Scalable Stochastic Block Partition

Ahsen J. Uppal[1], Guy Swope[2], and H. Howie Huang[1]

[1] The George Washington University
[2]The Raytheon Company

***Abstract*—The processing of graph data at large scale, though important and useful for real-world applications, continues to be challenging, particularly for problems such as graph partitioning. Optimal graph partitioning is NP-hard, but several methods provide approximate solutions in reasonable time. Yet scaling these approximate algorithms is also challenging. In this paper, we describe our efforts towards improving the scalability of one such technique, stochastic block partition, which is the baseline algorithm for the IEEE HPEC Graph Challenge [1]. Our key contributions are: improvements to the parallelization of the baseline bottom-up algorithm, especially the Markov Chain Monte Carlo (MCMC) nodal updates for Bayesian inference; a new top-down divide and conquer algorithm capable of reducing the algorithmic complexity of static partitioning and also suitable for streaming partitioning; a parallel single-node multi-CPU implementation and a parallel multi-node MPI implementation. Although our focus is on algorithmic scalability, our Python implementation obtains a speedup of 1.65x over the fastest baseline parallel C++ run at a graph size of 100k vertices divided into 8 subgraphs on a multi-CPU single node machine. It achieves a speedup of 61x over itself on a cluster of 4 machines with 256 CPUs for a 20k node graph divided into 4 subgraphs, and 441x speedup over itself on a 50k node graph divided into 8 subgraphs on a multi-CPU single node machine.**

I. Introduction

The analysis of graph data has become ubiquitous in a diverse array of fields, particularly when mining for hidden relationships and activities. But performance and scalability remain challenging issues because the computational complexity of many traditional algorithms, including graph partitioning (also known as community detection) and subgraph matching are NP-hard. For graph partitioning, the development of approximation algorithms [2] [3] has been critical in finding feasible solutions for real-world datasets. Unfortunately, there are major scalability challenges even for approximation algorithms. These challenges are made even more difficult for graph partitioning when the number of communities is not known a priori and there are no a priori cues about the membership of some of the nodes.

Furthermore, the streaming data adds to the complexity of the problem. While many theoretical algorithms assume that the entire graph is known at one time, real-world applications typically have data arrive in a streaming manner over time. One major challenge is to efficiently handle fresh streaming data, and merge its results with the results of the existing dataset.

II. Background

The stochastic block partition algorithm used as the GraphChallenge baseline uses a generative statistical models based on work by Peixoto [4][5][6].

Since the optimal number of blocks is not known a priori, the baseline algorithm first proceeds finding optimal partitions and entropies for partition sizes B of $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$, etc. until the partition with the minimum entropy is bracketed, and then switches to a Golden section search[7] to find the final partition and entropy.

For finding an optimal partition with a given block number of blocks, the baseline algorithm alternates between two program phases – agglomerative merging of existing communities, and nodal updates that move a vertex from one block to another. Nodal moves are accepted or rejected based on a probability derived from the resulting change in entropy. Changes that result in lower entropy are more likely to be accepted, in the manner of Metropolis-Hastings[8][9]. Agglomerative merges are greedy. That is potential merges are found for each source block. Then merges across all blocks are sorted by entropy and the ones with a minimum change in entropy are greedily chosen until the block count is sufficiently reduced. There is no sampling. The overall complexity of the algorithm is $O(N^2 \log^2 N) = O(E \log^2 E)$.

III. Approach

The C++ implementation of the baseline algorithm has excellent single-threaded performance on small graphs, but scaling this performance is difficult. In our single-machine testing, we found that the speedup of the baseline reference implementation saturates at 2.5x over sequential, even as the number of CPU cores increases from 4 to 8 to 16 to 32 as seen in Figure 1. The overhead from propagating nodal updates with message-passing on a multi-node system is even greater, thus limiting the performance of parallel MCMC updates.

In our initial profiling of the serial baseline algorithm (using the reference Python implementation), we discovered that for $N = 5000$ vertices, approximately 72% of the time is spent in nodal movements and 26% in nodal updates.

A. Agglomerative Merge Parallelism

Parallelization of the agglomerative merge portion is fairly straightforward. Most of the time in this phase is spent finding merge candidates for each current block. This problem decomposes cleanly into a parallel implementation, although the gains here are limited by Amdahl's law since this portion only represents a small part of the overall runtime. Our initial parallelization efforts using a pool of worker processes for this phase of the algorithm were relatively easy to implement, and provided the expected, albeit limited speed up of the overall program to approximately 35% for $N = 5000$ vertices.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

B. *Nodal Update and Shared State Batching*

Profiling reveals that the vast majority of the algorithm is spent performing nodal updates. One major challenge is that as large numbers of nodal updates are applied in parallel, write-intensive memory interconnect traffic will saturate the memory subsystem. But limiting the synchronization of these updates results in stale block membership assignments being visible to the other cores. Although not an issue for eventual convergence, our testing showed that using one iteration-old assignments resulted in a significant increase in the needed number of nodal updates and a consequent increase in runtime. Figure 2 shows the results of this testing using a single-threaded simulation of the effect of delayed nodal updates. Here, the update frequency is measured in terms of nodal proposals evaluated before accepted proposals are propagated. For $N = 1000$ nodes, a frequency of 1000 proposals indicates a one iteration delay before updates become globally visible. The results show that while the overall number of nodal moves increases with the length of the update interval, the overall runtime increases can increase significantly even with modest increases in the number of nodal moves. We interpret this to mean that the *quality* of nodal updates also increases as the update frequency increases, and a well-crafted parallel implementation should strive to minimize the latency of propagating nodal updates to all workers.

We designed a batch nodal update scheme to carefully balance the recency of updates against volume of updates. Armed with these results, we opted to make merge granularity and low-overhead for processing nodal updates a central feature of our nodal update efforts. We discuss this further in Section V.

IV. Top-Down Divide and Merge Algorithm

Parallelism from parallel nodal updates has its limitations due to traffic and computation induced from sending, receiving, and merging these updates. Even in a shared-memory implementation, there are significant overheads to locking and unlocking shared state in order to perform updates. Our major contribution is a top-down divide and merge algorithm to increase the isolation between parallel instances so that a minimum of shared state is be needed.

This top-down algorithm works by partitioning the input graph into several pieces, performing block partitioning on each piece, and then merging together the resulting blocks. Since the baseline algorithmic complexity is $O(N^2 \log^2 N)$, slicing N into K pieces results in at least a factor of $O(\frac{1}{K})$ savings aggregated across all instances assuming a fixed-cost merge overhead. The real-world benefit is greater though, because each instance of size $\frac{N}{K}$ can be executed entirely in parallel with no intercommunication.

A. *Subgraph Partitioning*

The initial partitioning into subgraphs is straight-forward. For every vertex in the original is mapped to $\{0 \ldots K-1\}$ subgraphs by taking its vertex id modulo K. For each subgraph, only edges whose endpoints are within that subgraph are preserved. This results in a large $\frac{1}{K^2}$ reduction in the number of edges that must be considered overall. If K is significantly less than B, the average number of communities per subgraph remains the same. However as K approaches B, the quality of subgraph partitions deteriorates rapidly.

B. *Subgraph Block Merge*

Although the basic idea is intuitive, there are implementation challenges that must be addressed to make it effective. First, the number of slices should be small relative to the number of true communities. Second, the stopping criterion and method of merging the results from each piece must be carefully designed to prevent errors from propagating into the final solution. Finally, the merge itself must be of sufficiently low overhead to realize these performance gains.

An approach that performs full block partitioning on each subgraph before merging is fast, but is very sensitive to partition errors in the subgraphs, particularly when the algorithm finds fewer communities in a subgraph than the true partition.

With these considerations in mind, we devised an approach that performs block partition on each graph piece, up until the point at which the Golden ratio bracket is established. At this establishment, each subgraph has an upper bound on the number of blocks, of approximately $2B^*$ where B^* is the true number of blocks. These proto-communities can be aggregated and the baseline algorithm can be allowed to continue with a merge-and-move approach of the baseline algorithm with the unified blocks.

But treating every block from the subgraph as an independent community (i.e. lumping), although accurate, is relatively slow because the resulting inter-block edge count matrix has dimensions $2KB^* \times 2KB^*$.

Merging partitions between subgraphs can be done much faster than the type of agglomerative merging used in the baseline algorithm. Consider two partitions from two subgraphs, $\mathbf{i}$, and $\mathbf{j}$ each of size $B_i = B_j$. A naive approach would treat these as independent and perform greedy agglomerative merging between elements of $\mathbf{i} \cup \mathbf{j}$. But since both partitions were generated from samples of the same graph data, processed the same way, the elements of $\mathbf{i}$ have a near one-to-one mapping to the elements of $\mathbf{j}$. (This is easy to see in the limiting case when each partition contains every community from the input graph.)

A rough agglomerative merge of partitions from two subgraphs can proceed as follows. First form the inter-block edge count matrix M based on each partition with dimensions $(B_i + B_j) \times (B_i + B_j)$. Then for each $r \in \mathbf{i}$, find the corresponding $s \in \mathbf{j}$ that minimizes the change in entropy when r is merged with s. The final merged partition has size B_j.

When generalized across subgraph partitions, the algorithm will take 16 partitions, merge down into 8, then lump all blocks from all eight together and proceed with stochastic block partition, as if the Golden bracket search has not started and the exponential block reduction is still in effect. Here we made one small change to use a smaller reduction rate, 0.35 instead of 0.50 as in the baseline algorithm. The reasoning is that merged partitions are much closer to the stopping point and hence should be searched more finely.

Thus our approach to top-down subgraph partitioning and merging is a combination of these ideas. It partitions the input graph, finds block partitions from each subgraph until the Golden ratio bracket is established, aggregates proto-communities from each piece, does a fast and rough agglomerative merge, and then continues with a merge-and-move approach of the baseline algorithm with the unified blocks. Because most of the runtime of the overall baseline algorithm is spent before the Golden ratio bracket is established, this approach yields considerable performance gains, while preserving accuracy.

Note that our approach, while fast, cannot arbitrarily divide the input graph into smaller and smaller pieces. At a point where the number of pieces approaches the number of actual block partitions, the community structure in each subgraph is lost. This algorithm is also amenable to efficiently merge in streaming graph data to an existing partition when the streamed pieces are sufficiently large. Although we do not examine streaming graph data here, we plan to explore this in future work.

V. Implementation

For our implementation, we started with the serial baseline algorithm implementation written in Python which we then modified and optimized. The major advantage of developing in Python is the ease of development and rapid prototyping relative to C++. Our implementation uses NumPy[10] for low-level array operations, Python multiprocessing with shared memory for single-node multi-core operation, and MPI for multi-node operation using mpi4py[11]. Although our focus in this work is on scalability and algorithmic performance, it is worth noting that the raw performance of optimized C++ is difficult to match with Python, especially for small graphs and low CPU counts where language runtime overheads are higher.

A. Nodal Updates with Shared Memory

Nodal updates are inherently write-intensive, have a relatively high ratio of communication to computation, and are dependent on updates from other workers. Global updates to shared state can be an expensive operation due to the volume of data, especially for the interblock edge count matrix.

Based on simulations of the impact of batch sizes for MCMC nodal updates, we designed our implementation to allow careful management of the granularity of messages in both directions, and with an effort to minimize the amount of data transferred.

Our design uses a pool of worker processes, each of which proposes nodal movements and computes the associated acceptance probability and accepts or rejects accordingly. Each worker is responsible for a group of graph vertices. A worker will report the changes in state back to a main aggregator. The aggregator reads nodal movements from each worker, computes the impact on shared global state, and uses a shared memory mechanism to efficiently pass these updates back to each worker. A worker only checks for changes in shared state at the beginning of an update cycle for a node, and makes a private copy only if the state has changed. Furthermore, in many cases, only those entries in the shared state that actually changed are copied. This is done by tagging each global update with an identifier, including which entries are affected.

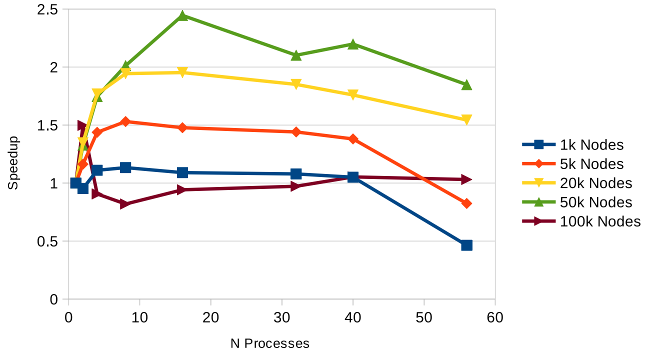

Fig. 1. Scalability of Baseline Parallel C++ Implementation Single-Node Multi-CPU.

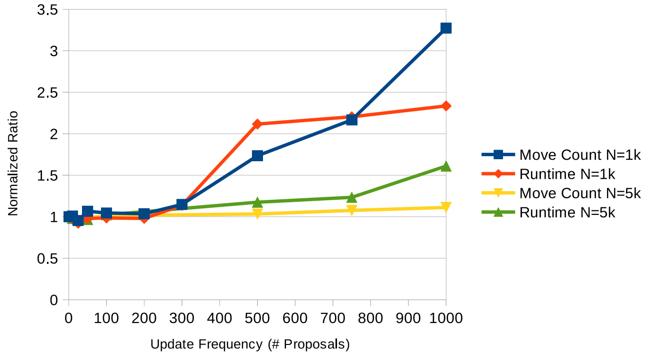

Fig. 2. Single-Threaded Effect of Nodal Update Frequency. The number of nodal moves and algorithm runtime are shown as ratios over the normal configuration which has an update frequency of 1 proposal (i.e. instant visibility).

Separating the aggregation of shared state from the workers proposing updates does create some overhead because certain information has to be re-computed. However these avoid heavy-weight synchronization mechanisms such as row locks, and allow careful batching of both shared state updates from main to the workers as well as nodal movement updates from the workers to main.

B. Multi-node MPI

Our top-down algorithm runs isolated block partition instances before a final merge, it makes it ideal for cluster computing situations where communication is through message passing that is relatively more expensive than shared memory IPC. To explore this configuration, we wrote an MPI implementation of our algorithm with mpi4py. To preserve performance, low-level nodal updates and agglomerative merging are still done with the multiprocessing library and using shared memory updates while high-level MPI-spawned instances handle different subgraph pieces on each compute node.

VI. Evaluation

Our static partitioning experiments use the baseline datasets from GraphChallenge, as well as some synthetic datasets we generated using the reference graph generator program.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

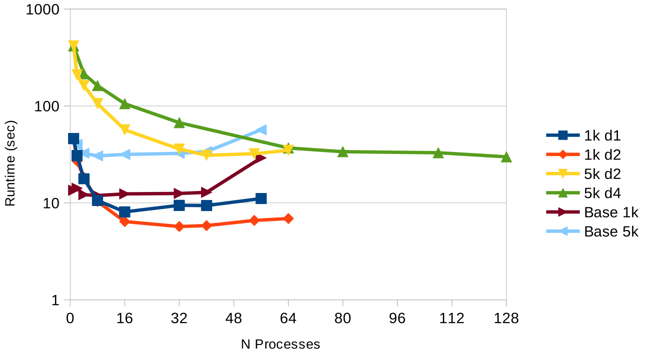

Fig. 3. Runtime of small graphs baseline and our implementation dividing into 1, 2, and 4 pieces.

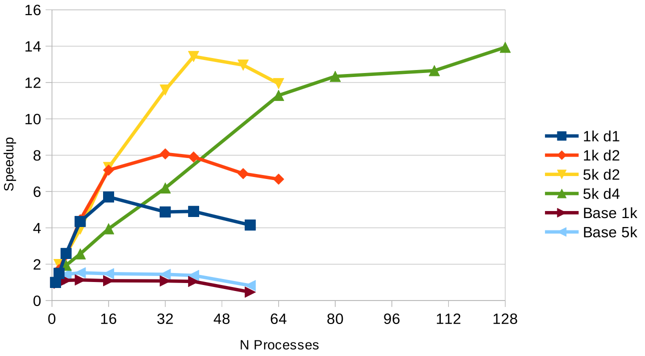

Fig. 4. Scalability of small graphs baseline and our implementation dividing into 1, 2, and 4 pieces.

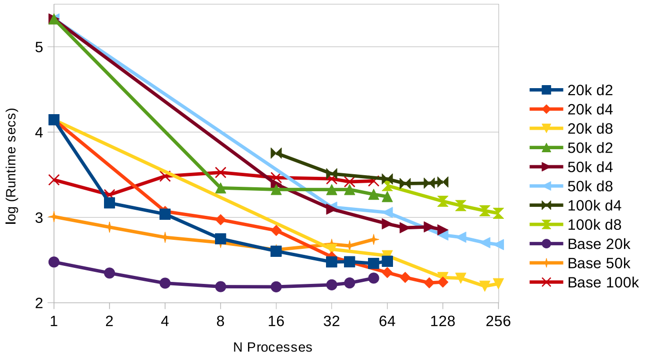

Fig. 5. Runtime of large graphs baseline and our implementation dividing into 2, 4, and 8 pieces. Note the log scale used on along both axes.

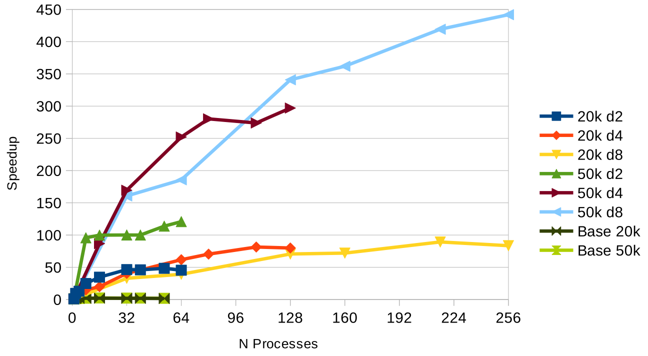

Fig. 6. Scalability of large graphs baseline and our implementation dividing into 2, 4, and 8 pieces.

Summary statistics for the static partition datasets are shown in Table I.

Our single-node test machine is an Intel Xeon E5-2683 server with 56 cores and 512GB RAM. Our multi-node cluster evaluation was done on Amazon EC2 with four m4.4xlarge instances, each with 64 CPUs and 64 GB RAM.

The runtimes for small-sized (N=1k, 5k) graphs are shown in Figure 3 while the associated speedups are shown in Figure Figure 4 as the number of processes increases. The different lines here show the two baseline C++ results compared to the different subgraph divide sizes for the top-down divide and merge algorithm. While the Python implementation is slower, the overall speedup as the number of processes increases shows much better speedup. Indeed, the speedup is robust to over-subscription of the number of processes relative to the number of CPUs (56 in this case). Our best parallel Python speedups over the fastest baseline parallel C++ run at the equivalent graph sizes are 2.09 and 1.02 for N=1k and 5k respectively.

The equivalent results for large-sized graphs (N=20k, 50k, 100k) show similar results in Figure 5 and Figure 6. The runtimes for our version greatly lag the baseline C++ at small numbers of processes, but scale much better. Our best parallel Python speedups over the fastest baseline parallel C++ run at the equivalent graph sizes are 0.984, 0.869, and 1.649, for N=20k, 50k, and 100k respectively.

The scalability for both large and small sizes also show the effect of the new top-down divide and conquer algorithm. The curves for speedups obtained by dividing into more pieces, come to dominate the curves for a smaller number of pieces. This shows how dividing the original graph into subgraphs reduces the overall amount of computation required. These gains are limited by the eventual loss of community structure from dividing the input into too many pieces. In our testing, dividing into 8 pieces preserves accuracy for large graph sizes while yielding good results.

Our scalability is also good for the MPI implementation running on our EC2 cluster. These results are shown in Figure 7 and Figure 8. For a 20k vertex graph, the speedup at 4 nodes (dividing into 4 subgraphs and using 256 processors total) is 61x faster than the single processor runtime.

Here we briefly discuss the precision and recall of our technique. These are summarized in Tables 2 and 3. Overall, our algorithm produces fairly good precision and recall numbers especially relative to the baseline C++ implementation. We

TABLE I. STATIC GRAPH STATISTICS

N Vertices	E Edges	Truth Partitions
1000	20135	11
5000	101973	19
20000	408778	32
50000	1018039	44
100000	2037415	56

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

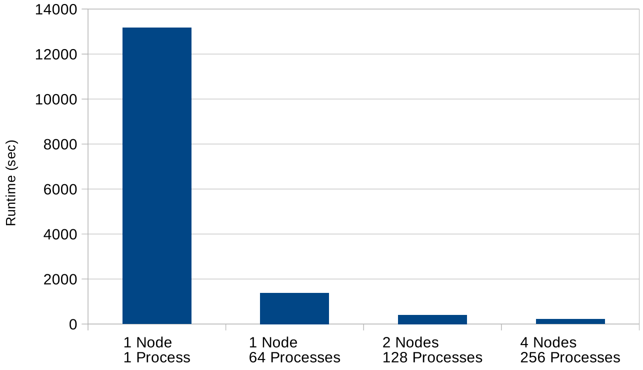

Fig. 7. Runtime of MPI implementation for 20k graph on 1, 2, and 4 nodes with 64 CPUs each.

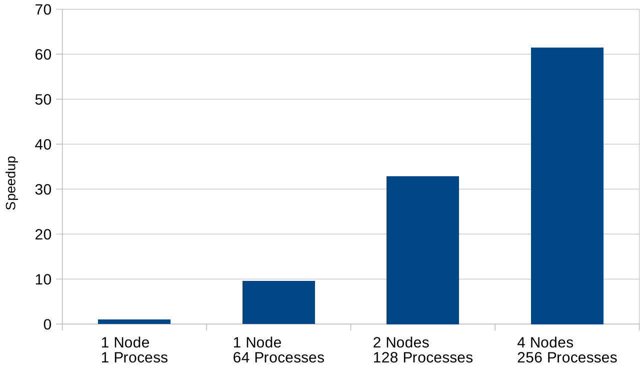

Fig. 8. Scalability of MPI implementation for 20k graph on 1, 2, and 4 nodes with 64 CPUs each.

have noticed that errors during top-down subgraph processing can propagate back to the solution, especially affecting the recall.

VII. Future Work

Our preliminary work in this area suggests several potentially fruitful areas for future work. First, an optimized C/C++ implementation is likely to be significantly faster than our Python implementation. Second, our top-down merge algorithm can likely be applied to efficiently process streaming graph data.

TABLE II. Pairwise Precision

N Vertices	Baseline	Div 2	Div 4
1000	1	1	-
5000	1	0.9314	1
20000	1	0.8445	0.9133
50000	1	0.7318	0.9536
100000	0 .9982	-	0.9113

TABLE III. Pairwise Recall

N Vertices	Baseline	Div 2	Div 4
1000	0.9425	1	-
5000	0.8822	0.9848	1
20000	0.8358	0.9634	1
50000	0.8443	0.9918	0.9401
100000	0.8736	-	0.9631

More prominently, the graphs in our synthetic datasets, as well typical graphs used in real-world community detection applications tend to be highly sparse. Indeed for the $N = 100000$ graph, there are $E = 2037415$ edges, or about 0.02 percent.

We experimented with using sparse array implementations to manage the inter-block edge count matrix, but found that current implementations, including linked-list, compressed sparse column, and compressed sparse row are very slow. This is because the matrix is traversed both by row and by column. A better data structure to support efficient traversal of non-zero entries across either dimension could be of great benefit.

VIII. Conclusion

We have described our preliminary work towards scaling up the stochastic block partition. We have developed a prototype that shows good scalability across single-node multi CPU and multi-node systems. In the future, we would like to develop and enhance this algorithm further while improving the accuracy further.

IX. Acknowledgment

Funding for this work was supported in part by The National Science Foundation and by The Raytheon Company.

References

[1] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith, “Streaming graph challenge: Stochastic block partition,” 2017.

[2] Y. Jin and J. F. Jaja, “A high performance implementation of spectral clustering on cpu-gpu platforms,” in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 825–834.

[3] H. Kanezashi and T. Suzumura, “An incremental local-first community detection method for dynamic graphs,” in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3318–3325.

[4] T. P. Peixoto, “Entropy of stochastic blockmodel ensembles,” *Physical Review E*, vol. 85, no. 5, p. 056122, 2012.

[5] ——, “Parsimonious module inference in large networks,” *Physical review letters*, vol. 110, no. 14, p. 148701, 2013.

[6] ——, “Efficient monte carlo and greedy heuristic for the inference of stochastic block models,” *Physical Review E*, vol. 89, no. 1, p. 012804, 2014.

[7] J. Kiefer, “Sequential minimax search for a maximum,” *Proceedings of the American mathematical society*, vol. 4, no. 3, pp. 502–506, 1953.

[8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[9] W. K. Hastings, “Monte carlo sampling methods using markov chains and their applications,” *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[10] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[11] L. Dalcín, R. Paz, M. Storti, and J. DElía, “Mpi for python: Performance improvements and mpi-2 extensions,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, 2008.