Accelerated Load Balancing of Unstructured Meshes

Gerrett Diamond, Lucas Davis, and Cameron W. Smith

Abstract Unstructured mesh applications running on large, parallel, distributed memory systems require the computational work related to mesh entities to be evenly distributed across processes in order to achieve maximal performance. To efficiently balance meshes on systems with accelerators the balancing procedure must utilize the accelerator. This work presents algorithms and speedup results using OpenCL and Kokkos to accelerate critical portions of the EnGPar diffusive load balancer.

1 Introduction

While common partitioning techniques such as multilevel or geometric methods are good for creating an initial distribution of load, those techniques are not as applicable to simulations where the mesh and load changes. These evolving simulations require dynamic load balancing techniques that are quick to improve the partition. Diffusive load balancing methods allow quick partition refinement for the relatively small changes to imbalance that are seen in adaptive mesh simulations. EnGPar's diffusive balancer has been shown to quickly produce high quality partitions at up to 512Ki processes [2].

2 EnGPar Dynamic Load Balancing

EnGPar is a partition improvement tool that utilizes a multi-hypergraph, called the N-graph, to describe the portions of the mesh that require load balancing. The N-

Gerrett Diamond e-mail: diamog@rpi.edu · Lucas Davis e-mail: davisl3@rpi.edu · Cameron W. Smith e-mail: smithc11@rpi.edu
Rensselaer Polytechnic Institute, Troy, NY

graph consists of vertices which represent the primary dimension entities of the mesh. The vertices are connected by hyperedges created from the secondary dimensions of the mesh that require load balancing.

EnGPar's diffusive algorithm is an iterative local refinement strategy. In each iteration the target criteria is improved until the imbalance is under a given tolerance or the imbalance cannot be improved further. Each iteration consists of three steps: targeting, selection, and migration. The targeting step gathers metrics on the part and its neighbors in order to determine which neighboring parts to send weight to and how much weight to send. The selection step is where graph vertices on the boundary are chosen to be sent to neighboring parts in order to satisfy the weights determined by the targeting phase. Finally, the migration step sends the graph entities that were selected to the destination parts and the graph is reconstructed.

In this work, we target accelerating distance computation and cavity selection. These two procedures consume up to 50% of the total execution time and are well suited to acceleration as they do not require inter-process communications [2].

Distance computation is performed during selection by ordering hyperedges on the boundary based on their distance from the center of the part from furthest to closest. EnGPar computes this distance with two breadth first traversals of the graph. The first traversal starts at the part boundary and works its way in while marking the depth of visited hyperedges. The second traversal starts from a hyperedge with the largest depth and works its way out to the boundary while marking the distance from the starting point.

Cavity selection determines if a cavity, defined as a hyperedge and the vertices that are connected by it, on the part boundary should be sent to one of the neighboring parts. A cavity is selected for migration if (1) the part that the hyperedge is shared with is a target part, (2) the target part has not been sent more weight than the limit, and (3) the size of the cavity is small.

3 Accelerating Distance Computation

Distance computation's breadth first traversal is accelerated with an OpenCL data-parallel ND-Range kernel for execution on GPUs and many-core processors. The host process calls the kernel once for each frontier in the traversal. The kernel implements a 'pull' based approach by iterating over the graph vertices pinned to each hyperedge twice. The first iteration determines if in the previous kernel call which vertices were updated. If a vertex is found, then the second iteration updates the distance of the other vertices.

The baseline OpenCL implementation uses a compressed sparse row (CSR) hypergraph representation and a pull based traversal. In Figure 1 the performance of optimized implementations relative to the baseline 'csr' implementation are shown. 'scg' in the name of the implementation indicates that use of the Sell-C-σ data structure [3], 'int' indicates use of four byte ints instead of eight byte ints, and 'unroll' indicates manual vertex loop unrolling. Runs were executed on graphs created from

meshes of the 2014 RPI Formula Hybrid suspension upright with up to 28M (million) tetrahedron (DOI: 10.5281/zenodo.1194576). All tests were executed on an NVIDIA 1080ti using CUDA 9.2. The chunk size of the 'scg' tests was fixed at 64; given the uniform degree of the element to vertex adjacencies, there was little performance difference between different chunk size settings. The given results are the average of three runs and include data transfers to and from the GPU. The OpenCL JIT compilation is not included in the timing as this one-time cost would be amortized across an entire run.

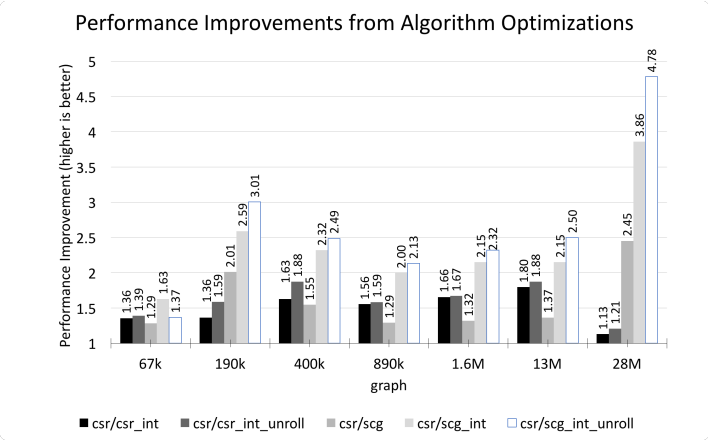

Fig. 1 Performance of breadth first traversal implementations.

On the 28M mesh, the 'scg_int_unroll' is 11 times faster than the serial, C++, push implementation, and 4.78 times faster than the 'csr' implementation. The performance boost given by loop unrolling and use of Sell-C-σ are the result of improved memory coalescing. Reducing the integer size by half improves performance by 24% for the 28 million element mesh.

4 Accelerating Cavity Selection

Accelerating the selection of cavities requires simultaneously evaluating many cavities. The current single threaded selection procedure evaluates cavities in order of their descending distance from the topological center. Since the ordered selection exposes no concurrency an alternative application of the topological distance is needed. The proposed approach applies a parallel topological distance sorting after a coloring based parallel cavity evaluation has executed.

Critical to concurrent cavity evaluation is avoiding race conditions when deciding which part to migrate a given graph vertex. Hyperedge coloring ensures that any two

hyperedges that share a common vertex will be assigned a different color, and thus entire sets of like-colored hyperedges can be evaluated concurrently.

The Kokkos-kernels graph coloring procedure [1] is used to color the hyperedges of the EnGPar hypergraph. This procedure is driven by a symmetric adjacency matrix. To color hyperedges we must create the hyperedge-to-vertex-to-hyperedge graph; the dual of the hypergraph. The dual graph has one vertex for each hyperedge, and an edge between two hyperedges if they share at least one common vertex.

The construction of the dual is listed in Algorithm 1. It starts by making a set, using a Kokkos `unordered_map`, that stores hyperedge-to-hyperedge adjacencies (l.2-l.12). A parallel reduction and prefix sum then compute the degree list `deg` (l.14-l.18). The hyperedge list, `edgeList`, is then filled with a parallel loop over the hyperedges. This loop utilizes a Kokkos atomic read-then-increment operation (l.23) to determine the position of the hyperedge in the list of adjacent hyperedges. The resulting CSR, `deg` and `edgeList`, are passed directly to the Kokkos coloring procedure.

Algorithm 1 Dual Graph Converter

```
 1: procedure DUAL(G = (V,E))
 2:     n = 0
 3:     parallel for v ∈ V do
 4:         for all (i,v) ∈ E do
 5:             for all (j,v) ∈ E \ {(i,v)} do
 6:                 n++
 7:     //n is an upper bound on the set's size
 8:     set of int pair m (n)
 9:     parallel for v ∈ V do
10:         for all (i,v) ∈ E do
11:             for all (j,v) ∈ E \ {(i,v)} do
12:                 m. insert ( (i, j) )
13:     N = |E|
14:     deg = [N + 1]
15:     parallel for k ∈ m do
16:         deg(k.first+1)++
17:     parallel for i = 0,1,...,N do
18:         deg[i] = sum(deg[0 : i])
19:     edgeList = [deg[N]]
20:     degreeCount = [N]
21:     parallel for k ∈ m do
22:         e = deg[k.first]
23:         i = degreeCount[k.first]++
24:         edgeList[e+i] = k.second
25:
```

The speedup of the dual and coloring procedures relative to serial implementations is shown in Figure 2. Tests were executed on the same system and series of graphs used in Section 3. Construction of the dual has a nearly flat speedup relative to the graphs. Conversely, the speedup of Kokkos coloring improves with graph size. Profiling of these tests is required to determine how effectively GPU resources are utilized and identify bottlenecks.

5 Closing Remarks

Speedup results of two critical procedures used by EnGPar were demonstrated. Parallel distance computation performance is over an order of magnitude greater and

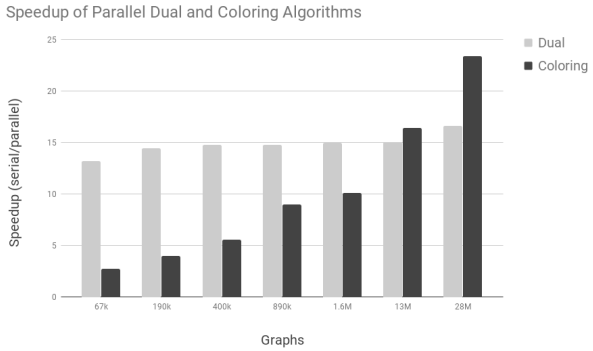

Fig. 2 The ratio of serial execution time to parallel for dual graph construction and graph coloring.

is a direct replacement for the existing procedure. Parallel coloring for the selection process demonstrates high performance across a range of mesh sizes. Ongoing work is focused on fully integrating these advances into the production version for evaluation of overall performance and partition quality.

Acknowledgements This research was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award DE-AC52-07NA27344 (FASTMath SciDAC Institute) and by the National Science Foundation under Grant No. ACI 1533581, (SI2-SSE: Fast Dynamic Load Balancing Tools for Extreme Scale Systems). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. DEVECI, M., BOMAN, E. G., DEVINE, K. D., AND RAJAMANICKAM, S. Parallel graph coloring for manycore architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2016), pp. 892–901.
2. DIAMOND, G., SMITH, C. W., AND SHEPHARD, M. S. Dynamic load balancing of massively parallel unstructured meshes. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (New York, NY, USA, Nov. 2017), ScalA '17, ACM, pp. 9:1–9:7.
3. KREUTZER, M., HAGER, G., WELLEIN, G., FEHSKE, H., AND BISHOP, A. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing 36*, 5 (2014), C401–C423.