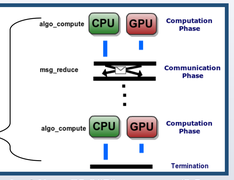
Exploring Hybrid Hardware and Data Placement Strategies for the Graph 500 Challenge

Experimental Platform

- The software we use is called TOTEM.
- TOTEM is a graph processing framework that simplifies the task of implementing graph algorithms for single-node platforms that include processing elements with separate memory spaces (e.g. discrete GPUs).
- We use a level synchronized Breadth First Search algorithm.
- The CPU and GPU each have their own kernel.
  - They are based on the implementations proposed by Hong et al. (See poster paper)
- TOTEM uses the Bulk Synchronous Parallel (BSP) model and provides key functionalities to implementers:
  - A common data representation,
  - Ghost nodes and their synchronization,
  - Hooks to callbacks that implement algorithm-specific CPU and GPU kernels,
  - Common optimizations (e.g. message aggregation, and overlapping computation with communication).
- TOTEM is open source and these kernels can be viewed online.
  - netsyslab.ece.ubc.ca/wiki/index.php/Totem

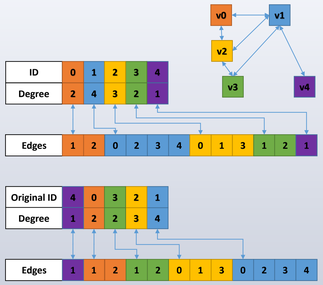
```
totem_config_t config = {
  graph,
  partitioning_strategy,
  algo_compute_func,
  msg_reduce_func,
  ...
};

totem_config(&config);
totem_execute();
```



- Our machine has two Intel Xeon E5-2670 processors (2.5 GHz, 10 cores, 512GB memory), and two Nvidia Kepler K40's (745 MHz, 2880 cores, 12GB memory).
- We generate graphs as for Graph500 (Similar to the Kronecker graph generator.) Graphs are power-law with an average node degree of 16.
- We focus on graphs of Scale 30: this means 1 billion vertices and 16 billion undirected edges.

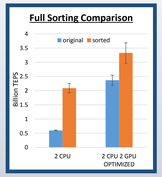
Memory Layout of the Graph

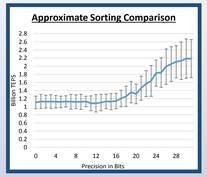
- We use the Compressed Sparse Row format to represent graphs, which is memory efficient. The vertices are represented as an array with the values being the degree of the vertex. The location of the vertices neighbours is the prefix sum of all previous vertices.
- The vertex ID is initially implicit, it is the index of the vertex. Each undirected edge in the graph is represented as two directed edges.
- The arrows here represent the location of a given vertices' edges on the array, based off the sum of degrees before the vertex. (A prefix sum)
- We can sort the CSR format by each vertices' degree, which rearranges the structure of the edge list. When we sort the vertices, we are only reordering the memory space - we keep the original graph by holding onto the original ID.
  - This will allow us to map back the result after executing an analysis on the graph.



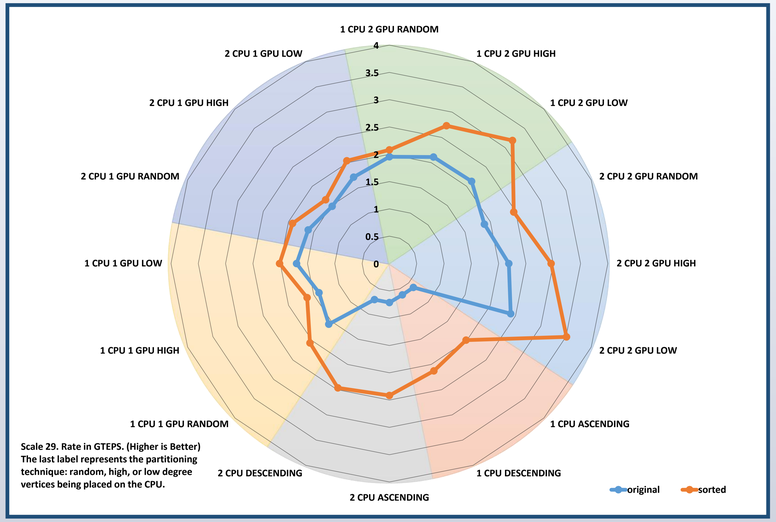
The Impact of Sorting

- We can sort during hybrid partitioning's setup phase, at negligible cost.
- Fully sorting the graph improves every type of configuration.
- We compared the precision of vertex sorting, by using a MSB radix-sort with variable precision.
  - Sorting using the first half of the bits barely showed improvement.
  - The results only approached the fully sorted improvement near complete precision.
  - We expect locality and caching to be responsible for this result.





The Configuration Space



Scale 29. Rate in GTEPS. (Higher is Better)
The last label represents the partitioning technique: random, high, or low degree vertices being placed on the CPU.

Configuration Analysis

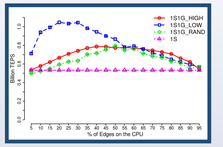
Lessons Learned

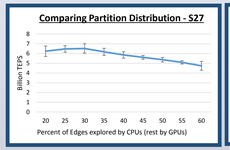
- Sorting graphs by vertex degree provides a boost to all configurations, and can be easily implemented with a memory map of the original ID.
- Intelligent partitioning of the graph is important, using 2 CPU's and 2 GPU's and randomly partitioning is no better than just using the 2 CPU's if we can sort!
- The hybrid configurations can take advantage of partitioning structure to achieve better performance on non-sorted graphs.

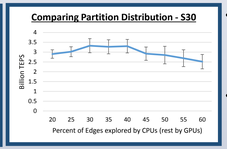
Going Forward

- The configuration space is large and significantly effects performance depending on many factors:
  - The partitioning of vertices across devices.
  - The order in memory of the graph arrays.
  - The memory access strategy for devices.
  - The degree of the vertex being explored.
- There are also some factors that are still available to explore:
  - A more fine grained distribution across devices.
  - The effect of the average node degree.
  - Does this sorting method help other algorithms?

Partitioning Across CPUs and GPUs

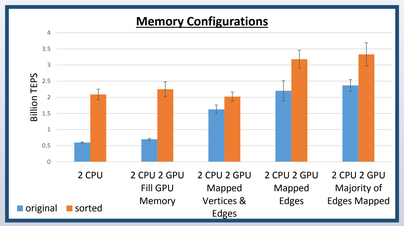






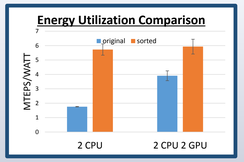
- When partitioning, our goal is to match hardware with workloads they can best fit to.
- A previous paper from our lab gave some initial intuition that if we can execute a large amount of the work on the GPU, it is beneficial to place the LOW degree vertices on the CPU. (Scale 25 undirected – similar machine)
- Our work confirms that the results hold true as we increase graph size.
- There is a sweet spot with ~30% of edges being explored by CPUs (and the rest by the GPUs)
  - The GPU is able to process a high degree vertices' many neighbours in parallel.
- These two graphs show Scale 27 and 30. The trend for this sweet spot continues to hold for our hardware across this full order of magnitude.

Using CUDA Mapped memory



- As we increase scale, the percentage of edges that can be offloaded to the GPU's device memory becomes tiny.
- We use mapped memory to overcome this limitation at the expense of increased communication overhead.
- If we memory map both the vertex array and the edge array, the hybrid platform performs worse on sorted graphs than if we just used the CPUs only!
- If we can at least keep the vertex array in the GPU memory, and map edges, we can achieve a boost in performance over the CPUs.
  - Filling the remaining GPU memory with a portion of the edges can give us another minimal boost.

The Energy Performance



- This graph shows a quick analysis of the performance metric of MegaTEPS per Watt, used in the Green Graph 500 challenge.
- We can see the results for adding GPUs is relatively close energy-wise in comparison to just CPUs in the best case.
- We believe the time and effort spent using Mapped Memory to the host DRAM contributes to not seeing a boost in energy savings when using GPUs.

Scott Sallinen, Daniel Borges, Abdullah Gharaibeh, Matei Ripeanu
netsyslab.ece.ubc.ca
Electrical and Computer Engineering
The University of British Columbia