Design Patterns for Multithreaded Algorithm Design and Implementation

Will Schroeder / Spiros Tsalikis

Thank You

- Scientific Computing and Imaging Institute
  University of Utah

- National Institute of General Medical Sciences of the National Institutes of Health: R24 GM136986

Reference Code

This presentation is meant to be independent of implementation details. Refer to these systems for concrete examples:

- vtkSMPTools - CPU-based
- vtk-m - accelerator / GPU-based

Simple Implementation Concepts

1. Parallel for loop - a functor is invoked simultaneously on subsets (subrange) of the range (0,N): For(0,N, functor)

2. Functor (invoked on each thread)
   - Initialize() - initialize thread local storage *(optional)*
   - operator() - operate on subrange
   - Reduce() - combine / composite each thread's output into final result *(optional)*

Will

3. Thread local storage - objects / variables local to each thread

4. Atomics - variables free from data races
   std::atomic

Spiros

5. Other common built-in functions: sort, fill, transform

kitware

Explained Through Case Studies

Will

1. Marching Cubes vs Flying Edges

2. Surface Extraction of 3D Unstructured Mesh

Spiros

Parallel For (over subranges $(begin_i,end_i)$)

For(0,N, functor) ⟹ $functor(begin_0,end_0)$ *thread 0*

⟹ $functor(begin_1,end_1)$ *thread 1*

⋮

⟹ $functor(begin_t,end_t)$ *thread t*

Common Design Patterns

- Remove data dependencies
  - Identify computational primitive(s)

- Multiple passes are typical:
  - Determine output shape and size
  - Precisely allocate output
  - Map input to output
  - Execute to produce output

Removing Data Dependencies

Trivial Parallelism:

Map n input primitives to m output primitives. The mapping is obvious, direct, and often implicitly defined. Typically only a single parallel pass is required.

E.g., compute vector magnitudes from vector field.

General Parallelism:

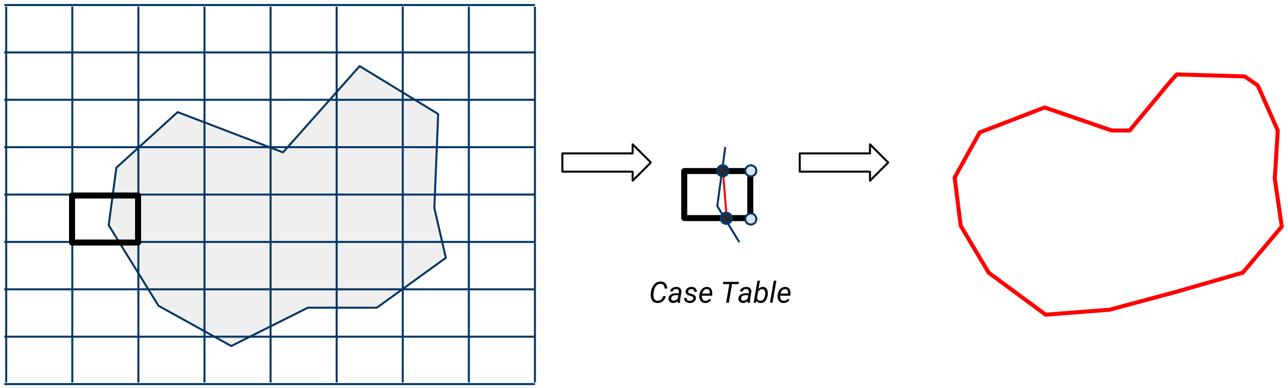
Mapping requires identifying data primitives, building explicit mapping, and possibly performing reduction / compositing. Multiple parallel passes are required.

E.g., isocontouring

Case Study #1: Marching Squares / Cubes

Given a scalar field, produce an (approximation) to the isosurface f(x) = constant (isovalue)

Output typically varies dramatically as the isovalue is varied.



Pixel square (or in 3D the voxel cube) is the computational primitive.

kitware

MC Algorithm

For each voxel cell in a volume:

- access eight voxel values
- compute case
- produce intersection points & triangles
- add points and triangles to output
- (optionally) merge coincident points

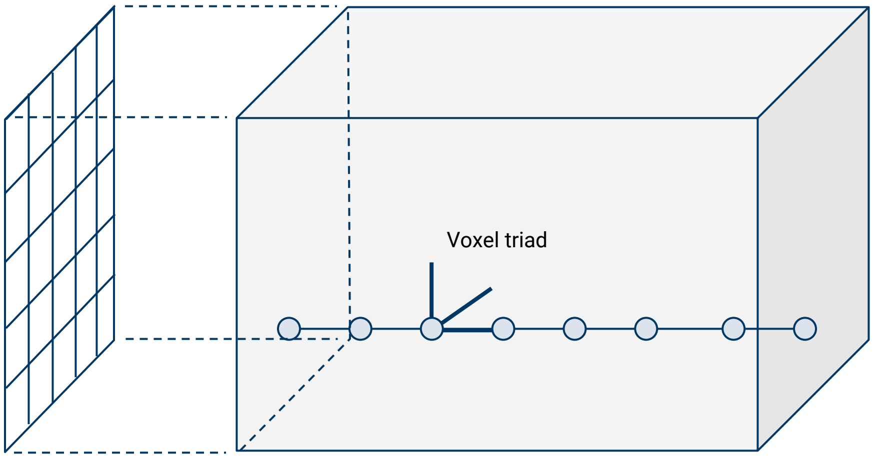
MC Parallelization Challenges

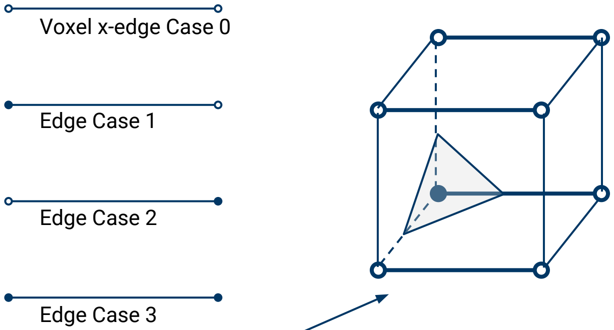
- Voxel values are accessed up to eight times
- Edge intersections are performed up to four times
- Dynamic arrays are needed to insert output points and triangles
  - Repeated resizing - blocked threads
  - Memory allocation is slow
- Point merging is a bottleneck (blocked threads)
  - Typically uses spatial or topological hash

Example: Flying Edges

- Four pass algorithm (requires only parallel For() loops)
  - Volume edges are the parallel primitive, i.e., edges are processed independently
- Visits voxel values only once
- Edge intersections performed only once
- Exact, one-time memory allocation
- The point merging bottleneck eliminated

Flying Edges: Definitions



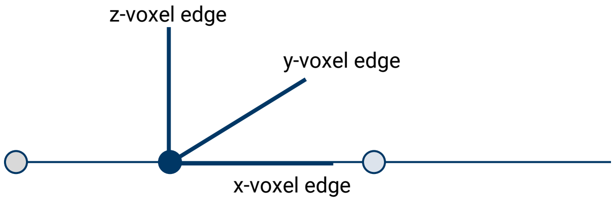


Volume-x-Edge metadata
(nXPts, nYPts,nZPts,ntris, xL,xR)

Computational primitive is volume x-edges.
Each voxel x-edge is classified
Four voxel x-edges are combined to produce MC case
MC case gives number of intersection points, triangles produced by each voxel
Auxiliary storage for edge metadata, and voxel value classification, is created and maintained
Trim edges (xL,xR) keep track of data location
Trim edges can be used to skip data (volume edges, volume slices)

kitware

Pass 1: Classify Voxel x-Edges



- For each volume x-edge
- Simply classify voxel value above or below isovalue to determine voxel-x-edge case
- Count number of voxel-x-edge intersections (edge case==1 || edge case==2)
- Keep track of edge trim (xL, xR)
- Update edge meta data, voxel triad classifications
- Note that volume voxel values are accessed only once

Pass 2: Classify y-z-Edges

- For each volume-x-edge metadata
- Note that edge trim can be used to skip over much of the volume
- Combine four edges forming a voxel cube to determine MC case
- Use modified MC case table to determine number of y, z intersections, and triangles generated
- Update edge metadata

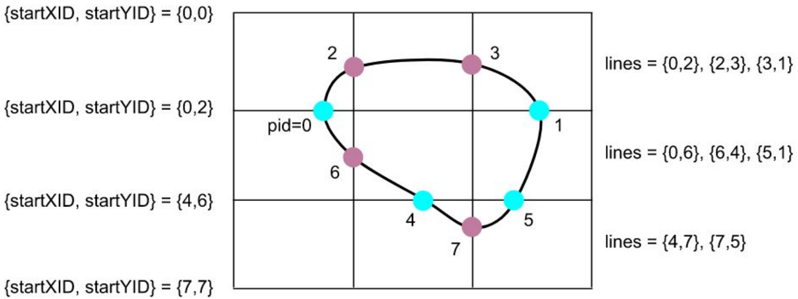
kitware

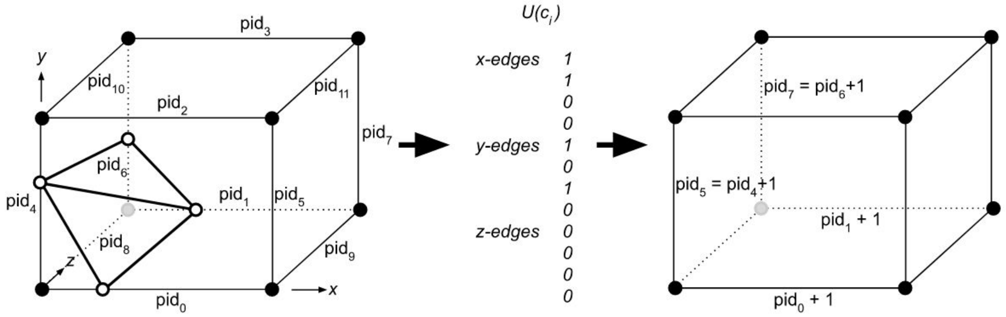
Pass 3: Compute Output Shape

- Perform prefix sum over all volume-x-edge metadata
    - Determines total number of output primitives (points, triangles)
    - Defines numbering for each point and triangle generated
    - Prefix sum often faster when performed sequentially

- One time, exact memory allocation can be performed

Pass 4: Generate Output





- For each volume x-edge
- Initialize output iterator with starting point id, triangle id
- Combine voxel-x-edge-cases to compute MC case
- Produce output points and triangles for each voxel triad
- Move to next voxel triad, updating point and triangle ids
- No point merging is required!!! Edge intersections computed only once!!!

Some Results

Algorithm	CT-angio	Supernova	Nano	Plasma
MC	1 (2.10s)	1 (2.667s)	1 (3.88s)	1 (69.86s)
MC-Opt	1.49	1.92	1.28	1.79
ST	1.44	1.90	0.54	3.51
FE	5.22	7.35	3.51	8.58

Sequential Speed ups

Algorithm	CT-angio	Supernova	Nano	Plasma
MC-Opt	1 (0.266s)	1 (0.266s)	1 (0.310s)	1 (4.56s)
ST	3.67	3.20	1.10	4.35
FE	8.26	9.45	4.49	11.05

Parallel Speed Ups (with 36 threads)

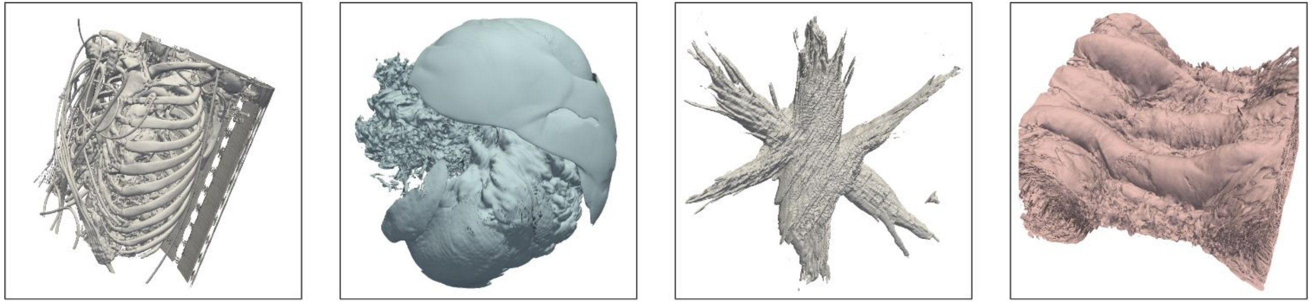


Figure 6: The four datasets used for testing. In reading order, the CT-angio, Supernova, Nano, and Plasma datasets.

Thread Local Storage

- Thread local storage (TLS) is the mechanism by which each thread in a given multithreaded process allocates storage for thread-local data. Thread-local data should be accessed only by one thread, to avoid no data races.
- Thread-local data can used to calculate the thread-local result, e.g. sum of numbers, which will be used at a reduce step to calculate the total result.
- Thread-local data along with a reduce step should be considered first over atomic variables, if possible.

kitware

A side note: Output Invariance

- Due to the "random" order in which threads are executed over subranges, output may change between runs.

- This can be managed in a number of ways
  - Explicit control of subranges
  - Explicit mapping of input -> output

In general, with finite precision arithmetic:

sum(a,b,c) + sum(d,e,f) ≠
sum(a,e,c) + sum(d,b,f)

Atomics

1. Atomics variables are used to ensure that operations like load, store, compare-and-swap (CAS), add, subtract, will be performed without a lock (mutex) and no data race will occur.
2. Atomics can also be used to create a spin-lock, i.e. mutex, which can yield better results if used appropriately, or a optimistic-lock (if determinism is not critical).
3. Atomics operations will be performed using a memory order:
   a. memory_order_relaxed # Only operation's atomicity is guaranteed, no ordering (e.g. counting)
   b. memory_order_acquire & memory_order_release # When you need a spin/optimistic lock
   c. memory_order_release & memory_order_consume # When you have a producer and a consumer
   d. memory_order_acq_rel # When ordering of operation of 1 atomic variable is required
   e. memory_order_seq_cst # When ordering of operations of > 1 atomic variables is required

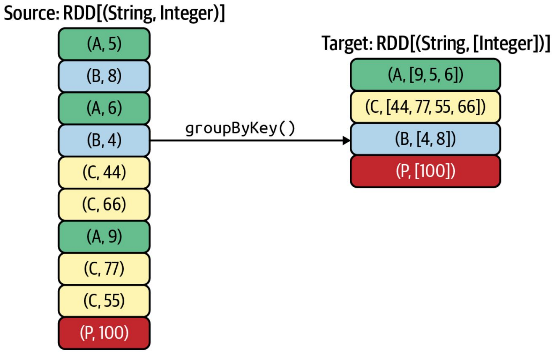
Common built-in parallel functions: Part 1

1. Fill(begin, end, value) helps you fill an array with a specific value.
   - This can be useful for initializing, e.g. counting
2. Copy(beginA, endA, beginB) helps you copy values from an array A to an array B.
3. Transform(beginA, endA, beginB, tranformFunctor) helps you transform an array A to an Array B (optionally in place).
   - This can be useful when you have thread local indices and you want to convert them to global indices.

kitware

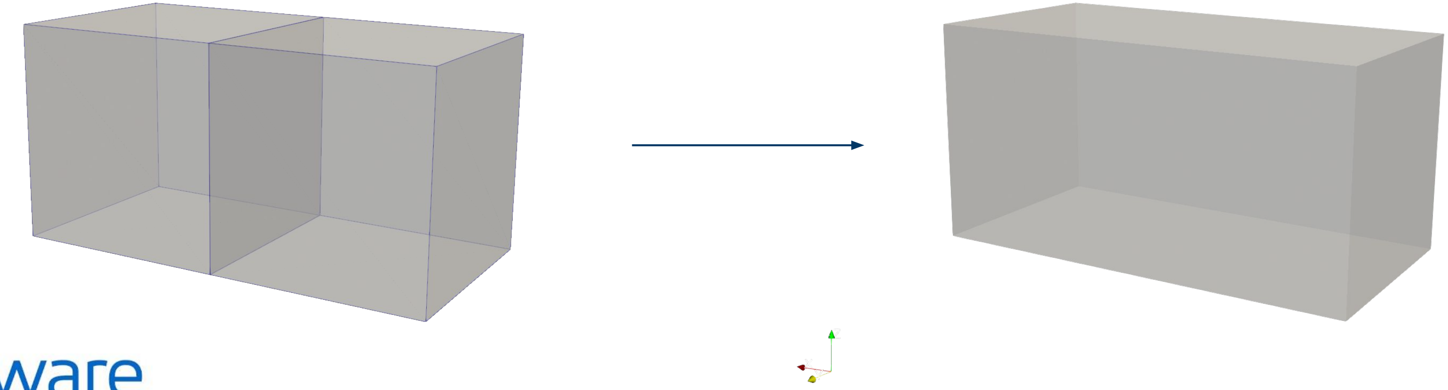
Common built-in parallel functions: Part 2

1. Sort(begin, end) helps you sort an array.
    - Be aware of O(n(log n)) and stable and un-stable versions.
    - For ReductionByKey operations, sorting can be avoided. Instead, you can build (more memory) a list of lists using counting (only for build), offsets and a flat list of lists that can be accessed using the offsets.
    - If the #keys < #uniqueKeys, a deque can later be used to keep track of the #uniqueKeys. Deque is preferred, because you can connect the thread-local deques in O(1).



Case Study #2: Surface Extraction of 3D Unstructured Mesh

1. Preserve *external* faces (used only once) and remove *internal* faces (used more than once)
2. Useful for:
   a. Rendering a 3D Unstructured Mesh without volume rendering
   b. Debugging Mesh Generation algorithms



SE Algorithm

1. For each *cell* in an unstructured mesh
    a. For each *face* in *cell*
        i. Extract the ids of the *face*
        ii. Rotate the ids so that the smallest id (p0) is first
        iii. Try to insert the *face* and compare with existing faces in FaceHashMap[p0] **list** (p0, is the key of the hash map)
            1. Remove an existing same or mirror face if one exists
            2. Else, insert it to the list by allocating space using a memory pool
2. For each *faceList* in FaceHashMap
    a. For each *face* in *faceList*
        i. Uniquely insert the points of the *face* and get their output point ids
        ii. Insert the *face* to the output cell array using the output point ids

kitware

SE Parallelization Challenges

- Modifying FaceHashMap[p0] is not thread-safe
- Allocating memory using a MemoryPool is not thread-safe
- Dynamic arrays are needed to insert output points and faces
  - Repeated resizing is not thread-safe
  - Memory-allocation is slow
- Point merging is not thread-safe
  - Typically uses spatial or topological hash

kitware

Multithreaded SE: Pass 1) Identify External Faces

- How to fix the unsafe modification of FaceHashMap and MemoryPool allocation? We have the following options:
  a. Spin-lock to Modify FaceHashMap[p0] and Spin-lock to allocate in MemoryPool.
  b. Spin-lock to Modify FaceHashMap[p0] and allocate using a thread-local MemoryPool.
  c. Modify thread-local FaceHashMap[p0] and allocate using a thread-local MemoryPool. Requires reduction to merge thread-local FaceHashMaps.
  d. Requires reduction of all faces by their key (hash value) to create a list faces (cellId, faceId) for each hash value. Iterate over the faces in each hash value, modify FaceHashMap[p0], and allocate using a thread-local MemoryPool.

kitware

Multithreaded SE: Pass 2) Compute Output Shape

1. Parse FaceHashMap ***sequentially*** and create a vector of faces.
2. Distribute the vector of faces to each thread, mark in a PointMap if an original point is used or not, and count the size of each thread's total faces, to know where to write in the output cell array.
3. Allocate output cell array
4. Parse the PointMap ***sequentially*** and assign output point ids to used original points and calculate the total number of output points
5. Allocate output points array

kitware

Multithreaded SE: Pass 3) Generate Output

1. Generate the output points arrays using the PointMap
   a. (*Optional*) Generate the output data related to points
2. Generate the output cell arrays using the PointMap and the distributed faces across each thread
   a. (*Optional*) Generate the output data related to cells

Parallel Efficiency & Speed-up

- *Speedup = Tsequential/Tparallel*
- *Parallel Efficiency = Speedup/Nthreads*
- *Acceptable* Parallel Efficiency >=%70
- If the Parallel efficiency is not good enough:
    a. Threads don't have the similar or enough amount of work (grain)
        i. Ensure that there is no *empty* work, over-decompose and distribute work, use a dynamic scheduler (load balancer)
    b. Memory reads/writes is more expensive than computation
        i. Try to access the memory in a *cache-friendly/*continuous way
    c. Synchronization, such as mutex, atomics (if used) is a bottleneck
        i. Minimize its usage or Remove completely if possible or worth it

Notes

- Parallel is not always faster, especially when the amount of work is small
- After designing a thread-safe parallel algorithm, you *should* analyze the performance using many **and** 1 thread(s).
  - Intel’s Vtune can be used to analyze performance.
- Debugging
  - Ensure memory access is thread-safe
  - Ensure system functions are thread-safe
  - Friend analysis tools (e.g., ThreadSanitizer and others)

kitware