Appendix A

Previous Designs and Implementations

Before working on ShareTrace, I did not have experience developing distributed algorithms. The approach proposed in **??** is my *fifth* attempt at defining a performant implementation of risk propagation that is also decentralized and online. The prior four attempts offered valuable learnings that guided me toward the proposed approach; however, only the latter supports truly decentralized, privacy-preserving contact tracing. To document my efforts in developing this thesis, prior designs and implementations are provided in this appendix.

A.1 Thinking Like a Vertex

The first iteration of risk propagation[1] utilized Apache Giraph[2], an open-source version of the iterative graph-processing library, Pregel (Malewicz et al., 2010), which is based on the bulk synchronous parallel model of distributed computing (Valiant, 1990). Giraph follows the *"think like a vertex" paradigm* in which the algorithm is specified in terms of the local information available to a graph vertex (McCune et al., 2015).

Risk propagation was implemented as defined by Ayday et al. (2020, 2021), using the factor graph representation of the contact network. Moreover, the implementation assumed the use of Dataswyft Personal Data Accounts[3], which provide a data-oriented interface to self-sovereign identity (Preukschat and Reed, 2021, pp. 98–99). However, because the Google/Apple API does not permit remotely persisting ephemeral identifiers, the implementation assumed that user geolocation data would be analyzed to generate the factor vertices in the factor graph (Appendix A.5). Figure A.1 describes the high-level architecture. Callouts 1, 2, and 4 were implemented using a fan-out design in which a *ventilator* Lambda function divides the work amongst *worker* Lambda functions.

[1] `https://github.com/cwru-xlab/sharetrace-giraph`

[2] `https://giraph.apache.org`

[3] `https://www.dataswyft.io`

[4] `https://aws.amazon.com/lambda`

[5] `https://aws.amazon.com/s3`

[6] `https://aws.amazon.com/emr`

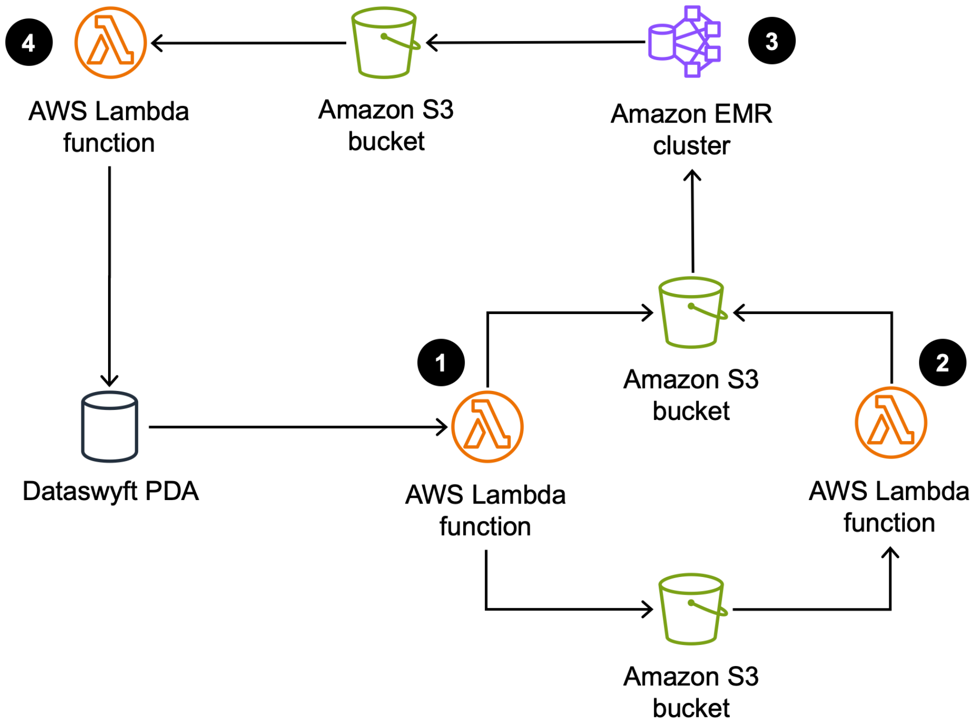

Figure A.1: ShareTrace batch-processing architecture. ❶ An AWS Lambda function[4] retrieves the recent risk scores and location data from the Dataswyft Personal Data Accounts (PDAs) of ShareTrace users. Risk scores are formatted as Giraph vertices and stored in an Amazon Simple Storage Service[5] (S3) bucket. Location data is stored in a separate S3 bucket. ❷ A Lambda function performs a contact search over the location data and stores the contacts as Giraph edges in the same bucket that stores the Giraph vertices. ❸ Amazon Elastic MapReduce[6] (EMR) runs risk propagation as a Giraph job and stores the exposure scores in an S3 bucket. ❹ A Lambda function stores the exposure score of each user in their respective PDA.

A couple factors prompted me to search for an alternative implementation.

1. *Dependency management incompatibility.* The primary impetus for reimplementation was the dependency conflicts between Giraph and other libraries. Despite several attempts (e.g., using different library versions, using different versions of Giraph, and forcing specific versions of transitive dependencies) to resolve the conflicts, a lack of personal development experience and stalled progress prompted me pursue alternative implementations.

2. *Implementation complexity.* For a relatively straightforward data flow, the architecture in Figure A.1 corresponded to over 4,000 lines of source code. In retrospect, AWS Step Functions[7] could have been used to orchestrate the workflow, including the fan-out design pattern, which would have simplified the Lambda function implementations. Regarding the implementation of risk propagation, one-mode projection (first used in Appendix A.4) would have simplified the implementation since it avoids types of vertices and messages.

A.2 Subgraph Actors

In an attempt to simplify the design in Appendix A.1, I rewrote risk propagation using the Ray Python library[8]. While it claims to support actor-based programming, Ray only offers coarse-grained concurrency, with each actor being mapped to a physical core. To achieve parallelism, the factor graph was

[7] `https://aws.amazon.com/step-functions`

[8] `https://www.ray.io`

partitioned amongst the actors such that each actor maintained a subset of variable vertices *or* factor vertices. The graph topology was stored in shared memory since it was immutable. The lifetime of this design was brief for the following reasons.

1. *Poor performance.* Communication between Ray actors requires message serialization. Moreover, partitioning the factor graph into subsets of factor vertices and variable vertices results in maximal interprocess communication. Unsurprisingly, this choice of partitioning manifested in poor runtime performance.

2. *Design complexity.* Not using a framework, like Giraph, meant that this implementation required more low-level code to implement actor functionality and message passing. Regardless of the performance, the overall design of this implementation was poorly organized and overthought.

A.3 Driver-Monitor-Worker Framework

Based on the poor runtime performance and complexity of the approach taken in Appendix A.2, I speculated that centralizing the mutable aspects of risk propagation (i.e., the current value of each variable vertex) would improve both metrics. With this is mind, I designed the *monitor-worker-driver* (MWD) *framework*, which draws inspiration from the *tree of actors* design pattern[9]. Figure A.2 describes the framework.

[9] `https://docs.ray.io/en/latest/ray-core/patterns/tree-of-actors.html`

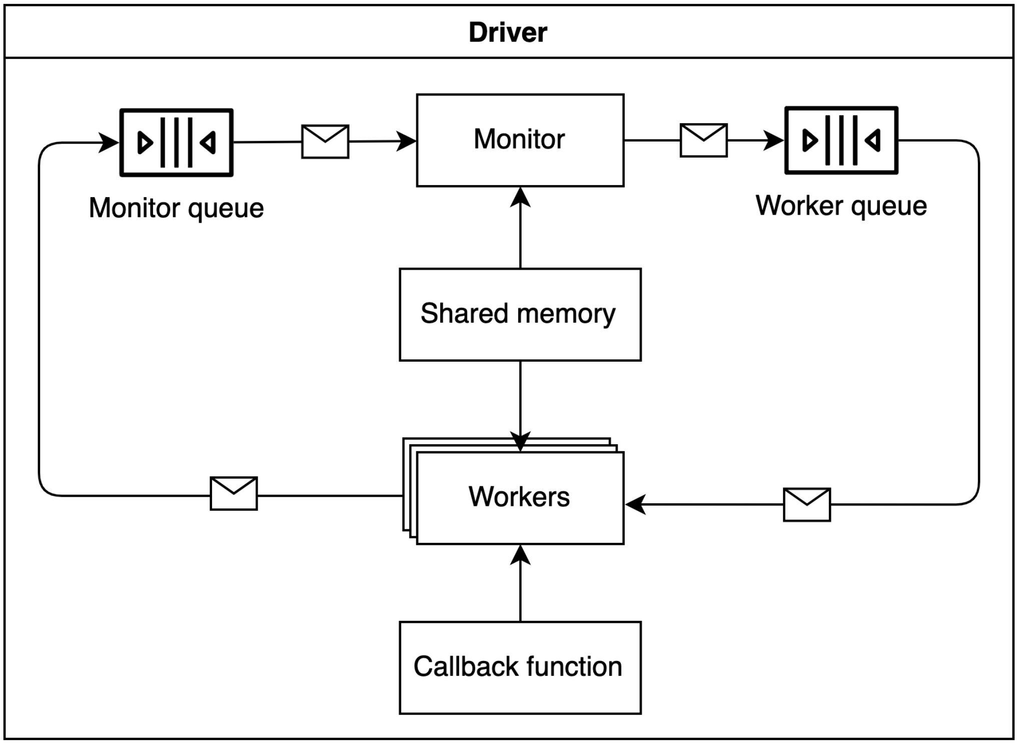

Figure A.2: Monitor-worker-driver framework. The *monitor* encapsulates the mutable state of the program and decides which messages are processed by workers. A *worker* is a stateless entity that processes the messages that the monitor puts in the *worker queue*. Worker behavior is defined by a *callback function*, which typically depends on the message contents. The side effects of processing a message are recorded as new messages and put in the *monitor queue*. This cycle repeats until some termination condition is satisfied. Any immutable state of the program can be stored in *shared memory* for efficient access. The *driver* is the entry point into the program. It initializes the monitor and workers, and then waits for termination.

For risk propagation, the driver creates the factor graph from the set of risk scores S and contacts C, stores the factor graph in shared memory, sets the initial state of the monitor to be the maximum risk score of each user, and puts all risk scores in the monitor queue. During message passing, the monitor maintains the maximum risk score (i.e., exposure score) for each variable vertex.

The MWD framework was the first approach that utilized the send coefficient to ensure the convergence and termination of message passing. However, because the MWD-based implementation assumed the factor graph representation of the contact network, the send coefficient was applied to both variable and factor messages.

Compared to Appendix A.2, this implementation provides a cleaner design and less communication overhead. However, what prompted me to consider (yet another) an alternative implementation was its scalability. Because the monitor processes messages serially, it is a bottleneck for algorithms in which the workers perform fine-grained tasks. Indeed, the Ray documentation[10] notes that the parallelization of small tasks is an anti-pattern because the interprocess communication cost exceeds the benefit of multiprocessing. Unfortunately, the computation performed by factor vertices and variable vertices is too fine-grained, so the scalability of the MWD framework was demonstrably poor.

[10] `https://docs.ray.io/en/latest/ray-core/patterns/too-fine-grained-tasks.html`

A.4 Projected Subgraph Actors

A.4.1 Evaluation

A.4.1.1 Experimental Design

Risk propagation requires a partitioning or clustering algorithm, as described in Algorithm **??**. We configured the METIS graph partitioning algorithm Karypis and Kumar (1998) to use k-way partitioning with a load imbalance factor of 0.2, to attempt contiguous partitions that have minimal inter-partition connectivity, to apply 10 iterations of refinement during each stage of the uncoarsening process, and to use the best of 3 cuts.

A.4.1.2 Synthetic Graphs

We evaluate the scalability and efficiency of risk propagation on three types of graphs: a random geometric graph (RGG) Dall and Christensen (2002), a benchmark graph (LFRG) Lancichinetti et al. (2008), and a clustered scale-free graph (CSFG) Holme and Kim (2002). Together, these graphs demonstrate some aspects of community structure Fortunato (2010) which allows us to more accurately measure the performance of risk propagation. When constructing a RGG, we set the radius to $r(n) = \min\left(1, 0.25^{\log_{10}(n)-1}\right)$, where n is the number of users. This allows us to scale the size of the graph while maintaining reasonable density. We use the following parameter values to create LFRGs: mixing parameter $\mu = 0.1$, degree power-law exponent $\gamma = 3$, community size power-law exponent $\beta = 2$, degree bounds $(k_{\min}, k_{\max}) = (3, 50)$, and community size bounds $(s_{\min}, s_{\max}) = (10, 100)$. Our choices align with the suggestions by Lancichinetti et al. (2008) in that $\gamma \in \mathbb{R}_{[2,3]}$, $\beta \in \mathbb{R}_{[1,2]}$, $k_{\min} < s_{\min}$, and $k_{\max} < s_{\max}$. To build CSFGs, we add $m = 2$ edges for each new user and use a triad formulation probability of $P_t = 0.95$. For all graphs, we remove self-loops and isolated users.

The following defines our data generation process. Let p be the probability of a user being "high risk" (i.e., $r \geq 0.5$) Then, with probability $p = 0.2$, we sample $L + 1$ values from the uniform distribution $\mathbb{U}_{[0.5,1)}$. Otherwise, we sample from $\mathbb{U}_{[0,0.5)}$. This assumes symptom scores and exposure scores are computed daily and includes the present day. We generate the times of these risk scores by sampling a time offset $t_{\text{off}} \sim \mathbb{U}_{[0\text{s};86,400\text{s}]}$ for each user such that $t_d = t_{\text{now}} + t_{\text{off}} - d$ days, where $d \in \mathbb{N}_{[0,L]}$. To generate a contact times, we follow the same procedure for risk scores, except that we randomly sample one of the $L + 1$ times and use that as the contact time.

We evaluate various transmission rates and send tolerances:

$$(\gamma, \alpha) \in \{0.1, 0.2, \ldots, 1\} \times \{0.1, 0.2, \ldots, 0.9\}.$$

For all γ, α, we set $n = 5,000$ and $K = 2$.

To measure the scalability of risk propagation, we consider $n \in \mathbb{N}_{[10^2,10^4]}$ users in increments of 100 and collect 10 iterations for each n. The number of actors we use depends on n such that $K(n) = 1$ if $n < 10^3$ and $K(n) = 2$ otherwise. Increasing K for our choice of n did not offer improved performance due to the communication overhead.

A.4.1.3 Real-World Graphs

We analyze the efficiency of risk propagation on three real-world contact networks that were collected through the SocioPatterns collaboration. Specifically, we use contact data in the setting of a high school (Thiers13) Fournet and Barrat (2014), a workplace (InVS15), and a scientific conference (SFHH) **?**. Because of limited availability of large-scale contact networks, we do not use real-world contact networks to measure the scalability of risk propagation.

To ensure that all risk scores are initially propagated, we shift all contact times forward by t_{now} and use ($t_{\text{now}} - 1$ day) when generating risk scores times. In this way, we ensure the most recent risk score is still older than the first contact time. Risk score values are generated in the same manner as described in Section A.4.1.2 with the exception that we only generate one score. Lastly, we perform 10 iterations over each data set to obtain an average performance.

A.4.2 Results

A.4.2.1 Efficiency

Prior to measuring scalability and real-world performance, we observed the effects of send tolerance and transmission rate on the efficiency of risk propagation. As ground truth, we used the maximum update count for a given transmission rate. Fig. A.3 indicates that a send tolerance of $\gamma = 0.6$ permits 99% of the possible updates. Beyond $\gamma = 0.6$, however, the transmission rate has considerable impact, regardless of the graph. As noted in Section **??**, send tolerance quantifies the trade-off between completeness and efficiency. Thus,

$\gamma = 0.6$ optimizes for both criteria.

Unlike the update count, Fig. A.3 shows a more variable relationship with respect to runtime and message count. While, in general, transmission rate (send tolerance) has a direct (resp. inverse) relationship with runtime and message count, the graph topology seems to have an impact on this fact. Namely, the LFRG displayed less variability across send tolerance and transmission rate than the RGG and CSFG, which is the cause for the large interquartile ranges. Therefore, it is useful to consider the lower quartile Q_1, the median Q_2, and the upper quartile Q_3. For $\alpha = 0.8$ and $\gamma = 0.6$, risk propagation is more efficient with $(Q_1, Q_2, Q_3) = (0.13, 0.13, 0.46)$ normalized runtime and $(Q_1, Q_2, Q_3) = (0.13, 0.15, 0.44)$ normalized message count.

A.4.2.2 Message Reachability

To validate the accuracy of **??**, we collected values of **??** and **??** for real-world and synthetic graphs. For the latter set of graphs, we observed reachability while sweeping across values of γ and α.

To measure the accuracy of **??**, let the *message reachability ratio* (MRR) be defined as

$$\mathrm{mrr}(u) := \frac{r(u)}{\hat{r}(u)}. \tag{A.1}$$

Overall, **??** is a good estimator of **??**. Across all synthetic graphs, **??** modestly underestimated **??** with quartiles $(Q_1, Q_2, Q_3) = (0.71, 0.84, 0.98)$ for the Equation (A.1). For $\alpha = 0.8$ and $\gamma = 0.6$, the quartiles of Equation (A.1) were $(Q_1, Q_2, Q_3) = (0.52, 0.77, 1.12)$ and $(Q_1, Q_2, Q_3) = (0.79, 0.84, 0.93)$, respectively. Table A.1 provides mean values of Equation (A.1) for both synthetic

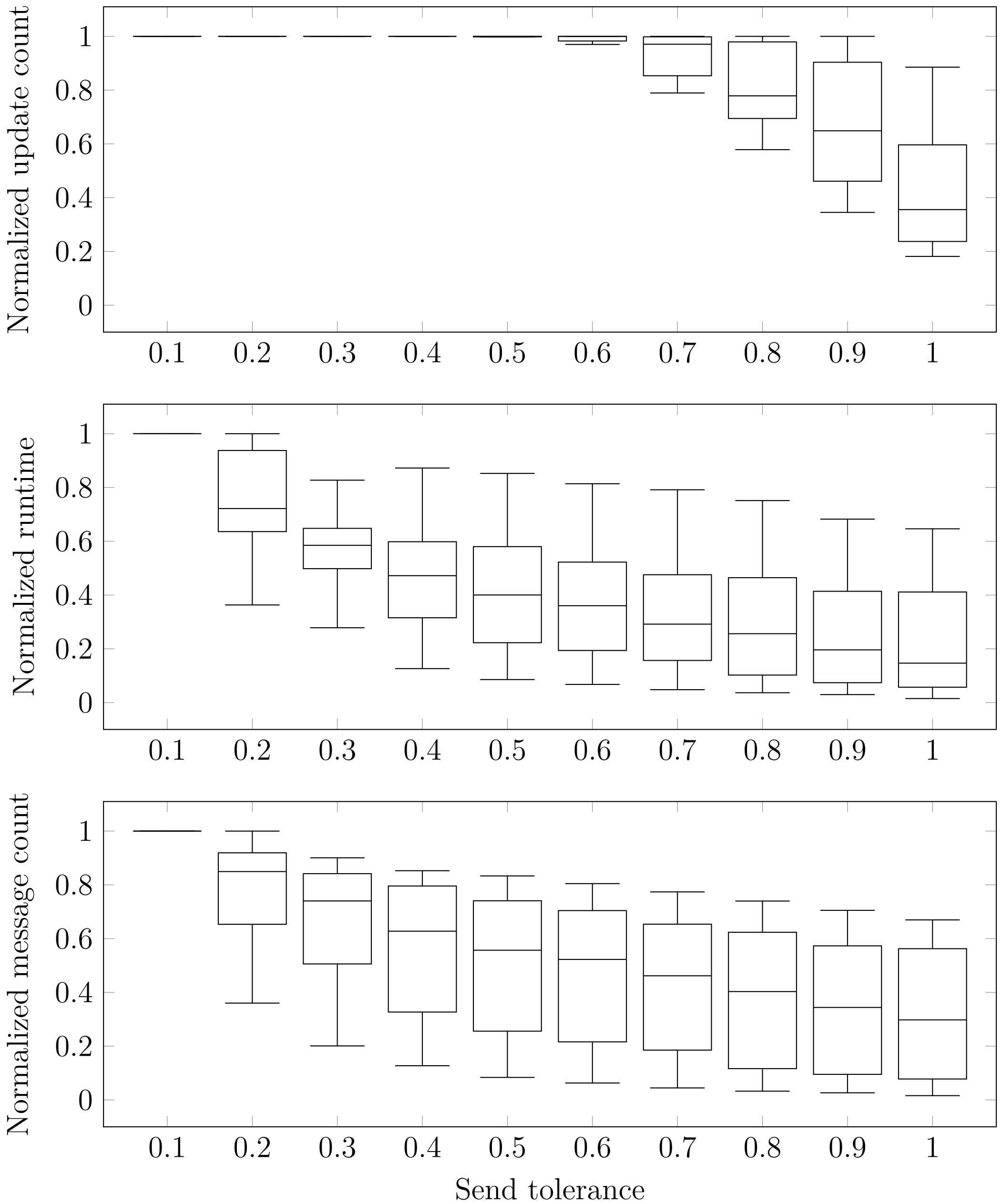

Figure A.3: Effects of send tolerance on efficiency. All dependent variables are normalized across graphs and transmission rates.

and real-world graphs. Fig. A.4 indicates that moderate values of γ tend to result in a more stable MRR, with lower (higher) γ underestimating (resp. overestimating) **??**. With regard to transmission rate, Equation (A.1) tends to decrease with increasing α, but also exhibits larger interquartile ranges.

Because **??** does not account for the temporality constraints **??** and **??**, it does not perfectly estimate **??**. With lower γ and higher α, **??** suggests higher MR. However, because a message is only passed under certain conditions (see Algorithm **??**), this causes **??** to overestimate **??**. While **??** theoretically is an upper bound on **??**, it is possible for **??** to underestimate **??** if the specified value of $s_0(v)$ overestimates the true value of $s_0(v)$. When computing Equation (A.1) for Fig. A.4, we used the mean $s_0(v)$ across all users v, so $\text{mrr}(u) > 1$ in some cases.

Setting	$\text{mrr}(u) \pm 1.96 \cdot \text{SE}$
Synthetic	
LFR	0.88 ± 0.14
RGG	0.74 ± 0.12
CSFG	0.90 ± 0.14
	$\mathbf{0.85 \pm 0.08}$
Real-world	
Thiers13	0.58 ± 0.01
InVS15	0.63 ± 0.01
SFHH	0.60 ± 0.01
	$\mathbf{0.60 \pm 0.01}$

Table A.1: Message reachability ratio for synthetic and real-world graphs ($\alpha = 0.8$, $\gamma = 0.6$). Synthetic (real-world) ratios are averaged across parameter combinations (resp. runs).

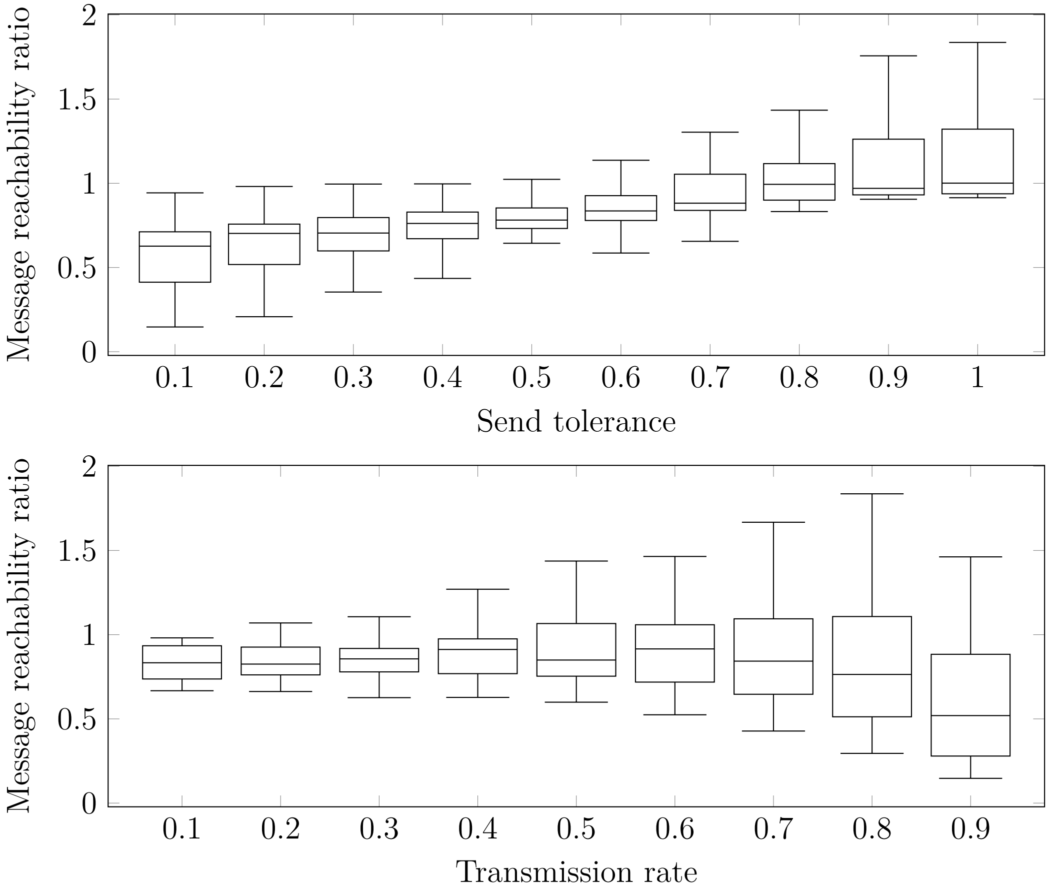

Figure A.4: Effects of send tolerance and transmission rate on the message reachability ratio. Independent variables are grouped across graphs.

A.4.2.3 Scalability

Fig. A.5 describes the runtime behavior of risk propagation. The runtime of CSFGs requires further investigation. A linear regression fit explains ($R^2 = 0.52$) the runtime of LFRGs and RGGs with a slope $m = (1.1 \pm 0.1) \cdot 10^{-3}$ s/contact and intercept $b = 4.3 \pm 1.6$s ($\pm 1.96 \cdot \mathrm{SE}$).

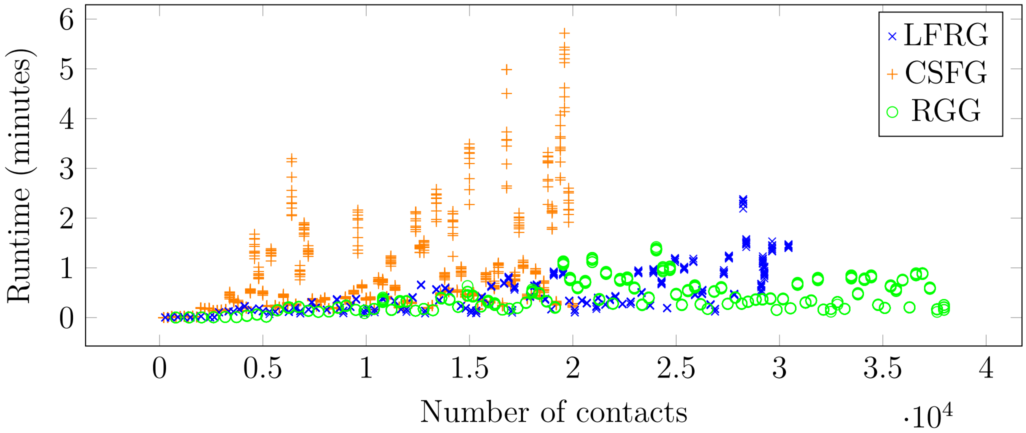

Figure A.5: Runtime of risk propagation on synthetic graphs containing 100–10,000 users and approximately 200–38,000 contacts.

A.5 Location-Based Contact Tracing

- Motivation: Google/Apple API prevents exporting Bluetooth EphIDs
- Other location-based contact tracing approaches

A.5.1 System Model

The system model is very similar to previous work Ayday et al. (2020, 2021) and designs. The only difference is that user geolocation data is collected instead of Bluetooth ephemeral identifiers. Figure A.6 shows the modified dataflow[11]. *Geohashing* is a public-domain encoding system that maps *geographic coordinates* (i.e., latitude-longitude ordered pairs (Sickle, 2004, p. 5)) to alphanumeric strings called *geohashes*, where the length of a geohash is correlated with its geospatial precision Morton (1966). To offer some basic privacy, a user's precise geolocation history is obfuscated on-device by encoding geographic coordinates as geohashes with 8-character precision which corresponds to a region of 730m^2.

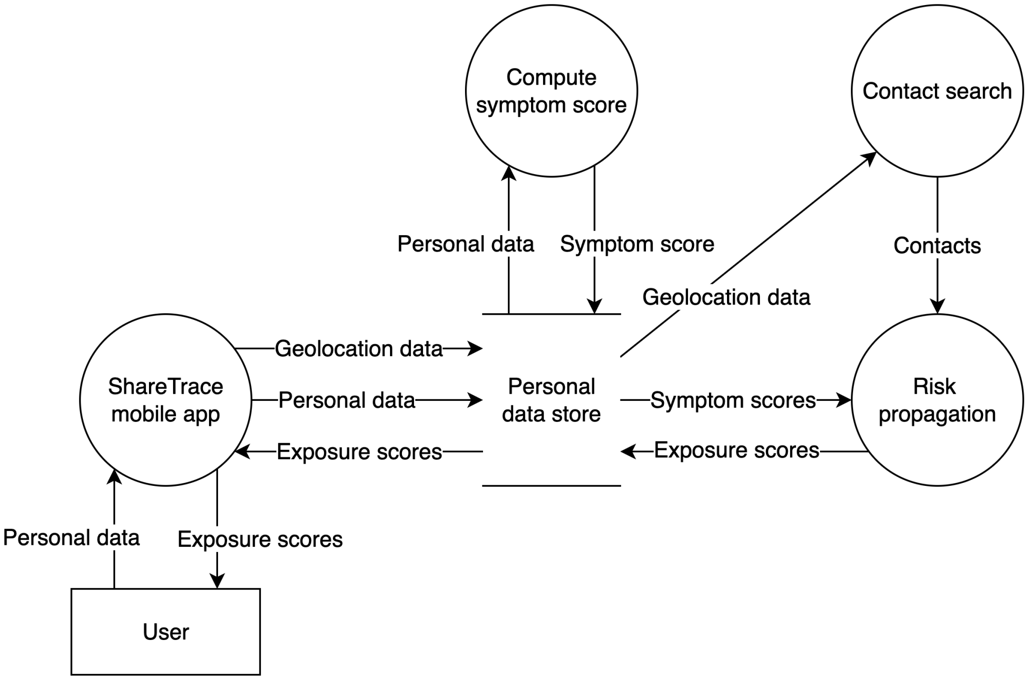

Figure A.6: Location-based ShareTrace dataflow. This requires that risk propagation is executed in a centralized setting since all user geolocation data is needed to construct the contact network.

[11] See **??** (p. **??**) for the definition of a dataflow diagram.

A.5.2 Contact Search

a temporally ordered sequence of timestamped geolocations. It is assumed that

1. geolocation histories are not recorded on a fixed schedule, and

2. a user remains at a geolocation until the next geolocation is recorded.

Finding the most recent contact between two users from their aligned geolocation histories is similar to finding the last k-length common substring between two strings, where each symbol represents a timestamped geolocation. The difference lies in how the start and end of the contact time interval is defined. By assumption 2, the start (end) of a contact time interval is defined as the earlier (ref. later) timestamp of the two first (ref. last) timestamped geolocations in the sequence where the two histories differ. Figure A.7 provides a visual example.

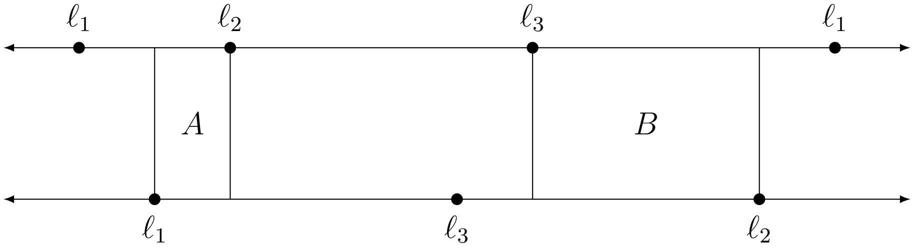

Figure A.7: Contact search with two geolocation histories. Each line denotes time, increasing from left to right. A point ℓ_i is a geolocation and occurs relative in time with respect to the placement of other points. Region B defines the contact interval as it is of sufficient duration and occurs after A.

A.5.2.1 Naive Contact Search

A naive approach to finding all contacts amongst a set of geolocation histories $\mathcal{H}$ is to compare all unordered pairs. For a given pair of aligned geolocation histories, the idea is to maintain a pointer to the previous and current index in each history, advancing the pair of pointers whose geolocation occurs later in time. Once a common geolocation is found, all pointers are advanced together until the geolocations differ. If the sequence is δ-contiguous, where a sequence of timestamped geolocations S is *δ-contiguous* if $S.length \geq \delta$ and then it is recorded. The latest such sequence is used to define the contact between the two users. Because only the most recent time of contact is of interest, the procedure can be improved by iterating in reverse and then terminating once a sequence is found. Regardless, this approach takes $\Theta(n^2)$ time, where $n = |\mathcal{H}|$, because all $\frac{n(n-1)}{2}$ unique pairs must be considered.

A.5.2.2 Indexed Contact Search

While the **for** loop in Naive-Contact-Search is *embarrassingly parallel* (Herlihy and Shavit, 2012, p. 14), the naive approach is neither scalable nor efficient. It can be improved by observing that it is necessary, but not sufficient, that a pair of ϵ-proximal geolocations exists between two geolocation histories for a contact to exist. Therefore, the geolocation histories $\mathcal{H}$ can be indexed into a spatial data structure $\mathcal{I}$ Mahmood et al. (2019); Mokbel et al. (2003); **?** and then only consider the geolocation-history pairs that share at least one ϵ-proximal geolocation pair. This approach is described by the Indexed-Contact-Search operation.

Line 2 executes a fixed-radius near-neighbors search (FR-NNS) Bentley (1975); Brin (1995) for each geolocation in the spatial index $\mathcal{I}$. Formally, given a set of geolocations $\mathcal{L} \subseteq \mathbb{L}$, a metric d, and a distance ϵ, the *fixed-radius near-neighbors* of a geolocation $\ell \in \mathcal{L}$ is defined as the subset of ϵ-proximal geolocations Brin (1995),

$$\mathcal{N}(\ell) = \{\ell' \in \mathcal{L} \mid d(\ell, \ell') \leq \epsilon\}$$

Note that the set of neighbors $\mathcal{N}(i)$ of user i corresponds to the geolocations that are ϵ-proximal to *any* of the geolocations in their geolocation history H_i,

$$\mathcal{N}(i) = \bigcup_{\ell \in H_i} \mathcal{N}(\ell).$$

On line 3, the operation Unique-Users maps these near-neighbors back to the associated users, removing any duplicates that may arise from mapping multiple geolocations to the same user. Finally, line 4 maps the set of users $\mathcal{U}$ back to their geolocation histories and runs Naive-Contact-Search on the resultant subset.

```
Indexed-Contact-Search($\mathcal{H}$)
 1: $\mathcal{I} \leftarrow$ Spatially-Index($\mathcal{H}$)
 2: $\mathcal{N} \leftarrow$ Fixed-Radius-Near-Neighbors($\mathcal{I}, \epsilon$)
 3: $\mathcal{U} \leftarrow$ Unique-Users($\mathcal{N}, \mathcal{H}$)
 4: return Naive-Contact-Search($\{H_i \in \mathcal{H} \mid i \in \mathcal{U}\}$)
```

To carry out FR-NNS, one approach is to use a *ball tree*, a complete binary tree that associates with each node a hypersphere that contains a subset of the data (Kibriya and Frank, 2007; Omohundro, 1989; **?**). Any metric can be used to perform FR-NNS on a ball tree. However, because geolocation is represented as geographic coordinates, metrics that assume a Cartesian coordinate system may be unsuitable. One of the simplest geometric models of the Earth is that of a sphere. Given two geographic coordinates, the problem of finding the length of the geodesic[12] between them is known as the *inverse geodetic problem* (**?**). Assuming a spherical Earth, the solution to the inverse problem is to find the length of the segment that joins the two points on a great circle[13].

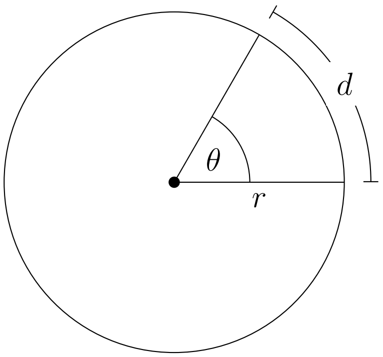

Figure A.8: Central angle of a great circle.

The *haversine*, or the half "versed" (i.e., reversed) sine, of a central angle θ is defined as

$$\operatorname{hav}\theta = \frac{\operatorname{vers}\theta}{2} = \frac{1-\cos\theta}{2} = \sin^2\frac{\theta}{2}. \tag{A.2}$$

The great-circle distance d between two points can be found by inversing (A.2)

[12] The *geodesic* is the shortest segment between two points on an ellipsoid (Lu et al., 2014).

[13] The *great circle* is the cross-section of a sphere that contains its center (Lu et al., 2014)

and solving,

$$d(\ell_i, \ell_j) = 2 \cdot \arcsin \sqrt{\sin^2 \frac{\phi_i - \phi_j}{2} + \cos \phi_i \cdot \cos \phi_j \cdot \sin^2 \frac{\lambda_i - \lambda_j}{2}},$$

where $\ell_i = (\phi_i, \lambda_i)$ is a latitude-longitude coordinate in radians (Brummelen, 2013, pp. 157–162).

The choice of the great-circle distance was primarily driven by its readily available usage in the scikit-learn **?** implementation of a ball tree. If such an approach for discovering contacts were to be used in practice, more advanced *geodetic datum* (Lu et al., 2014, pp. 71–130) could be used to provide better geospatial accuracy. Moreover, by projecting geodetic coordinates onto the plane, metrics that assume a Cartesian coordinate system could be used instead (Lu et al., 2014, pp. 265–326).

The running time of Spatially-Index is $\Theta\left(|\mathcal{H}| \cdot \log |\mathcal{H}|\right)$ Omohundro (1989). Assuming the ball tree is balanced[14], the running time of Fixed-Radius-Near-Neighbors to find the ϵ-proximal neighbors of all geolocations in the spatial index $\mathcal{I}$ is $\Theta(k\,|\mathcal{I}| \cdot \log |\mathcal{I}|)$, where k is the dimensionality of a tree element (i.e., $k = 2$ for geographic coordinates). The running time of Unique-Users is $\Theta(|\mathcal{N}|)$. While the running time of Naive-Contact-Search is $\Theta\left(|\mathcal{U}|^2\right)$, it is likely that $|\mathcal{U}| << |\mathcal{H}|$. Regardless of the input,

$$|\mathcal{H}| \geq |\mathcal{I}| \geq |\mathcal{U}|$$

since $|\mathcal{H}| = |\mathcal{I}|$ only if all geolocations in $\mathcal{H}$ are distinct, and $|\mathcal{I}| = |\mathcal{U}|$ only

[14] The ball-tree implementation provided by scikit-learn **?** ensures the tree is balanced.

if each user has exactly one geolocation that is distinct from all other users. Dependent upon the input, however, is $|\mathcal{N}|$. In the worst case, $|\mathcal{N}| \in O\left(|\mathcal{I}|^2\right)$ if each geolocation is ϵ-proximal to all other geolocations. This implies that the overall worst-case running time of Indexed-Contact-Search is $O(|\mathcal{H}|^2)$. In practice, the running time depends on the geohash precision as well as the geospatial density and mobility behavior of the user population.

Appendix B

Pseudocode Conventions

Pseudocode conventions are mostly consistent with Cormen et al. (2022).

- Indentation indicates block structure.
- Looping and conditional constructs have similar interpretations to those in standard programming languages.
- Composite data types are represented as *objects*. Accessing an *atttribute* a of an object o is denoted $o.a$. A variable representing an object is a *pointer* or *reference* to the data representing the object. The special value NIL refers to the absence of an object.
- Parameters are passed to a procedure *by value*: the "procedure receives its own copy of the parameters, and if it assigns a value to a parameters, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the attributes of the object are not." Thus, attribute assignment "is visible

if the calling procedure has a pointer to the same object.”

- A **return** statement “immediately transfers control back to the point of call in the calling procedure.”

- Boolean operators **and** and **or** are *short circuiting.*

The following conventions are specific to this work.

- Object attributes may be defined *dynamically* in a procedure.

- Variables are local to the given procedure, but parameters are global.

- The “$\leftarrow$” symbol is used to denote assignment, instead of “$=$”.

- The “$=$” symbol is used to denote equality, instead of “$==$”, which is consistent with the use of “$\neq$” to denote inequality.

- The “$\in$” symbol is used in **for** loops when iterating over a collection.

- Set-builder notation $\{\, x \in X \mid \textsc{Predicate}(x) \,\}$ is used to create a subset of a collection X in place of constructing an explicit data structure.

Appendix C

Data Structures

Let a *dynamic set* X be a mutable collection of distinct elements. Let x be a pointer to an element in X such that $x.key$ uniquely identifies the element in X. Let a *dictionary* be a dynamic set that supports insertion, deletion, and membership querying (Cormen et al., 2022).

- $\textsc{Insert}(X, x)$ adds the element pointed to by x to X.
- $\textsc{Delete}(X, x)$ removes the element pointed to by x from X.
- $\textsc{Search}(X, k)$ returns a pointer x to an element in the set X such that $x.key = k$; or NIL, if no such element exists in X.
- $\textsc{Merge}(X, x)$ adds the element pointed to by x, if no such element exists in X; otherwise, the result of applying a function to x and the existing element is added to X.
- $\textsc{Maximum}(X)$ returns a pointer x to the maximum element of the totally ordered set X; or NIL if X is empty.

Appendix D

Actor Model

The *actor model* is a local model of concurrent computing that defines computation as a strict partial ordering of events (Hewitt, 1977; Hewitt and Baker, 1977). An *event* is defined as "a transition from one local state to another" (Hewitt and Baker, 1977); or, more concretely, "a *message* arriving at a computational agent called an *actor*." The actor that receives the message of an event is called the *target* of the event.

Upon receipt, an actor may send messages to itself or other actors, update its local state, or create other actors.

May send messages to other actors (Clinger) Update its local state (Clinger) Create other actors

Events caused

An event may *activate* subsequent events

Ordering laws:

Passing messages between actors is the only means of communication. Thus, control and data flow are inseparable in the actor model (Hewitt et al., 1973).

A minimal specification of an actor includes its *name* and *behavior*, or how it acts upon receiving a message. Clinger (1981) explains,

> An actor's name is a necessary part of its description because two different actors may have the same behavior. An actor's behavior is a necessary part of its description because the same actor may have different behaviors at different times.

acquaintances, or a finite collection of names of the actors that it knows about (Hewitt et al., 1973).

Thus, the specification of an actor system consists of the following (Hewitt, 1977).

- Deciding on the natural kinds of actors (objects) to have in the system to be constructed.
- Deciding for each kind of actor what kind of messages it should receive.
- Deciding for each kind of actor what it should do when it receives each kind of message.

Bibliography

Erman Ayday, Fred Collopy, Taehyun Hwang, Glenn Parry, Justo Karell, James Kingston, Irene Ng, Aiden Owens, Brian Ray, Shirley Reynolds, Jenny Tran, Shawn Yeager, Youngjin Yoo, and Gretchen Young. Sharetrace: A smart privacy-preserving contact tracing solution by architectural design during an epidemic. Technical report, Case Western Reserve University, 2020. URL `https://github.com/cwru-xlab/sharetrace-papers/blob/main/sharetace-whitepaper.pdf`.

Erman Ayday, Youngjin Yoo, and Anisa Halimi. ShareTrace: An iterative message passing algorithm for efficient and effective disease risk assessment on an interaction graph. In *Proc. 12th ACM Con. Bioinformatics, Comput. Biology, Health Inform.*, BCB 2021, 2021.

Jon Louis Bentley. A survey of techniques for fixed radius near neighbor searching. Technical report, Stanford University, 1975.

Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, 1995.

Glen Van Brummelen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry*. Princeton University Press, 2013.

William Clinger. *Foundations of actor semantics*. PhD thesis, MIT, 1981. URL `http://hdl.handle.net/1721.1/6935`.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, fourth edition, 2022.

Jesper Dall and Michael Christensen. Random geometric graphs. *Phys. Rev. E*, 66, 2002. doi:10.1103/PhysRevE.66.016121.

Santo Fortunato. Community detection in graphs. *Phys. Rep.*, 486, 2010. doi:10.1016/j.physrep.2009.11.002.

Julie Fournet and Alain Barrat. Contact patterns among high school students. *PLoS ONE*, 9, 2014. doi:10.1371/journal.pone.0107878.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977. doi:10.1016/0004-3702(77)90033-9.

Carl Hewitt and Henry Baker. Laws for communicating parallel processes. Working paper, MIT Artificial Intelligence Laboratory, 1977. URL `http://hdl.handle.net/1721.1/41962`.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International*

Joint Conference on Artificial Intelligence, IJCAI'73, pages 235–245, 1973. URL https://dl.acm.org/doi/10.5555/1624775.1624804.

Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65, 2002. doi:10.1103/PhysRevE.65.026107.

George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998. doi:10.1137/S1064827595287997.

Ashraf M. Kibriya and Eibe Frank. An empirical comparison of exact nearest neighbour algorithms. In *Knowledge Discovery in Databases: PKDD 2007*, pages 140–151, 2007.

Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78, 2008. doi:10.1103/PhysRevE.78.046110.

Zhiping Lu, Yunying Qu, and Shubo Qiao. *Geodesy: Introduction to Geodetic Datum and Geodetic Systems*. Springer, 2014.

Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. Spatio-temporal access methods: a survey (2010 - 2017). *Geoinformatica*, 23:1–36, 2019.

Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Robert McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surveys*, 48, 2015. doi:10.1145/2818185.

Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26:1–7, 2003.

G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, International Business Machines, 1966.

Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

Alex Preukschat and Drummond Reed. *Self-Sovereign Identity: Decentralized digital identity and verifiable credentials*. Manning, 2021.

Jan Van Sickle. *Basic GIS coordinates*. CRC Press, 2004.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990. doi:10.1145/79173.79181.