MS46 Architecture-Aware Graph Analytics Part II of II:

Dynamic Load Balancing of Massively Parallel Graphs for Scientific Computing on Many Core and Accelerator Based Systems

Cameron W. Smith, Gerrett Diamond, George M. Slota, Mark S. Shephard

Scientific Computation Research Center
Rensselaer Polytechnic Institute

March 8, 2018

Outline

1. Partitioning and Load Balancing
2. EnGPar - a graph based diffusive load balancer
3. Comparison to ParMA
4. Improving Performance with Accelerators

Motivation

Many evolving distributed simulations have:

- Complex relational structures.
- Irregular forms of computational and communication costs.
- Evolving imbalance of work.
- Multiple criteria that need balancing simultaneously.

Common Methods for Partitioning

- Multilevel Graph Methods
  - ParMETIS
  - Zoltan
- Geometric Methods
  - Recursive Coordinate Bisection (RCB)
  - Recursive Inertial Bisection (RIB)
  - Multi-Jagged
- Diffusive Methods
  - Label Propagation

What is EnGPar?

- A partitioning tool to complement existing multi-level and geometric methods.
- Provides a diffusive load balancing algorithm for partition improvement and supports multi-criteria partitioning.
- Utilizes a specialized multigraph structure to represent relation based data.
- Implemented to support efficient data parallel operations on accelerators and vector units in many core processors.

Software

EnGPar's source can be found at `scorec.github.io/EnGPar/`.

- Written in C++ using MPI.
- Provides C/C++ and FORTRAN APIs.
- Uses PCU for sparse neighborhood exchange peer to peer communications.
  - Found at `github.com/SCOREC/core/tree/master/pcu`

N-graph

The N-graph is a multigraph with two modes of operation: traditional or hypergraph.

The N-graph is defined as the following:

- A set of vertices V representing the atomic units of work.
- If using the traditional graph mode:
  - N sets of edges $E_0, ..., E_{n-1}$ for each type of relation.
  - Each edge connects two vertices $u, v \in V$.
- If using the hypergraph mode:
  - N sets of hyperedges $H_0, ..., H_{n-1}$ for each type of relation.
  - N sets of pins $P_0, ..., P_{n-1}$ corresponding to each set of hyperedges.
  - Each pin in P_i connects a vertex, $v \in V$, to a hyperedge $h \in H_i$.

Mapping structures to the N-graph

To map to the N-graph simulations must:

- Define units of work as the vertices.
- Decide on the mode of edges to use.
- Create (hyper)edges between the vertices whose corresponding work relate to each other.

Mapping structures to the N-graph

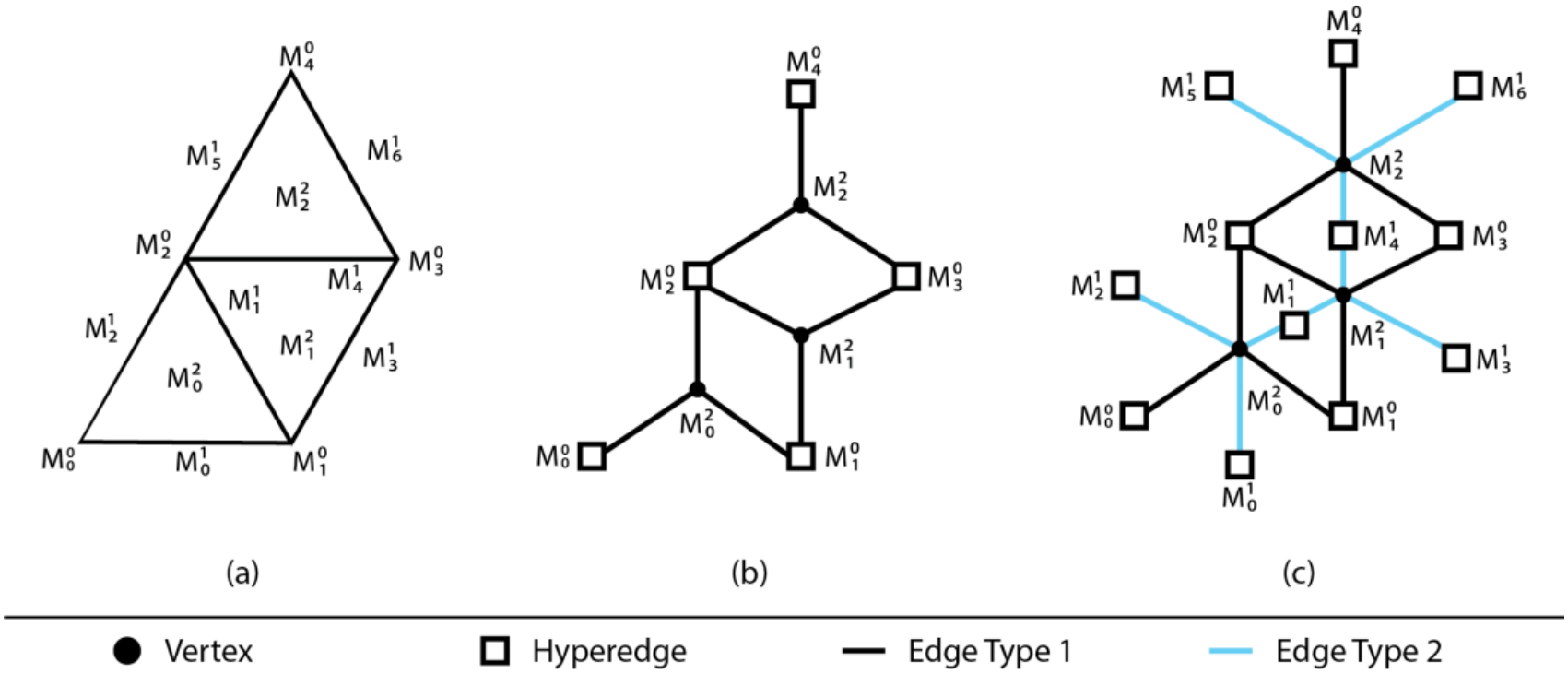


Figure: Converting a triangular mesh(a) to the N-graph with an edge type for mesh vertices (b) and an additional edge type for mesh edges (c).

Diffusive Terminology

- Sides
  - Each part determines which parts are its neighbors.
  - Determines a measurement of the area between each part.
- Cavity
  - Defined by (hyper)edges that cross a part boundary.
  - Includes all the vertices that bound the (hyper)edge.

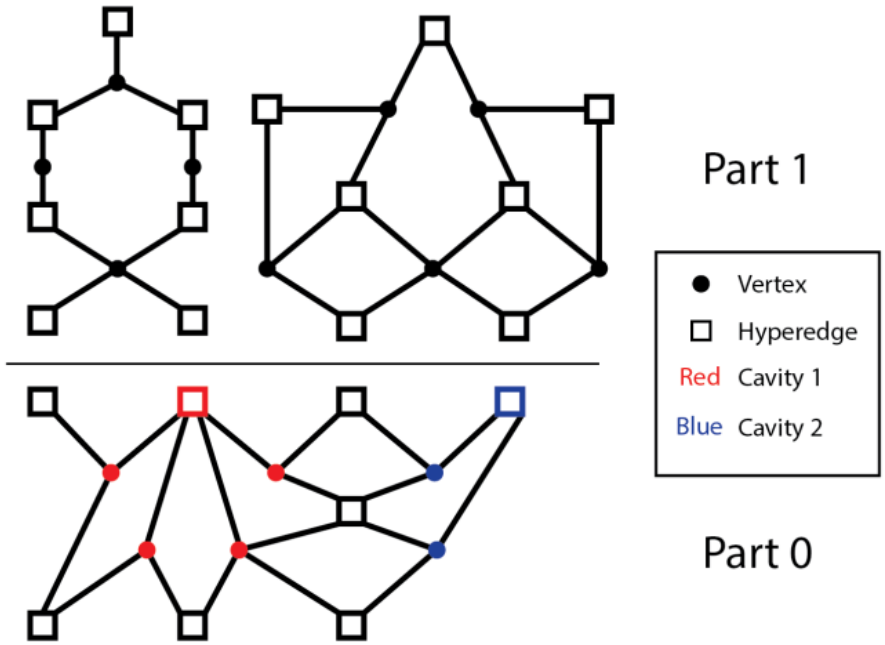


Figure: Two parts of an N-graph with a side of size 4. Two cavities are shown in red and blue for part 0.

Diffusive Terminology

- Weights
  - Each part computes its weight of the current target entity types, w_i.
  - This weight is shared with all of the part's neighbors(sides).
- Targets
  - The neighbors that the part will send weight to.
  - A part, i, will send weight to a neighbor, j, if:
    - ★ $w_i > w_j$
    - ★ the area between the parts (s_{ij}) is less than the average of all part boundaries.
  - Weight to send from part i to part j is $\alpha(w_i - w_j) * \frac{\text{size}(s_{ij})}{\text{size}(s)}$
    - ★ α is an input parameter that limits how much weight is sent in each iteration.

Diffusive Partitioning

Algorithm 1 Diffusive Load Balancing Framework

```
1:  procedure BALANCE(ngraph,entity_types)
2:      for all t ∈ entity_types do
3:          while imbalance of t > tolerance do RUNSTEP(ngraph,t)
4:              if Balancing Stagnates then
5:                  break
6:  procedure RUNSTEP(ngraph,t)
7:      sides = makeSides(ngraph)
8:      weights = makeWeights(ngraph, sides, t)
9:      targets = makeTargets(ngraph, sides, weights)
10:     queue = makeQueue(ngraph)
11:     plan = select(ngraph, targets, queue)
12:     ngraph.migrate(plan)
```

Queue

The queue provides an ordering of the (hyper)edges on the part boundary for selection.

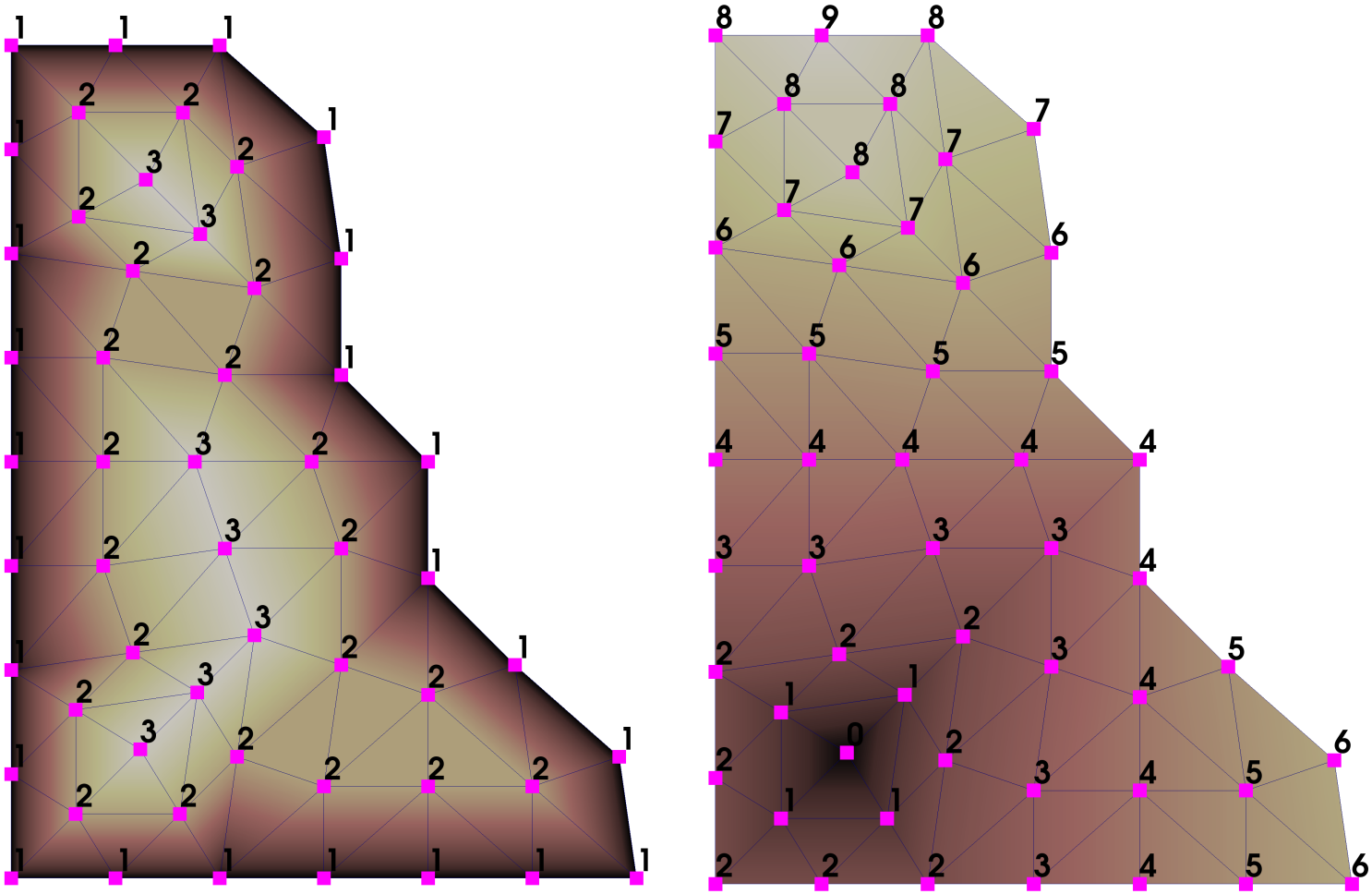
This is done in two steps:

- A breadth-first traversal starting at the part boundary to determine the furthest (hyper)edges as the center of the part.
- A breadth-first traversal starting at the center (hyper)edges to compute topological distance for each (hyper)edge on the part boundary.

When parts are not fully connected, this operation is performed on each component separately.

The queue is then ordered with shallowest components before deeper components.

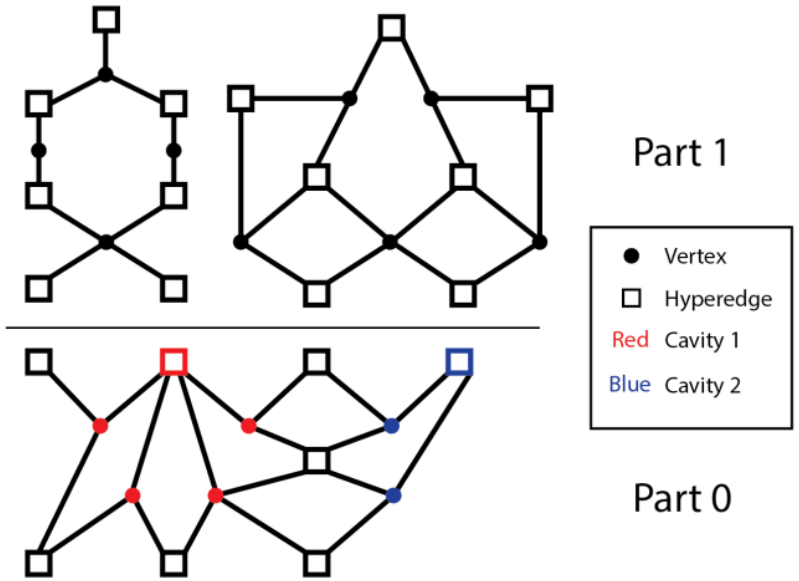
Queue



(left) The distance from each vertex to the boundary and (right) the distance from the core vertex (marked with a zero near the bottom left corner).

Selection

- Iterates over (hyper)edges that cross a part boundary.
- The cavity defined by the (hyper)edge is chosen for migration if:
  - The part that the (hyper)edge is shared with is a target part.
  - The target part has not been sent more weight than the limit.
  - The size of the cavity is small.



Comparison to ParMA

Problem Setup

We compare EnGPar's performance to its predecessor ParMA, which is built to operate directly on unstructured meshes.

ParMA and EnGPar are set to balance a mesh for a finite element analysis where:

- Scalability of matrix formation is sensitive to mesh element imbalance.
- Linear algebra routines are sensitive to the imbalance of degrees of freedom.
- Degrees of freedom are associated with mesh vertices.

ParMA and EnGPar first balance mesh vertices then elements; target imbalance of 1.05.
The imbalance of a given entity type (vtx, edge, face, or region) is defined as the max part weight divided by the avg part weight.

Problem Setup

Tests were run on a billion element mesh.
All experiments were run on the Mira BlueGene/Q system with one process per hardware thread.

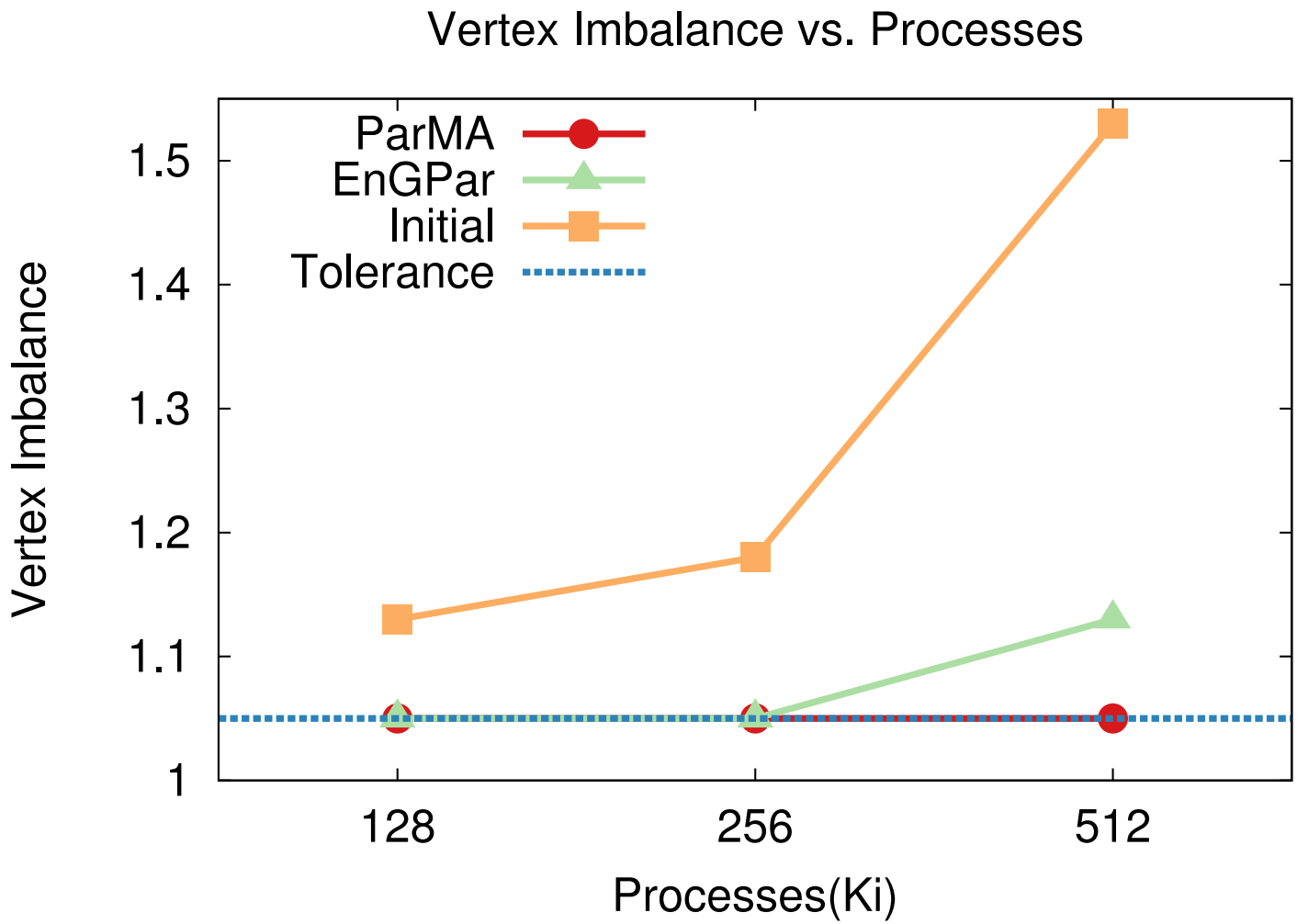
Initial partitions are built using:

- Global ParMETIS part k-way to 8Ki($8 * 2^{10}$) parts.
- Local ParMETIS part k-way from 8Ki to 128Ki, 256Ki, and 512Ki parts.

The partitions before using EnGPar or ParMA are as such:

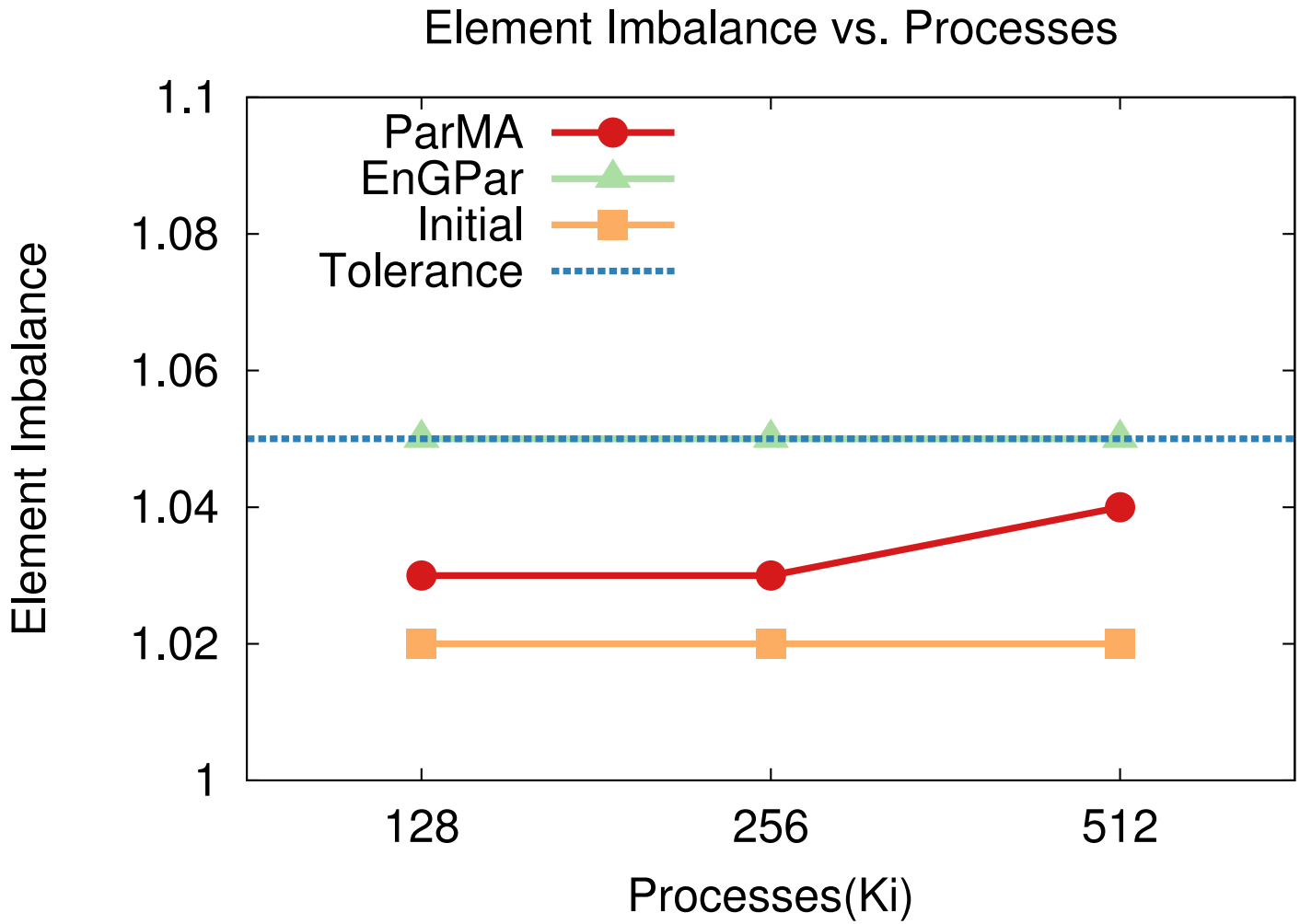
Number of Parts	128Ki	256Ki	512Ki
Elements per part	9,836	4,918	2,459
Vertex imbalance	1.13	1.18	1.53
Element imbalance	1.02	1.02	1.02

Mesh Vertex Imbalance

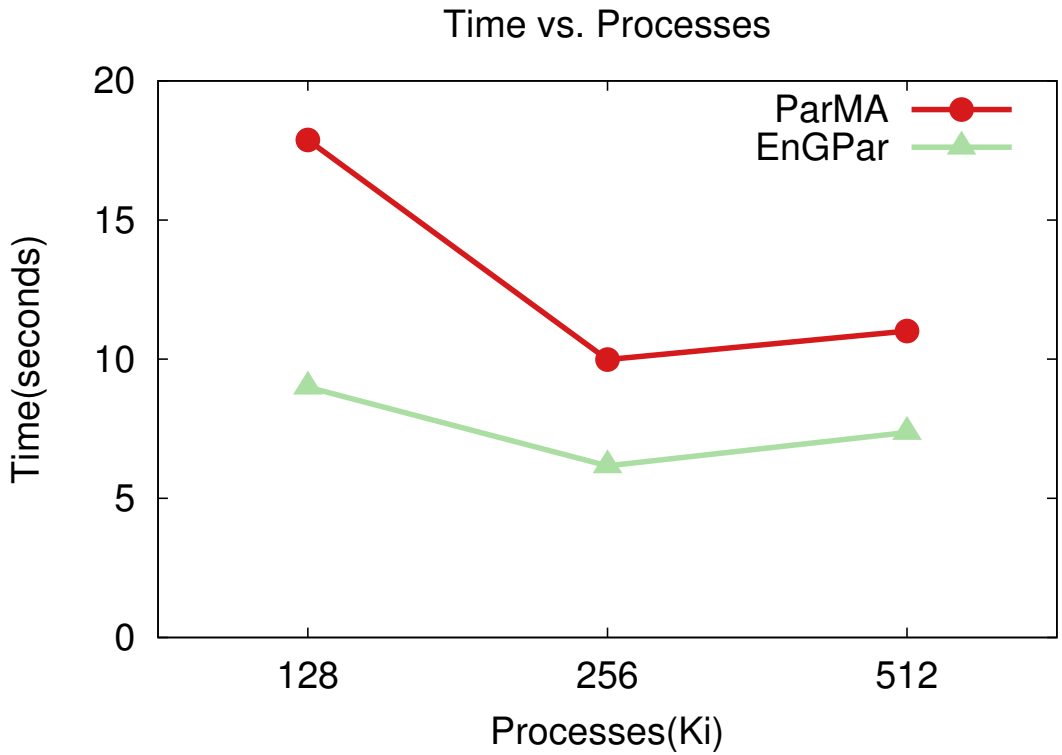


EnGPar does not reach the imbalance target yet - more logic to port over from ParMA.

Mesh Element Imbalance



Run Time Comparison



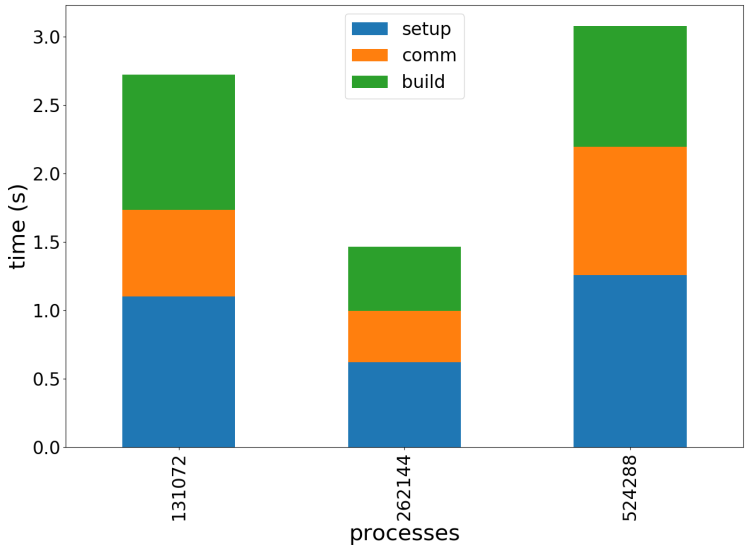
Given a tolerance they can both reach.
Local ParMETIS (embarrassingly parallel) creates the partition from 8Ki to 512Ki in 16 seconds, roughly the same amount of the time EnGPar spends to improve the mesh (requires global and neighborhood communications).

Improving Performance with Accelerators

Where is time currently spent in EnGPar?

Migration is 50% of total time at 512Ki, 48% at 256Ki, and 44% at 128Ki

- General implementation sends/receives vertices and their adjacent (hyper)edges to/from any neighbor - communications on host
- Each process rebuilds its hypergraph after communications complete



1B elm mesh balanced on 128Ki, 256Ki, and 512Ki with EnGPar

We want to improve the performance of these processes - that is a difficult task to start with.
First let's see if we can accelerate something easier; breadth first traversals of the hypergraph that we use to decide which vertices to migrate.

Related Work on Accelerated BFS

Lots of it - mostly focused on scale-free graphs and using many-core and GPUs

- SlimSell - custom data structure for algebraic BFS (Hoefler)
- Global and Local Manhattan Collapse - inner/frontier loop transformations to improve load balance and reduce irregular memory access (Slota)
- Merijn - runtime switching of graph structure and push/pull algorithm based on front size and other factors (M. Verstraaten)

Can we apply some of these approaches to a hyper multi-graph and run effectively on manycore CPUs, GPUs, and FPGAs?

Architecture Review

Manycore

- Tens of out-of-order cores with hardware level pipelining
- 1-2 TFLOPs per processor/socket, possible* to get near peak
- Programming models and languages: shared memory, distributed, SHMEM, OpenMP, PThreads, MPI, etc.
- Efficient use of available memory bandwidth often yields peak performance - supported by both data parallel and message passing models

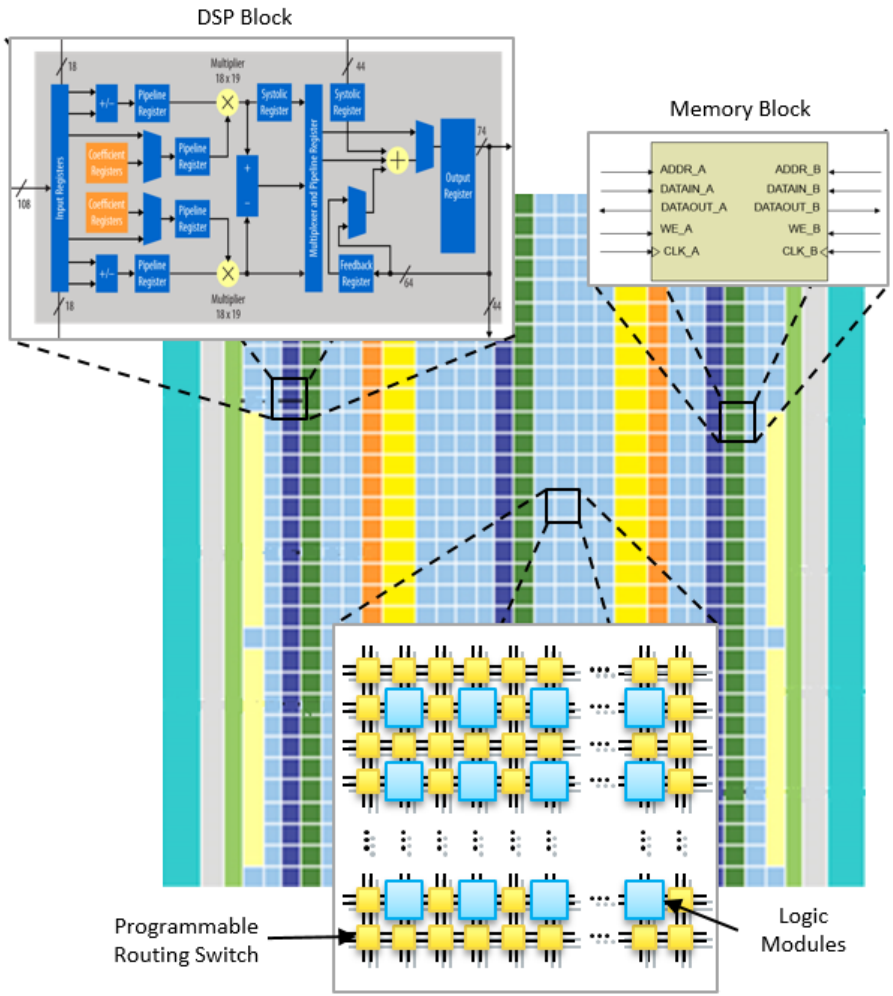
GPU

- Many small cores organized into groups that run in lock-step
- Multi TFLOPs per device, hard to get near peak
- Programming models and languages: OpenMP, CUDA, OpenCL, OpenACC
- Wide data parallelism yields maximum performance - requires memory coalescing and minimal divergence

Architecture Review (cont.)

FPGA

- Reprogrammable device consisting of DSPs, multipliers, registers, hundreds of thousands of logic elements
- Multi TFLOPs per device, possible* to get near peak performance
- Programming models and languages: OpenCL, initial OpenMP support
- Long pipelines yield highest performance - multiple streams progressing through the long pipeline concurrently; data parallel can also be effective



htt

BFS

Degree and diameter matters.

- Traversal of scale free graphs introduces additional complexities $\rightarrow$ specific algoritms to balance degree disparity.
- Traversal of low diameter graphs have larger frontiers and fewer synchronizations.

The selected method depends on architecture and graph characteristics

- Manycore - serial and shared memory parallel vertex-based, edge-based, push (vertices in the frontier push to next frontier), pull (all unvisited vertices check to for inclusion in next frontier)
- GPU - shared memory parallel versions of manycore procedures with data structure optimizations for coalesing, algebraic methods
- FPGA - all of the above... but pipelined push procedure may be best if pipeline loop and stalls (ii) can be minimized

Our Approach

We will focus on hypergraphs created from unstructured meshes

- Downward adjacencies have uniform degree! ... not so much for upward adjacencies
- Diameter greatly depends on the geometric model and the partition - high quality parts are compact and have low diameter

Use Sell-C-Sigma datastructure for coalesing

- Need weights for vertices and edges - lose some savings but retain improved vectorization
- Sorting (sigma) will be needed for meshes with semi-structured boundary layers (tet and wedge dominant with a handful of pyramids) and discrete event simulation graphs

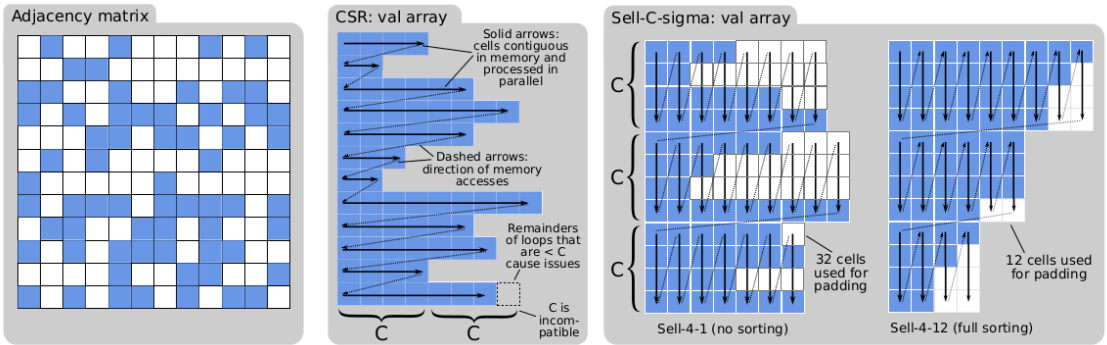
Quantifying progress

- Time to solution, energy usage, and GTEPS

Algorithms

The following variations of BFS were tested

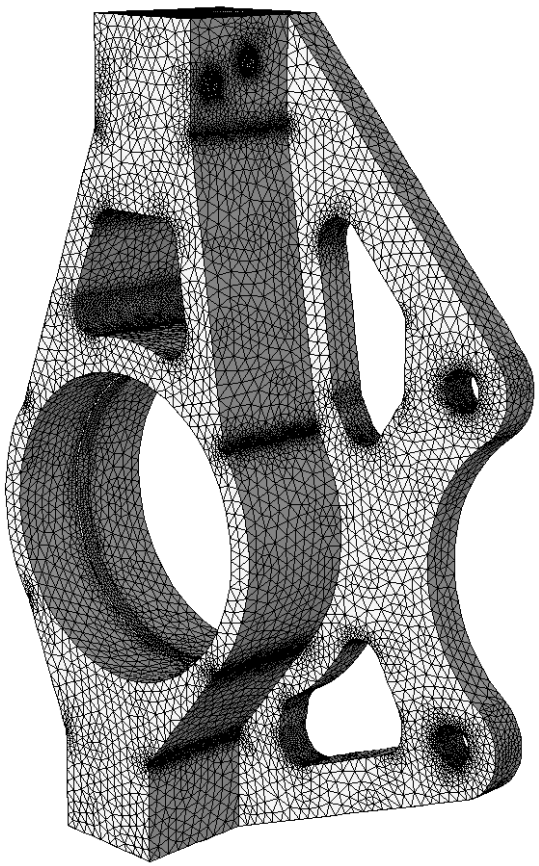
- push - C++ serial push
- pull - C++ serial 'pull' style vtx→hyperedge
- csr - OpenCL parallel 'pull' style vtx→hyperedge using CSR data structure
- scg - OpenCL parallel 'pull' style vtx→hyperedge using Sell-C-Sigma data structure
- `*_int` - use 4B int for adjacency and degree lists instead of 8B long
- `*_unroll` - unroll the loop over hyperedges adjacent to a vertex



Sell-C-Sigma data structure (Besta and Merending, et al.)

Test Hypergraphs

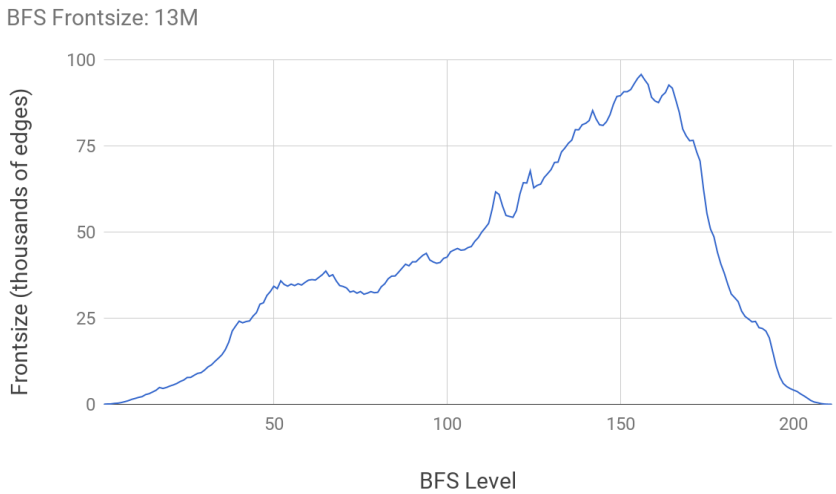
Hypergraphs created from unstructured tet mesh of automotive part:
mesh elements $\rightarrow$ graph vertices
mesh vertices $\rightarrow$ hyperedges



400k element mesh

graph	vertices	hyperedges	bfs levels	max frontier
67k	66433	15697	25	4432
190k	192728	40052	40	7992
400k	404613	88651	48	17316
890k	890925	187380	70	26664
1.6M	1580611	336215	82	45268
13M	12831104	2499193	82	95798
28M	27943315	5190006	210	

Hypergraphs



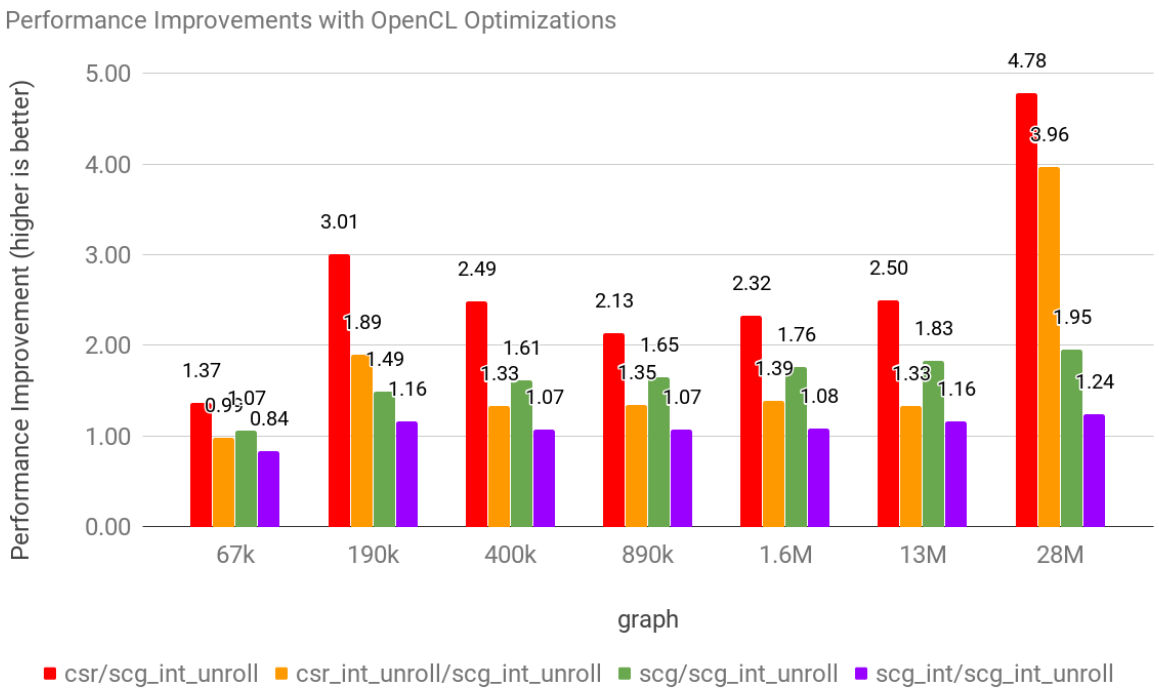
BFS Results

Timing comparison of different OpenCL BFS kernels on NVIDIA 1080ti

- Includes data transfers, but not kernel JIT compilation; average of three runs shown

`scg_int_unroll` is 4.78 times faster than csr on 28M graph and up to 11 times faster than serial push on Intel Xeon (not shown)

Memory coalescing is critical; `csr_int_unroll/scg_int_unroll`



Closing Remarks

EnGPar builds off of ParMA by:

- generalizing the diffusive load balancing algorithm for applications without an element-based unstructured mesh.
- improving the algorithm of key portions to improve overall runtime.

MPI tests show:

- EnGPar reduces the high mesh vertex imbalance from the 512Ki mesh from 1.53 to 1.13.
- EnGPar maintains the mesh element imbalance at the tolerance.
- EnGPar runs around 33% faster than ParMA.

Initial acceleration tests indicate:

- Utilizing known degree of downward adjacencies enables loop unrolling and improved load balance
- Accelerating other EnGPar graph methods could provide significant performance improvements $\rightarrow$ code rewrite, adjust abstraction layers
- GPU kernel on FPGA (Arria 10) is slower and uses more power than GPU... low hanging fruit remains for pipelined kernel

Future Work

Expanding the capabilities of EnGPar:

- Continue improving partition quality when parts are small (i.e., strong scaling)

Acceleration:

- Support migration - host communicates, device rebuilds (hyper)graph
- Focus on pipelined kernel implementation
- Test on bigger graphs from meshes and discrete event simulation

Applying EnGPar to other applications:

- CODES - a discrete event simulation for communication on supercomputer networks. `press3.mcs.anl.gov/codes/`
- FUN3D - a computational fluid dynamic simulation using a vertex-based partitioned mesh. `fun3d.larc.nasa.gov`
- PHASTA - massively parallel computational fluid dynamics. `github.com/PHASTA/phasta`

Thank You

Questions?

Acknowledgements:

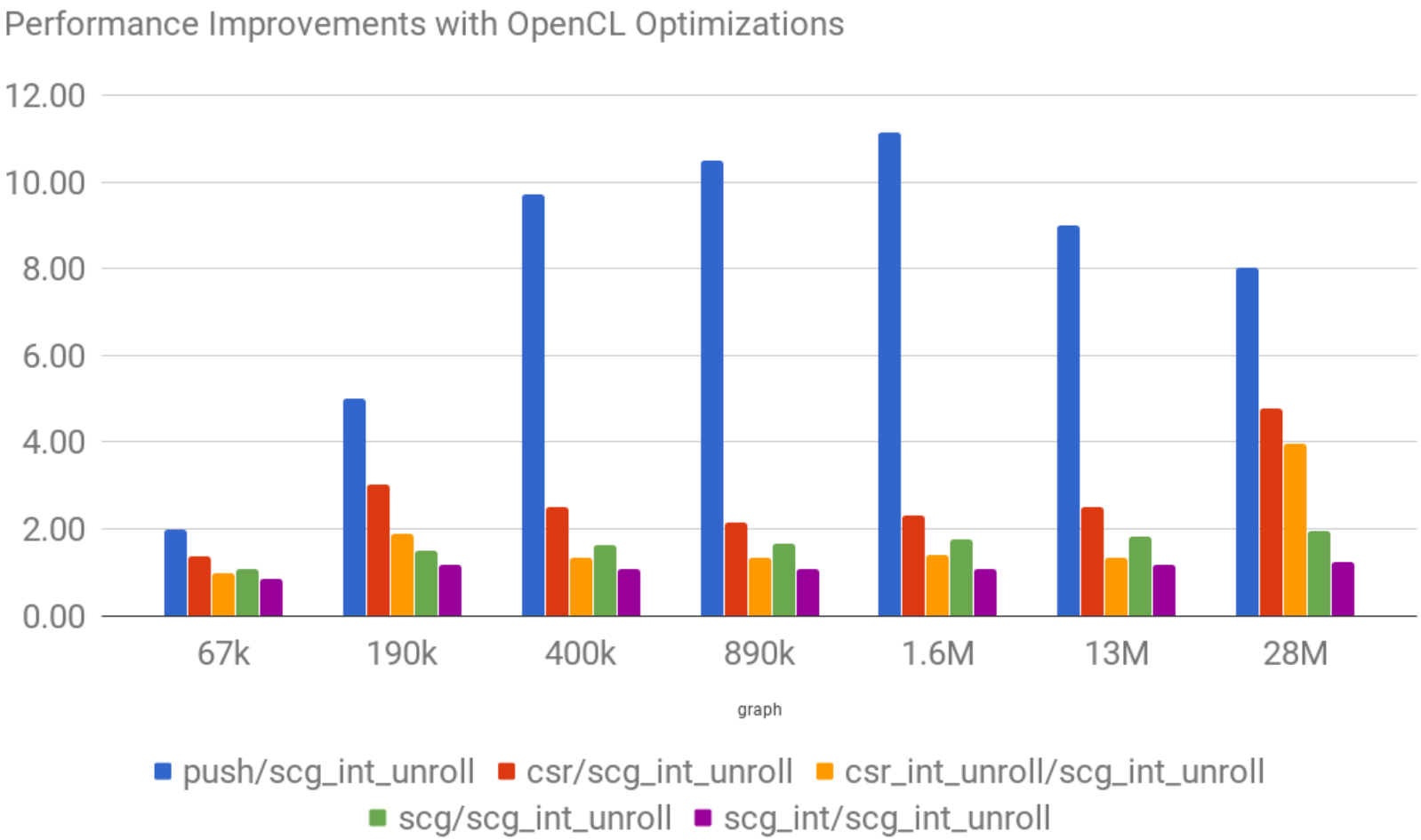
- NSF SI2-SSE: Fast Dynamic Load Balancing Tools for Extreme Scale Systems
- DOE FASTMath SCIDAC Institute
- CEED ECP
- Argonne National Laboratory - Kazutomo Yoshii

Extra Slides

Other View of Results

Timing comparison of different kernels on NVIDIA 1080ti - includes sequential push

- Includes data transfers, but not kernel JIT compilation
- The average of three runs is used for the comparison
- Higher is better



Gripes

OpenCL

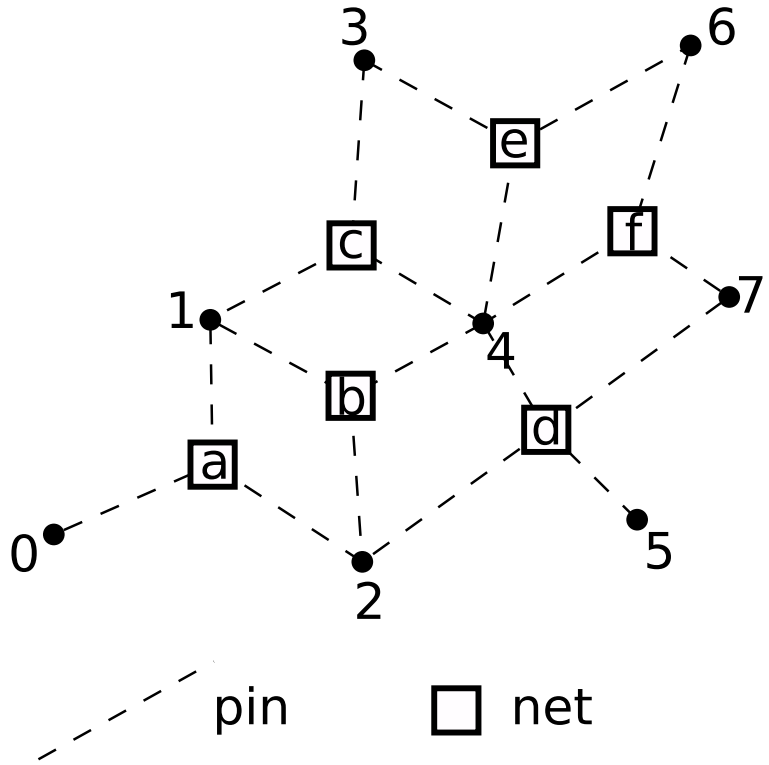
- Isn't supported on Power9+V100 deployments yet - can't build kernels
- OCLgrind was the only useful debugging tool I could find (github.com/jrprice/Oclgrind)
- Shiny (not so) new language features (e.g., team collectives) are not supported by devices I tested on

FPGAs

- 5hr+ compile time
- steep learning curve for development

Example of N-Graph CSR

Our current N-Graph data structure is two sets of compressed sparse row (CSR) structures



vtx id	0	1	2	3	4	5	6	7	
vtx-to-net offsets	0	1	4	7	9	14	15	17	19
vtx-to-net adjacencies	a	a,b,c	a,b,d	c,e	b,c,d,e,f	d	e,f	d,f	
net id	a	b	c	d	e	f			
net-to-vtx offsets	0	3	5	8	12	15	18		
net-to-vtx adjacencies	0,1,2	1,2	1,3,4	2,4,5,7	3,4,6	4,6,7			

BFS SCG Kernel and Loop

Kernel:

```
1  typedef int lid_t;
2
3  #define checkEdge(edge, depth, minDepthEdge, minDepth) { \
4    const int d = (depth[(edge)]);                         \
5    if ( d != -1 ) { /*visited*/                           \
6      if ( d < minDepth) {                                 \
7        /* this edge has the lowest */                     \
8        /* depth of those adjacent to vtx 'gid' */         \
9        (minDepthEdge) = (edge);                           \
10       (minDepth) = d;                                    \
11     }                                                    \
12   }                                                      \
13 }                                                        \
14
15 #define updateDepth(edge, depth, nextDepth, frontSize) { \
16   const int d = (depth[(edge)]);                         \
17   if ( d == -1 ) {  /*not visited*/                      \
18     (depth[(edge)]) = nextDepth;                         \
19     atomic_inc((frontSize));                             \
20   }                                                      \
21 }                                                        \
22
23 kernel void bfsScgKernel(global lid_t* restrict degreeList,
24                          global lid_t* restrict edgeList,
25                          global int* restrict depth,
26                          global int* restrict frontSize,
27                          const int numVerts,
28                          const int level)
29 {
30   lid_t chunkLength = get_local_size(0); // number of verts in a chunk
31   lid_t chunk = get_group_id(0); // which chunk is this work-item in
32   lid_t lid = get_local_id(0); // which vert in the chunk is assigned to this work-item
33   lid_t gid = get_global_id(0); // global vtx id
34   const lid_t LARGE_DEPTH = 1024*1024*1024;
35
36   //no work here - just a padded global entry
37   if( gid >= numVerts ) return;
38
39   const lid_t chunkStart = degreeList[chunk];
40   const lid_t maxChunkDeg = (degreeList[chunk+1] - degreeList[chunk])/chunkLength;
41   const lid_t chunkSize = chunkLength*maxChunkDeg;
42   const lid_t firstEdgeIdx = chunkStart+lid;
43   const lid_t lastEdgeIdx = firstEdgeIdx+chunkSize;
```

```
45   // loop through the edges adjacent to the vertex and find
46   // the one with the smallest depth
47   lid_t minDepthEdge;
48   int minDepth = LARGE_DEPTH;
49   for (lid_t j = firstEdgeIdx; j < lastEdgeIdx; j += chunkLength) {
50     const lid_t edge = edgeList[j];
51     // skip padded entries/edges
52     if (edge == -1) continue;
53     checkEdge(edge, depth, minDepthEdge, minDepth);
54   }
55
56   if (minDepth == level) {
57     const int nextDepth = level+1;
58     // a visited edge was found - loop through the adjacent
59     // edges again and set the depth of unvisited edges
60     for (lid_t j = firstEdgeIdx; j < lastEdgeIdx; j += chunkLength) {
61       const lid_t edge = edgeList[j];
62       // skip padded entries/edges
63       if (edge == -1) continue;
64       updateDepth(edge,depth,nextDepth,frontSize);
65     }
66   }
67 }
```

Loop:

```
221     cl::NDRange global(pg->num_vtx_chunks*pg->chunk_size);
222     cl::NDRange local(pg->chunk_size);
223
224     int maxLevel=1000;
225     int level=start_depth;
226     int h_frontSize;
227     do {
228       h_frontSize = 0;
229       // run the kernel once for each chunk
230       cl::Buffer* d_frontSize = copyToDevice<int>(
231           &h_frontSize,
232           1,
233           CL_MEM_WRITE_ONLY);
234       cl::Event e = bfsScgKernel(cl::EnqueueArgs(*engpar_ocl_queue, global, local),
235           *d_degreeList, *d_edgeList, *d_depth, *d_frontSize, pg->num_local_verts, level);
236
237       monitorPower(e);
238
239       copyFromDevice<int>(d_frontSize, &h_frontSize, 1);
240       printf("level %d frontSize %d\n", level, h_frontSize);
241       level++;
242     } while(h_frontSize && level < maxLevel);
243
244     copyFromDevice<int>(d_depth, in->visited, pg->num_local_edges[t]);
```

Power Consumption scg_int_unroll

device	1080ti	Arria 10	
graph	total power (W)	total power (W)	arria10/1080i
67k	1.17	0.52	0.45
190k	1.59	1.86	1.17
400k	1.68	4.40	2.61
890k	3.46	13.07	3.78
1.6M	5.64	26.48	4.69
13M	60.50	488.22	8.07
28M	151.32	1535.60	10.15

Based on GPU draw of 215W and FPGA draw of 30W
Measured with `nvidia-smi` and `aocl_mmd_getinfo`

Run time

device	i7	1080ti	Arria 10	i3
C		64	64	64
graph	push bfs (s)	scg_int_unroll (s)	scg_int_unroll (s)	scg_int_unroll (s)
67k	0.011	0.005	0.017	0.013
190k	0.037	0.007	0.062	0.069
400k	0.076	0.008	0.147	0.150
890k	0.169	0.016	0.436	0.530
1.6M	0.292	0.026	0.883	1.196
13M	2.535	0.281	16.274	22.727
28M	5.639	0.704	51.187	OOM

Includes data transfers, but not kernel JIT compilation

The average of three runs is used for the comparison