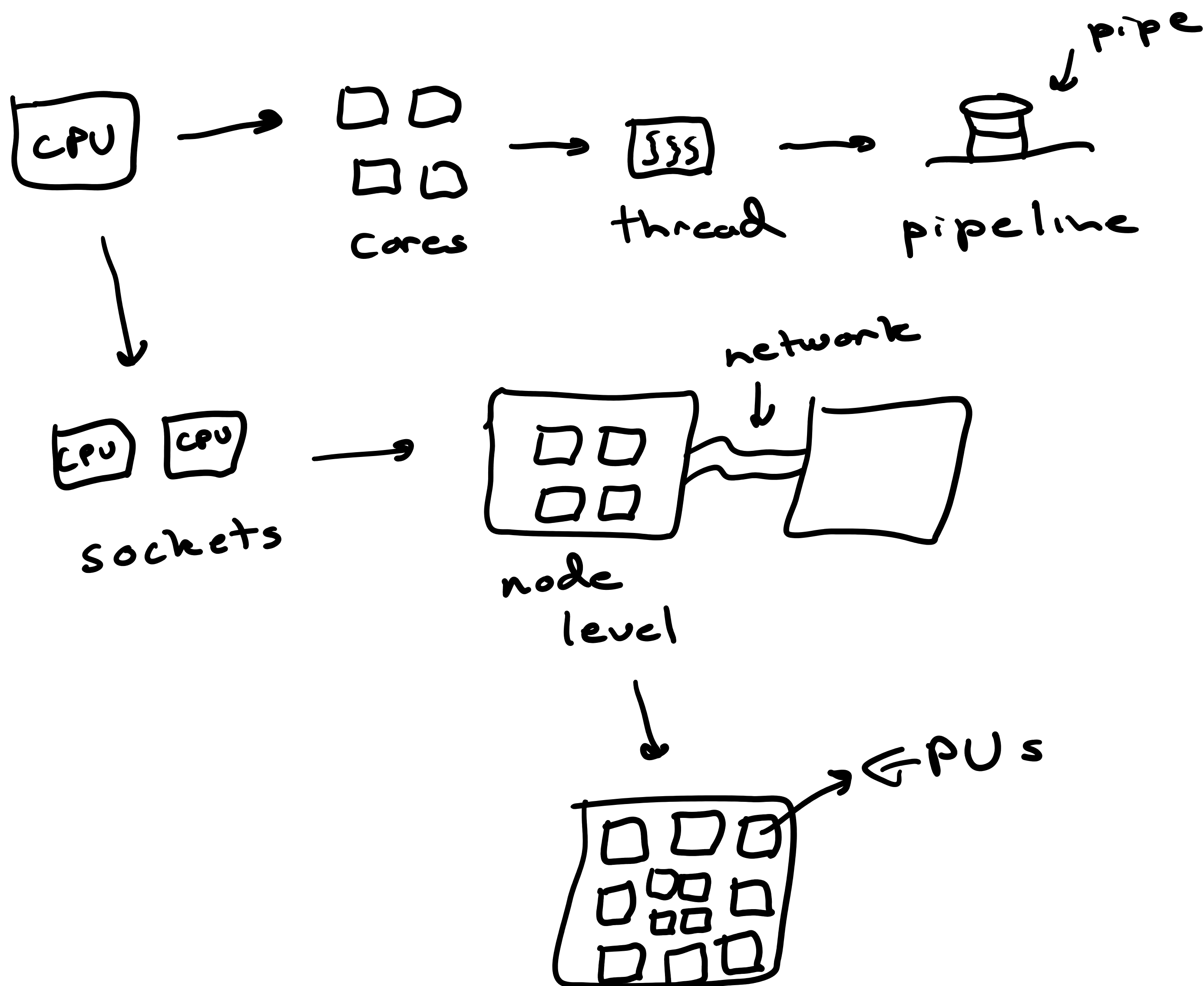
Lecture 20 - Parallel Graph Processing

Thursday, April 11, 2024 8:33 AM

Parallelization

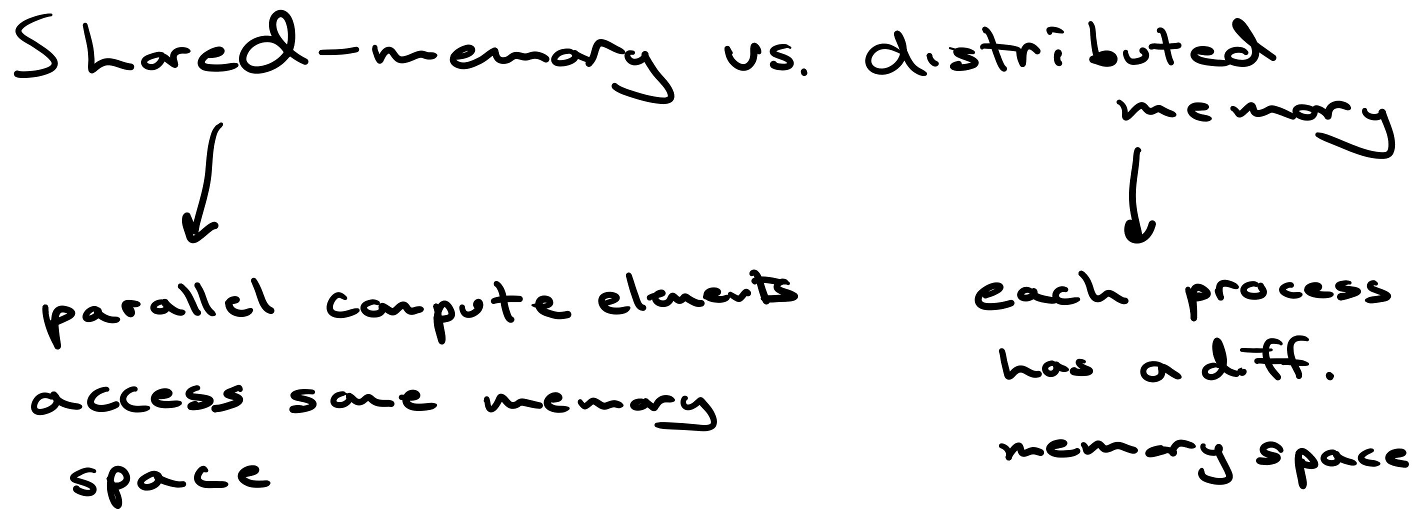
↳ Take many forms

Hierarchical



Goal: distribute graph computation on this hierarchy

on this hierarchy



Q: where do graphs and their computation fit here?

↳ multiple levels of parallelism

We start caring when:

→ Graphs get big

→ Problems get big (computationally)

However → going parallel also adds big challenges

Generally: parallel graph processing

AKA: addressing those challenges

Recall: vertex centric computations

$S[v] \leftarrow$ initialize()

For some # iter ← dependencies so can't

For some $v \in V(G)$ ← little better to parallelize

For $u \in N(v)$ ← generally, not enough work

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

$S[v] \leftarrow$ finalize()

where to include parallelization? (arrows with "?" pointing to the outer and middle loops)

Note: as $S[v]$ updates can be processed independently for each v (some nuance)

↳ we generally parallelize over the middle loop

Good: relatively easy to do

Good: relatively easy to do
can get good speedup
"easy" to ensure correctness

Bad: skewed degree distributions
↳ especially bad on GPUs
↓
a single thread/process updating a large degree vertex

(warp divergence)

if {} for {}
else {} while {}

To address these issues:

Hierarchical parallelization
↳ based on degree

* high degree vertices get inner loop parallelized

inner loop parallelize

* other vertices just parallelize the middle loop

Edge-based parallelization

(different graph structure)

For some # iter

For some $e \in E(G)$ ← parallel

$= (u,v)$

$S[u]$ ← update()

$S[v]$ ←

Loop collapse

↳ turn nested loops into just one loop

(add overhead but don't modify our graph)

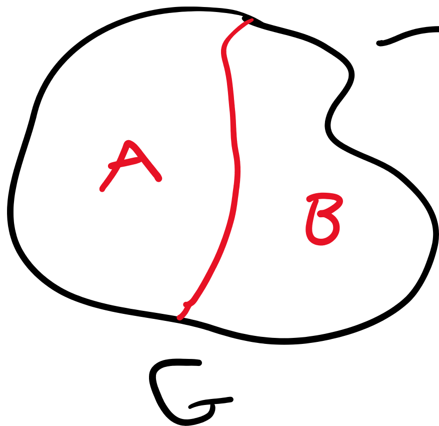
Most of the above are more relevant to shared-memory

Q: What about D-M?

distributed memory

Q What about D...

distributed memory

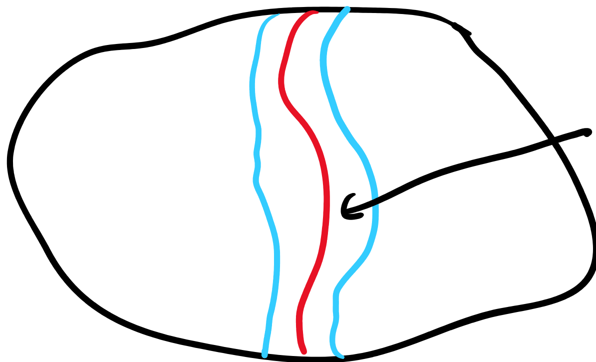


→ we partition $V(G)$ into disjoint sets

We compute on a subgraph induced on a set

We exchange state information via messages

For $v \in V(G)$
 For $u \in N(v)$ → not all u are on the same process as v
  $S[v]$ = update



we also need to include the 1-hop neighborhood in our partitioning (ghost vertices)

A given part/subgraph

(ghost vertices)

A given part/subgraph
- local vertex set
- edges incident on local vertices
- ghost vertices, endpoints on an edge but not in the local set
- State for all local+ghost vertices
  - ↳ we communicate state updates for boundary vertices
    - ↳ vertices adjacent to ghost

Our distributed vertex-centric approach

G_i ← local_graph(G)

i = rank or process ID

For $v \in V(G_i)$
    $S[v]$ = init()

while not done
    For v in local vertices
        For u in $N(v)$

$S[v] = update(S[v], S[u])$

Communicate all boundary $S[v]$

Reduce some stopping criteria

For v in $V(G_i)$

$S[v] = finalize()$

How to communicate

↳ MPI

(message passage interface)

<u>Simplest form</u>

Send (buffer, destination_process)

Recv (buffer, source_process)

Send ⟷ Recv

need to match

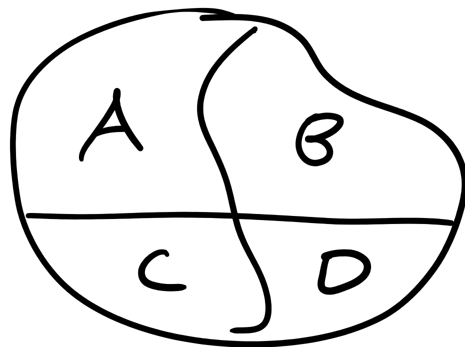
More complex communication patterns are built on the above

We also have reductions

All (or some) ranks 'reduce' some set of values collectively

↳ allreduce(queue_length, op=MPI.SUM)

→ all ranks sum together all value of each queue_lengths



Note: all ranks generally need to communicate to all other ranks

Next class: we'll be discussing and implementing BFS for HW4

install mpi4py

install MPI binaries

↓

OpenMPI

or

Mpich