SCALABLE ASYNCHRONOUS CONNECTED COMPONENTS DETECTION BASED ON A PARALLEL UNION-FIND ALGORITHM

BY

SENTHIL KUMAR KARTHIK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Laxmikant V. Kalé

ABSTRACT

Connectivity in a graph is a well-studied problem. Various parallel algorithms to detect and label connected components exist, many of which are optimized for a shared-memory environment. However, scientific and engineering applications today process large-scale graphs that do not fit in a single compute node. This calls for a highly scalable solution to the connectivity problem. We propose a novel distributed-memory parallel algorithm based on the Union-Find data structure and asynchronous messaging. We strengthen the scalability of our approach by introducing several optimization techniques for parallel execution. The algorithm is implemented as a library using Charm++, a migratable object-based parallel programming model, allowing any Charm++ application to easily perform connected components detection. MPI applications may also use the library either via Adaptive MPI, or by using interoperability features of Charm++. In addition, the library will also support reading data from the disk. As a driving use case we utilize the library in ChaNGa, a cosmological simulation framework, to detect clusters of stars and classify galaxies. We evaluate the performance of our algorithm for real and synthetic graphs, computing connectivity on a probabilistic mesh benchmark with over 250 million edges in under 10 seconds using 4,096 cores of the Blue Waters (Cray XE) Supercomputer.

To my parents, for their love and support.

ACKNOWLEDGMENTS

This work would not have been possible without the guidance of my advisor Professor Laxmikant Kalé. I would like to thank him for his insightful ideas and approaches on designing our algorithm and optimizing the implementation. His passion and energy has inspired me greatly and helped me progress consistently towards completing this work.

I would like to thank Professor Edgar Solomonik for his time and patience in analyzing the complexity of our algorithm, and also for his suggestions on implementing the global path compression optimization. Nihar Sheth has played a critical role in the development of this work. I would like to thank him for his help in adding TRAM support to the implementation and running various benchmarks on clusters for performance evaluation. I would also like to thank my good friend and colleague, Jaemin Choi, for his constant support and valuable suggestions on writing this thesis.

Much gratitude to my colleagues at the Parallel Programming Laboratory (PPL) for their generous help in solving any technical issues. I would also like to thank all my friends at UIUC for being helpful and cooperative. All the results presented in this thesis were performed on the Blue Waters Supercomputer maintained by the National Center for Supercomputing Applications.

Finally, I am extremely grateful to my parents for being a pillar of support in my life.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Identifying connected components of a graph is a fundamental problem in a variety of applications, such as finding communities from protein interaction graphs [1], identifying galaxies from the stars in the universe [2], and grouping connections in social network services [3].

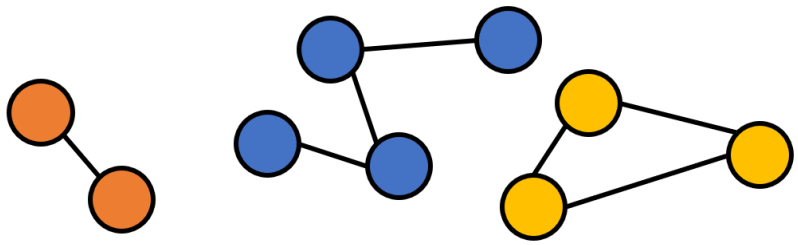


Figure 1.1: Connected components of a graph

Given an undirected graph $G = (V, E)$, with V as the set of vertices and E as the set of edges, a connected component of G is a maximal-size subgraph where a path exists between every pair of vertices in the subgraph [4]. The goal of connected components detection is to find all distinct connected components of the graph, which can be achieved in linear $O(|V| + |E|)$ time using a sequential algorithm [5]. At the end of the algorithm, each vertex has a representative vertex that represents the connected component it belongs to. It is worth noting that each vertex belongs to only one connected component. Two canonical approaches to solving the connectivity problem sequentially are (1) a graph traversal algorithm, i.e. breadth-first search (BFS) or depth-first search (DFS), and (2) a Union-Find [6] based algorithm. With large-scale graphs, however, the linear time complexity of sequential algorithms calls for parallelization either on a shared-memory or even a distributed-memory machine. Moreover, large-scale graphs in distributed-memory applications will be partitioned across multiple compute nodes, which would require connected components to also be processed in a distributed fashion for load balance and performance.

Parallel graph traversal algorithms [7] [8] perform multiple searches concurrently to identify a connected component that contains the source node. The inherent problem of these algorithms is that they are only able to process connectivity for a single component at a time. This is because multiple partial search trees in the same component created from parallel execution cannot be merged [5]. Our parallel algorithm takes the Union-Find approach instead, and is implemented as a library using the Charm++ [9] programming model.

The motivation to encapsulate the algorithm as a library comes from the significance of connected components detection in large-scale cosmological simulation applications such as ChaNGa [10]. Galaxy detection in ChaNGa is based on the Friends-of-Friends algorithm [11], which performs a series of Union-Find operations to detect clusters of stars and classify galaxies. Naturally, applications in other domains will be able to utilize the library and perform connected components detection in parallel with ease.

CHAPTER 2: BACKGROUND

2.1 SEQUENTIAL ALGORITHMS FOR CONNECTIVITY

2.1.1 Graph Traversal Methods

Either breadth-first search (BFS) or depth-first search (DFS) can be used to find connected components of an undirected graph. Initially, all nodes of the graph are marked as unvisited. We repeatedly pick an unvisited node, visit all reachable nodes by using either traversal method, and mark visited nodes with a unique identifier until all nodes are visited. Ultimately, nodes with the same identifier belong to the same connected component.

2.1.2 Union-Find Methods

Union-Find operations are used to maintain a collection of disjoint sets $\{S_1, S_2, ..., S_k\}$ of vertices $(x, y, ...)$ in a graph:

- $Union(x, y)$: merge S_i and S_j to $S_i \cup S_j$, where $x \in S_i \wedge y \in S_j$.
- $Find(x)$: return the unique identifier of the set containing x.

Each disjoint set is represented as a tree of vertices where each vertex maintains a pointer to its parent in the tree. The algorithm begins with all the vertices as singleton disjoint sets indicated by setting their parent pointers to null. Edges in the graph are then processed by issuing Find operations on the two endpoints of an edge. The returned identifiers can be compared to determine if the vertices of interest are in the same set. If they are different, the Union operation is performed to merge the two disjoint sets by setting the parent pointer of one of the set identifiers (root of the tree) to the other. This subsequently reduces the total number of disjoint sets in the domain. This algorithm can be applied to connected components detection, where the Find operations are used to identify two components that should be combined to a single connected component using the Union operation.

2.2 PARALLEL ALGORITHMS FOR CONNECTIVITY

Some of the well known parallel techniques for detecting connected components are explained in this section.

2.2.1 Parallel Graph Traversal

The parallel versions of graph traversal methods such as BFS and DFS can be used to compute connectivity of a graph. They have been found to be optimal and perform well for small-scale graphs [7], but suffer from the aforementioned deficiency of being able to determine connectivity for only one component at a time.

2.2.2 Shiloach-Vishkin Algorithm

The Shiloach-Vishkin algorithm [12] is a well-known PRAM algorithm used to detect connected components. It computes connectivity of all vertices in the graph concurrently in $O(\log n)$ time. The algorithm starts with all vertices as singleton trees, repeatedly performing *hooking* and *shortcutting* steps to combine the trees that are analogous to connected components. The hooking step is used to merge two connected components, and the shortcutting step is performed to reduce the amount of pointer chasing to find the root of a tree.

2.2.3 Parallel Union-Find Methods

One of the earliest efforts in parallelizing the classical Union-Find algorithm was done by Cybenko et al. [13]. They present parallel Union-Find algorithms for detecting connected components in both shared-memory and distributed-memory machines. Their hypercube-based algorithm for distributed memory replicates vertex information on all processors and constructs local spanning forests which are then aggregated together at one processor after $\log p$ steps. However, this approach did not show promising results as the execution time increased when more processors were used for a fixed size problem.

Anderson and Woll [6] present an efficient wait-free algorithm for the Union-Find data structure on shared-memory machines. They use atomic instructions to implement concurrent Find operations by using *path-halving* or pointer jumping. They analyze an algorithm to perform concurrent Union operations by maintaining consistent view of data across all threads while concurrently updating the data structures. While their asynchronous algorithms are applicable for a shared-memory programming model, they do not seem well suited for a distributed-memory setting.

Manne and Patwary [14] present a scalable parallel Union-Find algorithm specifically for distributed memory machines. They follow a two stage approach. In the first stage sequential Union-Find algorithm is used to build local spanning forests on each processor,

and these forests are combined in the second stage using classical or zigzag parallel Union-Find techniques.

Our algorithm and implementation are asynchronous in nature and our scalability analysis extends beyond those presented in previous studies [14, 13, 5].

2.3 THE CHARM++ PROGRAMMING MODEL

Charm++ is a migratable object and task based parallel programming model with an adaptive runtime system. The problem domain is decomposed into objects called *chares* containing data and methods that manipulate the data. The chares communicate with each other by asynchronously invoking methods in a message-driven fashion. The execution of an object's method corresponds to a task: the unit of execution that is scheduled by the runtime on a *processing element* (PE, typically a CPU core). Charm++ adopts *overdecomposition*, where there are generally many more objects than the number of PEs. This empowers the runtime system to make intelligent scheduling decisions and achieve overlap of computation and communication, as well as perform load balancing [15].

2.3.1 TRAM Library for Message Aggregation

Processing elements in the Charm++ model communicate using messages to transfer data and invoke methods on chares. Each message carries a certain overhead, mostly comprised of a fixed cost per message and a variable cost for the size of the message. The Topological Routing and Aggregation Module (TRAM) [16] improves performance of distributed Charm++ programs by bundling outgoing messages at each PE and routing them through a virtual N-dimensional mesh mapped on the processes. Each PE only sends and receives messages from its neighbors in the mesh. As each PE receives data from its peers, it either processes the data if it is the intended recipient, or buffers the data for forwarding if it is an intermediary destination. By aggregating all outgoing data submitted locally and received from peers, the per-message cost is significantly amortized while ensuring data is still received quickly (within N hops). In programs with large amounts of communication, the gains of amortizing message costs significantly outweigh the cost of increasing distance the average message travels.

CHAPTER 3: ALGORITHM

We implement a parallel algorithm based on the union-find (also called disjoint-set) data structures. The classical Union-Find algorithm maintains a collection of non-overlapping sets of elements. Each set is represented by a rooted inverted tree. More details about this method are given above in Section 2.1.2.

3.1 DESIGN CRITERIA

In theory the running time for the Union-Find algorithm is bound by

$$O(|V|\alpha(|E|, |V|))$$

where α is the slow growing inverse Ackermann function [17]. Many sequential optimizations like *Union-by-rank* and *path compression* are also used to ensure a rooted star structure for the inverted trees.

However one of the major challenges that we face in extending this algorithm into an asynchronous distributed environment is *race conditions*. Cyclic graphs with uniform decomposition into PEs lead to the formation of cycles in the inverted trees. An example is shown in Figure 3.1.

Other challenges include:

- fine-grained communication overhead
- equal distribution of Union and Find operations across PEs
- implementing well-known optimizations like path compression with no temporal or spatial locality

Our choice of the programming model and core algorithm design for the library tackles all these challenges in an effective manner. We also implement our own version of distributed asynchronous path compression as described in Section 3.3.

3.2 PARALLEL UNION-FIND ALGORITHM

Our algorithm detects connectivity in a graph by performing operations on vertices represented using distributed trees. We take the classical Union-Find approach but execute each query in parallel via asynchronous messages.

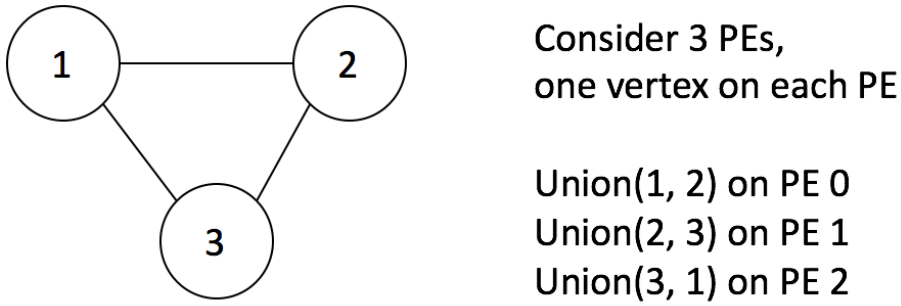


Figure 3.1: Cyclic graph with uniform distribution causing race conditions

Given a graph $G = (V, E)$, for every edge $e = (v_1, v_2)$ we execute a $Union(v_1, v_2)$ operation. The following is our proposed algorithm for performing this Union operation:

Algorithm 3.1 Message-based Asynchronous Union-Find

1: Message v_1 to perform $Find(v_1)$
2: Chain of messaging parent until root, where $boss_1 = Find(v_1)$
3: $boss_1$ messages v_2 to perform $Find(v_2)$, carrying information of $boss_1$
4: When $boss_2 = Find(v_2)$, set $boss_1$ as parent of $boss_2$

Effectively we are constructing a distributed forest of inverted trees where each tree represents a unique connected component. A vertex belongs uniquely to one of these trees. Figure 3.2 presents this algorithm graphically. Note that in Charm++, the messages in the algorithm can be sent asynchronously, enabling distributed parallel processing of all edges in the graph.

3.2.1 Solving Race Conditions

To prevent race conditions, we enforce a strict ordering in the Union operation based on the vertex ID. This brings in an additional min-heap like property to the inverted tree - the ID of the parent node is always smaller than the IDs of its children. A cycle edge can be detected if a node with a lower ID is asked to point to a node with a higher ID in the Union operation.

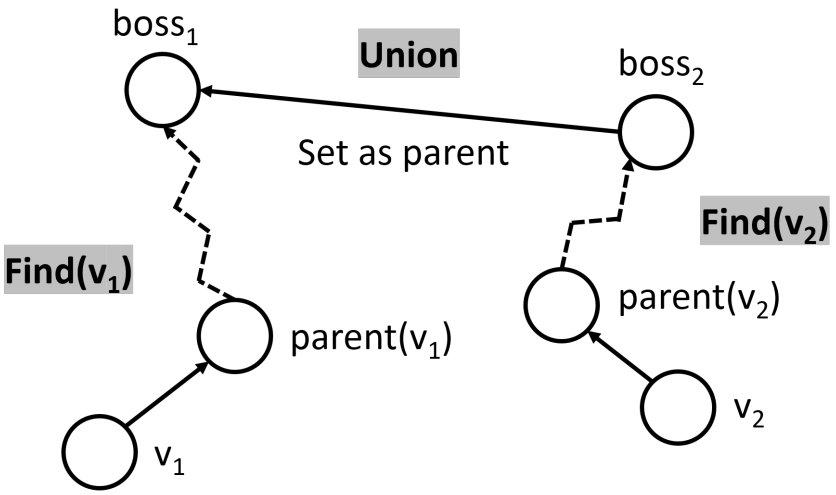


Figure 3.2: Message-based Union-Find operations

3.3 SELECTIVE ASYNCHRONOUS PATH COMPRESSION

We perform path compression at both local (within PE) and global (across PEs) scales during the *Find* operation. The local path compression algorithm ensures that all the trees within the PE are completely shallow i.e. rooted star.

Algorithm 3.2 Local Path Compression

1: Save a reference to the first vertex of path in v_0
2: **while** $parent(v_1)$ is on current PE **do**
3: $\quad v_1 = parent(v_1)$
4: **end while**
5: **while** $parent(v_0)$ is not v_1 **do**
6: $\quad tmp = parent(v_0)$
7: $\quad parent(v_0) = v_1$
8: $\quad v_0 = tmp$
9: **end while**

Global path compression is done towards the end of a *Find* operation to perform short-cutting of paths spanning across multiple PEs. The sender vertex is messaged to assign its parent pointer to a grandparent instead of the current immediate parent (pointer jumping).

CHAPTER 4: IMPLEMENTATION

The library is designed using the bound-array [18] concept in Charm++. There are three primary phases involved in our implementation:

- **Phase I**: Build forest of inverted trees using asynchronous Union-Find algorithm
- **Phase II**: Identify bosses of each component and label all vertices in that component in parallel
- **Phase III**: Prune out insignificant components based on a threshold (post-processing)

To detect the completion of each phase, we use in-built quiescence detection feature of Charm++. We have tested and verified the correctness of our implementation on real-world graphs, including protein structures obtained from RCSB Protein Data Bank [19].

4.1 ASYNCHRONOUS UNION-FIND

The first phase of the library implements the steps outlined in Algorithm 3.1. We check for strict ordering of vertex IDs for an edge during the Union and Find operations to prevent any race conditions and ensure correctness. We also handle prerequisites like library chare array initialization, obtaining local pointer to vertices and callback registration from the application in this phase. At the end of this phase program control is returned back to application indicating completion of the construction of the forest of inverted rooted trees, where each tree represents a connected component.

Listings 4.1 - 4.3 provide the high level pseudo-code for the three pivotal entry methods of this phase - $union\_request$, $find\_boss1$, and $find\_boss2$.

In addition to the above entry methods we added some optimizations to this phase in order to fine-tune the performance of tree construction for very large-scale graphs. These optimizations are explained below.

4.1.1 Locality-based Tree Climbing

For this optimization, we sequentially traverse the tree path instead of using costly messages until we come across a vertex that lies on a different chare. This increases the amount of work per chare but drastically reduces the total number of messages. After integrating this optimization, we observed a 25x speedup in this phase alone.

High level pseudo-code for Phase I entry methods

```
union_request(v1, v2) {
    if (v1.ID > v2.ID)
        union_request(v2, v1)
    else
        find_boss1(v1, v2)
}
```

Listing 4.1: union_request

```
find_boss1(v1, v2) {
    if (v1.parent == -1)
        find_boss2(v2, v1)          // v1 is boss1
    else
        find_boss1(v1.parent, v2)
}
```

Listing 4.2: find_boss1

```
find_boss2(v2, boss1) {
    if (v2.parent == -1) {
        if (boss1.ID > v2.ID)
           union_request(v2, boss1)
        else
           v2.parent = boss1
    }
    else
        find_boss2(v2.parent, boss1)
}
```

Listing 4.3: find_boss2

4.1.2 Message Aggregation

Our initial profiling and analysis showed that the tree construction phase is highly communication intensive as many fine-grained messages are created for every edge that is processed. For example, nearly 1.5 billion messages are generated for a graph with 16 million vertices and 6 million edges. We added support to perform topology aware routing and aggregation of messages using the TRAM [16] library in Charm++. This significantly reduced the communication overhead specifically for graphs running on many nodes.

4.2 PATH COMPRESSION

Asynchronous path compression techniques are also applied during the first phase of the library to ensure shorter tree paths thereby optimizing the Find operation. We perform path compression both at local and global scales, as explained below.

4.2.1 Local Path Compression

For this technique we implement the steps outlined in Algorithm 3.2. The active messages that drive the Find operation carry some additional data about the vertex that initiated the current path ($v0$). Accordingly this information is used to compress the path before the recursive calls are made in $find\_boss1$ and $find\_boss2$ functions. Local path compression increases the amount of serial work per chare but greatly boosts the speed of future Find operations that end up using the same path, as the inverted subtrees within the chare are completely shallow (rooted star).

4.2.2 Global Path Compression

To compress paths that are spanning across chares we implement pointer jumping operation [20] using asynchronous messaging between these chares. When a chare identifies that its parent vertex in Find operation is on a different chare it sends a message ($short\_circuit\_parent$) to the first vertex on path providing information of the grandparent vertex which is set as its new parent. While global path compression increases communication due to messages, we were able to bring down the overhead by aggregating messages using the TRAM library in Charm++.

Note on Global Path Compression

There are two other ways to implement global path compression during the tree construction phase. However our implemented pointer jumping based technique proves to be more efficient with least overhead.

1. Collect IDs of all vertices along the path during a Find operation. On reaching the boss of the component, this information is used to message each vertex on path to reassign the parent pointer. However, this introduces variable sized messages for the Find operation which cannot be used for aggregation in the TRAM library.

2. The boss of a tree can message the source leaf vertex to start another chain of messages for reassigning the parent pointers. However, the priority of these messages should be higher than those for Find operations to avoid ineffective path compression. This is because the asynchronous nature of the algorithm can proactively grow the tree even before the path compression messages are processed. Priorities are best effort semantics in Charm++ and thus they cannot be used to establish a formal proof of correctness of our algorithm.

4.3 COMPONENT LABELLING

In the second phase of the library we label each vertex of the graph with its component ID (identified by boss vertex) in a fully asynchronous distributed fashion. All the vertices in the inverted trees request their immediate parent for the component information. If a vertex does not possess this information when a request arrives then that request is buffered into a local *need_boss_requests* queue. These requests are addressed when a vertex receives its component ID, thereby propagating the information downwards in the trees. We again use the TRAM library in this phase to minimize the communication overhead involved in sending *need_boss* messages.

4.4 PRUNING INSIGNIFICANT COMPONENTS

Phase III of the algorithm accumulates the number of vertices in each component, and prunes out components with less than a threshold number of vertices. The counts of each component ID is then broadcast to each PE. The original approach involved a global reduction where each reducer built a map of component IDs to the number of vertices with that component ID on each chare, combining the number of vertices with the same component

Figure 4.1: Projections timeline view of Phase III with map-based reducers. The horizontal axis shows time in microseconds and the vertical axis represents the timeline for each PE. The red regions indicate the time spent inside reducer functions for merging maps contributed by different PEs. The green regions show the time taken to distribute and rebuild the final merged map on each PE.

ID on a separate target chare. The overhead in performing such a map-based reduction is shown in Figure 4.1. To improve the runtime of this phase, an additional preproccessing step is performed in the beginning of Phase II, where a parallel prefix scan is done to renumber the component IDs to be sequential across all chares. This allows the use of arrays indexed by the component IDs instead of maps in the reducer as it mitigates arbitrary component IDs. Using this extra optimization, execution time on 4,096 cores running on a graph with ∼64 million vertices and edges fell from just over 20 seconds to under 4.

4.5 INTEGRATION WITH CHANGA

Our library is used to implement a friends-of-friends based galaxy detection feature in ChaNGa. At the end of a standard ChaNGa simulation i.e. after all timesteps and generating output files, we initialize the librarys chare array to shadow the current state of the `TreePiece` array in the simulation. Each particle in the tree piece represents a unique graph vertex and the interactions between these particles represent the edges. We perform a simple traversal on the particles using the Smooth framework [21] to identify their neighbors and thereby stream edges into the library as Union operations. The threshold for pruning components is taken as an input parameter during initialization of the simulation.

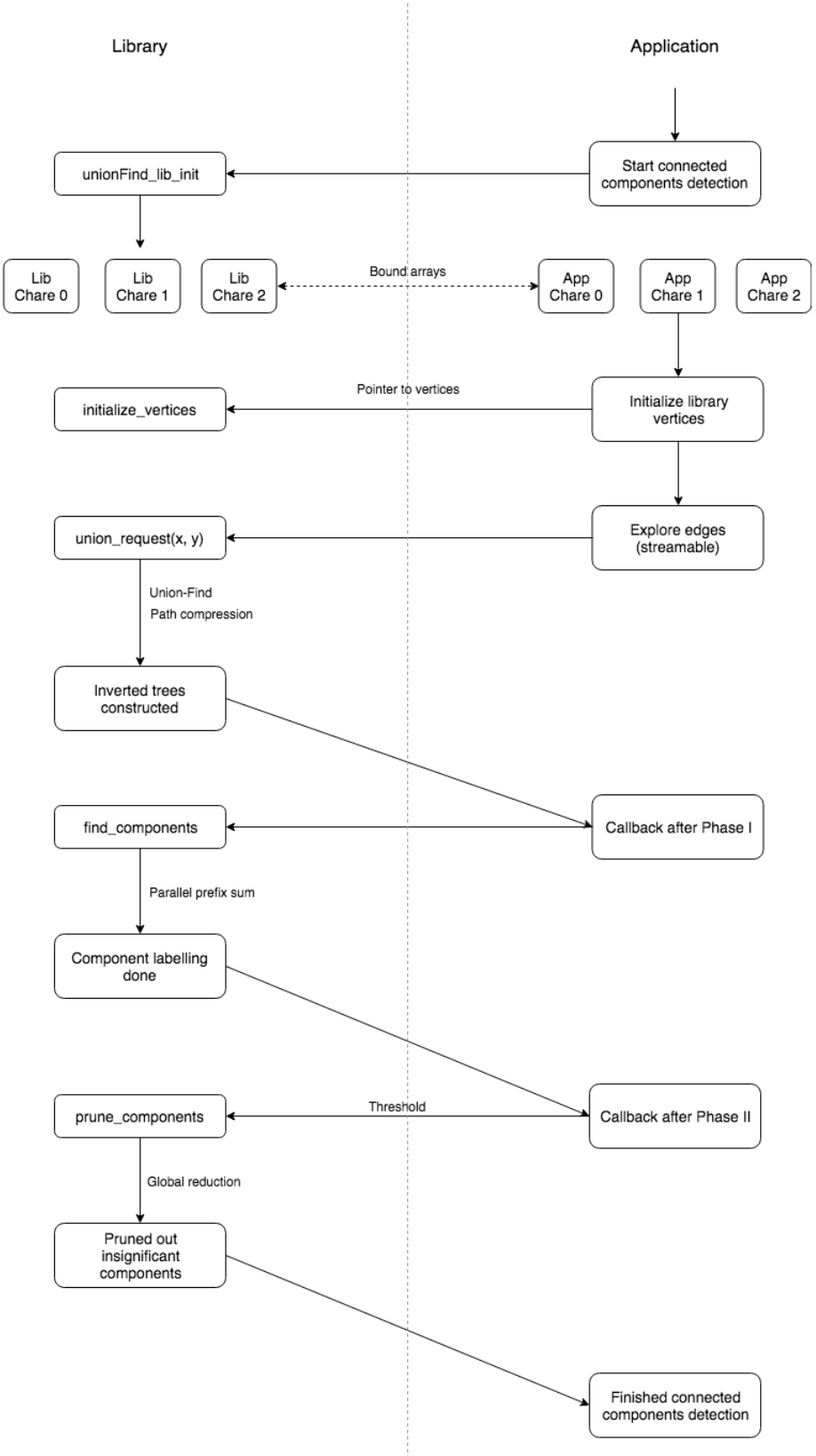


Figure 4.2: High level workflow diagram

CHAPTER 5: PERFORMANCE EVALUATION

We evaluate the performance of our library on the Blue Waters Supercomputer at National Center for Supercomputing Applications, using up to 128 Cray XE6 compute nodes with 32 cores each and 64GB of memory.

5.1 TEST PROBLEMS

The sample graphs for benchmarking performance were chosen from the following problem categories.

5.1.1 Probabilistic Mesh

To verify our algorithm, we use *probabilistic mesh* [22], which is a class of graphs motivated by cluster dynamics in computational physics (2D Ising model). It creates a random graph built on a lattice structure. An edge is created between two lattice points (vertices) based on a probability value determined by the coordinate positions. If the probability is low, the graph becomes sparsely connected and hence results in more components. The graph has less connected components otherwise.

With probabilistic meshes, we can easily scale the size of the input graph, verify the results and catch race conditions. If the probability and lattice size parameters are fixed, then the same graph is produced for different number of chares and PEs.

5.1.2 R-Mat Graphs

R-Mat [23] is a graph generator that chooses edges of a graph by taking an empty adjacency matrix, recursively partitioning it into 4 submatrices, and dropping edges into one of each four submatrices with some unequal probability. Through modifying the probabilities of choosing a partition, R-Mat graphs can closely model real-life graphs. It can match degree distributions of real graphs, display community-like structures, and create low-diameter graphs. The graph generator used to generate sample problems created symmetric graphs in the Problem-Based Benchmark Suite [24] format, which was converted into the undirected Protein Data Bank (PDB) graph standard. The symmetric nature of the PBBS graphs permitted a direct conversion without losing information about edges, and allows for conversion with low memory overhead even for exceedingly large graphs.

Graph	Vertices	Edges	Components
com-Amazon	334,863	925,861	70,533
com-Orkut	3,072,441	117,185,078	69

Table 5.1: Metrics of real world graphs for benchmark

5.1.3 Real World Graphs

Apart from synthetic graphs we also carried out performance analysis of the library with some large real world graphs taken from the SNAP datasets [25]. We picked the following two graphs for our benchmarks:

1. `com-Amazon:` A product network collected by crawling Amazon website to establish correlation between products commonly purchased by customers

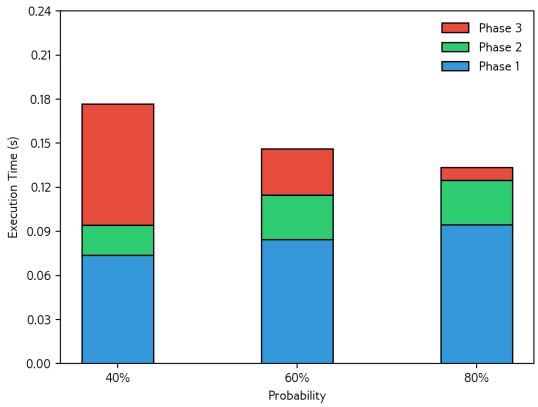
2. `com-Orkut:` An online social network where each vertex represents a user and an edge between users represents their friendship on the Orkut platform

Further details about these network graphs are provided in Table 5.1. The default format for SNAP graphs is a file containing the list of edges with one edge per line. However our benchmarking program expects the input graphs in the PDB file format. We carry out this translation in-memory by building the adjacency matrix representation of the graph.

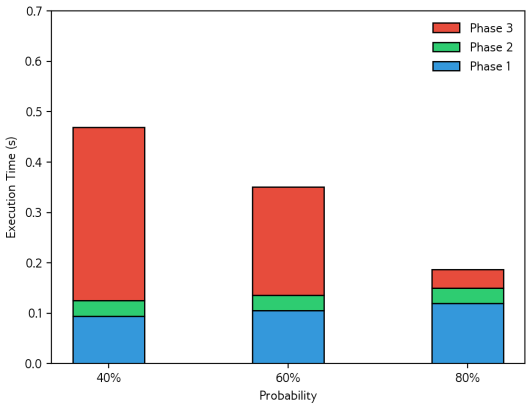
5.2 PHASE EXECUTION TIME

We carried out experiments where the mesh configurations were scaled from 1024^2 (1M) to 8192^2 (64M). The probability values used are 40%, 60%, and 80%. The mesh size per chare is fixed at 128^2 lattice points.

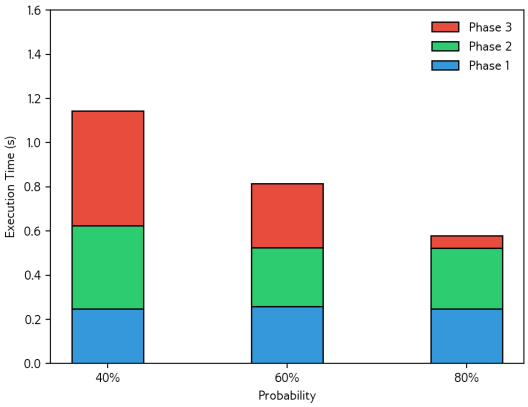
It can be seen from Figure 5.1 that the execution times of Phase I and Phase II stay relatively constant across different probability configurations and hence independent of the number of components in the graph. However, we notice that time taken for Phase III decreases as probability increases. This is expected behavior as the global reduction operation to collect component ID counts depends on the number of components in the graph. As the probability increases, the number of edges between lattice points increase and the resulting graph is more densely connected, leading to fewer number of components.



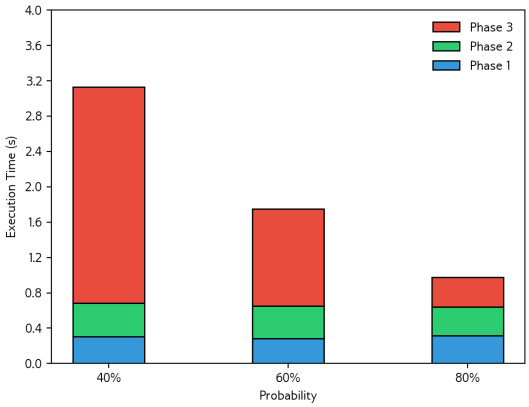
(a) Mesh size 1024^2 on 64 cores



(b) Mesh size 2048^2 on 256 cores



(c) Mesh size 4096^2 on 1024 cores

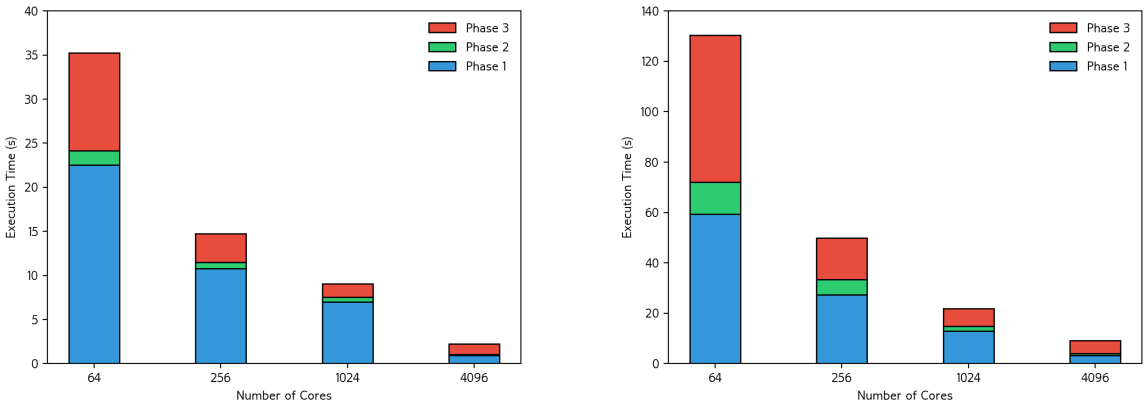


(d) Mesh size 8192^2 on 4096 cores

Figure 5.1: Phase execution time for different probabilities

5.3 STRONG SCALING WITH PROBABILISTIC MESH

We perform strong scaling analysis with meshes of size 8192^2 (64M) and 16384^2 (256M) using 60% probability, and the results are shown in Figure 5.2. We can observe good strong scaling up to 4096 cores of Blue Waters. Phase I executes much faster with a large number of cores as the amount of work per core decreases. Phase III shows a similar trend since more cores implies smaller number of components per chare thereby making the reduction operation faster and more efficient. We can also notice that the percentage of execution time spent in Phase III is higher for the larger mesh due to larger number of components (4x compared to mesh size 8192^2) which increases the overhead involved in the global reduction operation.



(a) Mesh size 8192^2 (b) Mesh size 16384^2

Figure 5.2: Strong scaling analysis with probabilistic meshes

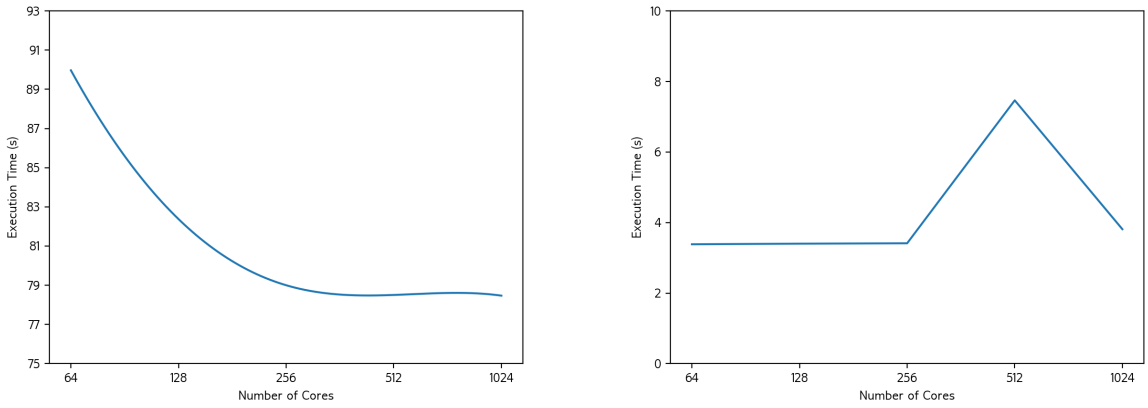
5.4 WEAK SCALING WITH R-MAT GRAPHS

Weak scaling runs were performed using R-Mat graphs, where the generator was specified with the default $A = 0.3, B = 0.1, C = 0.1, D = 0.5$. Tests were done with 32 cores per node and 1 chare per core. The generated graphs had moderate connectivity, with approximately 1.89 edges per vertex.

The results are presented in Table 5.2. While increasing problem sizes took multiplicatively longer despite increasing the core count proportionally, the scaling of each phase is approximately equal, at around 4-5x the previous size's runtime. Unlike probabilistic mesh, the R-Mat graphs were more likely to have localized communities among vertices with similar indices, and increasing problem sizes proportionally scaled the expected size of such communities. Larger communities leads to a larger number of Union operation requests to the roots of components, resulting in many localized messages to few chares and thereby significantly increasing the expected runtime.

Cores	**Graph Size**	**Phase 1 (in s)**	**Phase 2 (in s)**	**Phase 3 (in s)**	**Total Time (in s)**
32	2^{20}	2.128	0.654	0.069	2.852
128	2^{22}	7.458	2.319	0.341	10.119
512	2^{24}	29.186	9.517	1.677	40.380
2048	2^{26}	156.496	39.526	7.764	203.785

Table 5.2: Weak scaling results for R-Mat graphs



(a) Strong scaling of `com-Orkut` (b) Strong scaling of `com-Amazon`

Figure 5.3: Scaling analysis with real world graphs

5.5 SCALING REAL WORLD GRAPHS

We evaluate performance of the library on real-world graphs described in Section 5.1.3 by performing strong scaling runs ranging from 64 cores to 1024 cores. The computation grain size is fixed at one chare per core. The results are presented in Figure 5.3. `com-Orkut` being a dense graph shows a trend of decreasing execution time up to 1024 cores, however the scaling is not as good as the pattern observed with probabilistic meshes. `com-Amazon` on the other hand is a relatively sparse network and does not show scaling trends like `com-Orkut`. In fact we can observe from Figure 5.3b that the execution time remains consistent or increases as we increase the number of cores.

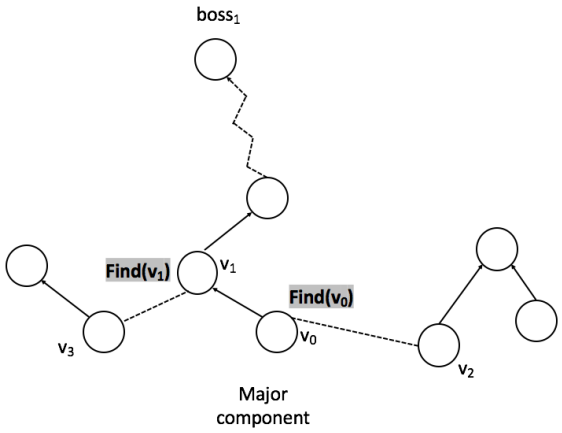


Figure 5.4: Bottleneck will be observed at $boss_1$ when edges (v_1, v_3) and (v_0, v_2) are processed during later stages of Phase I

We believe that the irregular nature of these graphs lead to potential bottlenecks at the roots of component trees. For graphs with very few number of components, the root of the biggest component tends to receive and process most of the Find operations and this can cause a serial congestion for the algorithm's progress in Phase I. An example scenario is highlighted in Figure 5.4 . We also observed that there can be cases where the component roots are unevenly distributed among the chares leading to load imbalance during component labelling in Phase II.

CHAPTER 6: CONCLUSION AND FUTURE WORK

We have introduced a scalable, asynchronous algorithm for connected components detection using a novel parallel Union-Find algorithm on distributed memory machines. We implemented it as a library using Charm++ and provided a simple ubiquitous interface for any general purpose application based on the programming model. We developed various optimizations to reduce communication overhead and speed up the critical path of our algorithm. The performance of the library is evaluated on the Blue Waters Supercomputer using various synthetic graph generators like probabilistic mesh and R-Mat, and also on real world graphs taken from the SNAP datasets. We presented a performance breakdown of the different phases of the library and demonstrated good strong scaling up to a large number of cores. Our results show that the library is able to detect connectivity accurately for a probabilistic mesh with over 250 million edges and 12 million components in just under 10 seconds using 4,096 cores. We have also integrated the library into the popular cosmology simulation framework (ChaNGa) to perform galaxy detection based on the friends-of-friends algorithm.

To the best of our knowledge, this work is the first implementation of a scalable, asynchronous Union-Find based connected components detection algorithm on large distributed memory machines. In the future, we plan on optimizing Phase I of the library for irregular graphs and improve weak scalability of the algorithm. Some initial ideas include algorithmic changes like preemptive Find operations to avoid serial bottleneck at the component root. We also plan on evaluating the performance and scalability of the library with very large (trillions of edges) ChaNGa datasets. Finally we would like to implement a Python interface for the library with integration into Charmpy [26] in order to explore avenues for solving the data clustering problem [27] which is widely used in many fields like machine learning and data mining.

REFERENCES

[1] L. Bennett, A. Kittas, S. Liu, L. G. Papageorgiou, and S. Tsoka, "Community structure detection for overlapping modules through mathematical programming in protein interaction networks," *PLOS ONE*, vol. 9, no. 11, pp. 1–15, 11 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0112821

[2] A. Hocking, J. E. Geach, Y. Sun, and N. Davey, "An automatic taxonomy of galaxy morphology using unsupervised machine learning," *Mon. Not. Roy. Astron. Soc.*, vol. 473, p. 1108, 2018.

[3] S. Dhingra, P. S. Dodwad, and M. Madan, "Article: Finding strongly connected components in a social network graph," *International Journal of Computer Applications*, vol. 136, no. 7, pp. 1–5, February 2016, published by Foundation of Computer Science (FCS), NY, USA.

[4] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, Aug. 1979. [Online]. Available: http://doi.acm.org/10.1145/359138.359141

[5] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, "An adaptive parallel algorithm for computing connected components," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, Sept 2017.

[6] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: ACM, 1991. [Online]. Available: http://doi.acm.org/10.1145/103418.103458 pp. 370–380.

[7] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063471 pp. 65:1–65:12.

[8] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810534 pp. 303–314.

[9] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/165854.165874 pp. 91–108.

[10] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kal, and T. R. Quinn, "Scaling hierarchical N-body simulations on GPU clusters," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[11] E. Tempel, R. Kipper, A. Tamm, M. Gramann, M. Einasto, T. Sepp, and T. Tuvikene, "Friends-of-friends galaxy group finder with membership refinement. Application to the local Universe," *Astron. Astrophys.*, vol. 588, p. A14, 2016.

[12] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0196677482900086

[13] G. Cybenko, T. G. Allen, and J. E. Polito, "Practical parallel union-find algorithms for transitive closure and clustering," *International Journal of Parallel Programming*, vol. 17, no. 5, pp. 403–423, Oct 1988. [Online]. Available: https://doi.org/10.1007/BF01383882

[14] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 186–195.

[15] J. Lifflander, G. C. Evans, A. Arya, and L. V. Kale, "Dynamic scheduling for work agglomeration on heterogeneous clusters," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 2404–2413.

[16] L. Wesolowski, R. Venkataraman, A. Gupta, J. S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing fine-grained communication with topological routing and aggregation of messages," in *2014 43rd International Conference on Parallel Processing*, Sept 2014, pp. 211–220.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[18] P. P. Laboratory, "Charm++ bound arrays," 2017. [Online]. Available: http://charm.cs.illinois.edu/manuals/html/charm++/13.html

[19] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The protein data bank," *Nucleic Acids Research*, vol. 28, no. 1, pp. 235–242, 2000. [Online]. Available: + http://dx.doi.org/10.1093/nar/28.1.235

[20] T.-S. Hsu, V. Ramachandran, and N. Dean, "Parallel implementation of algorithms for finding connected components in graphs," 1997.

[21] N-BodyShop, "Use the smooth framework for neighbors in changa," 2018. [Online]. Available: https://github.com/N-BodyShop/changa/wiki/Developer%27s-Guide#use-the-smooth-framework-for-neighbors

[22] S. S. Lumetta, A. Krishnamurthy, and D. E. Culler, "Towards modeling the performance of a fast connected components algorithm on parallel machines," in *Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 32–32.

[23] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, 2004, pp. 442–446. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[24] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312018 pp. 68–70.

[25] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[26] P. P. Laboratory, "Charmpy," 2018. [Online]. Available: http://charmpy.readthedocs.io/en/latest/

[27] W. contributors, "Cluster analysis — Wikipedia, the free encyclopedia," 2018, [Online; accessed 4-April-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Cluster_analysis