The Report Committee for Olivia Zhuhe Mitchell
certifies that this is the approved version of the following report:

Shortest-Path and Maximal Independent Set in PyKokkos

SUPERVISING COMMITTEE:

Milos Gligoric, Supervisor

George Biros, Reader

Shortest-Path and Maximal Independent Set in PyKokkos

by

Olivia Zhuhe Mitchell

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

The University of Texas at Austin

December 2023

Acknowledgments

Many thanks to all my long-suffering friends who has the patience to listen to all my techno-babble with minimal grimacing, and to my parents, grandparents, aunts, uncles and cousins, for all their support and encouragement in completing my degree. Also, to Nader Al Awar, Hannan Naeem, Aditya Thimmaiah and everyone else who set aside time to answer my questions (and let me into the lab when my ID didn't work!). Thank you George Biros for being my reader and Vijay Garg for both the knowledge shared in class, but also the knowledge shared outside of it. And lastly, to Milos Gligoric, who not only spent a great deal of time and effort helping shape a mess of code, comments and results into a readable paper, but also introduced me to new ways of thinking about code and also gave me the time and opportunity to play with an innovative, new library.

This material is partially based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

Abstract

Shortest-Path and Maximal Independent Set in PyKokkos

Olivia Zhuhe Mitchell, MS
The University of Texas at Austin, 2023

SUPERVISOR: Milos Gligoric

There are a wide variety of platforms to choose from when deciding to write a parallel algorithm. The chosen platform has a great impact on how developers write and run their code. Incompatibility between platforms is a major problem: in order for a developer to change systems or devices they often must rewrite their program from scratch.

PyKokkos is an alternative option to committing to a specific platform. It sits on top of the Kokkos framework that bridges the gap between various XPU devices. While Kokkos allows users to program parallel functions that can run expeditiously on both GPU and CPU in C++, PyKokkos also allows users to program in Python. Code written in PyKokkos has all the benefits of Python, while being both performant and portable.

Presented here are two examples of graph algorithms implemented for the first time in PyKokkos (or in Python): a parallel Maximal Independent Set algorithm and a parallel Shortest Path algorithm using Lattice-Linear Predicate. These implementations showcase the versatility of PyKokkos, where it makes writing parallel operations simple and easy, while not sacrificing performance.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1: Introduction

As computer systems get more complex and problem sizes increase, there is a greater need to run parallel computations both quickly and effectively. There is also a greater divide between the versatile CPU and the powerful but less flexible GPU. For developers who value flexibility, to run code on or across multiple device types, there are few good options. They must either choose and be bound to running on the CPU, or write code exclusively for a specific flavor of GPU.

Performance portability is the concept in which the same code written by developers can be run on different platforms without sacrificing the performance of writing directly for a specific CPU or GPU. The Kokkos [12, 28] framework is one such performance portable framework that not only allows the same parallel C++ code to be compiled to run on different platforms, but also automatically handles platform specific configurations such as how data must be organized and the abstraction of parallel threads.

PyKokkos [2, 3] makes performant portable programming accessible to developers new to the environment by allowing development to be done in Python; a language popular in the growing AI/ML space. Python has an enormous catalog of libraries available and allows for quick prototyping of a function or application. PyKokkos give users the ability to write code in Python, generate the device specific code using Kokkos, compile, and then bind and run the code and return the results back in Python, all in one step (from users' point of view). This allows PyKokkos users to quickly prototype parallel, performant code that can run on multiple devices.

Graph algorithms are a useful and popular family of problems. The two chosen for implementation here are the Maximal Independent Set and the ever popular Shortest Path problems. The Maximal Independent Set algorithm is based on a fast and highly parallel approach to solving MIS [9]. The Shortest Path algorithm is

implemented in a new format for deriving parallel algorithms that exist in a linear solution space. This specific MIS algorithm and Lattice-Linear Predicate format [14] in general has never been implemented in Python. These examples are the first graph algorithms implemented in PyKokkos. This is also the first approach that allows these particular algorithms to run on both CPU and GPU with the same source code.

The motivation behind this project is to illustrate the ease of which portable code can be written in a user friendly format, to demonstrate the performance differences between the same PyKokkos code on the GPU versus on the CPU, and to show the significant improvement in performance over existing Python implementations.

Chapter 2: Background

As the algorithms in this paper are implemented in PyKokkos [2, 3] and involve graphs, we provide some background on the Kokkos framework, PyKokkos framework, and graph algorithms.

2.1 Kokkos

Kokkos [12, 28] itself is a portable performance framework. It is implemented in C++, one of the more popular languages to write highly optimized and parallel code in.

Code written in Kokkos can be run on multiple device types, as Kokkos handles all the device specific libraries and limitations by using its own abstraction of memory spaces and parallelism. This allows for code developed in Kokkos to ignore complicated device details. An example of a Kokkos parallel section, taken from the Kokkos examples [12], is shown in Listing 2.1.

The Kokkos (and PyKokkos) *parallel_reduce()* or *parallel_for()* function contains a parallel section. In the parallel_reduce() in Listing 2.1, there are several parameters being passed in. The "02" is the optional label to the parallel code, mostly used for debugging or obtaining timing measurements. The N is how many threads are needed, or how many parallel instances of the code are needed per function call This is also equivalent to the number of indices in the array x since that is the size of the data being operated on. The rest is a pointer to the parallel code; in this case it is written within the function call. j denotes the *thread ID* which is unique to each otherwise identical copy of the parallel section. Since this is a parallel_reduce(), that means there must be a single result to the operation. The last parameter denotes the variable, *result* to hold that end result.

```
1 // Application: <y,Ax> = y^T*A*x
2 // Assuming M and N are passed in
3 // Initialize y, x, and A
4 double result = 0;
5 Kokkos::parallel_reduce("02", N, KOKKOS_LAMBDA(int j, double &update){
6   double temp2 = 0;
7
8   for (int i = 0; i < M; ++i){
9     temp2 += A(j, i) * x(i);
10  }
11
12  update += y(j) * temp2;
13 }, result);
14
15 // Output result.
16 if (repeat == (nrepeat - 1)){
17   printf(" Computed result for %d x %d is %lf\n", N, M, result);
18 }
```

Listing 2.1: Kokkos code example

Views are the Kokkos (and PyKokkos) equivalent of arrays or vectors of one or more dimensions. Variables x, y, and A are all initialized as Views prior to running the parallel section of code.

2.2 PyKokkos

PyKokkos [2, 3] is a performance portable framework implemented in Python that allows users to write parallel code and seemingly run it in a Python environment. Unlike C++, Python has a relatively low learning curve which makes it ideal for users that are more interested in quick prototypes rather than investing in writing complicated, highly optimized, device specific code.

One of the biggest drawbacks to using Python alone is that Python is limited to running on the CPU only, and due to constraints in the language implementation, often only on a single thread at a time. In order to allow code written in Python to run in parallel both on the CPU and on the GPU, different techniques are needed.

In many performant Python libraries, the library call is just a wrapper, or

translation layer, for an underlying and highly optimized device-specific libraries. It outsources the more compute intensive sections to some library written in a more optimized, parallel-friendly language and returns the results back to Python. This means that Python users are limited to either slower custom code or a limited selection of more performant code that may not be compatible on all device types.

PyKokkos gets around these limitations by translating fragments of Python code to Kokkos code, compiling it to the appropriate device, and linking it back into the user's script. This allows for functions to be seamlessly written in pure Python but have all the performance benefits of running in parallel on a CPU or GPU. In Listing 2.2, the equivalent PyKokkos parallel section to Listing 2.1 is shown.

```
1 # PyKokkos parallel section
2 @pk.workunit
3 def yAx(j: int,
4         acc: pk.Acc[float],
5         M: int,
6         y: pk.View1D[pk.double],
7         x: pk.View1D[pk.double],
8         A: pk.View2D[pk.double]):
9     temp2: float = 0
10    for i in range(M):
11        temp2 += A[j][i] * x[i]
12
13    acc += y[j] * temp2
14
15 # and would be called like so after initializing
16 # y, x, and A (M and N are passed in):
17 p = pk.RangePolicy(pk.get_default_space(), 0, N)
18 result = pk.parallel_reduce(p, yAx, M=M, y=y, x=x, A=A)
19 print(f"Computed result for {N} x {M} is {result}")
```

Listing 2.2: PyKokkos code example

Very similar in format to the Kokkos code in Listing 2.1, the PyKokkos code also has passed in a thread ID indicator, still j, and somewhere to store the results, *acc* (accumulator). Because the PyKokkos code is not in an inline format but in a separate function, the rest of the data must also be passed in as parameters. Like the Kokkos example, the Views x, y and A are initialized earlier in the method

before being passed into the parallel_reduce(). The *RangePolicy()* dictates what the parallelism will look like; how many parallel operations will be needed, on what device will this take place, and how threads can be distinguished in the parallel workload.

PyKokkos code sections are demarcated by the use of a decorator (like the *@pk.workunit* in the example code) for PyKokkos classes (*Workloads* or *Functors*) and functions (*workunit*). The decorators can also provide information to PyKokkos on what memory spaces need to be initialized or where data will be located. This can help fill in the gap between the Kokkos/C++ user managed memory and the PyKokkos/Python non-user managed memory.

2.3 Graphs and Graph Algorithms

Graph Algorithms are the family of problems that are related to a specific type of data structure called a *Graph*. Graphs are defined as a set of *Nodes* or *Vertices* and the *Edges* (indicating relationship) between them. In math terms, a graph can be defined like so and shown in Figure 2.1:

$G = (V, E)$

$V = \{n1, n2, n3, n4, n5\}$

$E = \{(n1, n2), (n1, n3), (n2, n3), (n2, n4)\}$

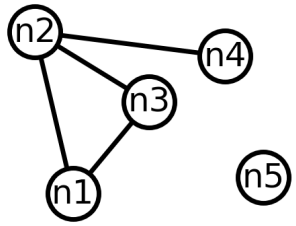

Figure 2.1: Example of a Simple Graph

Each edge can have a value and this value is called a *Weight*. This can be written as $e = (n1, n2, 4)$ with the last value, 4, representing the weight between nodes $n1$ and $n2$.

Edges can also have direction. Graphs with directed edges are called *directed graphs* and those without directed edges are called *undirected graphs*. Let us define $e1 = (n1, n2)$ and $e2 = (n2, n1)$. In an undirected graph, $e1 == e2$. In a directed graph, $e1 != e2$ since $e1$ is the path from $n1$ to $n2$ and $e2$ is the path from $n2$ to $n1$.

Graphs (edges) can be both weighted and directed at the same time, however graphs will not mix edge types; they are only directed or undirected. An example of a weighted, directed graph is shown in Figure 2.3.

In this paper, any non-weighted graphs will have a default edge weight of 1 and an edge weight of 0 will imply no relation or edge between the nodes.

2.3.1 Encoding Graphs

There are several formats in which to encode graphs. While the list of nodes and edges from the more formal definition of a graph is certainly an option, it is an inefficient format to run an algorithm with. A more natural format for computation is an array. If every index, n, of an N dimensional array or an NxN matrix represents a node on the graph, then the edges can be encoded with a non-zero value.

For example, a simple 3 node, $\{n0, n1, n2\}$, undirected graph with 2 edges, $\{(n0, n2), (n2, n1)\}$ (see Figure 2.2) and might look like this:

$$\begin{matrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{matrix}$$

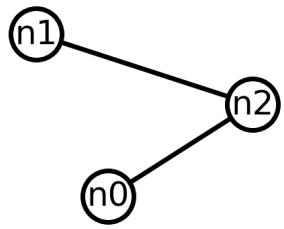

Figure 2.2: Example Graph from Encoded Matrix

If the same edges were directed and weighted, the graph would look like Figure 2.3 and the matrix written as:

$$\begin{matrix} 0 & 0 & 7 \\ 0 & 0 & 0 \\ 0 & 5 & 0 \end{matrix}$$

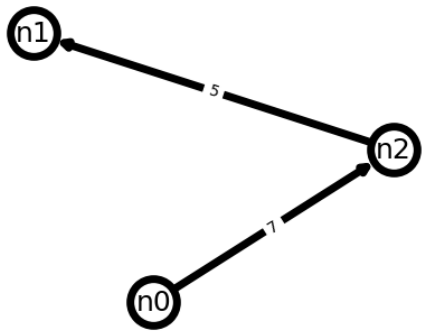

Figure 2.3: Example of a Weighted and Directed Graph from Matrix

2.3.2 Compressed Sparse Format(s)

One drawback to a full matrix representation of a graph is that most values in a real-world graph in matrix form are zeros. This type of matrix (or graph) is defined as *sparse*. A *dense matrix* would have more non-zero values than zeros values.

The solution to saving space in the graph encoding is to use a *Compressed Sparse Format*. This format only encodes the coordinates and non-zero data values of a matrix.

The two related Compressed Sparse Formats used here are the *Compressed Sparse Row (CSR)* and the *Compressed Sparse Column (CSC)* formats. Both work very similarly with the matrix being encoded into three separate arrays.

For the CSR format, the *data* or value array holds all the non-zero values of the matrix sorted in row then column order. The *indexpointer* (*indptr*) array which points at the beginning (inclusive) and ending (non-inclusive) indices for each row in the matrix on the other two arrays. The *column* (*col*, or more generally, *indices*)

array holds which column in the matrix that the equivalent data value belongs to. CSC format is equivalent except it uses row for column and column for row.

The above weighted and directed graph/matrix can be written in *CSR* as:

$indptr = \{0, 1, 1, 2\}$

$col = \{2, 1\}$

$data = \{7, 5\}$

The *CSC* equivalent would be written as:

$indptr = \{0, 0, 1, 2\}$

$row = \{2, 0\}$

$data = \{5, 7\}$

Notice that it is simple to find the total number of edges by getting the last value of the *indptr* array. For the CSR format, since the edges are first indexed by row, then column, all *out-bound* edges of a node (directed edges where the node is the source node of the edge) are grouped consecutively in the *data* and *row* arrays. Likewise, all *in-bound* edges of a node are grouped together in the CSC format.

2.3.3 NetworkX

Python has libraries implemented to read formatted graph files. One such library was used: the NetworkX [15] library handles reading in and writing to a multitude of different standardized formats. It can also convert the format once it is in NetworkX to the compressed sparse arrays, making it an ideal library to preprocess input data files. Additionally, it has the capability of creating graphs from both user input and a variety of graph generators. This makes it an excellent way to create graphs both trivial and small (for development and debugging), as well as large-scale (as will be discussed in later chapters).

Chapter 3: The Parallel Maximal Independent Set (MIS) Algorithm

3.1 MIS Background

An *independent* set of nodes is a set where all nodes in the set do not share any neighbors. A *neighbor* is defined as a pair of nodes that share an edge between them. While in directed graphs there are *in-bound* and *out-bound* neighbors where there is significance in edge direction; the Maximal Independent Set problem only applies to undirected graphs.

If there is a 5-node graph with $G = (V, E)$ with edges, $E = \{(A, B), (B, C), (C, D), (D, E)\}$, then the set of nodes $\{A, C\}$ would be a *independent set* since there are no shared edges between A and C. This graph is shown in Figure 3.1.

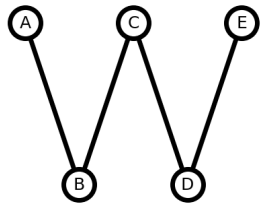

Figure 3.1: Example Graph

A *maximal* or *maximum* set refers to a set where nothing more can be added to the set and the condition still hold true. Another way of putting it is that a maximal set may not be a subset of another set that meets the same criteria. For example, the independent set of nodes $\{A, C\}$ is not maximal because the larger set of nodes $\{A, C, E\}$ is also an independent set. Since $\{A, C\}$ is a subset of $\{A, C, E\}$, then set $\{A, C\}$ cannot be a maximal independent set. The set of nodes $\{A, C, E\}$ is a maximal independent set because no more nodes, B or D, can be added and

the independent condition still hold true. For the same graph, there can be multiple maximal independent sets. For the example graph above, the set of nodes $\{B, D\}$ also is a valid solution since it is both independent and maximal. Some of the possible solutions are shown in Figure 3.2.

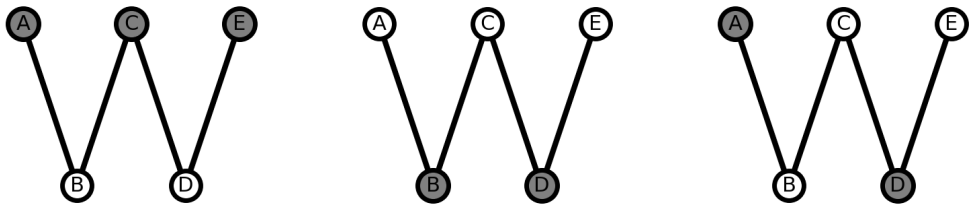

Figure 3.2: Possible MIS Solutions, MIS nodes in grey

3.2 The Parallel Maximal Independent Set Algorithm

The contribution of this report–the PyKokkos implementation of the algorithm–is based off the pseudocode in the paper: "A High-Quality and Fast Maximal Independent Set Implementation for GPUs" [9]. The parallel section of the pseudocode is shown in Listing 3.1.

The algorithm initializes by setting all nodes to *undecided* and also assigning a *random* priority to each node. The local highest priority node in the set of all *undecided* nodes then marks itself as within the MIS and all its neighbors as outside the set. This goes on till all nodes are marked as in or out of the set. Since each *undecided* node only needs to check its own neighbors to see if it is the local highest priority, the time cost for each iteration decreases as there are fewer remaining nodes with fewer unmarked neighbors. Since checking and marking a node happens with no dependencies on any non-neighboring values (and if marked, not only eliminates the MIS node from doing more work but also all its neighbors), the algorithm is highly parallel and will run in $O(log_2(N))$ time with high probability. While there is potential for two nodes in the MIS set to share a neighbor and mark it as not MIS,

this will not cause a race condition. Since they are writing the same value, it does not matter which node's thread actually succeeds.

```
1 // This is the parallel section
2 compute_kernel() {
3   //tid is thread ID
4
5   if(status[tid] == undecided) {
6     // find highest priority (random array) neighbor
7     best = get_best_neighbor(tid);
8     if(random[tid] > best) {
9       status[tid] = in;
10      mark_neighbor_out(tid);
11    } else {
12      need_another_round = true;
13    }
14  }
15 }
16
17 MIS() {
18  //allocate memory and transfer graph
19  //init status array to undecided
20  //init random array to random values
21  do {
22    need_another_round = false;
23    compute_kernel();
24  } while (need_another_round);
25 }
```

Listing 3.1: MIS pseudocode

In the pseudocode examples, *need_another_round* checks if any nodes are still *undecided* and if another round is needed. *get_best_neighbor()* checks the neighboring *undecided* nodes for the priority and returns the highest found, and *mark_neighbor_out()* marks all neighbors as out of the MIS.

3.3 Implementation

The PyKokkos implementation of the algorithm is fairly straightforward. The parallel section in PyKokkos as shown in Listing 3.2. The thread ID in this example is *i*. *done* is a View with a single index and *priority* and *status* are Views that hold

their respective value for each node.

```
# Initialized beforehand are:
# done: pk.View[int] value initialized to 0 (not done)
# priority: pk.View[int] set to random values
# status: pk.View[int] set to unmarked (-1)

# Graph CSR arrays (minus data)
# indices: pk.View[int]
# indptr : pk.View[int]

@pk.workunit
def compute_kernel(self, i: int):
  d : int = 0
  # check if self is undecided (-1)
  if self.status[i] == -1:
    best: int = 1
    # figure out if a neighbor has a higher priority
    for j in range(self.indptr[i], self.indptr[i+1]):
      neighbor: int = self.indices[j]
      if neighbor == -1:
        break
      if self.status[neighbor] != 0:
        if self.priority[i] < self.priority[neighbor]:
          best = 0
          break
    # if highest priority among neighbors, set self to in (1)
    # set all neighbors to out (0)
    if best == 1:
      self.status[i] = 1
      for j in range(indptr[i], self.indptr[i+1]):
        neighbor: int = self.indices[j]
        self.status[neighbor] = 0
    # might need another round to check resolution status (0)
    else:
      self.done[0] = 0
```

Listing 3.2: Parallel section of the PyKokkos MIS

Since MIS is only concerned with neighboring nodes, and not edge weight, a small optimization was made by removing the data, or value, array of the CSR format into the algorithm.

3.3.1 MIS Checker

A simple parallel MIS checker was also implemented in PyKokkos. Since the definition of a MIS set is that: for all nodes in the set, all neighbors must not be within the set, and for all nodes not within the MIS set, there must be at least one neighbor in the MIS set. It was trivial to write a parallel checker that took a single round to check for that condition and return *true* or $false$. The parallel section for the checker is in Listing 3.3. The only input for the checker is the graph and the MIS view output from the MIS algorithm.

```
@pk.workunit
def check_node(self, i: int):
  # in the mis set
  if mis[i] == 1:
    # check for no other included neighbors
    for j in range(indptr[i], self.indptr[i+1]):
      neighbor: int = self.indices[j]
      if self.mis[neighbor] == 1:
        self.valid[0] = 0
  # not in mis set, need at least one mis set neighbor
  else:
    found_mis: int = 0
    for j in range(indptr[i], self.indptr[i+1]):
      neighbor: int = self.indices[j]
      if self.mis[neighbor] == 1:
        found_mis = 1
    if found_mis == 0:
      self.valid[0] = 0
```

Listing 3.3: Parallel section of the PyKokkos MIS Checker

Chapter 4: The Parallel Lattice-Linear Shortest Path Algorithm

4.1 Algorithm Background

A *path* between 2 nodes in a graph is any list of edges that when taken in order will start at the source node and reach the destination node. A *shortest path* then is the optimal path between 2 nodes. In an undirected graph the path will be the same, regardless of which node is source and which node is the destination. In a directed graph, this does not hold true. For an unweighted graph, this will be the sets with the least number of edges. For a weighted graph, this will be the set of edges with the least total cost (or sum of the weight values for all the edges in the path). A shortest path algorithm is simply: the shortest (or least costly) way of getting between a pair or pairs of nodes in a graph.

If there is an unweighted, 3-node graph $G = (V, E)$ with
$E = \{(A, B), (A, C), (B, C)\}$ (see Figure 4.1). Then to go from A to C, the path $\{(A, C)\}$ would be taken. While there is another path to get from A to C:
$\{(A, B), (B, C)\}$ this would not be the shortest path since it would take two edges instead of one.

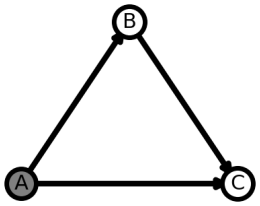

Figure 4.1: Example Graph

If a graph is weighted, the edges have cost associated with transversal, then the 'shortest' path might change. For example, if the same edges had a weight: $E =$

$\{(A, B, 2), (A, C, 7), (B, C, 2)\}$ (such as in Figure 4.2) then the path $\{(A, B), (B, C)\}$ would cumulatively cost $2+2 = 4$ and be the least cost path since the only alternative path has a total cost of 7.

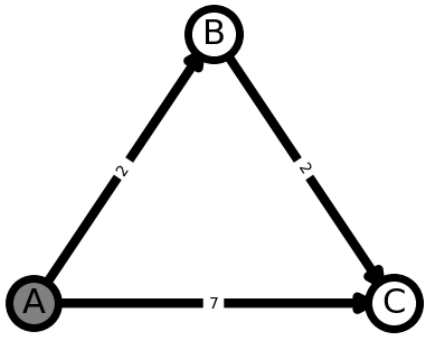

Figure 4.2: Example Weighted Graph

This type of problem applies to directed and undirected graphs alike and is widely applicable in many contexts from plotting an efficient road trip across the U.S. to packet routing in a network.

There are several subcategories of the shortest path problem:

- Single Source: from a single source node to all destinations
- Single Destination: shortest path from all sources to a single destination
- All Pairs: shortest path for every pair of nodes in the graph

In this paper the single source variant is discussed and implemented.

4.1.1 Dijkstra's and Bellman-Ford

The classical algorithm to solve the Shortest Path problem is Dijkstra's Algorithm [11]. It works by gradually working from the source nodes and searching for the neighboring edge of least weight or distance. The node at the other end of this

edge is marked as done since there is no path 'shorter' and its neighbors are also added to the search space. By greedily choosing the paths of least cost one by one, the algorithm finds the optimal paths between all nodes and the source node after all nodes have been marked.

Since Dijkstra's algorithm is a serial algorithm, only one edge is processed at a time and similarly, a single node marked. It will also not find the correct path for graphs of negative edge weights as it has a built in assumption that all edges in the path will have a greater cost the further away the path gets from the source.

This is where the Bellman-Ford Algorithm [7, 13] comes into play. Unlike Dijkstra's, the Bellman-Ford algorithm can handle not only negative edge weights, but also detect *negative cycles* in a graph. A negative cycle in a graph means that there is a path to and from the same node with a negative total cost and that there is no shortest path. Entering this cycle would have $-\infty$ total cost as the path continues to repeat it. An example of a negative cycle is shown in Figure 4.3.

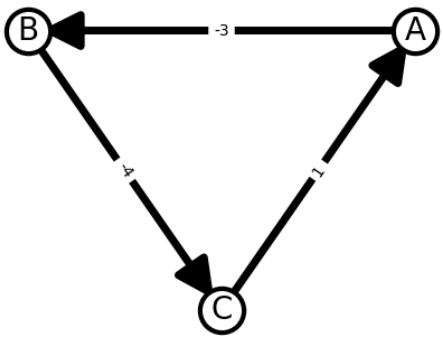

Figure 4.3: Example Negative Cycle

The Bellman-Ford algorithm has more overhead than Dijkstra's since it both adds new edges to the search space, and updates all existing paths with the new information. This handles the scenario of discovering a better path due to the inclusion of a negative edge weight to lower total cost. Since all paths must be found after all

edges have been traversed, the algorithm ends when either no updates can be made or it has run that many times. Should a change be made to the total node cost in any one node after all the edges have been explored, then there must be a negative cycle in the graph and the shortest path problem for that graph is unsolvable.

While Bellman-Ford is originally a serial algorithm, it is easily made parallel by distributing the work of checking and then updating the node information across multiple threads.

4.1.2 Lattice-Linear Predicate (LLP)

Lattice-Linear Predicate [14] is a new parallel approach to solving problems with a distributed linear solution space.

For linear problems, the solution can take the form of an array or vector. All potential solutions defined by this vector form a *lattice*. The point at which the algorithm is at in the lattice is called the *global state*. The nodes in the lattice are every permutation of the global state. The edges are the relationships, or paths, between these states.

```
1 #global state vector
2 G
3
4 # done in parallel with regards to
5 # the size of the global state vector
6 do {
7   # check to see if the predicate is satisfied
8   # for all indices of the vector
9   forbidden: bool = predicate(G)
10
11  # if the state is forbidden,
12  # advance in the lattice.
13  if forbidden:
14    advance(G)
15  } while (forbidden && G is changing)
```

Listing 4.1: Generalized LLP parallel section

To determine if the current global state is the solution or if the algorithm

must continue searching, the solution must satisfy the *predicate*. If the predicate is not satisfied, then the state is *forbidden* and the global state should be adjusted or *advanced*. The advance is the change made to the global state from one or more threads that advances the global state closer to a solution that satisfies the predicate. Listing 4.1 shows the general format of an LLP algorithm.

4.2 LLP Shortest Path

For the shortest path problem, the global state consists of the vector of total cost it would take to reach each node and is initialized to: $(0, \infty, \infty, ..., \infty)$ (assuming the first node is the start node). The solution space would be from ∞ to $-\infty$ for all nodes (also assuming negative costs are possible).

The predicate for shortest path would be that for every node in the graph (excluding the start) there shall be no path to that node from any of its neighbors that costs less than what it has in the global state: *for graph* $G(V, E)$ *for all edges* $(i, j) \in E : G[j] <= G[i] + weight(i, j)$ (assuming also that total number of iterations does not exceed the total number of edges).

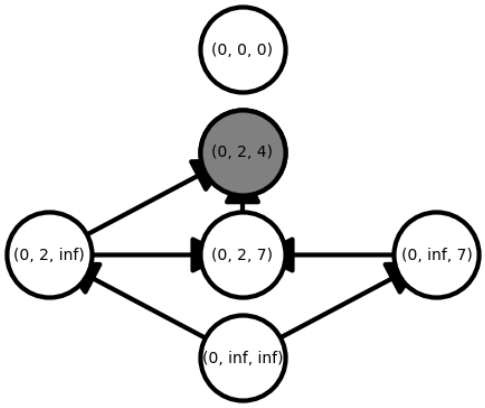

Figure 4.4: Moving Through a Lattice (Global State)

Let us consider the graph shown in Figure 4.2. As an example of how this applies to the lattice, the total cost to reach each node, excluding the start node

which is always 0, is somewhere between $-\infty$ and ∞. The initial global state would then be $(0, \infty, \infty)$. Depending on algorithm execution, one or more of the nodes could update their own total cost in the global state. B could update itself to 2 and C could update itself to 7. Eventually, this leads to the final global state of $(0, 2, 4)$. The relevant subset of this lattice is shown in Figure 4.4. The state $(0, 0, 0)$ is also shown in the figure as it is a valid possible solution in the overall lattice.

```
# global state vector or current node in lattice
G

do{
  # check to see if cost for this node
  # is greater than the cost it takes to
  # get to this node from a neighbor
  # for all neighbors n
  forbidden: if G[tid] > G[n] + weight(n, tid)

  # update the Global State and ensure
  # this node has the min cost of
  # the path from all neighbors n
  if forbidden:
    G[tid] = min(G[n] + weight(n, tid))
  } while(forbidden && total iterations <= number of edges)
```

Listing 4.2: Shortest Path LLP parallel section

Putting the Shortest Path specific details and the generalized LLP formula gets an LLP-Shortest Path parallel section shown in Listing 4.2.

4.2.1 Implementation

In the PyKokkos implementation, the global state vector is initialized to infinity, with the exception of the start node which is set to 0. There is an additional vector keeping track of the previous node to later build the full shortest path from, this is also initialized. Also initialized to $false$ is a variable to indicate if the advance was made (if the previous global state was forbidden), and if another round of the parallel section should be run.

The rest of the code is essentially the same as the above LLP Shortest Path

in Listing 4.2: check if forbidden by the shortest path predicate, if not, update the global state with the advance. The advance for shortest path is simply updating the previous node and the new total cost to the previous node vector and global state vector. This is done per thread which is per index of the global state vector (per node). The parallel section of the PyKokkos LLP-Shortest Path implementation is shown in Listing 4.3.

```
@pk.workunit
def compute_kernel(self, i: int):
  if i == self.start:
    return

  # get min{G[in-bound neighbor] + weight[in-bound neighbor, i] for all in-bound
      neighbors}
  curr_cost: float = self.G[i]
  prev_node: int = self.prev_nodes[i]

  # get G[in-bound neighbor] + weight[in-bound neighbor, i]
  for j in range(self.indptr[i], self.indptr[i+1]):
    neighbor: int = self.indices[j]
    neighbor_cost: pk.float = self.G[neighbor]
    this_weight: pk.float = self.weights[j]
    this_cost: pk.float = neighbor_cost + this_weight

    # min{} check
    if this_cost < curr_cost:
      prev_node = neighbor
      curr_cost = this_cost

  # LLP format: check if forbidden, if so, advance

  # Global state is forbidden if
  # G[i] > min{}
  if self.G[i] > curr_cost:
    # advance is set G[i] to min{}
    self.done[0] = 0
    self.prev_nodes[i] = prev_node
    self.G[i] = curr_cost
```

Listing 4.3: Parallel section of the PyKokkos Shortest Path

Chapter 5: Evaluation

Kokkos and PyKokkos allow code to be compiled and run on a variety of platforms. The variable that controls which platform (CPU, GPU, etc.) and how it is run (OpenMP, Serial, POSIX Threads, etc.) is called a 'space'. We ran the algorithms in three different spaces: (1) Serial, which is on the CPU, (2) OpenMP (OMP), which allows for multi-threading on the CPU, and (3) CUDA, which is on an Nvidia GPU. Also being compared is the NetworkX implementation of the algorithms, which is in Python and thus single-threaded on CPU by default. OpenMP restricts the number of threads allowed through the OMP_NUM_THREADS variable. If left unset, it will be configured automatically. For OpenMP multithreaded runs, OMP_NUM_THREADS is left unset, and for OpenMP with a single thread it is set to 1.

All runs were done on Frontera [27] and are the average of 5 or more runs. The Frontera nodes have Intel Xeon CPU E5-2620 v4 processors (2 sockets with 16 cores each) and 4 NVIDIA Quadro RTX 5000 per node. The PyKokkos version used is commit 1c6e3e6c, which was the latest on main at the time of our experiments.

5.1 Graphs

The graphs used to test the algorithms were pulled/generated from a variety of different sources. The characteristics of the graphs used and their sources (Matrix Market (MM), NetworkX (NX) and Stanford Network Analysis Project (SNAP)) are shown in Table 5.1.

5.1.1 Matrix Market (MM)

The initial set of graphs came from Matrix Market [8]. It is a National Institute of Standards and Technology (NIST) repository of sparse matrices and generators. The Matrix Market files are encoded in specific, well-defined formats (coordinates

Table 5.1: Graphs Used to Evaluate Implemented Algorithms

Graph	#Nodes	#Edges	Source	Type
e05r0000	236	5,846	MM	directed, weighted
reg_ud_1000_n	1,000	50,000	NX	undirected
rand_ud_1000_n	1,000	100,000	NX	undirected
e20r0000	4,241	131,412	MM	directed, weighted
s1rmq4m1	5,489	262,411	MX	directed, weighted
e30r0000	9,661	305,794	MX	directed, weighted
reg_ud_10000_n	10,000	5,000,000	NX	undirected
rand_ud_10000_n	10,000	10,000,000	NX	undirected
Oregon-1	11,492	23,409	SNAP	undirected
e40r0100	17,281	553,562	MM	directed, weighted
e40r0100_P	17,281	553,562	MM	directed, weighted
reg_ud_100000_n	100,000	50,000,000	NX	undirected
com-Amazon	334,863	925,872	SNAP	undirected
as-Skitter	1,696,415	11,095,298	SNAP	undirected
com-Orkut	3,072,441	117,185,083	SNAP	undirected
com-LiveJournal	3,997,962	34,681,189	SNAP	undirected

and a value or as a column-oriented array). Reading from and writing to this format is already implemented by NetworkX.

As all the graphs in this set have negative edge weights and self-directed edges, they are unsuitable for the MIS algorithm. Since the matrices in this data set were not originally meant to be interpreted as graphs, the actual listing of values in the matrix do not strictly translate to edges in a graph. Coordinates which list an edge weight of zero are disregarded as a non-edge when transforming the matrix to a graph. The tables in this chapter show the adjusted edge counts taken after the file is read in as a graph. e40r0100_P is e40r0100 modified to have all edges made positive, which we use for detailed performance analysis.

5.1.2 Stanford Network Analysis Project (SNAP)

When running performance testing on the algorithms, larger datasets are required. The Stanford Network Analysis Project (SNAP) [21] has a wide variety of

well-cataloged networks in its database. This provided some good examples of real-life datasets to run performance comparisons on. These larger graphs are collected from various internet community relationships ('com' prefixed) [30] or routing networks (Oregon-1, as-Skitter) [20], and thus are all undirected and unweighted. These particular graphs were chosen out of the database because they represent a wide range of large graph sizes and are compatible to both of the algorithms. We used as-Skitter as one of the graphs the MIS paper tested their OpenMP and CUDA implementations (ECL-MIS) on. All SNAP graphs were pre-translated to Matrix Market format as part of the SuiteSparse Matrix Collection [10].

5.1.3 NetworkX

The NetworkX graphs were all randomly generated by NetworkX's graph generators [15]. The 'rand' or 'reg' prefix distinguishes if the graph was generated as a purely random graph or if it is a random regular graph. The number indicates the number of nodes.

5.2 MIS

Listed in Table 5.2 are the average run times of the parallel sections for each of the PyKokkos spaces and configurations (i.e., OPM_NUM_THREADS) as well as the run times of NetworkX's implementation of a MIS algorithm. All values are in milliseconds and rounded to one decimal place. Figure 5.1 is a bar chart showing the MIS algorithm times from Table 5.2. NetworkX took over ∼13hrs to run before timing out without completion on the com-LiveJournal graph. Measurement for PyKokkos only includes the time spent executing the parallel sections of the Kokkos code. For NetworkX, only the algorithm's run time was measured. The loading of the graph into NetworkX or PyKokkos was excluded. For performance runs, we also exclude running the MIS checker on the results. It is expected that performance on smaller graphs be comparatively worse on multithreaded and GPU runs due to the overhead

of setting up the parallelism outweighing any parallel performance benefits.

Table 5.2: MIS Times (in ms)

Graph	**Serial**	**OMP Single**	**OMP Multi**	**GPU**	**NetworkX**
reg_ud_1000_n	0.1	0.1	1.3	2.0	0.7
rand_ud_1000_n	0.1	0.1	1.7	2.1	0.7
reg_ud_10000_n	1.1	1.2	3.0	4.5	23.7
rand_ud_10000_n	1.2	16.4	3.3	6.6	28.2
Oregon-1	0.2	0.3	0.6	1.8	734.9
reg_ud_100000_n	16.8	20.6	11.6	5.7	760.2
com-Amazon	13.2	13.8	3.8	2.0	354,762.2
com-Orkut	442.6	483.6	71.6	24.0	10,913,850.5
com-LiveJournal	261.8	318.7	49.5	13.9	timeout

In Table 5.3, we show the speedup ratio of the various times compared to the Serial results. For the GPU Ratio, this would be calculated with Serial time divided by GPU time.

Table 5.3: Ratio of MIS Speedups to Serial

Graph	**OMP Single**	**OMP Multi**	**GPU**	**NetworkX**
reg_ud_1000_n	0.6	0.0	0.0	0.1
rand_ud_1000_n	0.6	0.0	0.0	0.1
reg_ud_10000_n	0.9	0.4	0.2	0.0
rand_ud_10000_n	0.1	0.4	0.2	0.0
Oregon-1	0.9	0.4	0.1	0.0
reg_ud_100000_n	0.8	1.5	3.0	0.0
com-Amazon	1.0	3.5	6.6	0.0
com-Orkut	0.9	6.2	18.5	0.0
com-LiveJournal	0.8	5.3	18.8	N/A

Table 5.4 is the comparison of ECL-MIS implemented both in CUDA and OpenMP vs PyKokkos. as-Skitter was a SNAP graph used by the original paper in their performance tests. For the OpenMP ECL-MIS, the parameters also required a thread-count as an input. For ECL Single, 1 was passed in and for ECL Multi, 36 (the number of processor cores) was used.

Table 5.4: MIS vs ECL-MIS (in ms) on as-Skitter

OMP Single	ECL Single	OMP Multi	ECL Multi	GPU	ECL GPU
94.8	70.8	18.1	40.9	3.5	1.4

5.3 LLP-Shortest Path

Similarly to the MIS tables, tables 5.5 and 5.6 show the time and speed up ratios for the Shortest Path algorithm and Figure 5.2 is a chart of the PyKokkos times. The largest of the graphs were omitted for the sake of time as this algorithm runs much longer than MIS, and for this algorithm graphs with negative edge weights (marked with a †) were included.

Table 5.5: Shortest Path Times (in ms)

Graph	Serial	OMP Single	OMP Multi	GPU	NetworkX
e05r0000†	37.3	40.9	73.6	66.7	69.1
reg_ud_1000_n	0.6	0.5	0.6	2.2	45.9
rand_ud_1000_n	1.0	1.0	1.4	2.2	102.9
e20r0000†	14,923.3	15,912.1	14,059.1	1,554.6	347.2
s1rmq4m1†	58,623.6	58,731.5	39,368.7	4,127.6	905.2
e30r0000†	79,915.4	81,012.0	50,326.0	5,831.5	14,242.9
reg_ud_10000_n	57.0	56.2	12.6	15.1	6,533.4
rand_ud_10000_n	109.9	106.6	22.8	18.7	13,569.7
Oregon-1	0.8	1.0	0.8	3.9	57.4
e40r0100†	257,923.1	266,121.0	104,526.0	16,368.6	29,729.7
e40r0100_P	88.6	86.5	26.8	23.3	6,846.5
com-Amazon	244.8	297.5	24.9	25.2	3,719.5

For graphs: e05r0000, e20r0000, e30r0000, e40r0100 and s1rmq4m1, there are negative cycles meaning that Bellman-Ford was forced to run number of $edges + 1$ times to detect the cycle and thus the lack of a solution. When edges are all positive, the performance of the PyKokkos implementations significantly increased as shown by the run times of e40r0100 compared to e40r0100_P.

NetworkX has a library of a variety of Shortest Path Algorithms. The one run

Table 5.6: Ratio of Shortest Path Speedups to Serial

Graph	OMP Single	OMP Multi	GPU	NetworkX
e05r0000†	0.9	0.5	0.6	0.5
reg_ud_1000_n	1.0	1.0	0.3	0.0
rand_ud_1000_n	1.0	0.7	0.5	0.0
e20r0000†	0.9	1.1	9.6	43.0
s1rmq4m1†	1.0	1.5	14.2	64.8
e30r0000†	1.0	1.6	13.7	5.6
reg_ud_10000_n	1.0	4.5	3.8	0.0
rand_ud_10000_n	1.0	4.8	5.9	0.0
Oregon-1	0.9	1.1	0.2	0.0
e40r0100†	1.0	2.5	15.8	8.7
e40r0100_P	1.0	3.3	3.8	0.0
com-Amazon	0.8	9.8	9.7	0.1

here is the Bellman-Ford implementation which returns all the previous nodes in the path in addition to the total cost per node; this is most similar to the implemented PyKokkos algorithm. Since NetworkX constantly maintains the full paths for each node, it performs better on graphs with negative cycles, as it does negative cycle checking in each iteration as opposed to waiting for the last iteration.

An implementation of LLP Bellman-Ford was done in Java (by a fellow student) that returns only the total cost of the input graph. The comparison is shown in Table 5.7. As the algorithm only runs on graphs with integer edge weights, it was run on the smaller of the unweighted (thus weight 1) graphs.

Table 5.7: Shortest Path vs Java (in ms)

Graph	OMP Single	OMP Multi	GPU	LLP Java
reg_ud_1000_n	0.5	0.6	2.2	1,041.8
rand_ud_1000_n	1.0	1.4	2.2	969.6

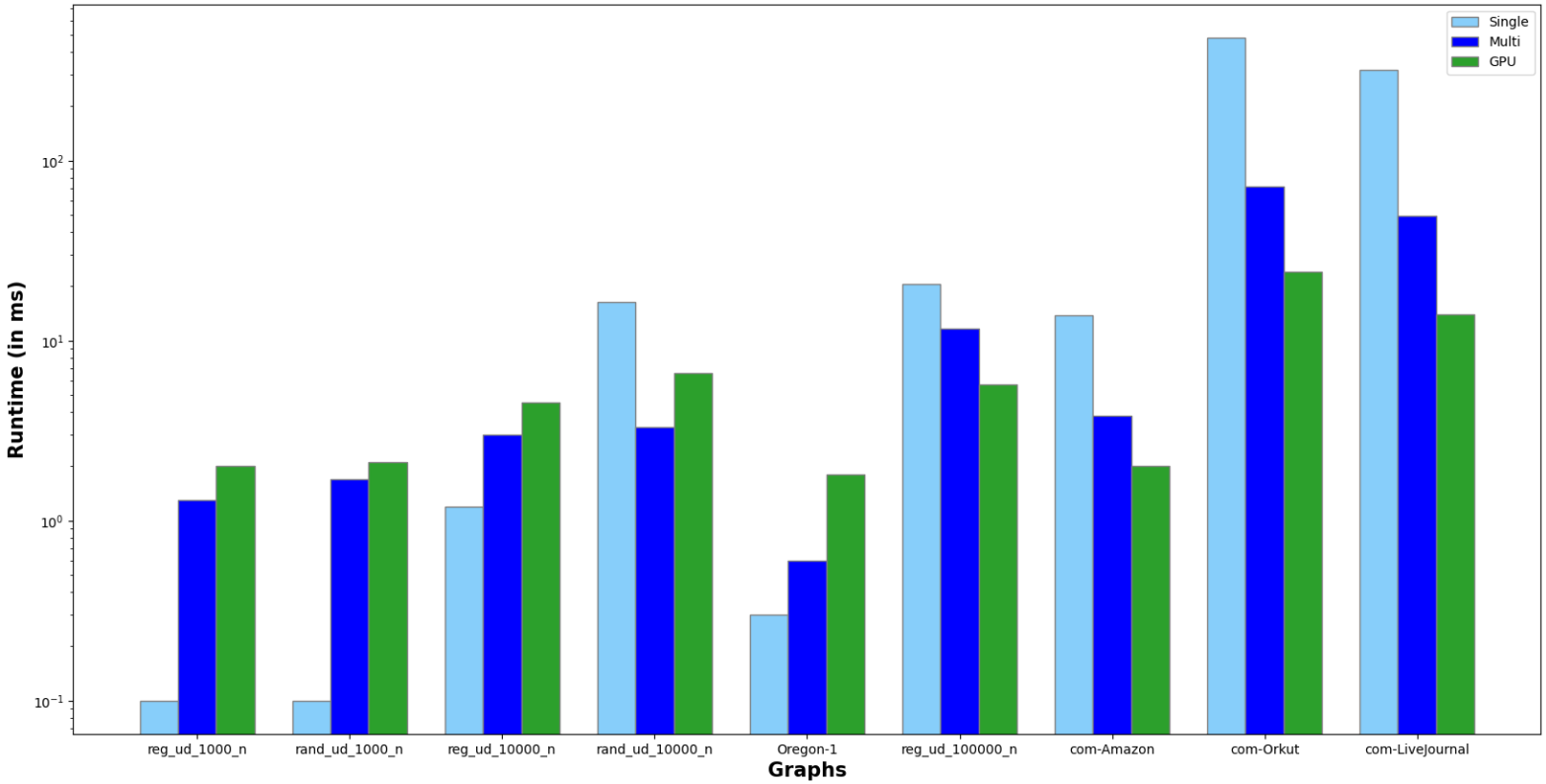

Figure 5.1: MIS Time Comparisons

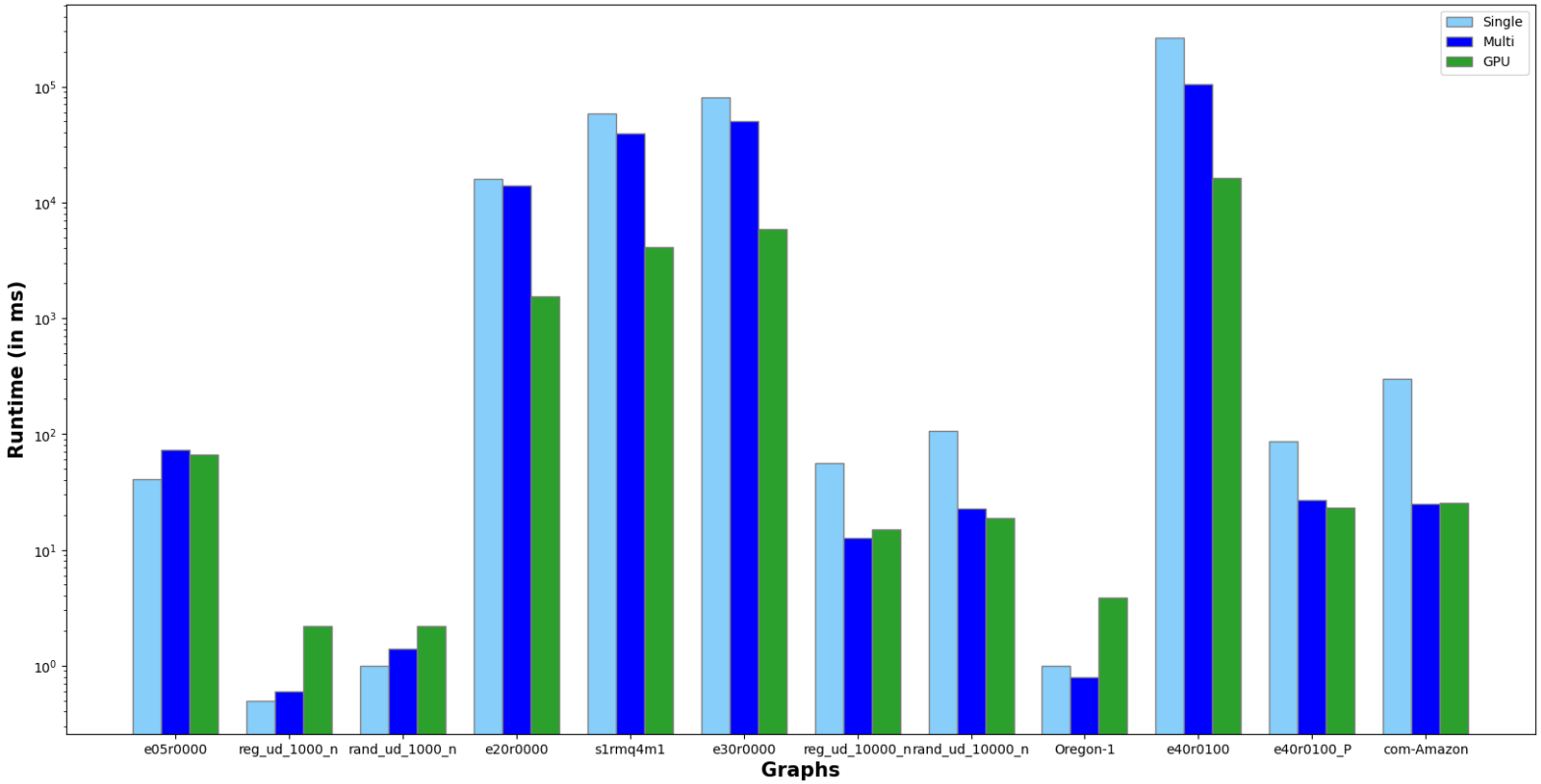

Figure 5.2: Shortest Path Time Comparisons

Chapter 6: Abstracting the Creation of Views in PyKokkos

6.1 Improving View Creation Abstraction

While *Views* in PyKokkos could be built by hand, much of the time, data is already in an array format such as an numpy array [16] or a cupy array [23].

Initially when first starting out on PyKokkos, there was no unified way of bringing data from either the GPU or CPU space (numpy or cupy arrays respectively) without calling the space specific functions.

In the course of implementing the graph algorithms, using one or the other of these functions made writing code for either CPU or GPU more difficult. A small improvement to the PyKokkos API was made by abstracting both of these functions, which allows users to call the same function to handle either device's arrays in one function call. A comparison of before and after the change is shown in Listing 6.1.

```
1 # before:
2 view_c = pk.from_cupy(cupy_array)
3 view_n = pk.from_numpy(numpy_array)
4
5 # after:
6 view_c = pk.array(cupy_array)
7 view_n = pk.array(numpy_array)
8 view_l = pk.array([int1, int2, ..., intn])
```

Listing 6.1: PyKokkos View array() changes

Chapter 7: Related Work

CPython [29] is the official implementation of the Python language specification and the most popular and prevalent Python runtime. One drawback to this implementation is the Python Global Interpreter Lock (GIL). While the GIL simplified the safe handling of memory in Python, it is also the main restriction on parallelism as it blocks multiple threads from running at the same time. Alternative Python compilers do not share this restriction. Numba [18] compiles Python code to LLVM. Cython [6], which is a superset of the Python specification, compiles the same as C++ and has C++ like calls. There is also an accepted proposal, PEP 703, to add an option in CPython to run without the GIL, which would allow for true multithreading in Python.

Libraries with Python Wrappers mitigate the GIL issue in Python by outsourcing parallel operation to languages that do support multithread CPU or GPU operations. Well-known AI/ML libraries in Python such as scikit-learn [26], PyTorch [25] and TensorFlow [1] handle their large scale computations in this way. In addition to having a database of large graph datasets, Stanford Network Analysis Project (SNAP) [22] also provides a Python wrapper to its collection of graph algorithms in C++.

In the area of Portable Performance Graph Algorithms, there is an implementation of a parallel MIS in Kokkos which runs on both CPU and GPU [17]. IrGL [24] is a graph algorithm specific compiler that creates optimized GPU code from an intermediate representation.

For other models that provide portable performance, Legion [5] has a data-centric design that tries to solve the high overhead cost of moving around large volumes of data. It gives users the ability to describe data attributes to optimize parallel operations on the data in C++. Legate [4] is in Nvidia project that sits on top of

Legion and also allows users to program in a familiar Python interface through importing Legate core libraries instead of numpy and other common libraries.

Parla [19] is a Python orchestration layer that handles data and task management between heterogeneous compute nodes. Paired with PyKokkos, distributed portable and performant code could be run over a system with many nodes.

Chapter 8: Conclusion

In this report, we demonstrated two performant and portable parallel algorithms. One is the first implementation of the Maximal Independent Set (MIS) algorithm by Burtscher et al. in Python. The other is the first implementation of an algorithm using Lattice-Linear Predicate in Python or on a GPU. Both of these algorithms, MIS and LLP-Shortest Path, are also the first graph algorithms to be written in PyKokkos.

The performance of the algorithms on graphs of various sizes demonstrate the versatility of PyKokkos allowing users to write code once and run on whichever platform is most convenient. The portability of PyKokkos does not come at the cost of readability or performance. PyKokkos code is not only readable, it also shows significant performance improvements, even run serially, over 'pure' Python code alternatives. This holds true both on moderately sized graphs and on larger graphs when either GPU or multithreaded CPU is enabled.

Despite the performance of the algorithm itself, the biggest slowdowns of running the algorithm came from the NetworkX preprocessing of the graphs. The inclusion of a graph library into PyKokkos would be of great performance benefit to future graph algorithms and make PyKokkos even more attractive to parallel application developers.

Works Cited

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. A performance portability framework for Python. In *International Conference on Supercomputing*, pages 467–478, 2021.

[3] Nader Al Awar, Steven Zhu, Neil Mehta, George Biros, and Milos Gligoric. Pykokkos: Performance portable kernels in python. In *International Conference on Software Engineering, Tool Demonstrations Track*, pages 164–167, 2022.

[4] Michael Bauer and Michael Garland. Legate NumPy: Accelerated and distributed array computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[7] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[8] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. In *Working Conference on Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[9] Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. A high-quality and fast maximal independent set implementation for gpus. *ACM Trans. Parallel Comput.*, 5(2), 2018.

[10] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), 2011.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.

[12] H. Carter Edwards and Christian R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop*, pages 18–24, 2013.

[13] L. R. Ford. *Network Flow Theory.* RAND Corporation, Santa Monica, CA, 1956.

[14] Vijay K. Garg. Predicate detection to solve combinatorial optimization problems. In *Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2020.

[15] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Python in Science Conference*, pages 11–15, 2008.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[17] Brian Kelley and Sivasankaran Rajamanickam. Parallel, portable algorithms for distance-2 maximal independent set and graph coarsening. In *International Parallel and Distributed Processing Symposium*, pages 280–290, 2022.

[18] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In *Workshop on the LLVM Compiler Infrastructure in HPC*, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Hochan Lee, William Ruys, Ian Henriksen, Arthur Peters, Yineng Yan, Sean Stephens, Bozhi You, Henrique Fingler, Martin Burtscher, Milos Gligoric, Karl Schulz, Keshav Pingali, Christopher J. Rossbach, Mattan Erez, and George Biros. Parla: A python orchestration system for heterogeneous architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022.

[20] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *International Conference on Knowledge Discovery in Data Mining*, pages 177–187, 2005.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[22] Jure Leskovec and Rok Sosič. SNAP: A general-purpose network analysis and graph-mining library. *Transactions on Intelligent Systems and Technology*, 8(1):1, 2016.

[23] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Workshop on Machine Learning Systems (LearningSys)*, 2017.

[24] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. *SIGPLAN Not.*, 51(10):1–19, 2016.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., 2019.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Dan Stanzione, John West, R. Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. Frontera: The evolution of leadership computing at the national science foundation. In *Practice and Experience in Advanced Research Computing*, pages 106–111, 2020.

[28] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. The Kokkos ecosystem: Compre-

hensive performance portability for high performance computing. *Computing in Science and Engineering*, pages 1–9, 2021.

[29] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[30] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

Vita

Olivia Zhuhe Mitchell is a software developer at Intel Corporation. There she was on Intel® Cluster Checker development team and is currently working with reporting the performance of Intel's parallel communication libraries. She received her undergraduate degree in Computer Engineering at University of Illinois at Urbana-Champaign.

Address: zhvhem@gmail.com

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.