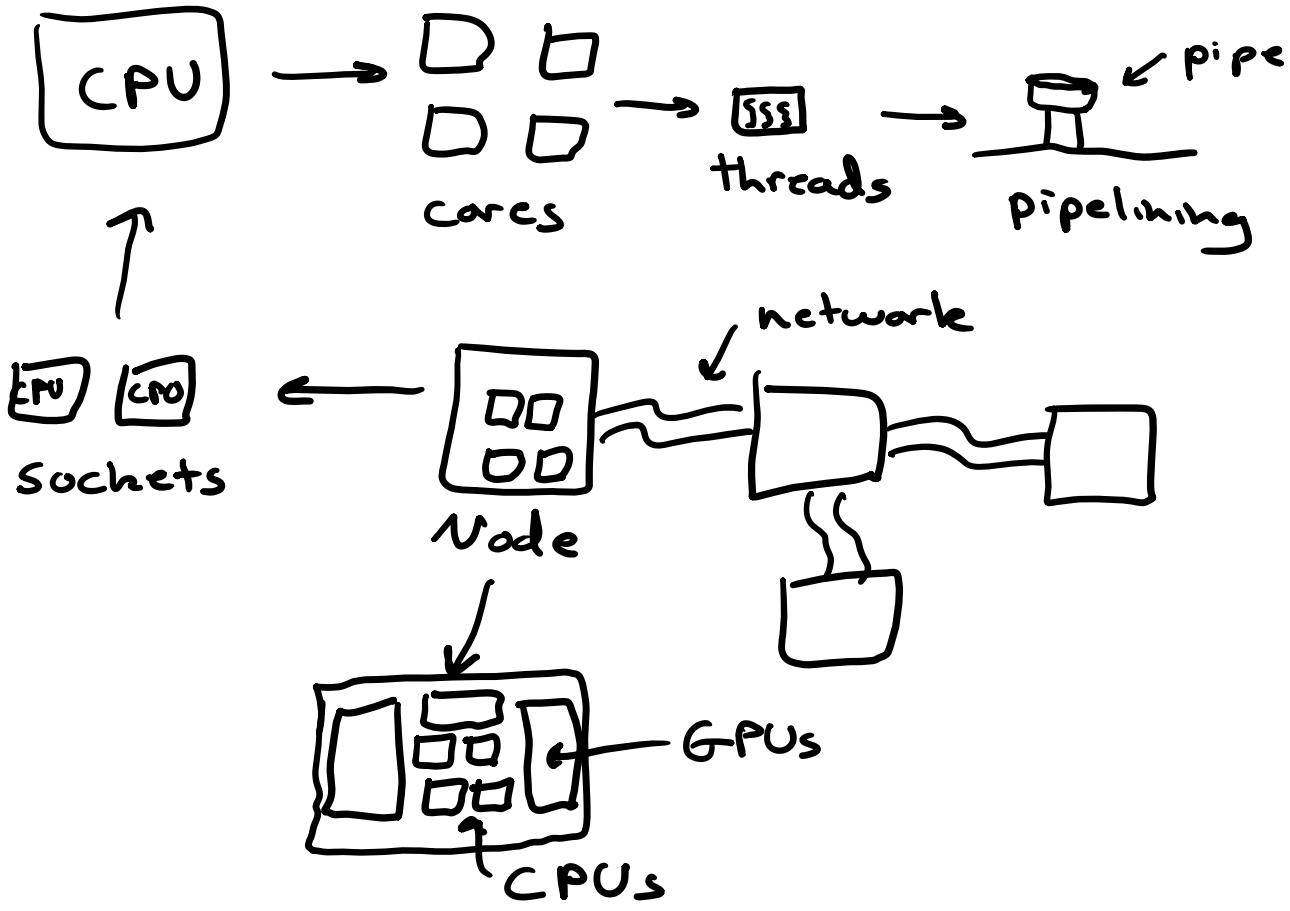
Parallelization:

Takes many forms



Goal: distribute graph computations to this hierarchy

shared-memory → parallel compute elements

distributed-memory → each process

parallel compute elements use the same memory space

each process has a different space → explicit comms. is needed

Recall: vertex-centric computations

```
S[v] = init()

For some # iter                          ← dependencies, so can't
    For some v ∈ V(G)                    ← better, but imbalanced
        For u ∈ N(v)                     ← not a lot of work
            S[v] ← update(S[v], S[u])

    S[v] = finalize()
```

Q: where to parallelize? → For some $v \in V(G)$

Note: as $S[v]$ updates can be processed independently, and there's enough work to benefit, we parallelize over the middle loop

Good: relatively easy to do and works well enough

Bad: skewed degree distributions

↓

think: single thread processing a low vs. high degree vertex

(warp divergence)

if {}	for {}	iter = 5
else {}	while {}	iter = 50,000,000

To address this issue:

* Hierarchical parallelization
  - ↳ based on degree
  - * High degrees get the inner loop processed in parallel
  - * Small degrees get the middle loop parallelized

middle loop parallelized

★ Edge-based parallelization

For some # iter

in parallel

For some $(u,v) \in E(G)$

$S[u] \leftarrow$ update$(S[u], S[v])$

(using thread-safe method)

How: specific data structures

using thread-safe updates

OR: loop collapse

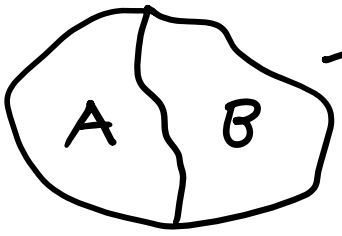
Loop collapse

↳ turn nested loops into one big outer loop

Usually adds some overhead, especially if our workload across iterations

What about distributed memory?

Step 1: partition the graph



G

→ we partition V(G) into some k disjoint sets

each rank updates its "owned" vertices

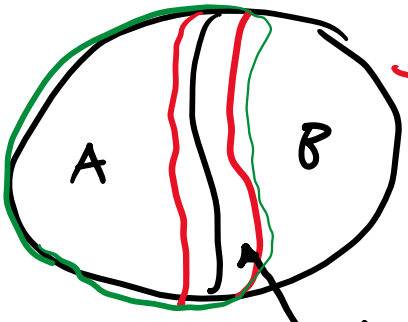
we exchange all necessary updates

↙ our owned subgraph

For $v \in V(G_i)$

For $u \in N(v)$ → Note: u might not be owned by rank i

$S[v] \leftarrow$ update



→ we need to include the one-hop neighborhood

all of A, effectively

ghost vertices for A

A given part/subgraph owned by same rank:

- owned vertex set → states we update
- edges incident on owned vertices
- ghost vertices → states updated via communication
- States for all owned+ghost vertices

↳ we update our owned
we recieve updates for ghosts

Our Distributed vertex-centric graph processing approach

For some rank i:

$G_i \leftarrow \text{localGraph}(G)$

For $v \in V(G_i)$
$S[v] = \text{init}$ ← Note: can usually init ghost states w/o communicating

For some # iter:

For $v \in V(G_i)$

For $v \in V(G_i)$

For $(v, u) \in E(G_i)$

$S[v] = \text{update}(S[v], S[u])$

Note: the core processing is the same

* communicate all updates

(any $v \in V(G_i)$ where v is a ghost on some other G_j)

$S[v]$ is exchange to all $u \in N(v)$ where $u \in V(G_j)$ (owned by)

How to communicate?

→ MPI

Message passing interface

Simplest form of communications

Send(buffer, destination)

Recv(buffer, source)

send(*, 1) ⟷ recv(*, 0)
on rank 0 ↑ on rank 1

generally, each send needs a recv

More complex communications

↳ Reductions

All or some subset of ranks where we 'reduce' some set of values collectively

allReduce(buffer, op=MPI_Sum, ranks)

Reduce: some collective operation over a set of values

i.e., sum, min, max, etc.

Note: generally, all ranks within

a graph computations need to communicate with all other ranks

→ $O(p^2)$ expected messages

↑ p ranks all pairwise communicating

So our communication pattern:

For i = 0...(p-1)

  For j=(i+1)...(p-1)

    communicate pairwise between ranks i,j

Next class: we'll implement distributed BFS

install mpi4py

install mpi binaries (might be included in mpi4py)

install mpi binaries (might be included in mpi4py)

↓

OpenMPI

or

Mpich