Parallel bfs + partitioning

Friday, May 26, 2017 11:05 AM

efficient RAM algorithms do not easily translate into "good performance" on current computing platforms. This mismatch arises due to the fact that current architectures lean towards efficient execution of regular computations with low memory footprints, and heavily penalize memory-intensive codes with irregular memory accesses. Graph traversal problems such as BFS are by definition predominantly memory access-bound, and these accesses are further dependent on the structure of the input graph, thereby making the algorithms "irregular".

Algorithm 1 Serial BFS algorithm.

Input: $G(V, E)$, source vertex s.
Output: $d[1..n]$, where $d[v]$ gives the length of the shortest path from s to $v \in V$.

```
1: for all v ∈ V do
2:     d[v] ← ∞
3: d[s] ← 0, level ← 1, FS ← φ, NS ← φ
4: push s → FS
5: while FS ≠ φ do
6:     for each u in FS do
7:         for each neighbor v of u do
8:             if d[v] = ∞ then
9:                 push v → NS
10:                d[v] ← level
11:    FS ← NS, NS ← φ, level ← level + 1
```

is slightly different from the widely-used queue-based serial algorithm [14]. We can relax the FIFO ordering mandated by a queue at the cost of additional space utilization, but the work complexity in the RAM model is still $O(m + n)$.

Vertices processed currently clearly separated from those to be processed in next (easier parallelisation?)

Parallelisation (related works)

decades [31, 32]. The classical parallel random access machine (PRAM) approach to BFS is a straightforward extension of the serial algorithm presented in Algorithm 1. The graph traversal loops (lines 6 and 7) are executed in parallel by multiple processing elements, and the distance update and stack push steps (lines 8-10) are atomic. There is a barrier synchronization step once for each level, and thus the execution time in the PRAM model is $O(D)$, where the D is the diameter of the graph. Since the PRAM model does not weigh in synchronization costs, the asymptotic complexity of work performed is identical to the serial algorithm.

Majority of novel implementation follow this level synchronous layout. Main directions or optimizations:

- ensuring that parallelization of the edge visit steps (lines 6, 7 in Algorithm 1) is load-balanced,
- mitigating synchronization costs due to atomic updates and the barrier synchronization at the end of each level, and
- improving locality of memory references by modifying the graph layout and/or BFS data structures.

Shared memory optimizations

Multithreaded systems: Bader and Madduri [4] present a fine-grained parallelization of the above level-synchronous algorithm for the Cray MTA-2, a massively multithreaded shared memory parallel system. Their approach utilizes the support for fine-grained, low-overhead synchronization provided on the MTA-2, and ensures that the graph traversal is load-balanced to run on thousands of hardware threads. The MTA-2 system is unique in that it relies completely on hardware multithreading to hide memory latency, as there are no data caches in this system. This feature also eliminates the necessity of tedious locality-improvement optimizations to the BFS algorithm, and Bader and Madduri's implementation achieves a very high system utilization on a 40-processor MTA-2 system. Mizell and Maschhoff [29] discuss an improvement to the Bader-Madduri MTA-2 approach, and present performance results for parallel BFS on a 128-processor Cray XMT system, a successor to the Cray MTA-2.

fine-grained, low-level synchronization on Cray MTA-2

GPGPU quite similar environment to Cray MTA-2, you have to take care that threads perform regular and conignous memory access to achieve high utilization.

- neighbours might be coalesced, but frontier is not
- Intensive synchronization needed

the d array in the level synchronous algorithm. Harish and Narayanan [22] discuss implementations of BFS and other graph algorithms on NVIDIA GPUs. Due to the comparably higher memory bandwidth offered by the GDDR memory, they show that the GPU implementations outperform BFS approaches on the CPU for various low-diameter graph families with tens of millions of vertices and edges. Luo et al. [27] present an improvement to this work with a new hierarchical data structure to store and access the frontier vertices, and demonstrate that their algorithm is up to 10× faster than the Harish-Narayanan algorithm on recent NVIDIA GPUs and low-diameter sparse graphs. You et al. [39] study BFS-like traversal optimization on GPUs and multicore CPUs in the context of implementing an inference engine for a speech recognition application.

Very intersting articles, those about GPU

How CPUs compare?

Multicore systems: There has been a spurt in recent work on BFS approaches for multicore CPU systems. Current x86 multicore architectures, with 8 to 32-way core-level parallelism and 2-4 way simultaneous multithreading, are much more amenable to coarse-grained load balancing in comparison to the multithreaded architectures. Possible

Multithreading architectures – probably means GPGPU and Cray MTA-2

nization overhead. However, due to the memory-intensive nature of BFS, performance is still quite dependent on the graph size, as well as the sizes and memory bandwidths of the various levels of the cache hierarchy. Recent work

Memory is bottleneck for this problem (GPUs, high bandwidth)

of the various levels of the cache hierarchy. Recent work on parallelization of the queue-based algorithm by Agarwal et al. [1] notes a problem with scaling of atomic intrinsics on multi-socket Intel Nehalem systems. To mitigate

trinsics on multi-socket Intel Nehalem systems. To mitigate this, they suggest a partitioning of vertices and corresponding edges among multiple sockets, and a combination of the fine-grained approach and the accumulation-based approach in edge traversal. In specific, the distance values (or the "visited" statuses of vertices in their work) of local vertices are updated atomically, while non-local vertices are held back to avoid coherence traffic due to cache line invalidations. They achieve very good scaling going from one to four sockets with this optimization, at the expense of introducing an additional barrier synchronization for each BFS level. Xia and Prasanna [37] also explore synchronization-

Answer: partition across cores and hold back modification to non-local vertices to avoid cache-line bouncing

level. Xia and Prasanna [37] also explore synchronization-reducing optimizations for BFS on Intel Nehalem multicore systems. Their new contribution is a low-overhead "adaptive barrier" at the end of each frontier expansion that adjusts the number of threads participating in traversal based on an estimate of work to be performed. They show significant performance improvements over naïve parallel BFS implementations on dual-socket Nehalem systems. Leiser-

Adaptive barrier, which adjusts number of threads working on the problem

implementations on dual-socket Nehalem systems. Leiserson and Schardl [25] explore a different optimization: they replace the shared queue with a new "bag" data structure

Distributed memory

Distributed memory systems: The general structure of the level-synchronous approach holds in case of distributed memory implementations as well, but fine-grained "visited" checks are replaced by edge aggregation-based strategies. With a distributed graph and a distributed d array, a processor cannot tell whether a non-local vertex has been previously visited or not. So the common approach taken is to just accumulate all edges corresponding to non-local vertices, and send them to the owner processor at the end of a local traversal. There is thus an additional all-to-all communication step at the end of each frontier expansion. In-

But what for? Other processor also has those edges, why send them?

We probably parallelize on frontier vertices level (alternative– gather all neighbours and partition them). Each node visits only it's local nodes – so if our local node was discovered by someone else (through non-local node), we need to be informed about it to handle it during the following iteration.

Costs of all-to-all depends on:

- Network
- Graph partitioning

2D partitioning to the rescue

tributed memory system. The implementation by Yoo et al. [38] for the BlueGene/L system is a notable distributed memory parallelization. The authors observe that a two-dimensional graph partitioning scheme would limit key collective communication phases of the algorithms to at most $\sqrt{p}$ processors, thus avoiding the expensive all-to-all communication steps. This enables them to scale BFS to process concurrencies as high as 32,000 processors. However,

But there are problems with their paper:

- they assume uniform degree distribution and base bounds calculation on this assumption

realizable for graphs with skewed degree distributions. Furthermore, the computation time increases dramatically (up

- to 10-fold) with increasing processor counts, under a weak scaling regime. This implies that the sequential kernels and data structures used in this study were not work-efficient.

Section 4. Cong et al. [12] study the design and implementation of several graph algorithms using the partitioned global address space (PGAS) programming model. Recently, Edmonds et al. [16] gave the first hybrid-parallel 1D BFS implementation that uses active messages.

PGAS - is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process, thread, or processing element.[1] The novelty of PGAS is that the portions of

the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference.
What does this now mean for programmers and end-users? Both RMA and

PGAS are programming interfaces for end-users and offer several higher-level constructs such as remote read, write, accumulates, or locks. RDMA is often used to implement these mechanisms and usually offers a slimmer interface such as remote read, write, or atomics. RDMA is usually processed in hardware and RMA/PGAS usually try to use RDMA as efficiently as possible to implement their functions.

https://htor.inf.ethz.ch/blog/index.php/2016/05/15/what-are-the-real-differences-between-rdma-infiniband-rma-and-pgas/

External memory algorithms

External memory algorithms: Random accesses to disk are extremely expensive, and so locality-improvement optimizations are the key focus of external memory graph algorithms. External memory graph algorithms build on known I/O-optimal strategies for sorting and scanning. Ajwani and Meyer [2, 3] discuss the state-of-the-art algorithms for BFS and related graph traversal problems, and present performance results on large-scale graphs from several families. Recent work by Pierce et al. [30] investigates implementations of semi-external BFS, shortest paths, and connected components.

Algorithms alternate to level-synchronous approach

plementations of these algorithms. The fastest-known algorithm (in the PRAM complexity model) for BFS repeatedly squares the adjacency matrix of the graph, where the element-wise operations are in the min-plus semiring (see [17] for a detailed discussion). This computes the BFS ordering of the vertices in $O(\log n)$ time in the EREW-PRAM model, but requires $O(n^3)$ processors for achieving these bounds. This is perhaps too work-inefficient for traversing large-scale graphs. The level synchronous approach is

ing large-scale graphs. The level synchronous approach is also clearly inefficient for high-diameter graphs. A PRAM algorithm designed by Ullman and Yannakakis [35], based on path-limited searches, is a possible alternative on shared-memory systems. However, it is far more complicated than the simple level-synchronous approach, and has not been empirically evaluated. The graph partitioning-based strategies adopted by Ajwani and Meyer [3] in their external memory traversal of high-diameter graphs may possibly lend themselves to efficient in-memory implementations as well.

Other work

Other Related Work: Graph partitioning is intrinsic to distributed memory graph algorithm design, as it helps bound inter-processor communication traffic. One can further relabel vertices based on partitioning or other heuristics [13, 15], and this has the effect of improving memory reference locality and thus improve parallel scaling.

A sparse graph can analogously be viewed as a sparse matrix, and optimization strategies for linear algebra computations similar to BFS, such as sparse matrix-vector multiplication [36], may be translated to the realm of graph algorithms to improve BFS performance as well. Recent research shows prospects of viewing graph algorithms as sparse matrix operations [8, 19]. Our work contributes to that area by exploring the use of sparse-matrix sparse-vector multiplication for BFS for the first time.

The formulation of BFS that is common in combinatorial optimization and artificial intelligence search applications [5, 24] is different from the focus of this paper.

Looks interesting, especially view as sparse matrix

Their algorithm

With 1D partitioning

tithreaded processors. The distance array is distributed among processes. Every process only maintains the status of vertices it owns, and so the traversal loop becomes an edge aggregation phase. We can utilize multithreading within a process to enumerate the adjacencies. However, only the owner process of a vertex can identify whether it is newly visited or not. Thus, all the adjacencies of the vertices in the current frontier need to be sent to their corresponding owner process, which happens in the All-to-all communication step (line 21) of the algorithm. Note that the only thread-level synchronization required is due to the barriers. The rest of the steps such as buffer packing and unpacking can be performed by the threads in a data-parallel manner. The key aspects to note in this algorithm, in comparison to the serial level-synchronous algorithm (Algorithm 1), is the extraneous computation (and communication) introduced due to the distributed graph scenario: creating the message buffers of cumulative size $O(m)$ and the All-to-all communication step.

Algorithm 2 Hybrid parallel BFS with vertex partitioning.

Input: $G(V, E)$, source vertex s.
Output: $d[1..n]$, where $d[v]$ gives the length of the shortest path from s to $v \in V$.

1: **for all** $v \in V$ **do**
2: $d[v] \leftarrow \infty$
3: $level \leftarrow 1$, $FS \leftarrow \phi$, $NS \leftarrow \phi$
4: $op_s \leftarrow find\_owner(s)$
5: **if** $op_s = rank$ **then**
6: push $s \rightarrow FS$
7: $d[s] \leftarrow 0$
8: **for** $0 \leq j < p$ **do**
9: $SendBuf_j \leftarrow \phi$ ▷ p shared message buffers
10: $RecvBuf_j \leftarrow \phi$ ▷ for MPI communication
11: $tBuf_{ij} \leftarrow \phi$ ▷ thread-local stack for thread i
12: **while** $FS \neq \phi$ **do**
13: **for** each u in FS **in parallel do**
14: **for** each neighbor v of u **do**
15: $p_v \leftarrow find\_owner(v)$
16: push $v \rightarrow tBuf_{ip_v}$
17: Thread Barrier
18: **for** $0 \leq j < p$ **do**
19: Merge thread-local $tBuf_{ij}$'s **in parallel**, form $SendBuf_j$
20: Thread Barrier
21: **All-to-all collective step** with the master thread: Send data in $SendBuf$, aggregate newly-visited vertices into $RecvBuf$
22: Thread Barrier
23: **for** each u in $RecvBuf$ **in parallel do**
24: **if** $d[u] = \infty$ **then**
25: $d[u] \leftarrow level$
26: push $u \rightarrow NS_i$
27: Thread Barrier
28: $FS \leftarrow \bigcup NS_i$ ▷ thread-parallel
29: Thread Barrier

- FS processed in parallel (probably means thread-level parallelism)
- Does FS consist only of the local vertices?
- communication
  - Even local vertices go through send
  - Separate buffers for each node
  - Each thread builds batch separately, but they are collected and sent together
- All vertices to be processed during next iteration should be in recv buffer
- How it works (roughly):
  - Each node iterates over frontier he owns
  - Neighbours are sent to owners
  - Owners mark neighbours as visited and at the to frontier for the next iteration

With 2D partitioning

toring out the underlying algebra, each BFS iteration is computationally equivalent to a sparse matrix-sparse vector multiplication (SpMSV). Let A denote the adjacency matrix of the graph, represented in a sparse boolean format, x_k denotes the kth frontier, represented as a sparse vector with integer variables. It is easy to see that the exploration of level k in BFS is algebraically equivalent to $x_{k+1} \leftarrow A^{\mathsf{T}} \otimes x_k \odot \overline{\bigcup_{i=1}^{x_i}}$ (we will omit the transpose and assume that the input is pre-transposed for the rest of this section). The syntax $\otimes$ denotes the matrix-vector multiplication operation on a special , $\odot$ denotes element-wise multiplication, and overline represents the complement operation. In other words, $\overline{v}_i = 0$ for $v_i \neq 0$ and $\overline{v}_i = 1$ for $v_i = 0$. This algorithm becomes deterministic with the use of (select,max)-semiring, because the parent is always chosen to be the vertex with the highest label. The algorithm does not have to store the previous frontiers explicitly as multiple sparse vectors. In practice, it keeps a dense $parents = \bigcup_{i=1}^{x_i}$ array, which is more space efficient and easier to update.

- If we take A, 1 in ith row means that vertex is start of and edge
  in case of A^T vertex is end of the edge
- if we are multiplying ith row of the matrix, 1 @ j position means I-th vertex is end of and edge (which starts at j)
  if frontier vector has corresponding 1 => it's frontier vertex => edge in question goes from frontier vertex to vertex I => I is belongs to new frontier
- If in result we have x at k-th position, this vertex is end of x edges starting at previous frontier vertices
- Without vector-vector multiplication we get correct frontier matrices, but not distance
- If we multiply each frontier by level and sum them all, we should get distance vector
- How to get parent vector? (predecessors in BFS path)
  - X(k+1) <- A^T * x(k)
  - If A^T(ij) * xk(j) == 1 we have edge from j (frontier) to I (new frontier)
  - I is visible in the x(k+1), but j is not – it only appears as 1 in previous frontier vector

Algorithm 2 Distributed Breadth-First Expansion with 2-D Partitioning

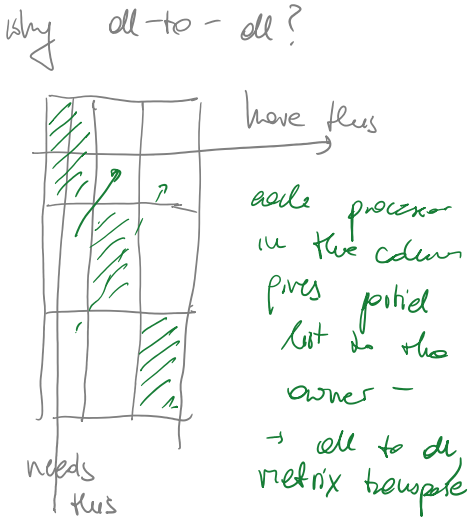
Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s \\ \infty, & \text{otherwise} \end{cases}$
for $l = 0$ to ∞ **do**
  $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
  if $F = \emptyset$ for all processors **then**
    Terminate main loop
  end if
  for all processors q in this processor-column **do**
    Send F to processor q
    Receive $\bar{F}_q$ from processor q (The $\bar{F}_q$ are disjoint)
  end for
  $\bar{F} \leftarrow \bigcup_q \bar{F}_q$
  $N \leftarrow \{$neighbors of vertices in $\bar{F}$ using edge lists on this processor$\}$
  for all processors q in this processor-row **do**
    $N_q \leftarrow \{$vertices in N owned by processor $q\}$
    Send N_q to processor q
    Receive $\bar{N}_q$ from processor q
  end for
  $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
  for $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
    $L_{v_s}(v) \leftarrow l + 1$
  end for
end for

(algorithm from different paper on the same topic)

- first owners request from all nodes in the column rest of the edges for vertices they are responsible for (expand step) (but no need to ask for those that are not on the frontier)
  - only actual edges are distributes, not whole submatrices (not surprising since submatrices are probably not even stored)
  - often implemented by allgather - but why? P(0,0) doesn't need rest of the edges for vertices he is not responsible for - each processor in column should execute gather for part he is interested in?
    - instead of allgather on the whole column, we could do it as all-to-all, where each vertex gives to each processor
  - you could do that without asking first, but then rest of the columns have to send all vertices in given range, not only those that are in frontier
- at the end they distribute found neighbours to nodes responsible for them (fold step, identical to the one occuring in 1D version)

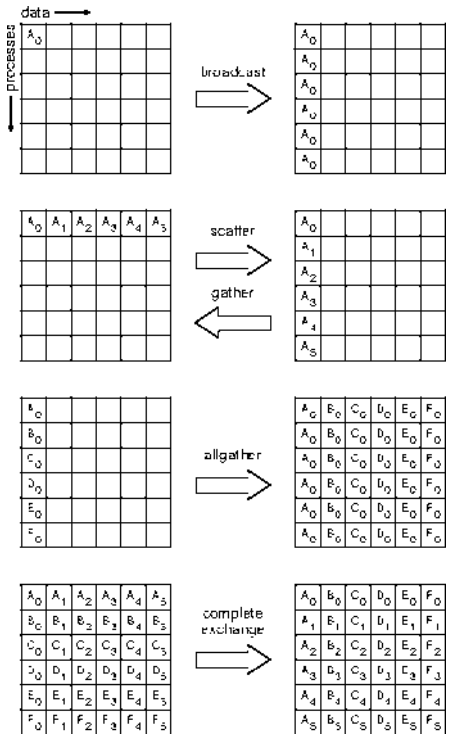
MPI collective communications routines (http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node95.htm#Node95):

- barrier
- bcast - one to many
- gahter - many to one



    - so each node has some data, and target gets array
- allgather - same as gather, but copy of result is deilvered to all vertices
- reduce
    - could be treated as special case of reduce when gather simply concatenates values
- allreduce - result returned to all
- alltoall - complete exchange
    - scatter + gather
    - result is as if all nodes performed scatter, but then each single result would be in separate data structure; so gather merges them (but only concpeptually, communication not needed?)
    - https://stackoverflow.com/a/15049670 - this seems consistent with my understanding:
      Instead of providing a single value that should be shared with each other process, each process specifies one value to give to each other process.
- scan, exscan - prefix reduction (reduction but only on part of the data?)

All collective routines have non-blocking variants (even barrier synchronization)



algorithm from this paper:
succint overview:

The parallel BFS algorithm with 2D partitioning has four steps:

- **Expand:** Construct the current frontier of vertices on each processor by a collective *All-gather* step along the processor column $P(:, j)$ for $1 \leq j \leq p_c$.
- **Local discovery:** Inspect outgoing edges of vertices in the current frontier.
- **Fold:** Exchange newly-discovered vertices via an collective *All-to-all* step along the processor row $P(i, :)$, for $1 \leq i \leq p_r$.
- **Local update:** Update distances and parents locally for newly-visited vertices.

details:

Algorithm 3 Parallel 2D BFS algorithm.

Input: A: undirected graph represented by a boolean sparse adjacency matrix, s: source vertex id.
Output: π: dense vector, where $\pi[v]$ is the predecessor vertex on the shortest path from s to v, or -1 if v is unreachable.

1: **procedure** BFS_2D(A, s)
2: $f(s) \leftarrow s$
3: **for** all processors $P(i, j)$ **in parallel do**
4: **while** $f \neq \emptyset$ **do**
5: TRANSPOSEVECTOR(f_{ij})
6: $f_i \leftarrow$ ALLGATHERV($f_{ij}, P(:, j)$)
7: $t_i \leftarrow A_{ij} \otimes f_i$
8: $t_{ij} \leftarrow$ ALLTOALLV($t_i, P(i, :)$)
9: $t_{ij} \leftarrow t_{ij} \odot \overline{\pi_{ij}}$
10: $\pi_{ij} \leftarrow \pi_{ij} + t_{ij}$

- f - current frontier, f_i - frontier part for which i-th row is responsible
- t - temporary parent information, for this iteration only
- TransposeVector - redistributes the vector so that the subvector owned by the ith processor row is now owned by the ith processor column
    - now this image starts making sense - the input shows transposed matrix, but vector ownership is changed at the beginning of iteration

- P(:,j) - processors than own f(i,:) after transpose - so we gather from the column all components needed for multiplication
  - this really is an all-gather operation - for multiplication each processor in column needs f_i
  - this operation plays different role than in previous algorithm - here we collect frontiers vector, there we collect rest of the edges (in prevous implementation only fraction of frontier vector n/p is held on each proc, here after allgather we have n/columns)
- despite slightly different partitioning, it still holds that we need to redistribute data only within row

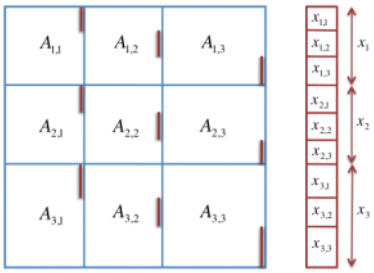


Figure 1: 2D vector distribution illustrated by interleaving it with the matrix distribution.

- this image shows transposed matrix, but vector is transposed at the beginning of the algorithm
- why they distribute vector even though they need to allgather it later on?
  alternative: store only on one node and then broadcast (which, according to authors is cheaper?)
  - n+1 frontier needs to be copied back to one node instead of many?
  - ever, vertex ownerships are more flexible. One practice is to distribute the vector entries only over one processor dimension (p_r or p_c) [23], for instance the diagonal processors if using a square grid ($p_r = p_c$), or the first processors of each processor row. This approach is mostly adequate for sparse matrix-dense vector multiplication (SpMV), since no local computation is necessary for the "fold" phase after the reduce step, to which all the processors contribute. For SpMSV, however, distributing the vector to only a subset of processors causes severe imbalance as we show in Section 4.3.

Graph representation

On the contrary, a CSR-like representation is too wasteful for storing sub-matrices after 2D partitioning. The aggregate memory required to locally store each submatrix in CSR format is $O(n\sqrt{p}+m)$, while storing the whole matrix in CSR format would only take $O(n+m)$. Consequently,

1D partitioning is not wasteful, we store all vertices exactly once (in offset table). In two 2D for each column we store sqrt(p) additional offset lists (edges are not duplicated)

Formats:

- triplets - (i, j, value) set
- CSR (Compressed Sparse Rows) - array of offsets + array of values
  - offset array is dense - is we have empty row is still needs to be stored
- CSC - transpose of CSR
- DCSC (doubly compressed sparse columns) - compress offset array, good for hyperspace matrices with empty columns
  - we only store

Performance

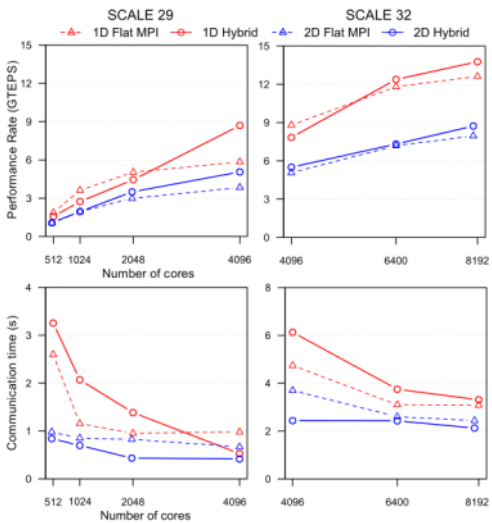


Figure 2: BFS 'strong scaling' results on Franklin for R-MAT graphs: Performance rate achieved (in GTEPS) on increasing the number of processors (top), and the corresponding average inter-node MPI communication time (in seconds, bottom).

- TEPS:
  - flat first win, but at some point hybrid overtakes it
    how cores are increased? more nodes or more cores per machine?
    theory: hybrid - additional processes consume performance single one could utilize, but reduce communication? so more cores = more communication in flat case so at some point benefits from more processing power per process are shadowed by communication overhead
    but communication time drops - why could that be?
    1D has better rate than 2D and the more cores they use they bigger the difference
- Comms time:
  - flat smaller than hybrid up to a point
  - 2D spends less time on communication
- despite spending more time on communication, 1D has better results (hybrid benefitial on more cores)
  - if the time of 2D is not occupied by communication, it must spend more time on computations (maybe that whole matrix-vertex thing slows him down)
- hybrids initially slower, but then catch up and overtake

Their explanation:
why 2D slower?

> 2D hybrid algorithm tends to outperform the flat 2D algorithm, but can not compete with the 1D algorithms on this architecture as it spends significantly more time in computation. This is due to relatively larger cache working sizes, as captured by our model in Section 5.

so communication time overshadowed by calculations

why hybrids initially slower, then faster?

> concurrencies. We attribute this effect partially to bisection bandwidth saturation, and partially to the saturation of the network interface card when using more cores (leading to more outstanding communication requests) per node. The

low cores => less machines => more vertices per machine => (unless lucky) more communication per node
with more machines we should get network saturation, but maybe topology is good enough that it doesn't happen

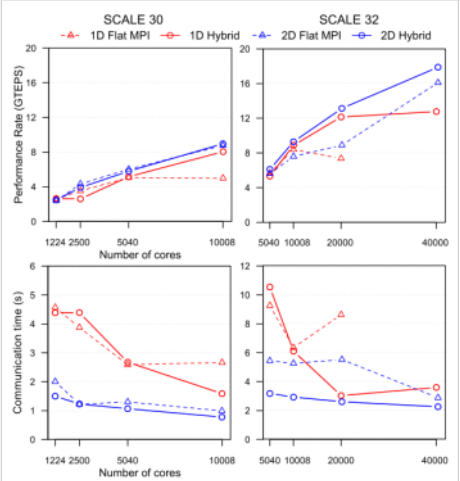


Figure 3: BFS 'strong scaling' results on Hopper for R-MAT graphs: Performance rate achieved (in GTEPS) on increasing the number of processors (top), and the corresponding average inter-node MPI communication time (in seconds, bottom).

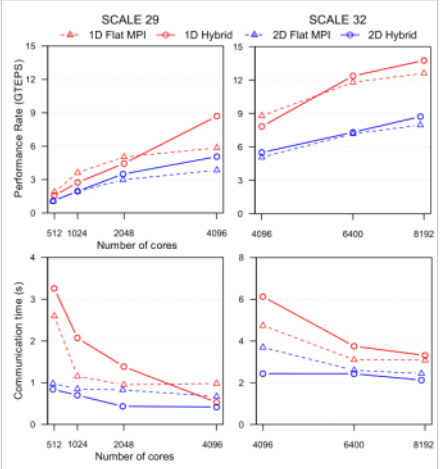


Figure 2: BFS 'strong scaling' results on Franklin for R-MAT graphs: Performance rate achieved (in GTEPS) on increasing the number of processors (top), and the corresponding average inter-node MPI communication time (in seconds, bottom).

- better peak results of TEPS,
- comms times initially larger, then comparable
- flats far more inefficient when it comes to comms time
- 2D is slightly faster, especially for larger concurrencies
- flat variant either comparable of slower (1D in upper-left corner is an exception)

> In contrast to Franklin results, the 2D algorithms perform better than their 1D counterparts. The more sophisticated Magny-Cours chips of Hopper are clearly faster in integer calculations, while the overall bisection bandwidth has not kept pace. We did not execute the flat 1D algorithm on 40K cores as the communication times already started to increase when going from 10K to 20K cores, consuming more than 90% of the overall execution time. In contrast, the percentage of time spent in communication for the 2D hybrid algorithm was less than 50% on 20K cores.

so computations here are faster, so the algorithm can win?

Strong vs weak scaling
Strong scaling spread the same size problem across more nodes. Weak scaling keeps the problem-per-node size constant while increasing the number of nodes, thus solving a larger problem size.

From <https://www.quora.com/What-is-the-difference-between-strong-and-weak-scaling>

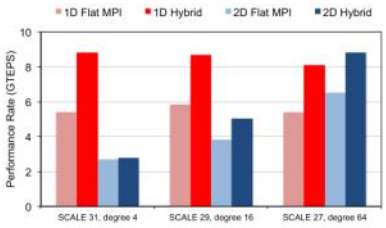


Figure 5: BFS GTEPS performance rate achieved on 4096 cores of Franklin by varying the average vertex degree for R-MAT graphs.

Senitivity of algorithm to densities

- 1D performs worse, 2D better as degree increases
- what we need to communicate:
  - get rest of frontier from column (sparse vector, so larger frontier = more communication)
  - distribute ends to form new frontier (more edges = more sparse 1s to transfer = more communication)
- if bound by communication larger degrees should result in performance loss (1D?)
- if bound by computation, larger degrees should not bring speed up - denser matrices so more to multiply (unless multiplication is pretty stupid and multiplies even zeros, so amount of computation to be performed doesn't change)
  - oh, they decreased the number of vertices instead of adding edges - so resulting matrices/vectors were smaller

directons of improvements they want to pursue

distributed-memory graph algorithms.

Exploiting symmetry in undirected graphs. If the graph is undirected, then one can save 50% space by storing only the upper (or lower) triangle of the sparse adjacency matrix, effectively doubling the size of the maximum problem that can be solved in-memory on a particular system. The algorithmic modifications needed to save a comparable amount in communication costs for BFS iterations is not well-studied.

Exploring alternate programming models. Partitioned global address space (PGAS) languages can potentially simplify expression of graph algorithms, as inter-processor communication is implicit. In future work, we will investigate whether our two new BFS approaches are amenable to expression using PGAS languages, and whether they can deliver comparable performance.

Reducing inter-processor communication volume witl graph partitioning. An alternative to randomization of vertex identifiers is to use hypergraph partitioning software to reduce communication. Although hypergraphs are capable of accurately modeling the communication costs of sparse matrix-dense vector multiplication, SpMSV case has not been studied, which is potentially harder as the sparsity pattern of the frontier vector changes over BFS iterations.

Interprocessor collective communication optimization. We conclude that even after alleviating the communication costs, the performance of distributed-memory parallel BFS is heavily dependent on the inter-processor collective communication routines All-to-all and Allgather. Understanding the bottlenecks in these routines at high process concurrencies, and designing network topology-aware collective algorithms is an interesting avenue for future research.

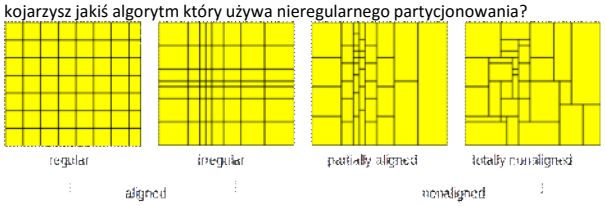
questions:

Plan na najbliższy czas

- validator do kolorowania (+ interfejs validatorów, pewnie też muszą być rozproszone)
- spróbować to odpalić na jakimś nieco większym grafie
- RMA wersja kolorowania
- implementacja BFS, w tym z partycjonowaniem 2D by się przyjrzeć problemowi

Pytania duże

- kojarzysz jakiś algorytm dla którego nieciągła numeracja globalna byłaby problemem?
  nawet jeśli tak, to potem można dorobić flexible partitioning
- przetwarzanie asynchroniczne - korzystamy z wersji iXXX, potem test i callbacki- czy są jakieś powody dla których nie miałoby sensu?
- kojarzysz jakiś algorytm który używa nieregularnego partycjonowania?



- jeśli tak to można mieć bardziej szczegółowe interfejsy, które będą miały adapter do ogólnego dla tych algorytmów, które tej szczegółowości nie potrzebują
- wiesz o jakichś narzędzich którymi możnaby mierzyć czas komunikacji MPI czy trzeba zrobić własną instrumentację?
- multithreaded architectures (Cray-MTA2) - rzeczy takie jak GPGPU? jak bardzo obecnie stosowane w kontekście

obliczeń grafowych?

Pytania małe

- mpi - stworzenie od razu zablokowanego okna
- alltoall - how different from multiple scatters?
- scan, exscan - MPI routines - częściowy reduce?

how this impact our abstraction

A couple of problems:

- lack of support for 2D partitioning
  - for local vertices only local edges might be returned (locally stored edges doesn't mean that vertices to which they point must also be stored locally, so still full ids needed)
  - some information about partitioning details might be needed (i.e. where is edge?)
- lack of matrix perspective on graph
  - dense matrix perspective doesn't really make sense, does it? only for fast prototyping of algorithms using this perspective (so we can use standard libraries? but how standard library'll support our custom nodes, we'd have to return dense matrices)
  - adjacency lists form nice sparse representation, but somehow we might need transpose
- graph transposing - should we support it as operation? maybe we should specify at the beinning that we want graph to be loaded in transposed form? (modifications to the interface would be required)

todo

- why allgather
- interface saturation - other extrema is network saturation?
- how exactly they are merged afterwards?
- why they distribute vector even though they need to allgather it later on?