Skywing: A Software Platform for Decentralized Computing

ICERM, Asynchronous Methods for Numerical Linear Algebra

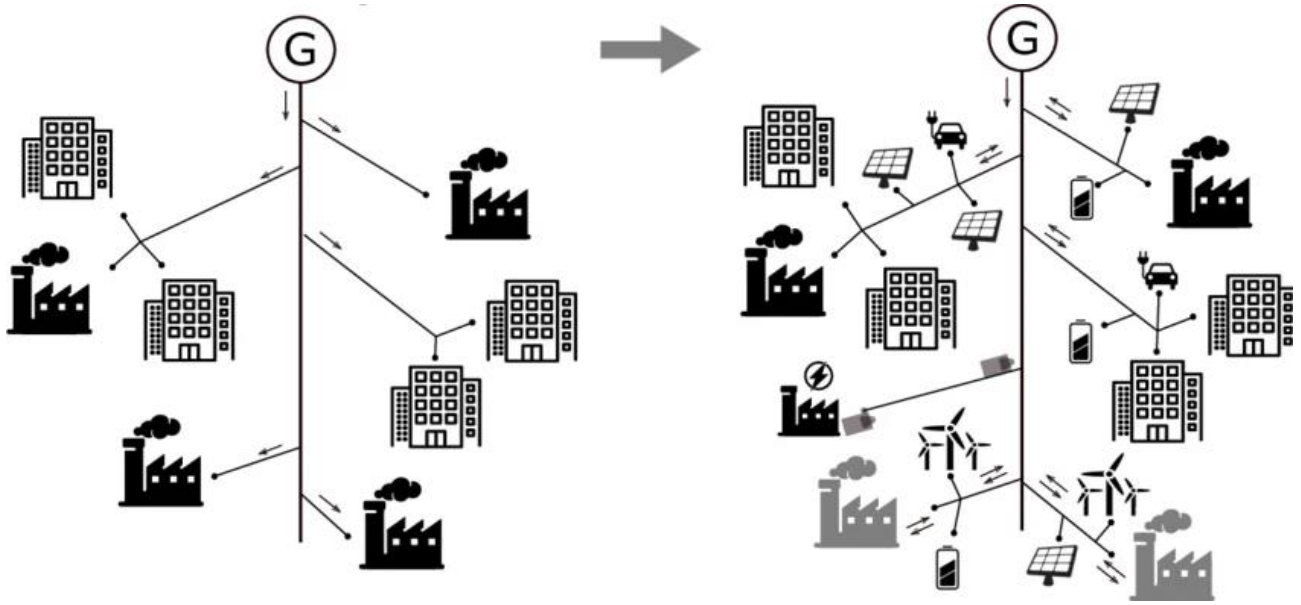
May 4-8, 2026

Annika Mauro | Global Security Computer Applications Division

Lawerence Livermore National Laboratory

Prepared by LLNL under Contract DE-AC52-07NA27344.

Emerging computing environments are increasingly **decentralized**



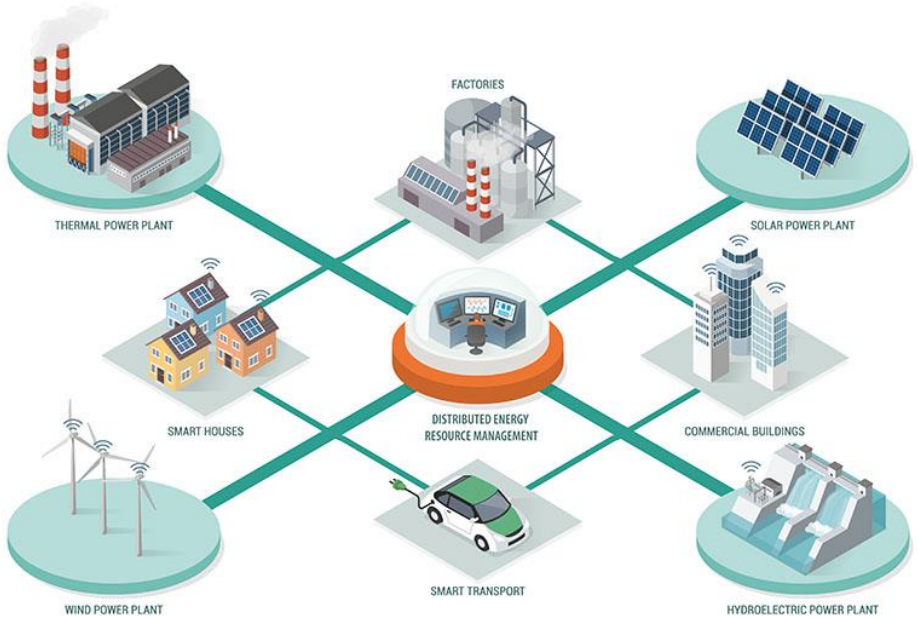
Centralized control

Distributed energy resources
Goal: Decentralized control

We would like to utilize these computing environments, but there are challenges

Key challenges in decentralized environments:

- No central controller
- Constantly changing participation and device disconnections
- Unreliable network communication
- Streaming, potentially lossy data
- Heterogenous device capabilities

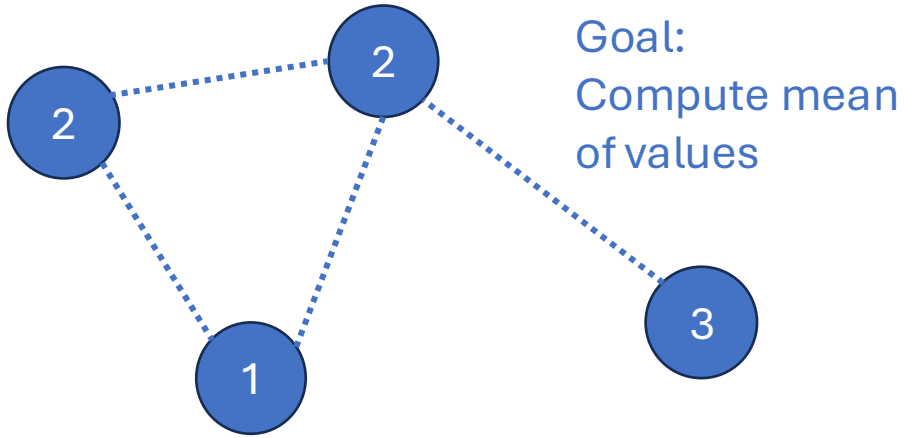


These challenges have created **gaps in decentralized algorithms and decentralized software platforms**

Gap 1: Algorithmic gap
Creating decentralized algorithms is hard

Not only do our algorithms need to be asynchronous and distributed, but also decentralized!

- No central processing
- No full communication of data
- No synchronization points
- Data can be updated in a streaming fashion
- Ideally, fault tolerant



We need research into **decentralized algorithms** to be able to utilize emerging computing environments

Gap 2: Software platform gap

Existing software is not designed for decentralized environments

MPI and OpenMP
Even with non-blocking, needs some centrally coordination, not lightweight

Ray core
Much less lightweight, uses distributed object store, ML and large-scale cluster focus

We need a flexible **platform designed for decentralized computation** to be able to utilize emerging computing environments

Skywing

https://github.com/llnl/Skywing

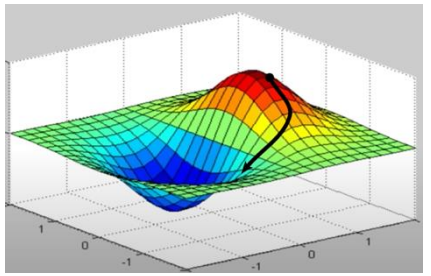
Skywing is an open-source Python-based software platform for **decentralized, unreliable environments**, with an emphasis on **asynchronous iterative methods**.

Application layer

User interfaces

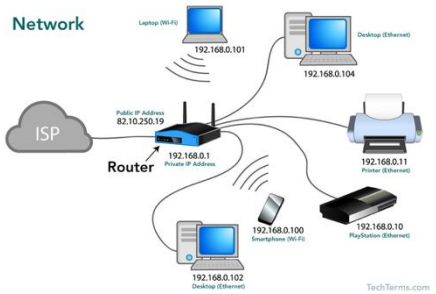
Algorithm layer

Repository of algorithmic building-blocks

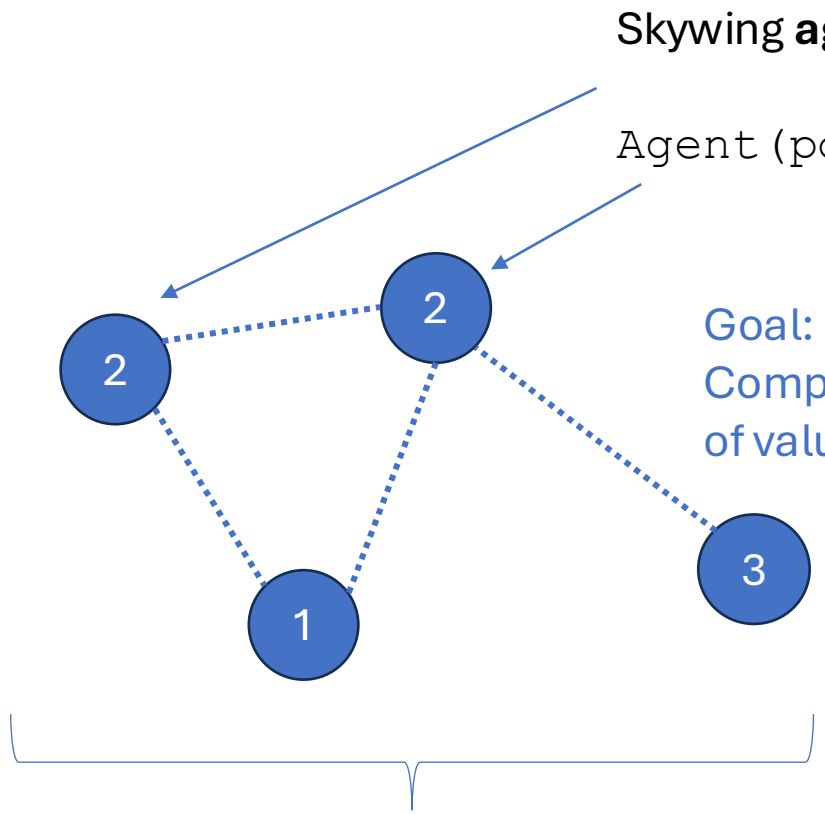


Communication layer

Network communication, message handling



Skywing terms



Accomplished with an **Iteration**, initialized with a specific **Processor**/ algorithm

```
Iteration(MeanProcessor)
```

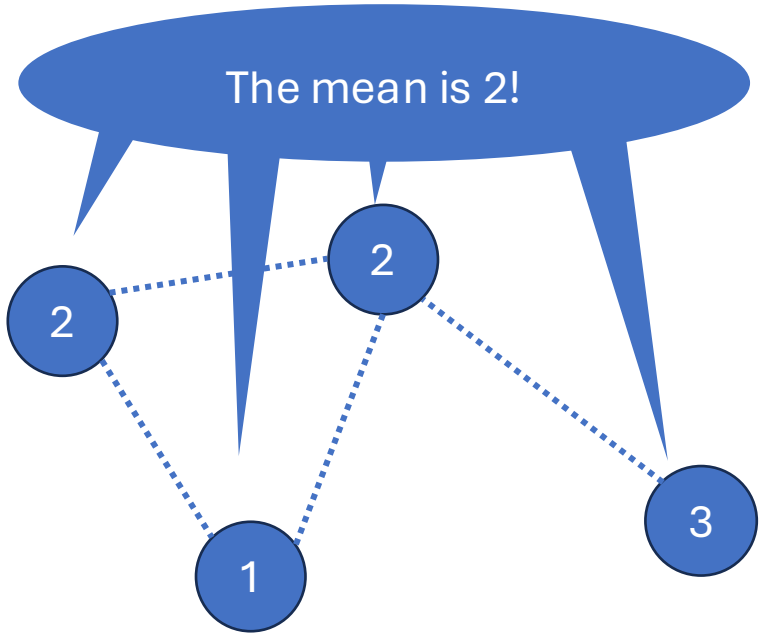
Agents publish and subscribe to **tags**, which are labels for data streams

Application layer, simple mean example

Construct agent, iteration, processor

Launch – runs agents in the background

But, we still can interact if we want




```
agent = Agent(addr, port, nbrs)
mean_iteration = Iteration(MeanProcessor,
agent)
mean_iteration.launch(data)
sleep(60)
result = mean_iteration.get_result()
while True:
    sleep(60)
    result = mean_iteration.get_result()
    some_control_logic(result)
    mean_iteration.update_data(get_new_data())
```

Application layer, multiple iteration example

Goal: find variance of the data
The mean of the squared distance of each value from the mean

```
agent = Agent(addr, port, nbrs)
mean_iteration = Iteration(MeanProcessor, agent)
mean_iteration.launch(data)
variance_iteration = Iteration(MeanProcessor, agent)
variance_iteration.launch()
while true:
      mean_result = mean_iteration.get_result()
      sq_dist_from_mean = (mean_result - data)^2
      variance_iteration.update_data(sq_dist_from_mean)
      variance = variance_iteration.query()
```

First mean processor will calculate mean

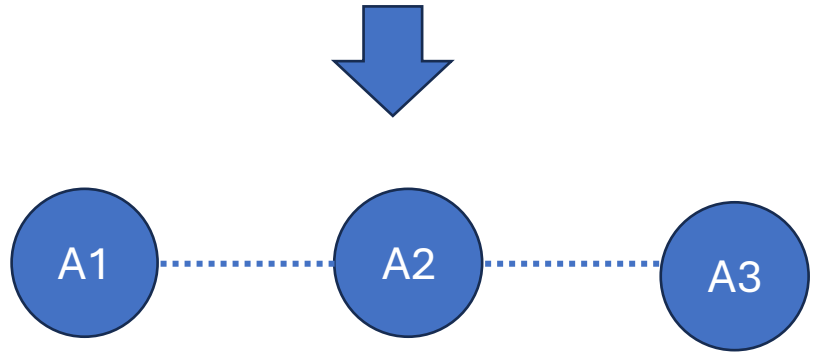
Second mean processor will calculate variance

Output of one iteration becomes the input to another with some frequency

Application layer – driver interface

- We have some drivers to help launch agents and record iterate history results
- Helps support more traditional research workflows
- For example: Generate a 9 x 9 matrix with condition number 100, partition by column into 3 portions, generate a Skywing collective of 3 agents connected in a line, run Jacobi with certain asynchronous delays, record results for each agent, and plot convergence profile

$$A = \begin{bmatrix} | & | & | \\ A_1 & A_2 & A_3 \\ | & | & | \end{bmatrix}$$



Algorithm layer - typical distributed Jacobi code

```
# Assume know local_rows of A and b, have U and L
local_diag_inv = 1.0 / A[local_rows, local_rows]
for iteration in range(max_iters):
    recv_data = get_neighbor_updates()

    for (neighbor_x, neighbor_rows) in recv_data:
        x_global[neighbor_rows] = neighbor_x

    local_x = local_diag_inv * (b[local_rows] - (U+L)[local_rows,
    :] @ x_global)
    x_global[local_rows] = local_x

    send_data_to_neighbors(local_x, local_rows)

return x_global
```

initialize

process update

prepare for publication

get result

Same Jacobi algorithm, broken into sections

```
class JacobiProcessor:
    def __init__(self):
        local_diag_inv = 1.0 / A[local_rows, local_rows]

    def update_data(self, data):
        A,b = data

    def get_result(self):
        return x_global

    def process_update(self, my_tag, recv_data)
        for (neighbor_x, neighbor_indices) in recv_data:
            x_global[neighbor_indices] = neighbor_x
        local_x = local_diag_inv * (b[local_rows] - (U+L)[local_rows, :]
        @ x_global
        x_global[local_rows] = local_x

    def prepare_for_publication(self):
        return local_x, local_rows
```

Iterative method main loop

Re-combines processor sections into a typical interactive method interface

```
while True:
    data = processor.prepare_for_publication()
    publish(my_tag, data)
    for tag in other_tags:
        if has_data(tag):
            # Retrieves any new data received under this tag
            tag_data = get_data_if_present(tag)
            iter_data[tag] = tag_data
    processor.process_update(my_tag, iter_data)
```

`get_result()` and `update_data()` allow the user interact with the iteration whenever they want to

Algorithm layer – interface summary

```
class Processor:
    def update_data(self, data):
        # For algorithms that handle data updates
    def get_result(self):
        # return current result
    def prepare_for_publication(self):
        # return data to publish
    def process_update(self, my_tag, recv_data):
        for nbr_tag, data in recv_data:
                # handle nbr data
```

This interface separates out algorithm logic with common iterative method structure

Benefits:

- Allows the user to call get_result() or update_data() whenever they want for streaming workflows
- Creates a common interface of well-defined components
- Allows **composability**

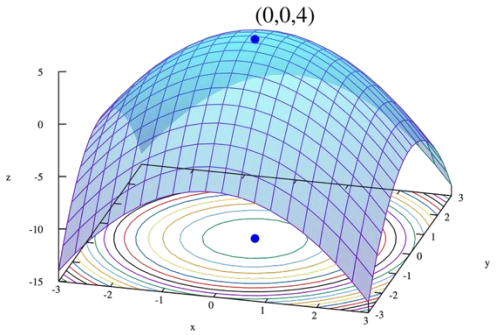
Algorithm layer - composing algorithms

Goal: combine a mean and a count processor into a sum processor

- Every processor has a standard interface, so it is easy to compose processors!

```
class SumProcessor:
    def __init__(self)
        self.mean_processor = MeanProcessor(data)
        self.count_processor = CountProcesor()
    def update_data(self, data):
        mean_processor.update_data(data)
    def get_result(self):
        mean = meanProcessor.get_result()
        num_agents = count_processor.get_result()
        sum = mean * num_agents
    def process_update(self, my_tag, recv_data)
        return mean_processor.process_update(recv_data[0]),
               count_processor.process_update(recs_data[1])
    def prepare_for_publication(self):
        return mean_processor.prepare_for_publication(),
               count_processor.prepare_for_publication()
```

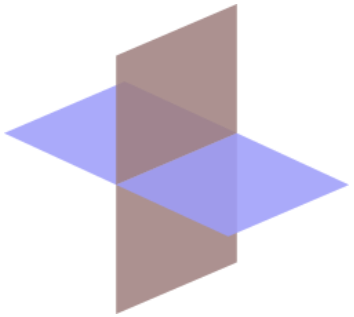
Algorithm layer – supported algorithms



Optimization

- ADMM
- Asynchronous SGD
- CoLa: Communication-Efficient Decentralized Linear Learning (work in progress)
- ASY-SONATA: Asynchronous distributed Successive cONvex Approximation algorithm over Time-varying digrAphs

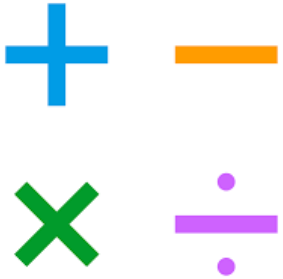
More to come!



Linear Solvers

- Asynchronous Jacobi

More to come!



Foundational math

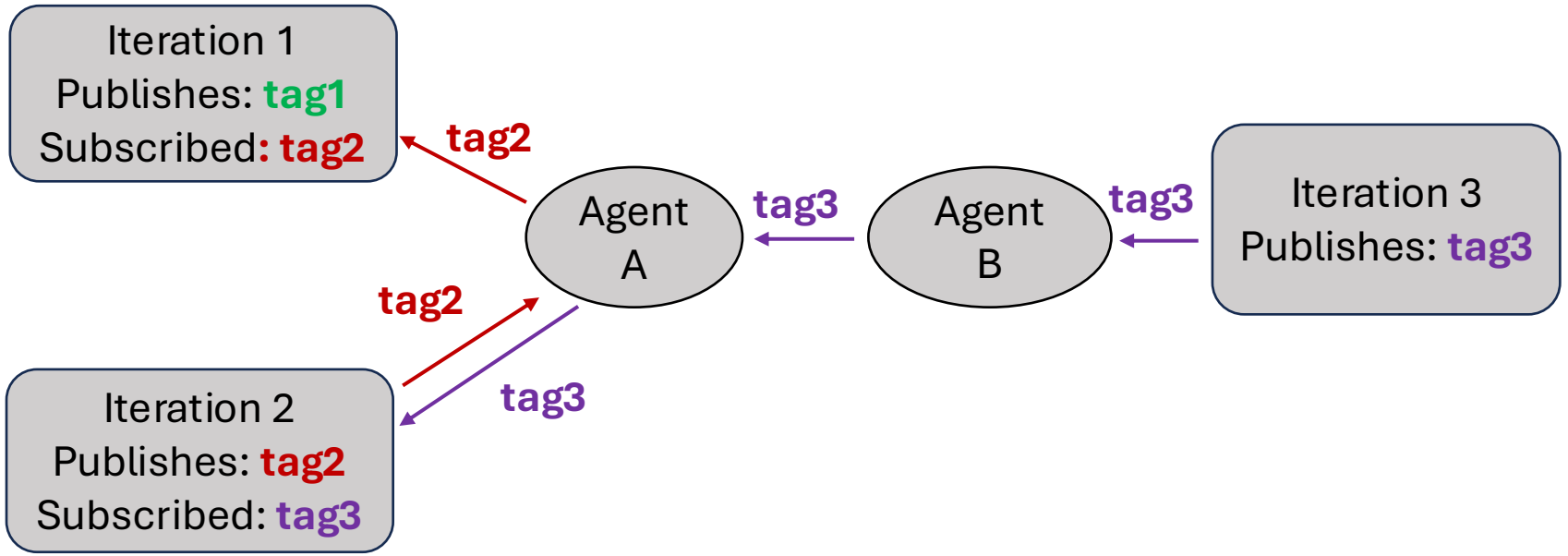
- Multiple mean/sum options
- Max/min supporting data updates

More to come!

Communication layer

Basic communication layer provides

- Communication/networking protocols
- Publish/Subscribe functionality
- Encode/decode functionality

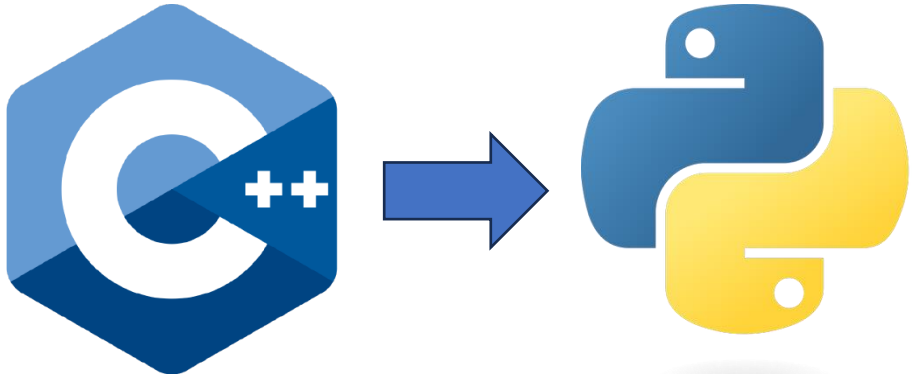


Python Skywing redesign/rewrite

Skywing originally was written in C++, but …

- C++ was less accessible to both develop and use
- Some features initially developed ended up being more cumbersome than useful
- Computation speed likely not the bottleneck compared to network speed
- C++/python combo still harder to develop

Taking all that we learned from the C++ version, we decided to rewrite and refine the software design in Python



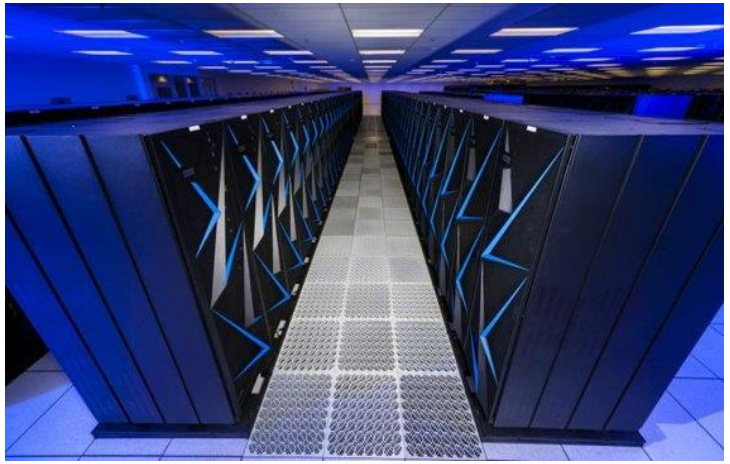
This python-only Skywing will be released this summer
(email me if you want to be notified of this: mauro3@llnl.gov)

Skywing supports both research and deployment settings

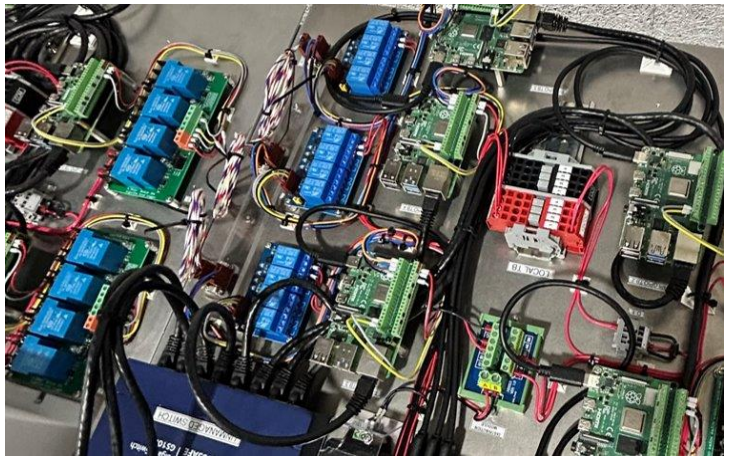
LLNL HPC

Testbeds

Field pilots



Development



Testing

Pilot deployment

Algorithms developed and tested in Skywing can then be used in applications

Skywing testing – Raspberry PI testbeds

- Contains real **hardware** - Raspberry Pis, Switches, Relays
- Each building/Pi runs a **Skywing agent**
- Validates Skywing in a more realistic deployment setting

Skywing takeaways

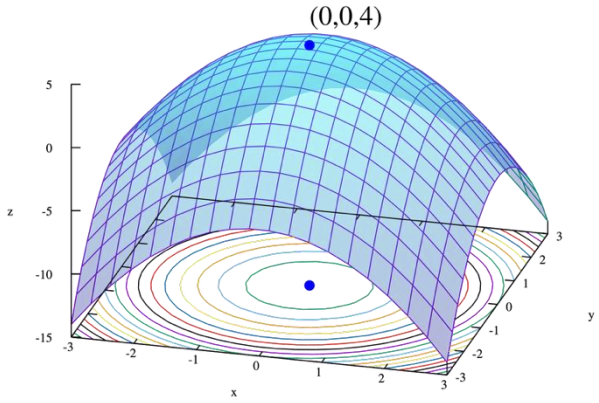
Open-source, python-based

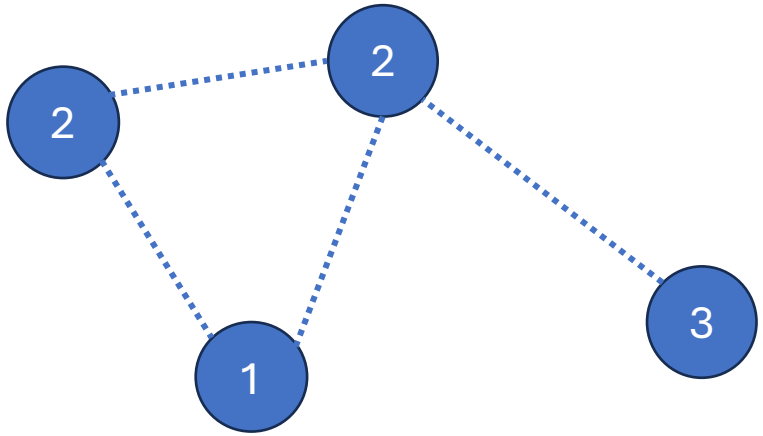
Fully decentralized structure

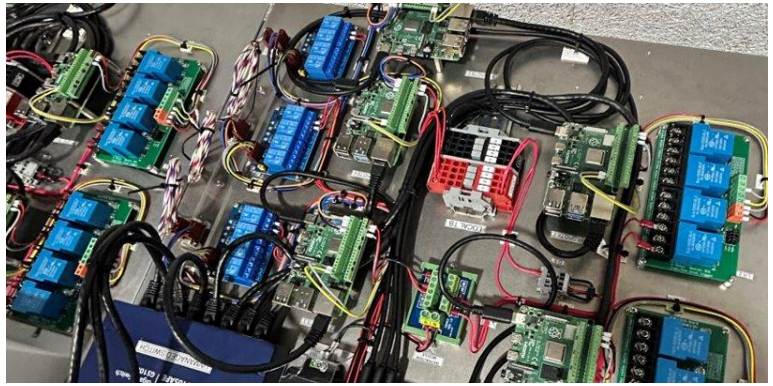
Lightweight

Supports research and deployment workflows

Toolbox of compossible algorithms







Current Team:
Annika Mauro - mauro3@llnl.gov
Alyson Fox
Wayne Mitchell
Sheyna Kapadia
Colin Ponce
Sarah Osborn
Tom Benson

Funding: LDRD ER 24ERD030

https://github.com/llnl/Skywing

References

Fox, Alyson, et. al. 2022. "Algorithmic Development for Unreliable Computing Environments." *ASCR Workshop on Cybersecurity and Privacy for Scientific Computing Ecosystems.*
Erlandson, Luke, et. al. 2023.

"Resilient sACD for Asynchronous Collaborative Solutions of Systems of Linear Equations." 1*8th Conference on Computer Science and Intelligence Systems FedCSIS*

Vogl, Christopher, et. al. 2024. "Modifying the Asynchronous Jacobi Method for Data Corruption Resilience." *SIAM Journal on Scientific Computing (SISC).*