A New Parallel Execution Process For Suggar++

Ralph Noack, Ph.D.
President

Celeritas Simulation Technology, LLC

www.CeleritasSimTech.com

Outline

- Background: Overset parallel execution
- New Suggar++ approach
  - Decomposition
  - Execution architecture
  - Observations
- Summary

Background

Typical Flow Solver Work and Communication

- Work and Data
  - Solving PDE at cell/element
  - Work and required data is element and its neighbors
  - Balanced by putting equal numbers of elements on each parallel processor
  - Work and communication pattern is static and consistent
- Communication
  - Comprised of data exchanged between elements at partition boundaries
  - Proportional to the number of grid element faces on the boundary between grid partitions
  - Minimized by minimizing number of faces on partition boundaries
- Decomposition is static

Overset Connectivity Work and Communication

- Two different types of work
  - Hole-Cutting (Direct Cut)
    - Cutting face: find elements intersected by face
  - Donor Search
    - Fringe X,Y,Z: Find containing donor element
- Data required: grids that overlap
- Connections/communications are spatial
  - Given x,y,z find containing donor element
- Spatial Decomposition Volume (SDV) approach needed to reduce communications

Overset Connectivity Work And Communication Is Variable And Dynamic

- Spatially variable
  - Region of domain has no overlapping grids (no work)
  - Other regions have lots of overlapping grids
  - Work varies with grid type
    - Cartesian grid is light and fast
    - General polyhedral grid is heavy and slow
- Temporally variable/dynamic for moving bodies
  - Grids/elements that overlap will change in time
  - Decomposition must change in time for load balance
    - Data/Grid migration is overhead and complex

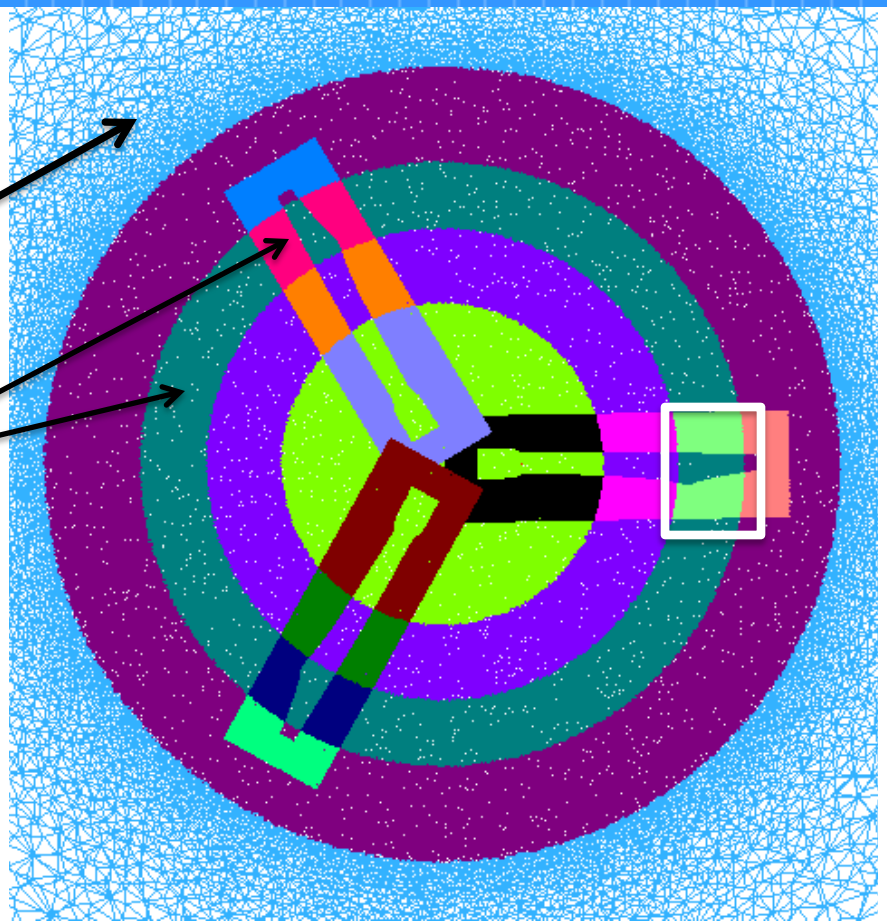
Scalability of Overset Assembly Work

- Not scalable like flow solver for dynamic problems
  - Scalable in some cases with proper SDV partitioning
  - Data memory increases with # ranks
  - Overhead increases with # ranks
    - Communication and threads
- Appearance of scalable Wall Time with idle resources
  - HyperThreads or other cores for overhead work
- Scalability improves with more memory usage

Overset Parallel Decomposition

Spatial Decomposition Volume Cylinder

- Cylindrical slices assigned to different ranks
- Outer (Cyan) portion of background grid is inactive
- Elements overlapping cylinder are assigned to rank
- Slice of (rigid) blade will always overlap slice of background grid
- Fringes & donors on same rank



Dynamic SDV Decomposition

- YOGA (2016 Cameron Druyor) uses a dynamic SDV decomposition approach
  - Decompose work
  - Request overlapping grid elements (dynamically partition data)
    - Gather elements overlapping work and build ADT
  - Uses multiple threads to try to hide communication time

Suggar++ Approach

Dynamic SDV Decomposition Suggar++ Approach: Work

- Decompose work
  - Cutting face, fringe xyz
  - Use binary tree to spatially partition work
    - Option 1: sort data and split at median
      - Perfectly balanced (off by 1 if odd number)
      - Cycle directions
    - Option 2: split largest direction at center
      - Imbalanced
      - Spatially more compact/cubic

Dynamic SDV Decomposition Suggar++ Approach: Data/Grids

- A priori decompose grids into “snippets”
  - Reduce need to search entire grid for elements overlapping work
  - Partition using octree
    - Refine until leaf contains specified number of elements
- Snippet minimal load state
  - Cartesian Bounding Box
  - List of elements overlapping the snippet
- Snippet maximal load state
  - Add grid points, connectivity, IBLANK, DSF, ADT required by work

Work Subdivision
Set of Tasks

- Task generator (Gtor)
  - Has a parent grid or dynamic body
  - Generates a set/queue of Tasks
    - Type of Work (hole cut or donor search)
    - Work Items (Cutting faces or fringes)
- Task worker
  - Needs to fetch Task from a Gtor
- Dynamic work allocation/load balance
  - Worker requests new task when needed

Execution Process

- Get Task
  - From local Queue
  - Fetch from Gtor
- Fetch Snippets overlapping Task
  - Locally build ADT
    - Used for searches
- Perform Work
- Return results

Overhead when Work and Data are not on the same rank

Actual Work

How to Reduce Overhead

- Assign Task Queues to ranks
  - Improve localization by working on Tasks that are spatially nearby
  - Tasks can be sent to non-assigned rank if needed for work balance
- Keep Grid Snippets used on rank in memory
  - Avoids Snippet fetch in subsequent tasks
  - Similar to data migration in static partitioning
  - Can lead to large increases in memory
  - Use reference count to free when not needed

Execution Pipeline

- Four stage pipeline
  - Each stage is a persistent/resident thread
    - Avoid cost of create/join threads when not busy
    - Has a Task queue
  - Task completed in one stage is inserted into queue for next stage
  - Hope is that overhead communication/work is hidden/overlapped by main work thread
    - Requires extra/idle compute resources

Stage 0: Fetch Task

- Fetch Task if queue is low
  - Looks at next two stages
  - Do not want to fetch too many tasks
  - Send message to Gtor
    - Reply contains: Task | GetWorkFrom | NoWorkLeft
- Increase Reference Count for Snippets overlapping Task
  - Want to keep Snippets in memory to avoid fetch
- Insert Task in queue for Stage 1

Stage 1: Fetch Grid Snippets

- Get Task from queue
- Fetch Snippets overlapping Task
  - Not needed if Snippet is resident on rank
  - Send request for Snippet & unpack (Communication)
  - Build ADT (Work)
- Insert Task in queue for Stage 2

Stage 2: Execute Work

- Get Task from queue
- Perform Task Work
  - Hole Cut: Cut elements in Snippets
  - Donor Search: find Snippet elements containing fringe
- Decrease Reference Count for Snippets overlapping Task
  - Free Snippet memory when Reference Count is zero
- Insert Task in queue for Stage 3

Stage 3: Return Result Data

- Get Task from queue
- Return results of Task Work
  - Hole Cut: elements in Snippets cut by faces to Snippet parent grids
  - Donor Search: Best Donor to Gtor/parent grid
- Delete Task

Communication of Tasks and Snippets

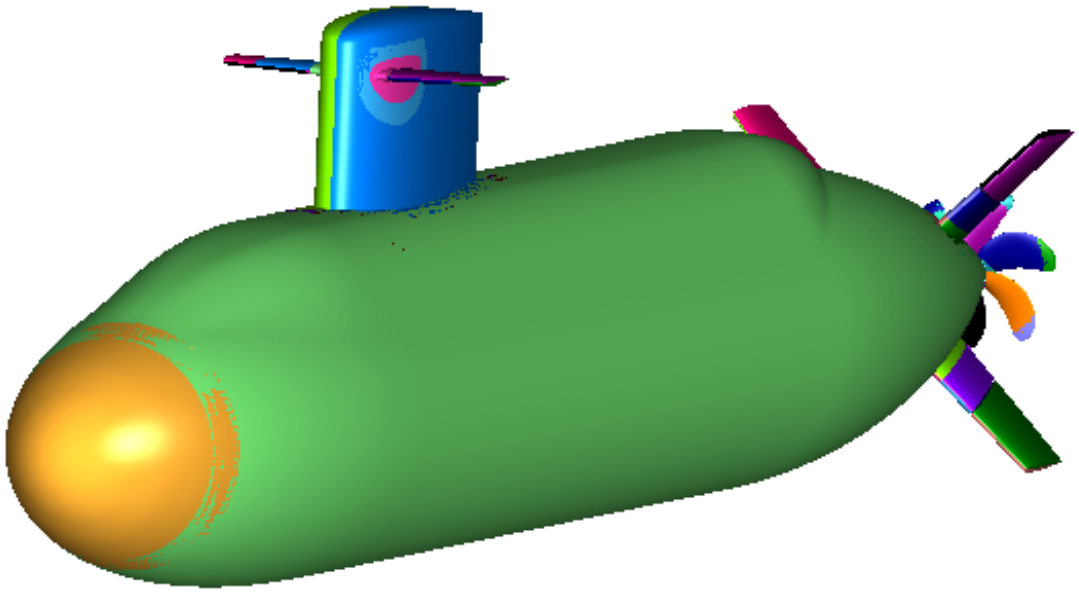
- ZeroMQ is used for asynchronous communication
- Rank 0 is Broker
  - Maintains loose state of ranks with work
  - Shuts down execution when work is finished
- Main thread/socket listens for incoming messages
- Transfer threads/sockets for data requests
  - Task, Snippet, returned data, …

Number of Threads on Each Rank

- 1 for Main/control thread
- 4 for pipeline stages
- 1 for asynch send messages
- N for TransferSockets
  - N = 1 to # ranks
- M for ZeroMQ

Observations

Test Case: Node Centered Submarine with Structured Grids



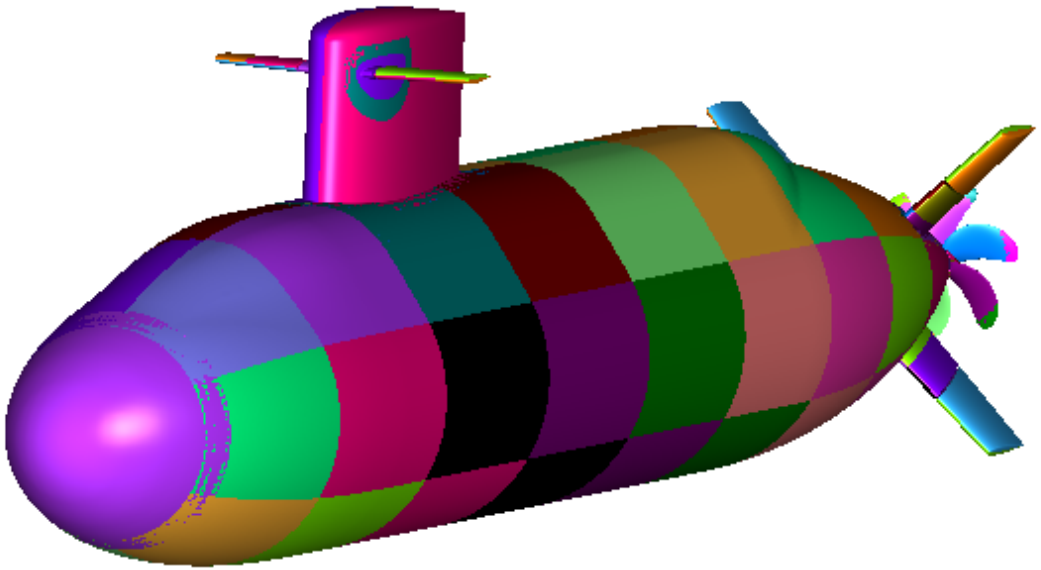
- 46 grids
- # of grid points
  - Total 130E6
  - Min 437,532
  - Max 7 grids between 1.1E6 and 1.8E6 points
  - Ave 2.8E6

Largest grid is hull body: curvilinear
Next 6 are Cartesian grids: Background and refinement

Problems with Large Grids: Background/Refinement

- Many/All other grids overlap it
  - Many/All ranks will request data/snippets from it
    - Socket contention if use only main thread/socket
    - Lots of overhead work for rank: gather & send Snippets
- Data imbalance
- Work imbalance
  - Other ranks may need to get work from this rank
- Communication bottleneck
  - Need multiple Transfer Sockets/threads

Test Case: Submarine with Structured Grids Decomposed



- 561 grids
- # of grid points
  - Total 134E6
  - Min 123E3
  - Max 1.6E6
  - Ave 239E3

Bottleneck is reduced but still present for some grids

Problems Encountered

- Debugging deadlocks
  - Multiple ranks with many threads
    - Launch xterm running gdb for each rank
- Problems with ZeroMQ
  - V4.2.1 would deadlock reading /dev/random
    - Fixed in V4.2.3
- Problems with Intel TBB
  - Submission to workgroup does not start immediately
  - Switched to Suggar++ Thread class using pthread

Observations

- ZeroMQ transfer rate
  - Slower between ranks than within a rank
  - Difficult to measure/understand bottleneck/overhead
  - MPI transfer may be faster
    - MPI for large buffer is 10X than ZeroMQ for multiple smaller transfers
- Overhead of thread create
  - Use of threads can be slower than serial

Observations

- Stage 0: Fetch Task wall time is generally small
- Build ADT (Stage 1) is small < ~10% of Task execution
- Stage 3: Return Results wall time is generally small
  - Can be large for hole cutting

To Do

- Examine effect of parameters
  - # items in work Task
  - # elements in Snippet
  - # Tasks in queue
  - Using MPI to bulk send initial set of Snippets
- Direct socket access
  - Replace some use of ZeroMQ
- Investigate use of multi-threaded MPI
  - OpenMPI 3.1.2

Summary

- Discussed overset parallel execution work and decomposition
  - Cannot scale like flow solver due to overhead
- Presented architecture for new parallel approach in Suggar++
  - Dynamic decomposition and distribution of work and data
  - Multi-threaded with asynchronous communications using ZeroMQ
  - Presented observations on process
  - Work in progress

Acknowledgements

- Thank you to Pablo Carrica, Univ. Iowa for test case and access to cluster

Questions?

Commercial distribution and support
for Suggar++ provided by

Celeritas Simulation Technology, LLC

http://www.CeleritasSimTech.com

Exportable under an EAR-99 license