses` function is an optimized routine for detecting intersections between bathymetric transects. By combining spatial indexing with optional multithreading and efficient geometric tests, the implementation scales well to large datasets and offers significant performance benefits over naïve approaches.