Parallelisation of ANUGA

Stephen Roberts[1]

[1]Department of Mathematics
The Australian National University

September 15, 2008

Development Team

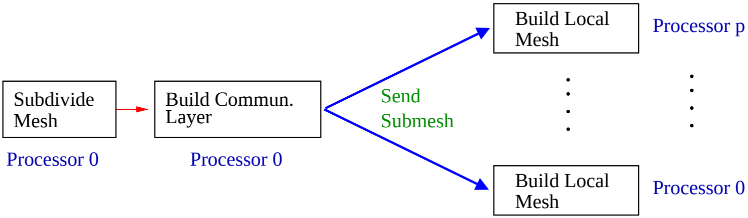
- GA: Ole Nielsen
- ANU: Stephen Roberts, Linda Stals, Jack Kelly

Update Step

```
def evolve_one_euler_step(self, yieldstep, finaltime):
    """
    One Euler Time Step
    Q^{n+1} = E(h) Q^n
    """

    # Compute fluxes across each element edge
    self.compute_fluxes()

    # Update timestep to fit yieldstep and finaltime
    self.update_timestep(yieldstep, finaltime)

    # Update conserved quantities
    self.update_conserved_quantities()

    # Update ghosts
    self.update_ghosts()

    # Update time
    self.time += self.timestep

    # Update vertex and edge values
    self.distribute_to_vertices_and_edges()

    # Update boundary values
    self.update_boundary()
```

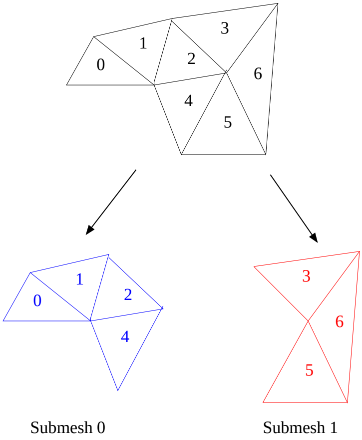
Parallelisation of the Algorithm

1. partition the mesh into a set of non-overlapping submeshes
2. build a ‘ghost’ or communication layer of triangles around each submesh and define the communication pattern
3. distribute the submeshes over the processors,
4. and update the numbering scheme for each submesh assigned to a processor.



The main steps used to divide the mesh over the processors.

Ghost Triangles

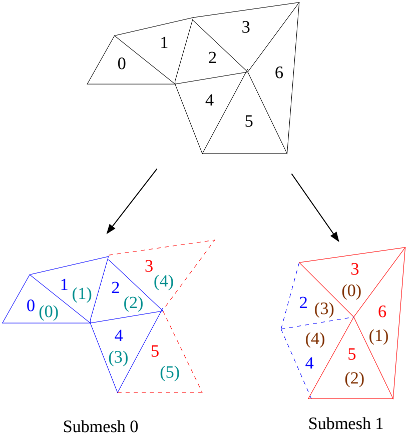


An example subpartitioning of a mesh.

Ghost Triangles

- During the `evolve` calculations Triangle 2 in Submesh 0 will need to access its neighbour Triangle 3 stored in Submesh 1.
- The standard approach to this problem is to add an extra layer of triangles, which we call ghost triangles.

Ghost Triangles



An example subpartitioning with ghost triangles. The numbers in brackets shows the local numbering scheme that is calculated and stored with the mesh.

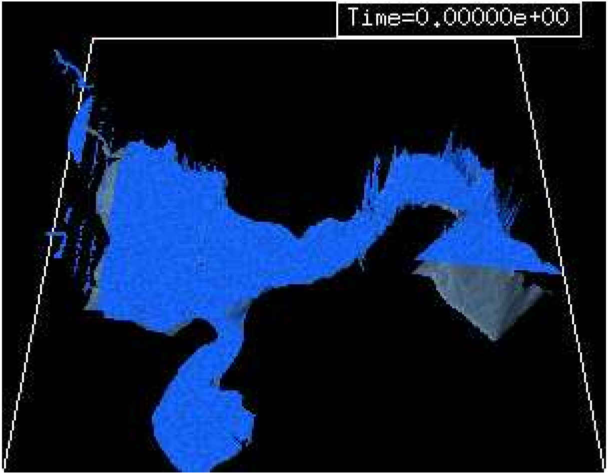
Ghost Triangles

- The ghost triangles are read-only
- They are only there to hold any extra information that a processor may need to complete its calculations.
- The ghost triangle values are updated through communication calls.
- After each `evolve` step Processor 0 will have to send the updated values for Triangle 2 and Triangle 4 to Processor 1, and similarly Processor 1 will have to send the updated values for Triangle 3 and Triangle 5
- This happens in the self .update_ghosts() of the evolve step

Mesh Partitioning

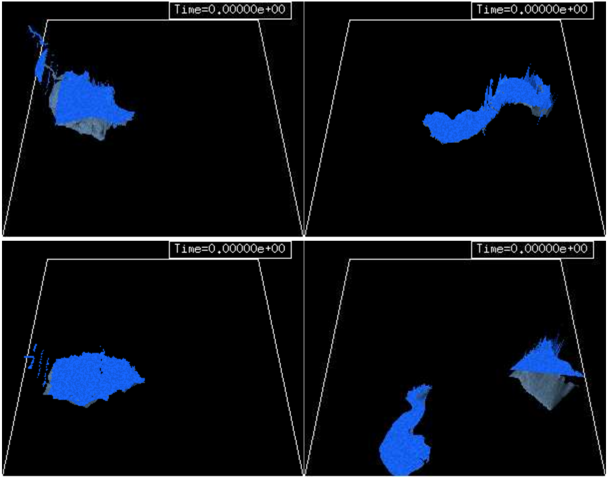
- We use Metis partitioning library.
- Hierarchical partitioner
- `glaros.dtc.umn.edu/gkhome/metis/metis/overview`
- See George Karypis and Vipin Kumar.
  A fast and high quality multilevel scheme for partitioning irregular graphs.
  SIAM Journal on Scientific Computing, 20(1):359–392, 1999.
  `http://glaros.dtc.umn.edu/gkhome/fetch/papers/mlSIAMSC99.pdf`

Mesh Partitioning: Example



The Merimbula mesh.

Mesh Partitioning: Example



The Merimbula grid partitioned over 4 processors using Metis.

Mesh Partitioning: Example

CPU	0	1	2	3
Elements	2757	2713	2761	2554
%	25.6%	25.2%	25.6%	23.7%

4-way test of Meribula Mesh

CPU	0	1	2	3	4	5	6	7
Elements	1229	1293	1352	1341	1349	1401	1413	1407
%	11.4%	12.0%	12.5%	12.4%	12.5%	13.0%	13.1%	13.0%

8-way test of Meribula Mesh

Performance Analysis

- Ran on a cluster of four nodes connected with PathScale InfiniPath HTX.
- Each node has two AMD Opteron 275 (Dual-core 2.2 GHz Processors) and 4 GB of main memory.
- The system achieves 60 Gigaflops with the Linpack benchmark, which is about 85% of peak performance.
- For each test run we evaluate the parallel efficiency as

$$E_n = \frac{T_1}{nT_n} 100,$$

where $T_n = \max_{0 \leq i < n}\{t_i\}$, n is the total number of processors (submesh) and t_i is the time required to run the `evolve` code on processor i.

Performance Analysis: Advection Rectangular

n	T_n (sec)	$E_n(\%)$
1	36.61	
2	18.76	98
4	10.16	90
8	6.39	72

n	T_n (sec)	$E_n(\%)$
1	282.18	
2	143.14	99
4	75.06	94
8	41.67	85

n	T_n (sec)	$E_n(\%)$
1	2200.35	
2	1126.35	97
4	569.49	97
8	304.43	90

Parallel Efficiency Results for the Advection Problem on a Rectangular Domain with (1) $\texttt{N} = 40$, $\texttt{M} = 40$, (2) $\texttt{N} = 80$, $\texttt{M} = 80$ and (3) $\texttt{N} = 160$, $\texttt{M} = 160$.

Performance Analysis: Advection Rectangular

- The examples where $n \leq 4$ were run on one Opteron node containing 4 processors, the $n = 8$ example was run on 2 nodes (giving a total of 8 processors).
- The communication within a node is faster than the communication across nodes, so we would expect to see a decrease in efficiency when we jump from 4 to 8 nodes.
- Furthermore, as `N` and `M` are increased the ratio of exterior to interior triangles decreases, which in-turn decreases the amount of communication relative the amount of computation and thus the efficiency should increase.
- The efficiency results shown here are competitive.

Performance Analysis: Merimbula

n	T_n (sec)	$E_n(\%)$
1	145.17	
2	77.52	94
4	41.24	88
8	22.96	79

n	T_n (sec)	$E_n(\%)$
1	7.04	
2	3.62	97
4	1.94	91
8	1.15	77

Parallel Efficiency Results for (1) the Advection Problem and (2) the Shallow Water Problem on the Merimbula Mesh.

Performance Analysis

- The efficiency results are not as good as initially expected
- The profiled code indicated that the problem is with the `update_boundary` routine.
- On one processor the `update_boundary` routine accounts for about 72% of the total computation time.
- When metis subpartitions the mesh it is possible that one processor will only get a few boundary edges (some may not get any) while another processor may contain a relatively large number of boundary edges.
- The profiler indicated that when running the problem on 8 processors, Processor 0 spent about 3.8 times more doing the `update_boundary` calculations than Processor 7.
- This load imbalance reduced the parallel efficiency.

Code

```
#------------------------------------------------------------------------
# Setup computational domain
#------------------------------------------------------------------------
points, vertices, boundary = rectangular_cross(10, 10) # Basic mesh
domain = Domain(points, vertices, boundary) # Create domain

#------------------------------------------------------------------------
# Setup initial conditions
#------------------------------------------------------------------------
domain.set_quantity('elevation', topography) # Use function for elevation
domain.set_quantity('stage', expression='elevation') # Dry initial stage

#------------------------------------------------------------------------
# Create the parallel domain
#------------------------------------------------------------------------
domain = distribute(domain, verbose=True)

#--------------------------------------------------------------------------
# Setup boundary conditions
# This must currently happen *after* domain has been distributed
#--------------------------------------------------------------------------
Br = Reflective_boundary(domain)      # Solid reflective wall
Bd = Dirichlet_boundary([-0.2,0.,0.]) # Constant boundary values

domain.set_boundary({'left': Br, 'right': Bd, 'top': Br, 'bottom': Br})
```