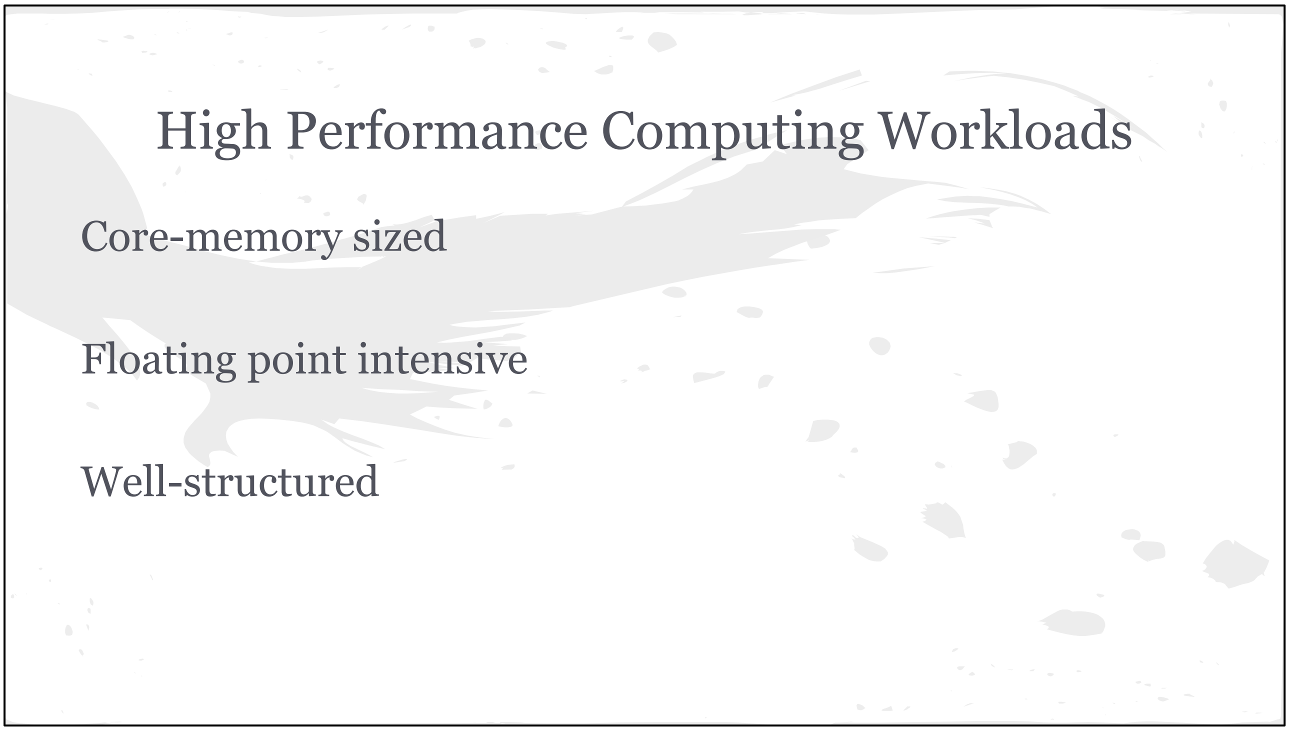
Challenges in large-scale graph processing on HPC platforms and the Graph500 benchmark

by
Nkemdirim Dockery



High Performance Computing Workloads

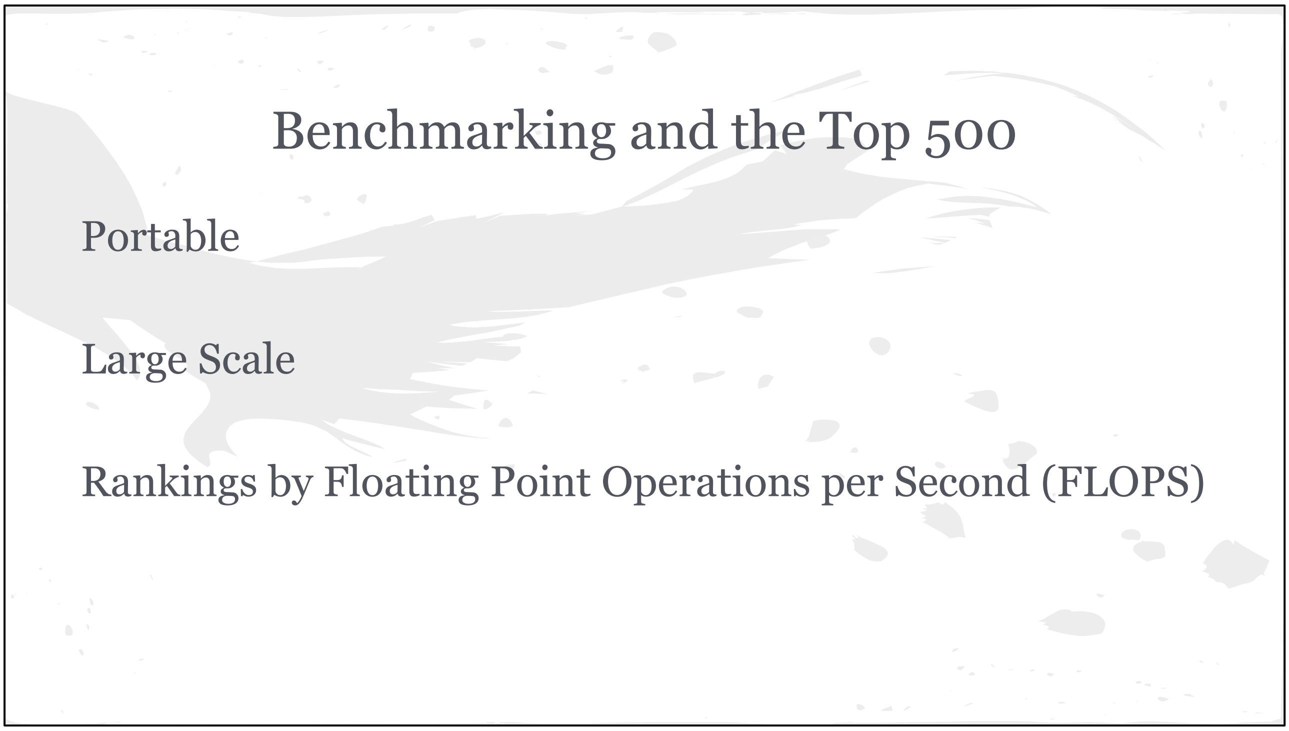
Core-memory sized

Floating point intensive

Well-structured

def high performance computing
    zone in on parallel/distributed nature and the amount of communication
describe conventional workload

High performance computing typically refers to the connection of a large amount of processing nodes over a high bandwidth, low latency network.
The most common model of machine for high performance computing is the distributed memory model, where the processing nodes are fairly independent. Programming for this model is typically done using MPI. These programs have processing nodes alternating between working independently on local data and coordinating with other nodes for collective communication. Data is usually transmitted at pauses between computations, and while this shrinks overall latency costs, it does this at the expense of more fine grained control. This loss of algorithmic control is not usually a problem for a typical HPC problem such as a physics simulation, which are core-memory sized, floating point intensive, and well-structured, because programs can take advantage of the temporal and spatial locality inherent in these problems.



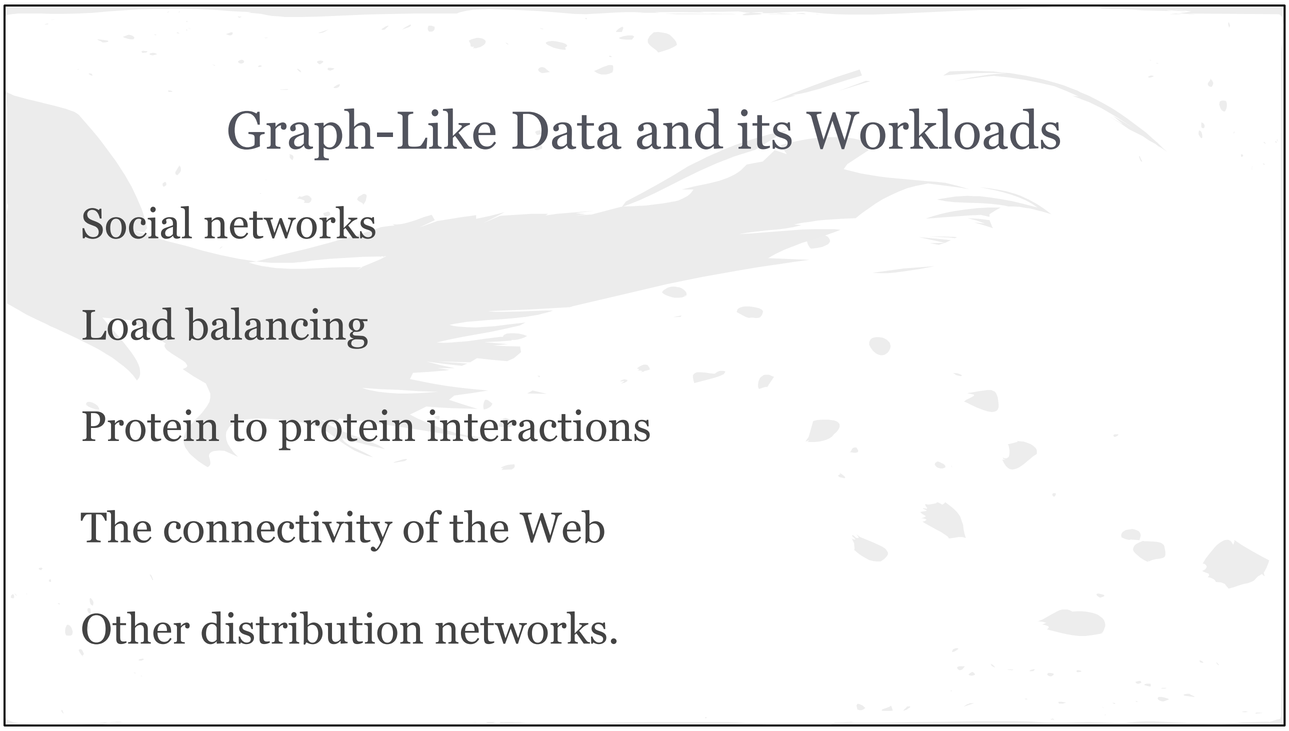
Benchmarking and the Top 500

Portable

Large Scale

Rankings by Floating Point Operations per Second (FLOPS)

Benchmarking in terms of high performance computing is essentially the process of taxing an untested system with workloads that are both general enough to apply across development environments and use cases, and large enough to give an indication of an untested system's capabilities. The Top 500 project aims to provide a reliable basis for tracking and detecting trends in high-performance computing and bases rankings on HPL,[1] a portable implementation of the high-performance LINPACK benchmark written in Fortran for distributed-memory computers. **HPL** is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. The HPL package provides a testing and timing program to quantify the **accuracy** of the obtained solution as well as the time it took to compute it. The best **performance** achievable by this software on your system depends on a large variety of factors.but with some assumptions on the interconnection network, the HPL implementation is **scalable** in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage. This means that as the expected capabilities of untested machines rises, the benchmarking implementation can successfully scale up in order to provide an appropriate workload.



Graph-Like Data and its Workloads

Social networks

Load balancing

Protein to protein interactions

The connectivity of the Web

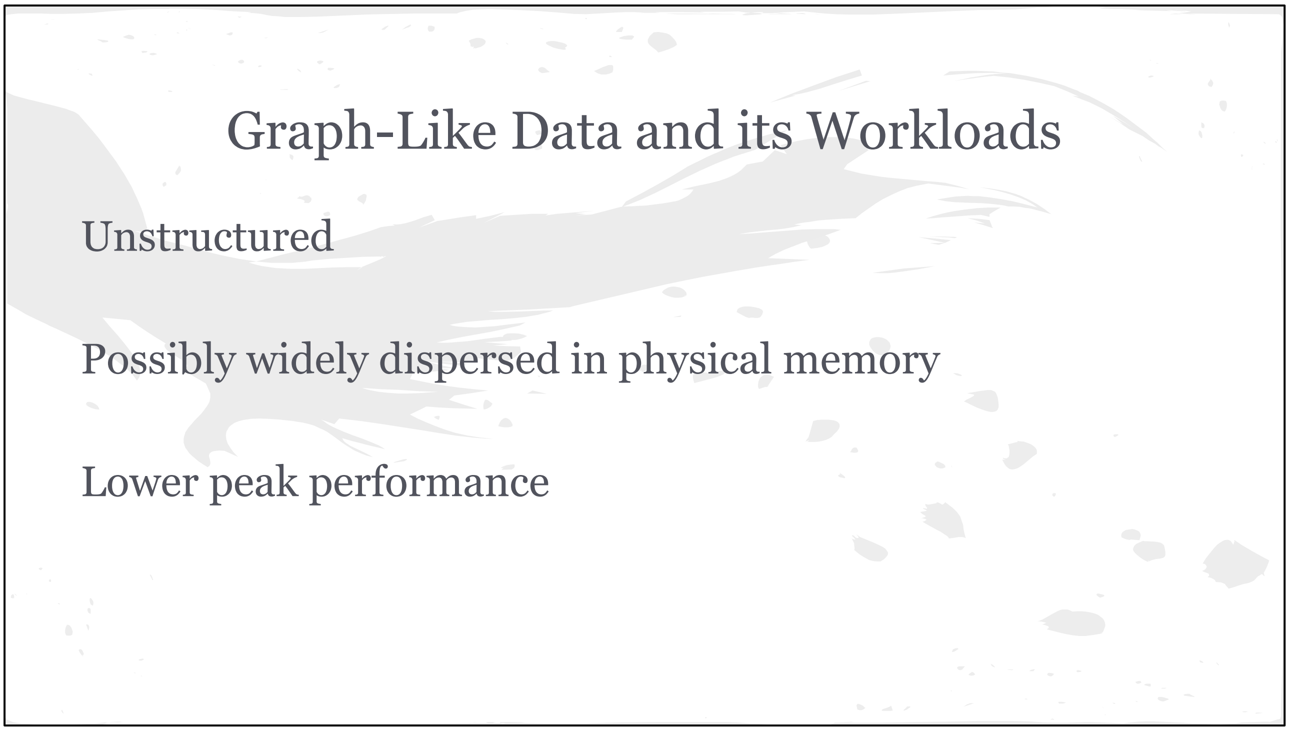
Other distribution networks.

describe emergence of graph like data

    zone in on unstructured nature

Graph like data comes out of the analysis of real world phenomenon such as social networks, load balancing, protein to protein interactions, the connectivity of the Web, and other distribution networks.

Graphs are applicable in such diverse settings because they're an abstract way of describing interactions between entities.
The data itself represents the interactions between discrete points. The purpose surrounding these emerging datasets is not typically to ask a "what if," but to search through the dataset in order to discover hypotheses to be tested. While these graphs allow for a lot of insight into a large amount of data, they tend to be out of core, integer oriented, and unstructured.



Graph-Like Data and its Workloads

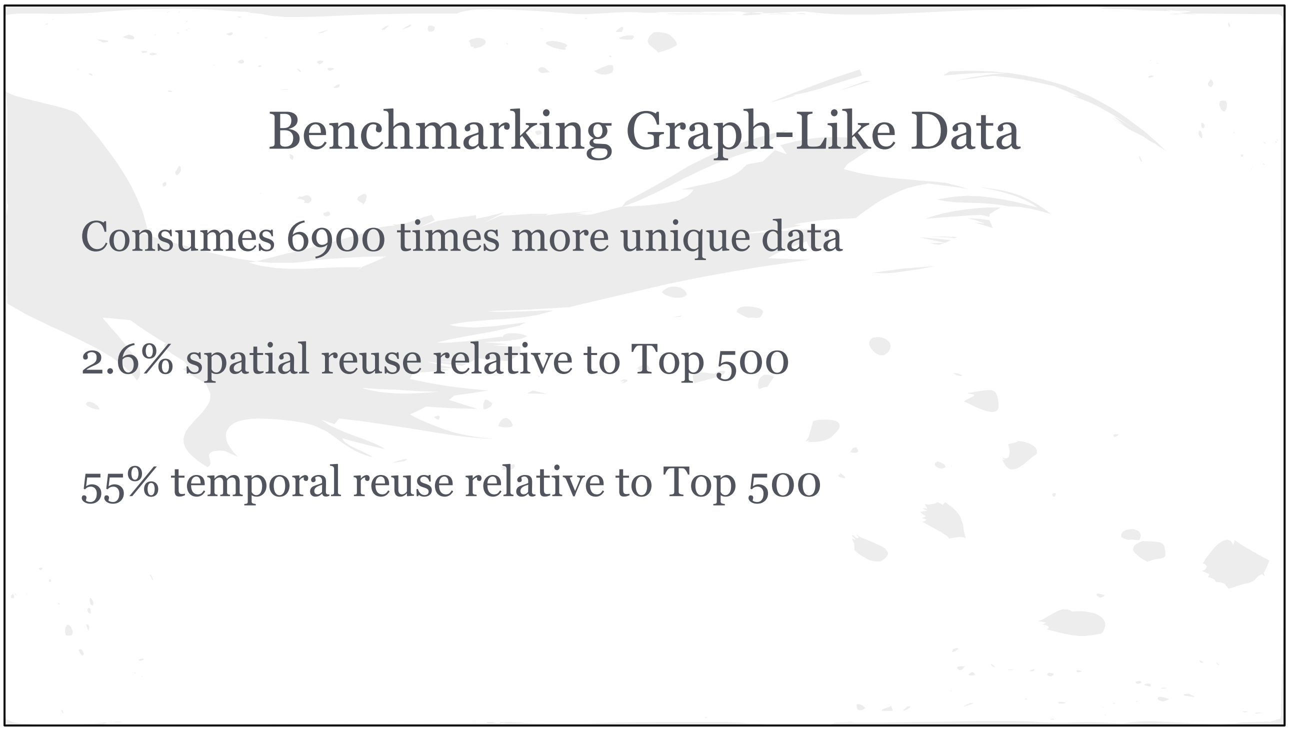
Unstructured

Possibly widely dispersed in physical memory

Lower peak performance

describe stress unstructured nature places on communication

In many emerging applications such as social and economic modeling, the graph has very little exploitable structure. In fact, in such settings, a vertex v's neighbors can be widely dispersed in global memory. This leads to data access patterns that make very poor use of memory hierarchies, which in turn can result in idle processors. Because access patterns are data dependent, since they are a function of the graph's edge structure, standard prefetching techniques are often ineffective. In addition, graph algorithms typically have very little work to do when visiting a vertex, so there is little computation for each memory access. For all these reasons, graph computations often achieve a low percentage of theoretical peak performance on traditional processors.



Benchmarking Graph-Like Data

Consumes 6900 times more unique data

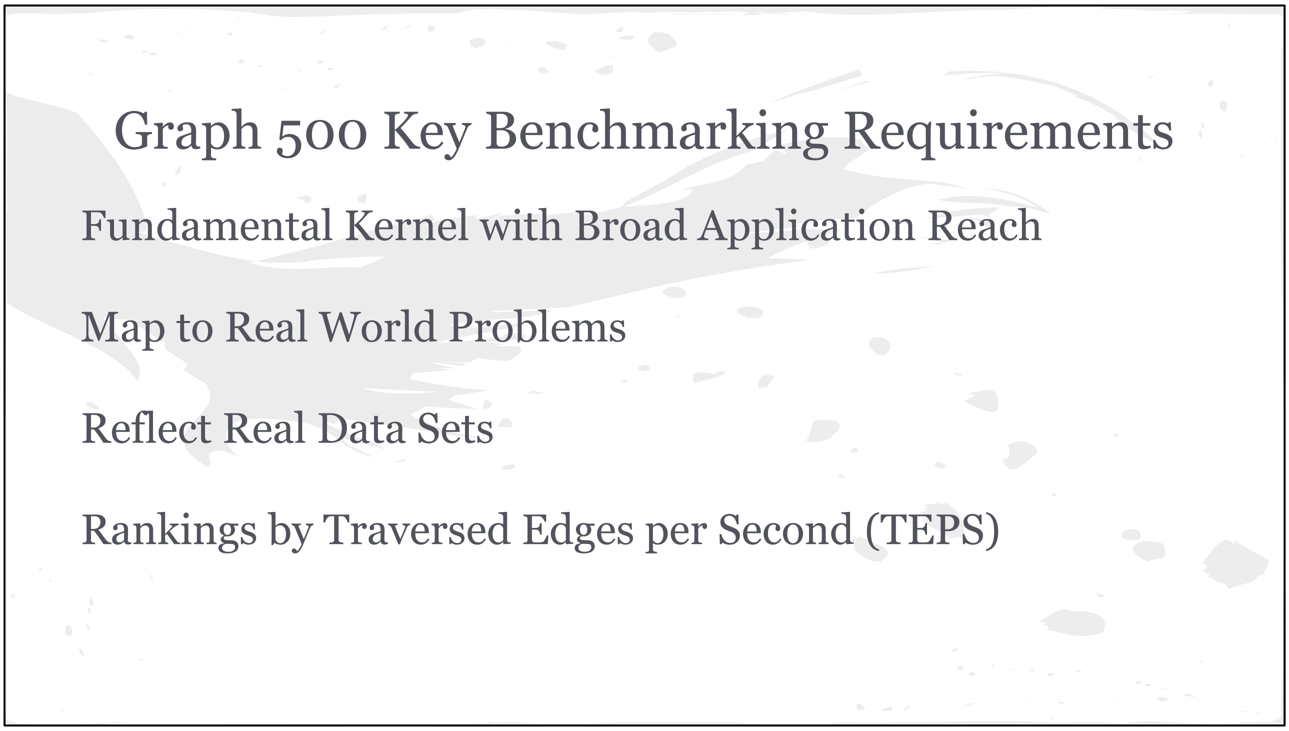
2.6% spatial reuse relative to Top 500

55% temporal reuse relative to Top 500

pose problem of benchmarking, highlighting shortcomings of the top500, which benchmarks against FLOPS, with regards to communicationn
introduce graph500

So given that complex graph applications exhibit very low spatial and temporal locality, meaning low reuse of data near data already
used and low reuse of data over time respectively respectively, how do we evaluate an HPC system’s capability to handle the processing of graph-like data? The top500 benchmarks mimic more traditional workloads where these localities are present, but these applications also exhibit significantly larger datasets than is typical for real-world physics applications or industry benchmarks. Compared to the LINPACK benchmark that defines the Top 500 list, an example graph problem consumes 6,900 times the unique data, and exhibits 2.6% of the temporal reuse and 55% of the temporal reuse.

The Graph 500 benchmarking list aims to bridge this gap by producing scalable benchmarks that are in line with the characteristics of graph-like data analysis.



Graph 500 Key Benchmarking Requirements

Fundamental Kernel with Broad Application Reach

Map to Real World Problems

Reflect Real Data Sets

Rankings by Traversed Edges per Second (TEPS)

Here are some key requirements identified for any benchmark of graph-like data:

1. There should be a Fundamental Kernel with Broad Application Reach: rather than a single transitory point application, the benchmark must reflect a class of algorithms that impact many applications.graph-like
2. It should Map to Real World Problems: results from the benchmark should map back to actual problems, rather than exhibiting a theoretical or “pure computer science” result.
3. It should Reflect Real Data Sets (that are important!): similar to mapping to real problems, the data sets should exhibit real-world patterns. Application performance for many real-world problems
is highly input dependent.

Graph 500 Example Datasets

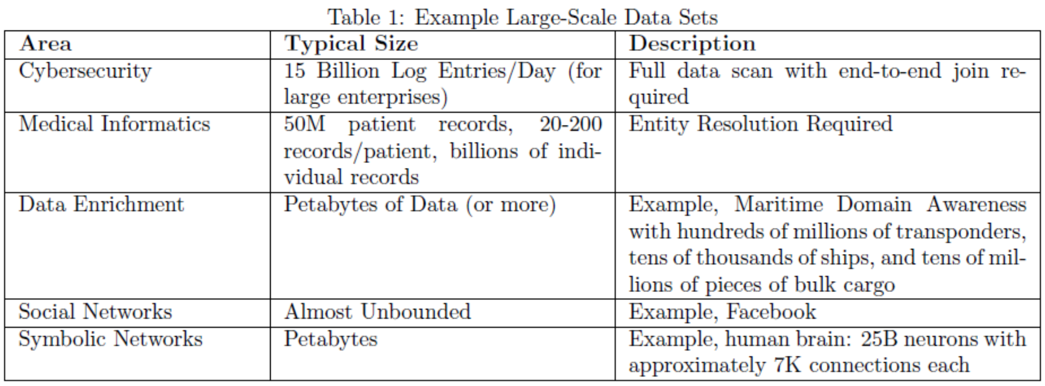
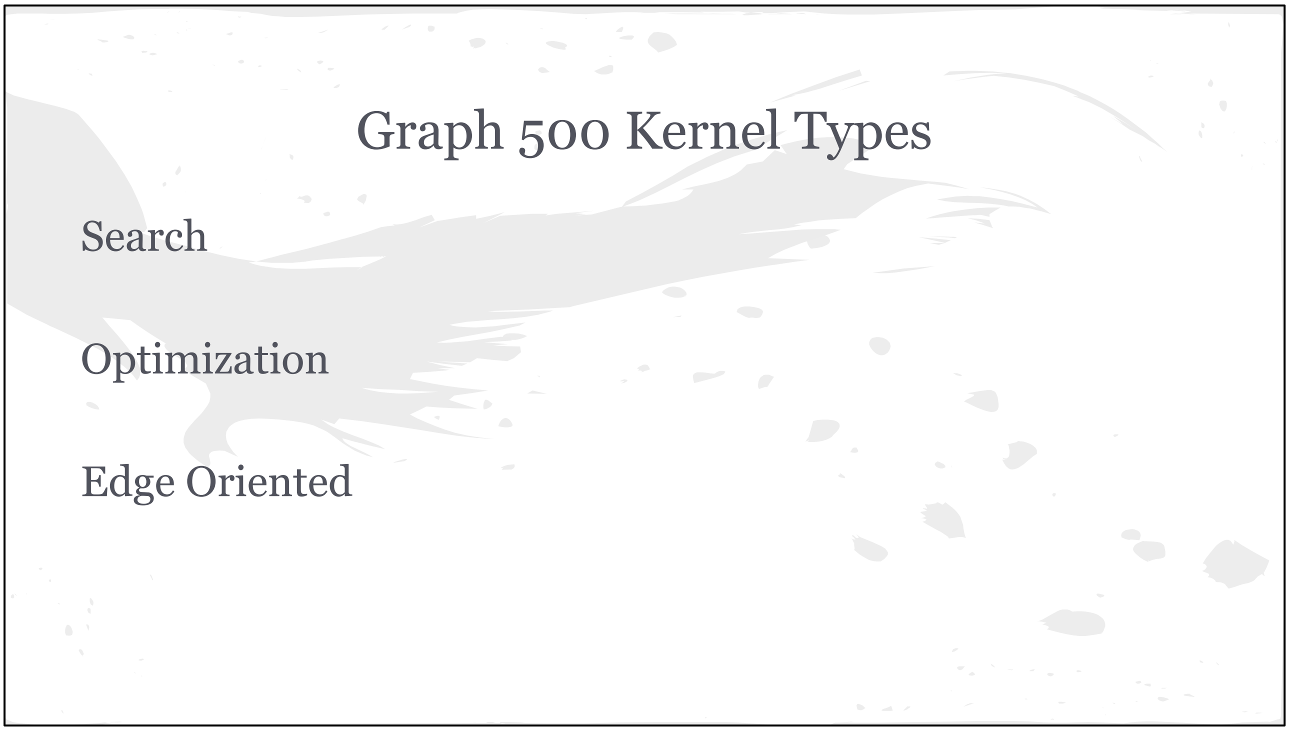


Table 1: Example Large-Scale Data Sets

Area	**Typical Size**	**Description**
Cybersecurity	15 Billion Log Entries/Day (for large enterprises)	Full data scan with end-to-end join required
Medical Informatics	50M patient records, 20-200 records/patient, billions of individual records	Entity Resolution Required
Data Enrichment	Petabytes of Data (or more)	Example, Maritime Domain Awareness with hundreds of millions of transponders, tens of thousands of ships, and tens of millions of pieces of bulk cargo
Social Networks	Almost Unbounded	Example, Facebook
Symbolic Networks	Petabytes	Example, human brain: 25B neurons with approximately 7K connections each

Some datasets include && table 1



Graph 500 Kernel Types

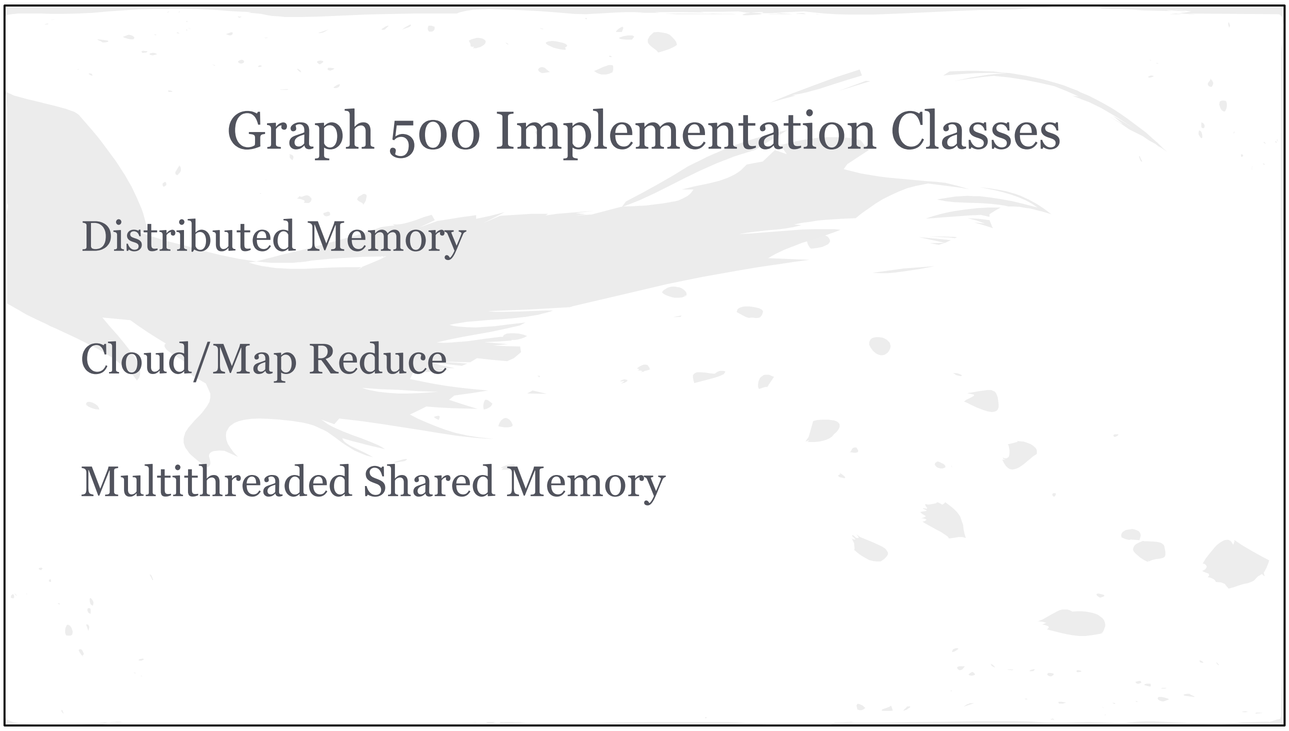
Search

Optimization

Edge Oriented

Here are the types of kernels chosen to satisfy the identified benchmarking requirements of a broad application reach, and a real world mapping

1. Search: Concurrent Search, requiring traversal of the graph and labeling or identification of a result.
2. Optimization: Single Source Shortest Path is a core example optimization problem proposed by the Graph 500 committee
3. Edge Oriented: Things like identifying Maximally independent sets - independent set in a graph that is not a subset of any other independent set.



Graph 500 Implementation Classes

Distributed Memory

Cloud/Map Reduce

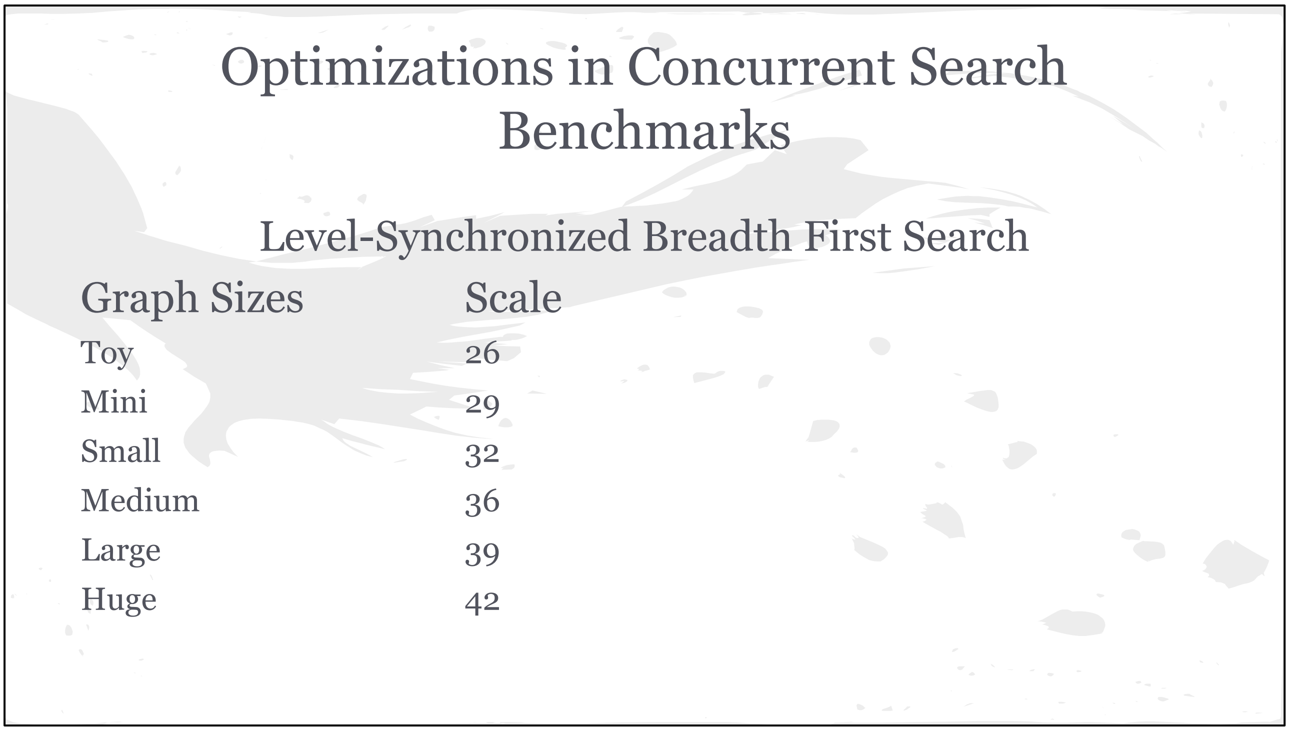
Multithreaded Shared Memory

The Graph500 has reference implementations of the benchmarks for multiple High performance computing scenarios:
Distributed Memory: This is the dominant model for high performance computing

Cloud/MapReduce: More reflective of industry trends, and represents a key programming model for loosely coupled architectures.

Multithreaded Shared Memory: This reference implementation will support large-scale shared memory machines, like SMPs or the Cray XMT. Currently,
this is the preferred programming model for many graph applications.

The benchmark will also allow for implementations for more customized platforms as well as optimizations to the reference implementations

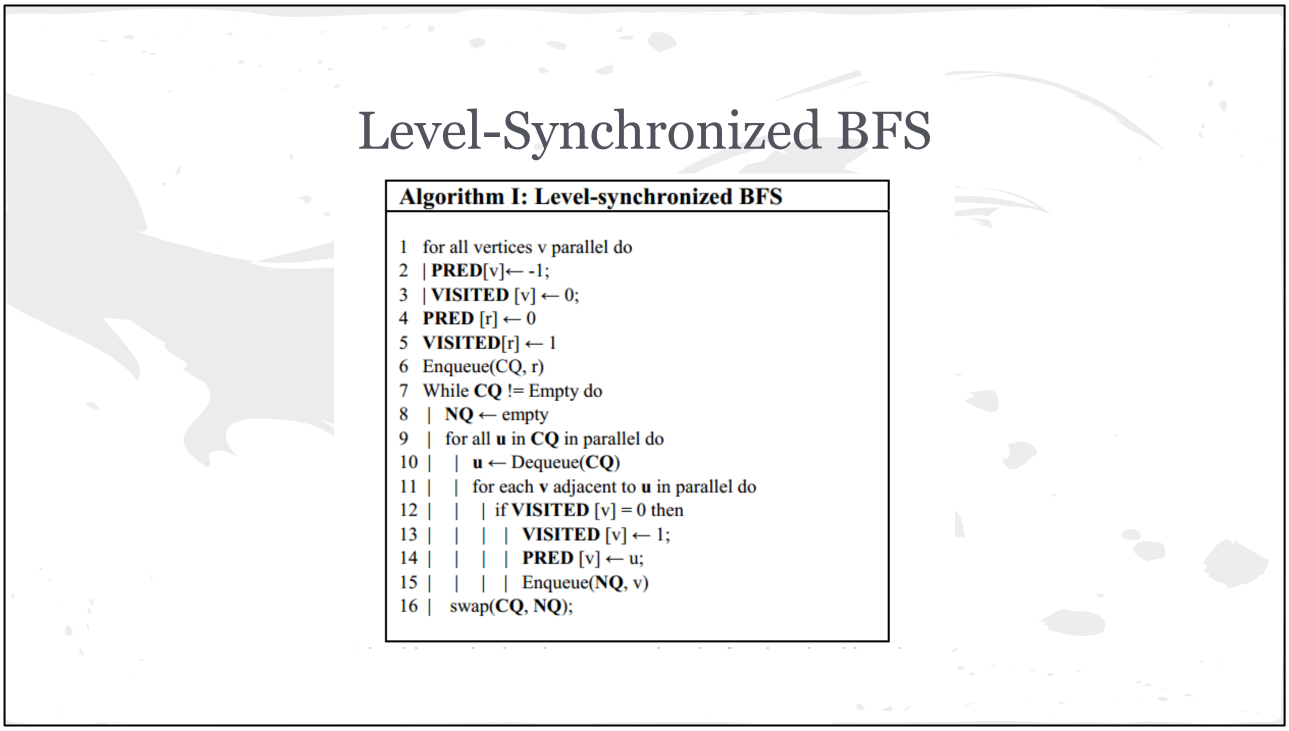


Optimizations in Concurrent Search Benchmarks

Level-Synchronized Breadth First Search

Graph Sizes	Scale
Toy	26
Mini	29
Small	32
Medium	36
Large	39
Huge	42

One set of the Graph500's reference search-type benchmarks performs breadth-first searches in large undirected graphs generated by a scalable data generator. There are six problem sizes: toy, mini, small, medium, large, and huge. Each problem solves a different size graph defined by a Scale parameter, which is the base 2 logarithm of the number of vertices. For example, the level Scale 26 for toy means 2^26 and corresponds to 10^10 bytes occupying 17 GB of memory. The six Scale values are 26, 29, 32, 36, 39, and 42 for the six classes. The largest problem, huge (Scale 42), needs to handle around 1.1 PB of memory. Scale 38 is the largest that has been solved by a top-ranked supercomputer.



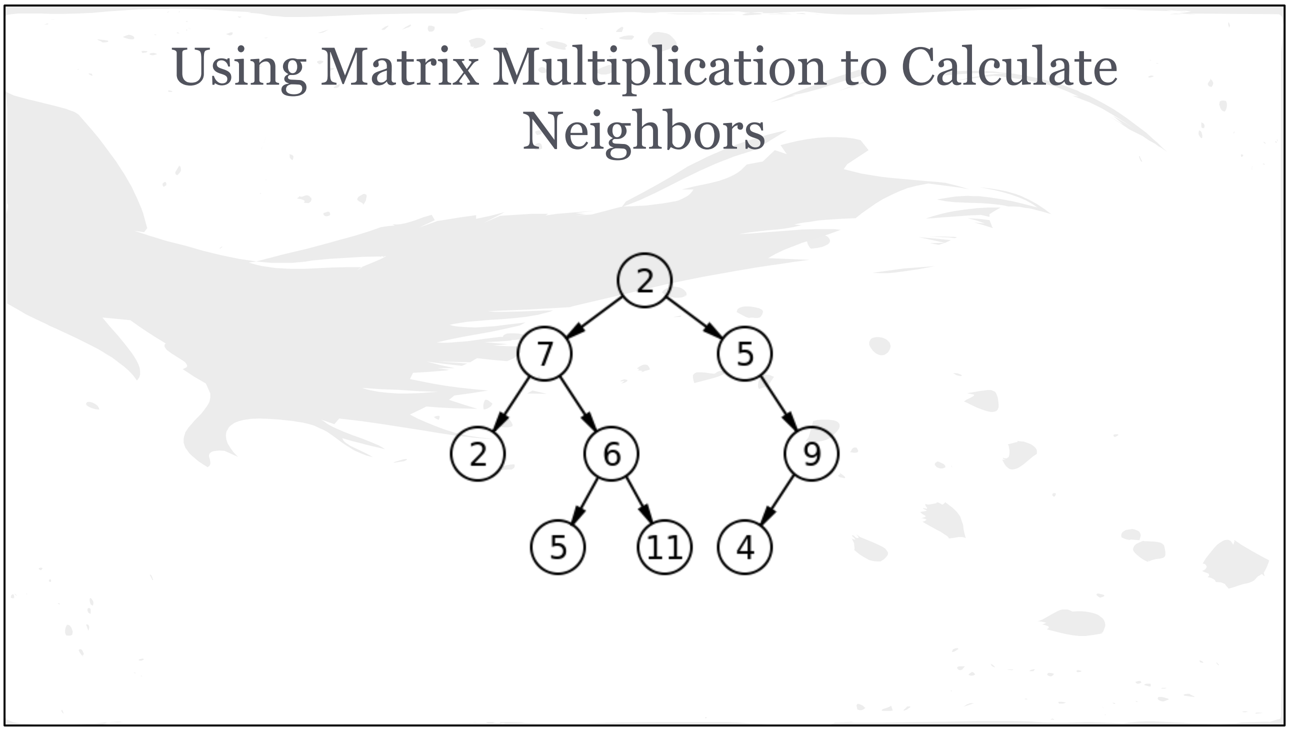
Level-Synchronized BFS

Algorithm I: Level-synchronized BFS

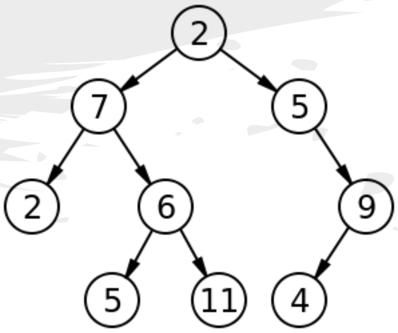
```
for all vertices v parallel do
|   PRED[v] ← -1;
|   VISITED [v] ← 0;
PRED [r] ← 0
VISITED[r] ← 1
Enqueue(CQ, r)
While CQ != Empty do
|   NQ ← empty
|   for all u in CQ in parallel do
|   |   u ← Dequeue(CQ)
|   |   for each v adjacent to u in parallel do
|   |   |   if VISITED [v] = 0 then
|   |   |   |   VISITED [v] ← 1;
|   |   |   |   PRED [v] ← u;
|   |   |   |   Enqueue(NQ, v)
|   swap(CQ, NQ);
```

As far as the searching kernels are concerned, the reference implementations for the different platforms are all based on the level-synchronized breadth first search. ‘Level-synchronized’ means that even though work is being done in parallel, all the nodes of a level are processed before any child nodes are processed.

Here is the pseudocode for the algorithm that implements level-synchronized BFS. Each processor has two queues, Current Queue and Next Queue, and two arrays, PRED for a predecessor array and VISITED to track whether or not each vertex has been visited. At the end of the Breadth First Search, the Predecessor array will contain the breadth first search tree.



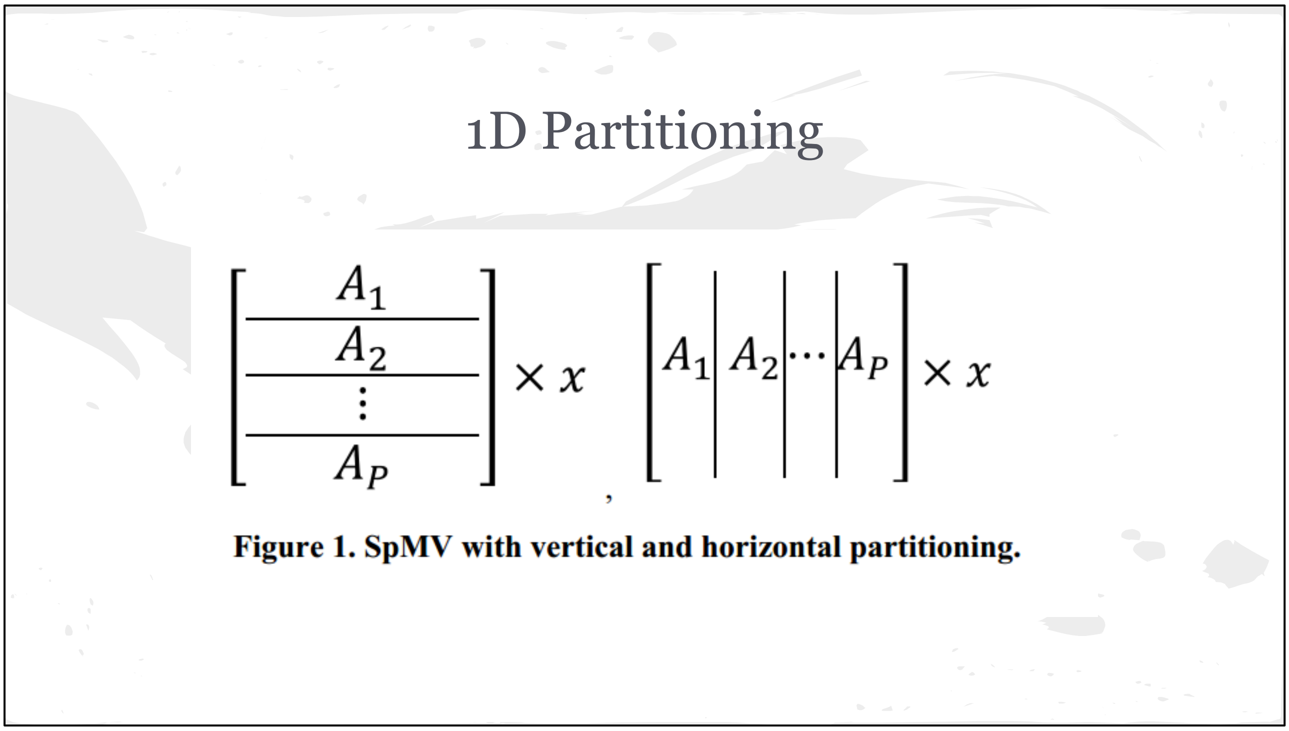
Using Matrix Multiplication to Calculate Neighbors



However, instead of an iterative loop, one can use Matrix multiplication in order to calculate the neighbors of a given set of vertices

Will demonstrate

This offers some advantages in that adjacency matrices are fairly sparse, enabling the use of compressed representations for the multiplication. The matrix representation also allows us to partition the adjacency matrix, and distribute the work amongst multiple processors.



1D Partitioning

$$\begin{bmatrix} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{bmatrix} \times x \quad , \quad \left[\begin{array}{c|c|c|c} A_1 & A_2 & \cdots & A_P \end{array}\right] \times x$$

Figure 1. SpMV with vertical and horizontal partitioning.

Highlight scalability issues with 1D partitioning
The reference implementation of the breadth first search comes in three versions. The R-CSR (replicated-(Compressed Sparse Row)), and the R-CSC (replicated-(Compressed Sparse Column)) partition vertically, while the SIM (simple) method partitions horizontally

For the R-CSR and R-CSC methods that divide an adjacency matrix vertically, the Current Queue is duplicated to all of the processes. Then
each processor independently computes NQ for its own portion. Copying CQ to all of the processors means each processor sends its own portion of NQ to all of the other processors. CQ (and NQ) is represented as a relatively small bitmap. For relatively small problems with limited amounts of distribution, the amounts of communication data are reasonable and this method is effective.
However, since the size of CQ is proportional to the number of vertices in the entire graph, this copying operation still leads to a large amount of communication for large problems with large distribution.

For the simple method that partitions horizontally, there is still a large amount of communication that doesn't scale. Each process has to send off the borders of its neighborhood to the processes that need it. This results in an all-to-all communication that falls apart at higher scales.

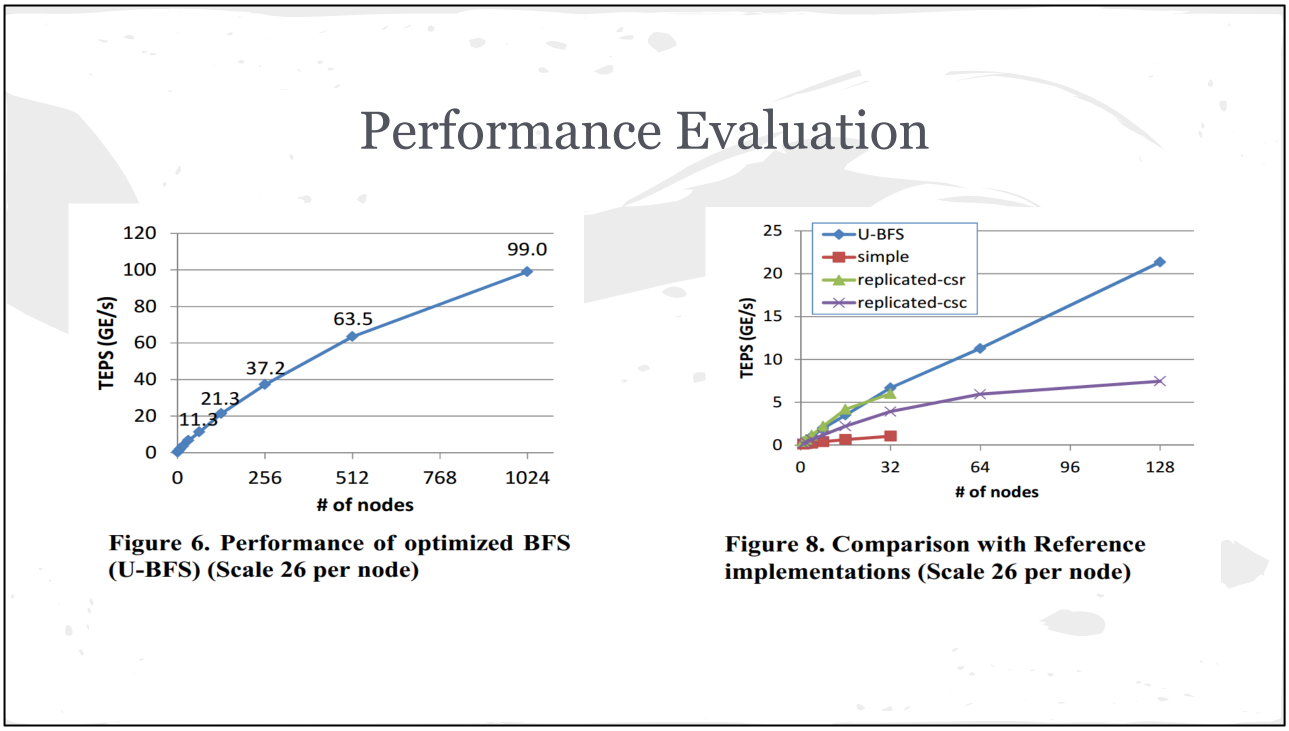
2D Partitioning

$$
\left[\begin{array}{c|c|c|c}
A_{1,1}^{(1)} & A_{1,2}^{(1)} & \cdots & A_{1,C}^{(1)} \\ \hline
A_{2,1}^{(1)} & A_{2,2}^{(1)} & \cdots & A_{2,C}^{(1)} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A_{R,1}^{(1)} & A_{R,2}^{(1)} & \cdots & A_{R,C}^{(1)} \\ \hline
A_{1,1}^{(2)} & A_{1,2}^{(2)} & \cdots & A_{1,C}^{(2)} \\ \hline
A_{2,1}^{(2)} & A_{2,2}^{(2)} & \cdots & A_{2,C}^{(2)} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A_{R,1}^{(2)} & A_{R,2}^{(2)} & \cdots & A_{R,C}^{(2)} \\ \hline
 & & & \\
 & & & \\ \hline
A_{1,1}^{(C)} & A_{1,2}^{(C)} & \cdots & A_{1,C}^{(C)} \\ \hline
A_{2,1}^{(C)} & A_{2,2}^{(C)} & \cdots & A_{2,C}^{(C)} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A_{R,1}^{(C)} & A_{R,2}^{(C)} & \cdots & A_{R,C}^{(C)}
\end{array}\right]
$$

Figure 4. 2D Partitioning Based BFS [4]

With 2D partitioning, however, each processor only communicates with the processors in the processor-row and the processor-column.

Ueno and Suzumura's optimized breadth first search benchmark uses 2D partitioning as its base, along with a small compression scheme for larger
transfers of vertex information, parallelization of communication and processing by the use of per processor buffers, and cache hit optimizations by visiting row indexes in a sorted ordering local to each 2D partition



Performance Evaluation

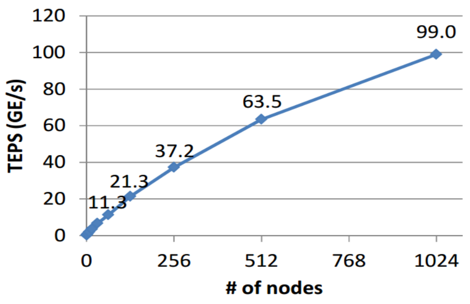


Figure 6. Performance of optimized BFS (U-BFS) (Scale 26 per node)

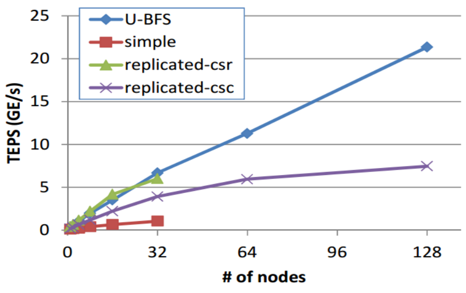
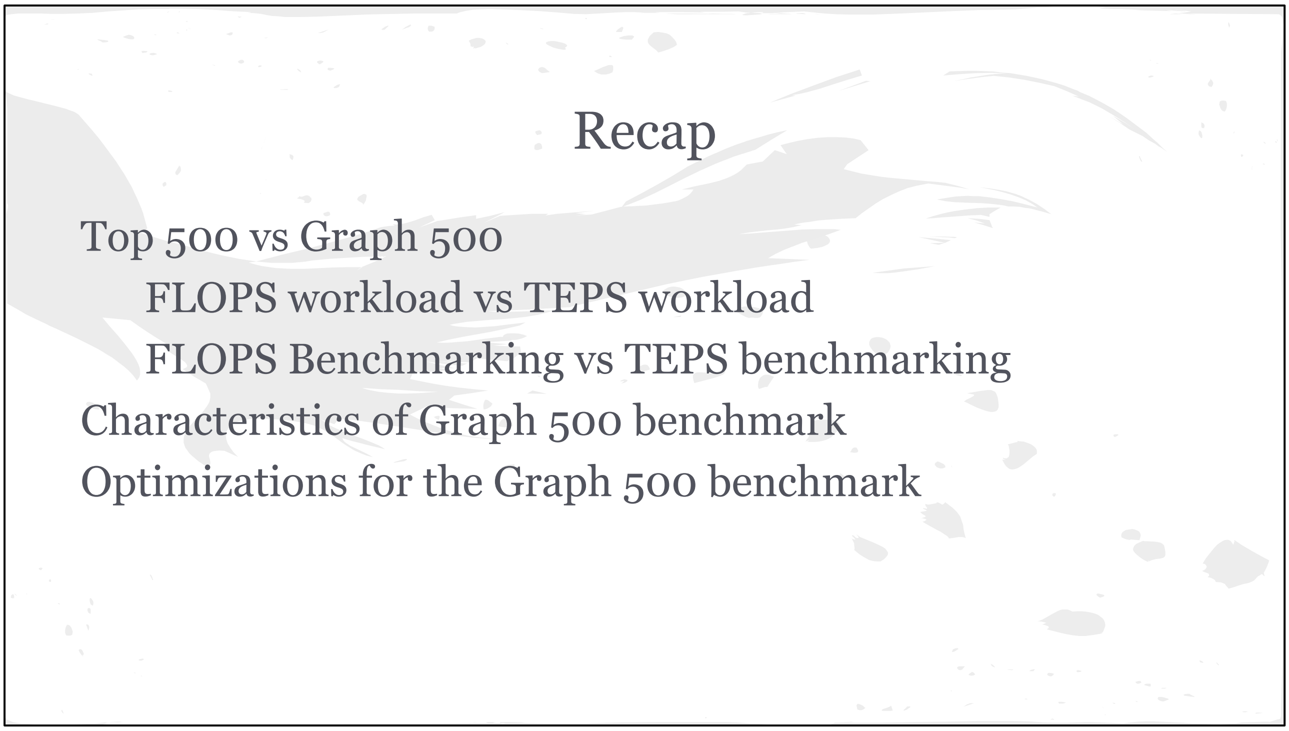


Figure 8. Comparison with Reference implementations (Scale 26 per node)

Performance evaluation

There is a definite increase in speed and capability, as you all can see. In terms of capability, you can see where the reference implementations meet their limits. In terms of speed, you've got a pretty straightforward comparison, where for the graph500 benchmark, speed is measured in the number of traversed edges per second rather than floating point operations per second.



Recap

Top 500 vs Graph 500

- FLOPS workload vs TEPS workload
- FLOPS Benchmarking vs TEPS benchmarking

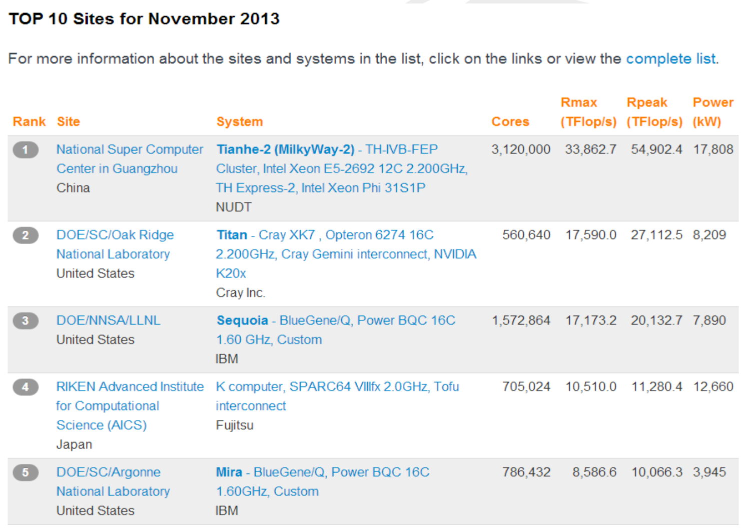
Characteristics of Graph 500 benchmark

Optimizations for the Graph 500 benchmark

So we have contrasted the workloads of typical high performance computing and emerging graph-like data problems. We've learned that there is a huge loss of locality and structure, meaning that the strategies that usually take advantage of those qualities are no longer effective. There is also a huge increase in the amount of data as well as the amount of potential communication. Since the top 500 benchmarks are geared towards typical high performance computing workload, they don't test a machine's capability to handle these newer types of applications. This is why the graph 500 came about. The graph 500's kernels are general enough to cut across many graphical applications and targets multiple platforms. It also scales well into the larger problem sizes. For the largest graphs, however, the reference implementations became unreliable. This is where the optimizations are supposed to start playing a larger role. We took a look at Ueno and Suzumiya's 2D partitioning based level-synchronized breadth first search. It uses 2D partitioning along with further compression of data transmissions, and local sorted ordering in order to extend the reach of the reference implementations breadth first search.

Top 500

November 2013

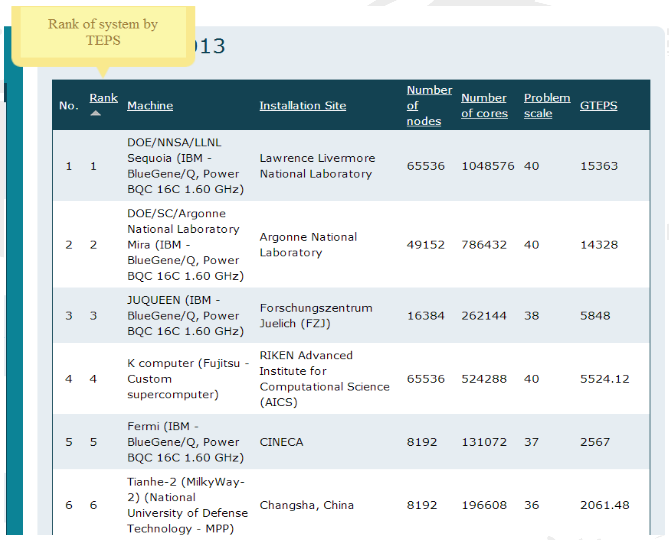


TOP 10 Sites for November 2013

For more information about the sites and systems in the list, click on the links or view the complete list.

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

Graph 500
November 2013



No.	Rank	Machine	Installation Site	Number of nodes	Number of cores	Problem scale	GTEPS
1	1	DOE/NNSA/LLNL Sequoia (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)	Lawrence Livermore National Laboratory	65536	1048576	40	15363
2	2	DOE/SC/Argonne National Laboratory Mira (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)	Argonne National Laboratory	49152	786432	40	14328
3	3	JUQUEEN (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)	Forschungszentrum Juelich (FZJ)	16384	262144	38	5848
4	4	K computer (Fujitsu - Custom supercomputer)	RIKEN Advanced Institute for Computational Science (AICS)	65536	524288	40	5524.12
5	5	Fermi (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)	CINECA	8192	131072	37	2567
6	6	Tianhe-2 (MilkyWay-2) (National University of Defense Technology - MPP)	Changsha, China	8192	196608	36	2061.48

Here is a look at the graph 500 for November in 2013