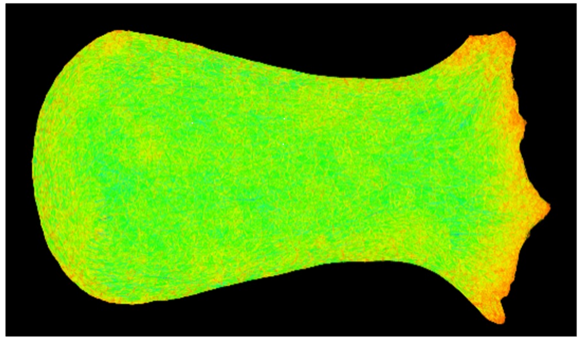
Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism

Quan M. Nguyen and Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

HPCA 2023
Wednesday, 1 March 2023

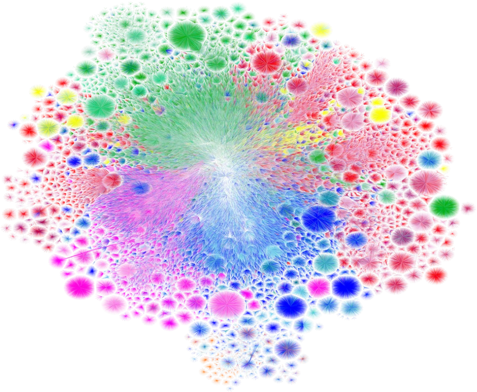
Irregular applications difficult to accelerate

- Data-dependent memory accesses and control flow
- Recent hardware support for irregular application pipelines
- **Problem: many ways to map pipelines**



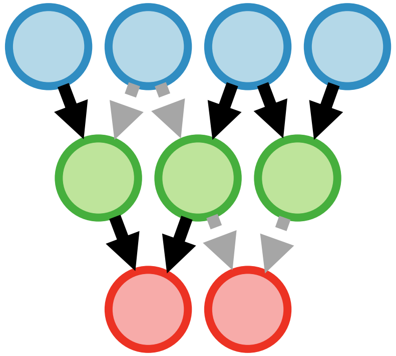
Dynamic simulation

https://sparse-files-images.engr.tamu.edu/Um/2cubes_sphere_graph.gif



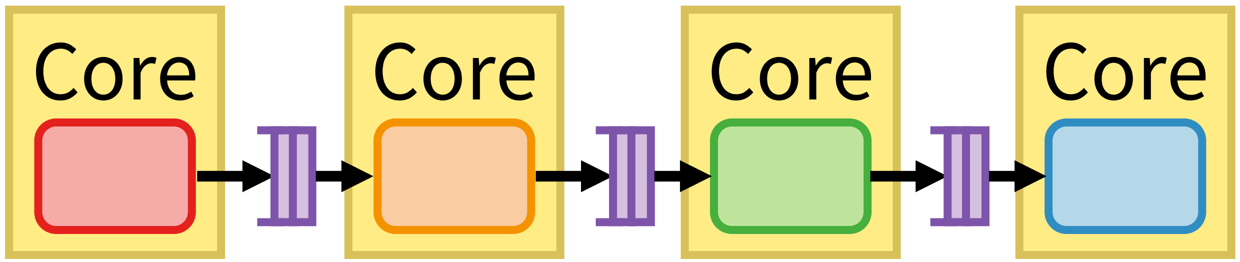
Graph processing

By Barrett Lyon / The Opte Project
Visualization of the routing paths of the Internet.

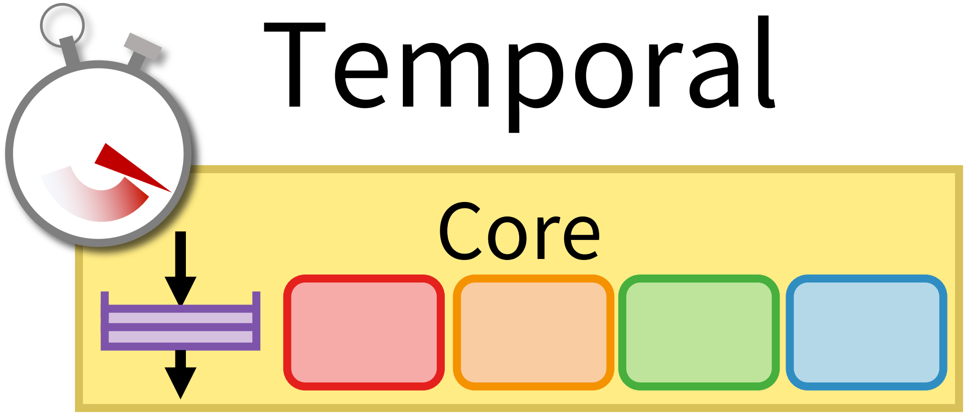


Sparse deep learning

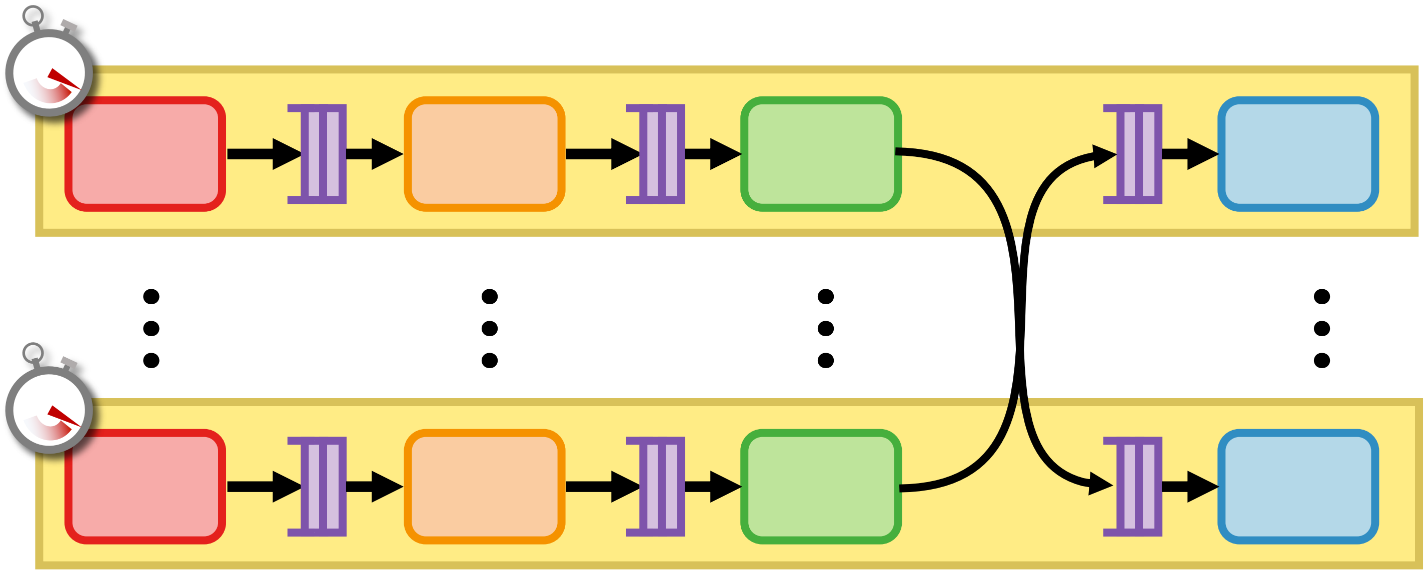
Spatial



Temporal

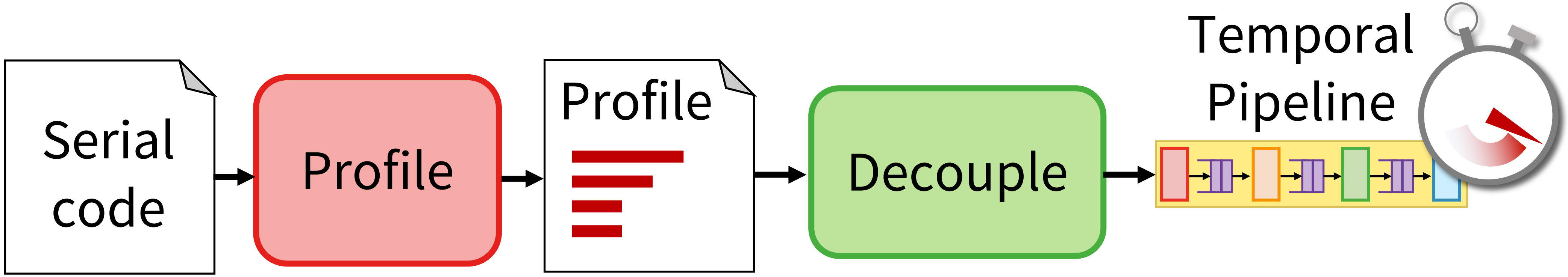


Both!



New tradeoffs call for a new compiler: Phloem

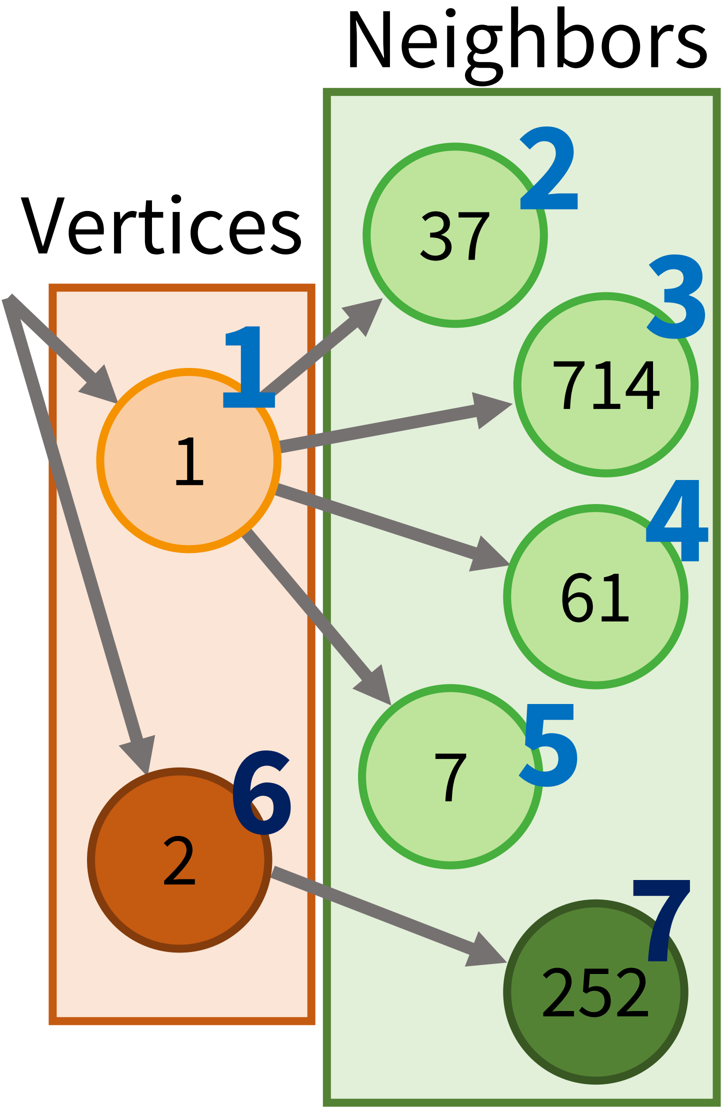
- Temporal pipelines have fundamentally different tradeoffs
- Phloem systematically creates efficient temporal pipelines
- Apply as static transformation or use profile-guided optimization
- **Phloem improves performance by 1.7x gmean, 80% of manually tuned code**



Agenda

Intro → **Background** → Phloem → Evaluation

The perils of irregularity

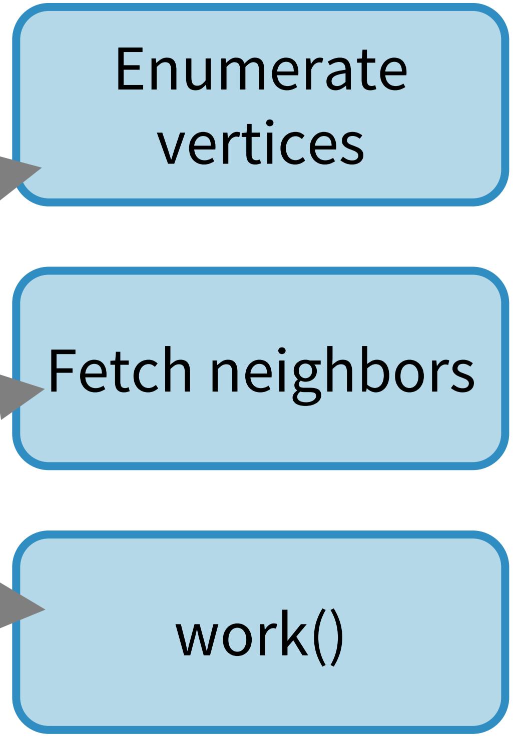


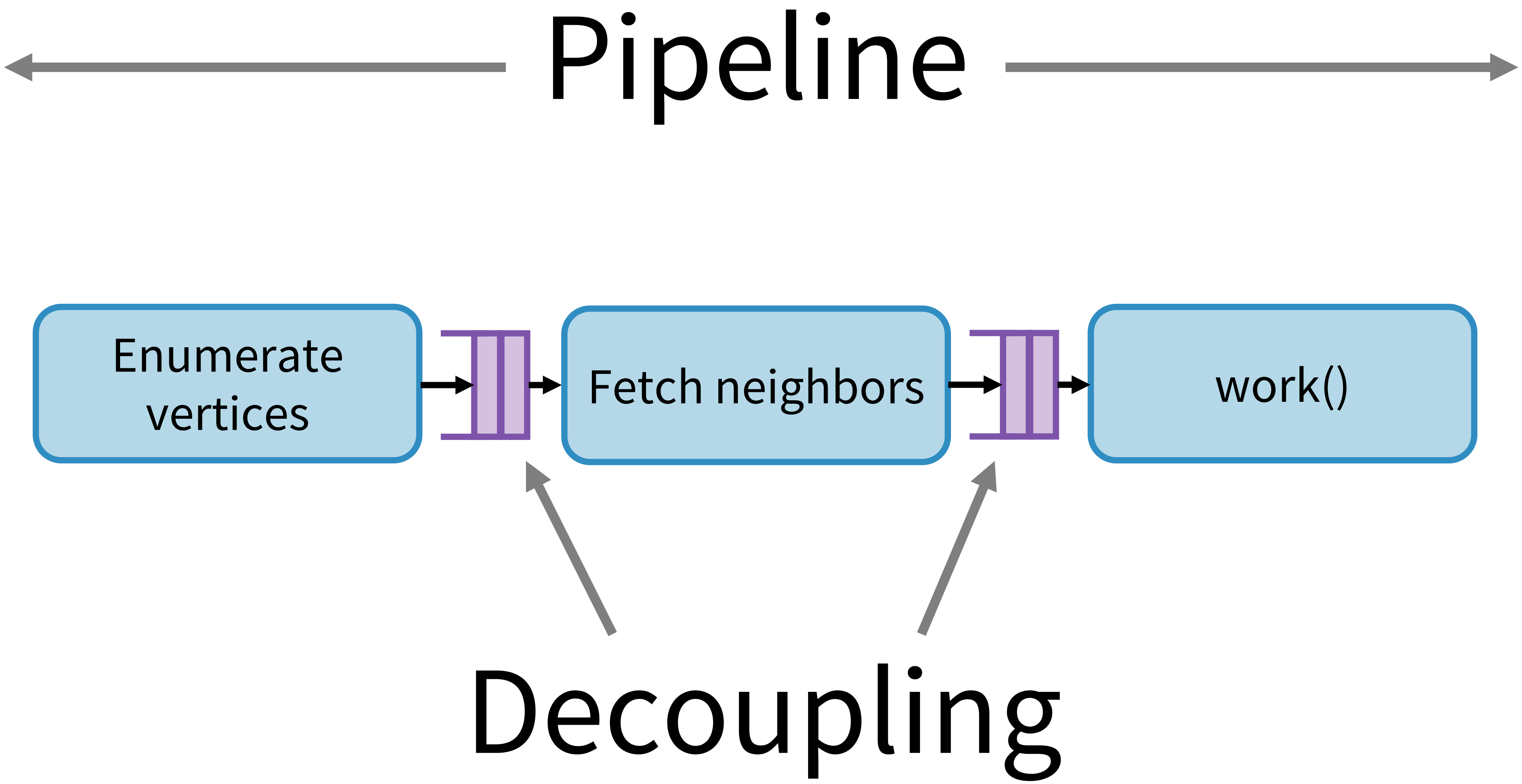

```
for vtx in vertices:
    for ngh in neighbors[vtx]:
        work(vtx, ngh)
```

data-dependent access, control

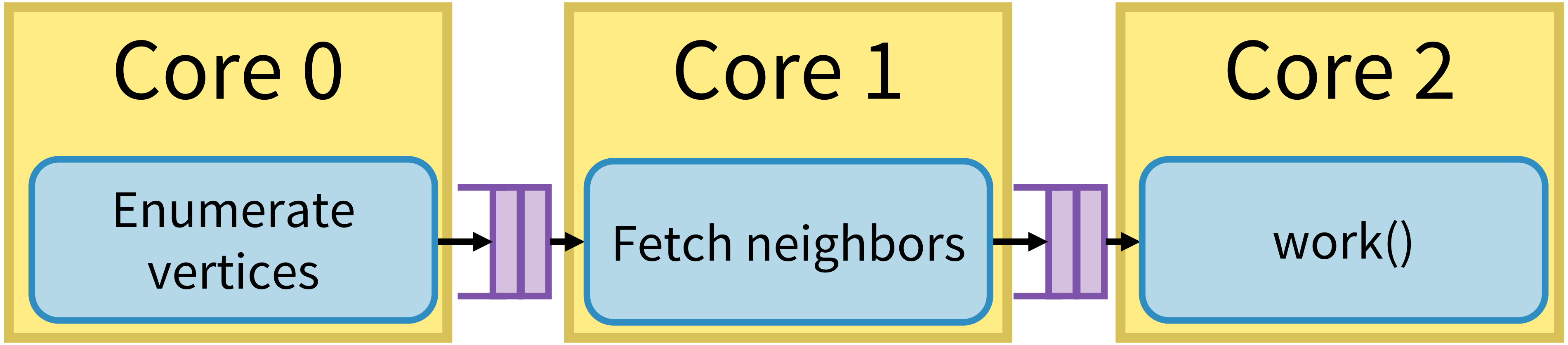
As a pipeline (in separate cores)

```
for vtx in vertices:
    for ngh in neighbors[vtx]:
        work(vtx, ngh)
```



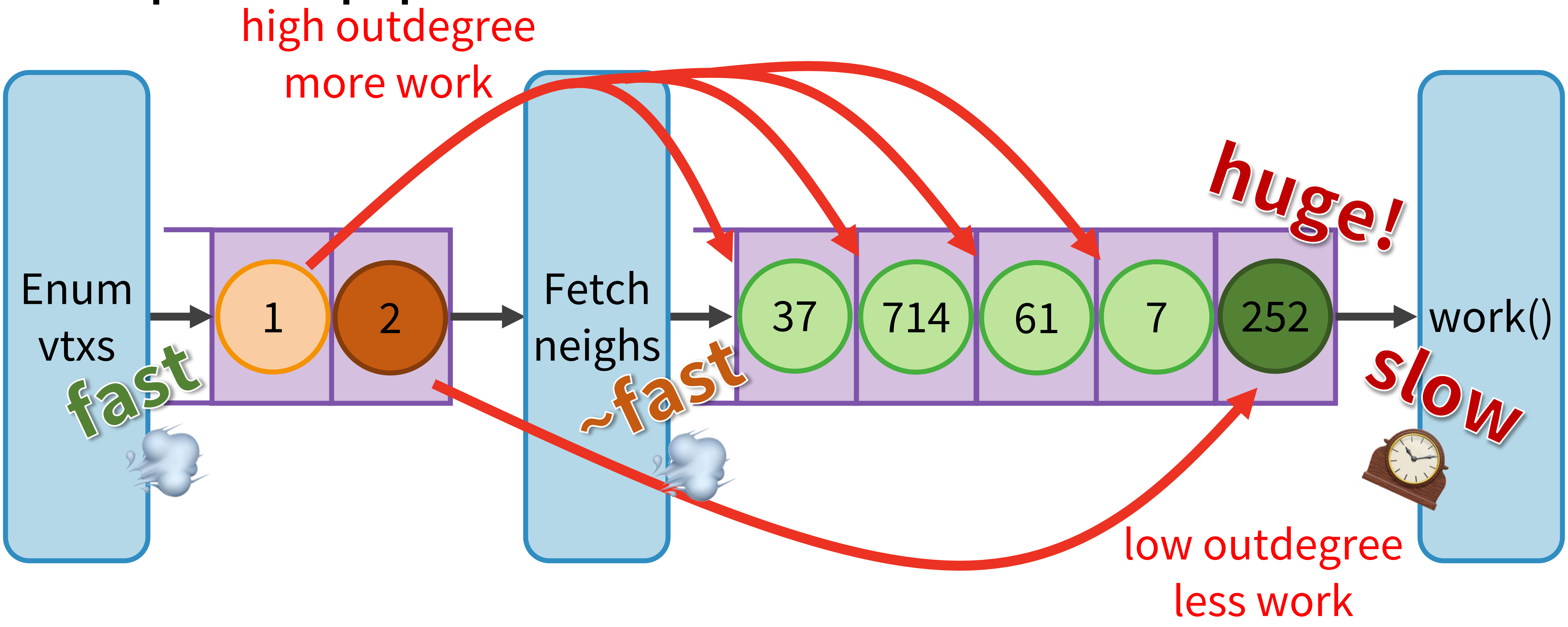

Pipeline
Enumerate vertices
Fetch neighbors
work()
Decoupling

Decoupling is not new, but...



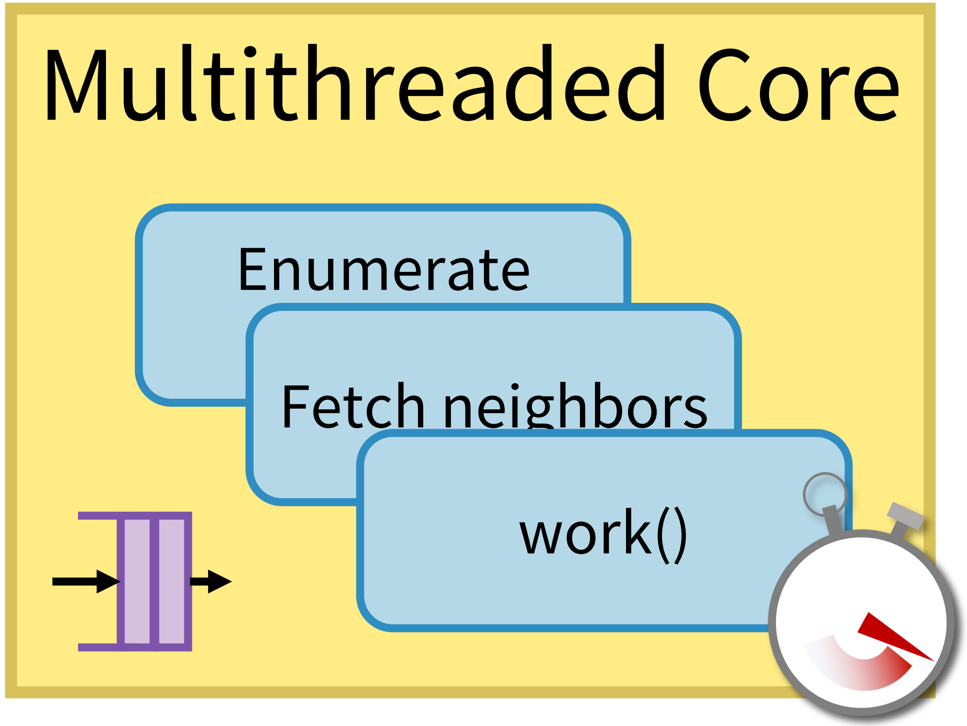
(DAE [ISCA'82] , PIPE [ISCA'85], ZS-1 [ASPLOS'87], ACRI-1 [HPCN'95], MT-DCAE [PACT'01], Raw [MICRO'02], Merrimac [SC'03], DSWP [PACT'04,MICRO'05,CGO'10], Outrider [ISCA'11], MPPA [HPEC'13], HELIX [CGO'12,ISCA'14], DeSC [MICRO'15], ...)

Spatial pipelines cause load imbalance



Dynamic temporal pipelines effectively handle irregularity

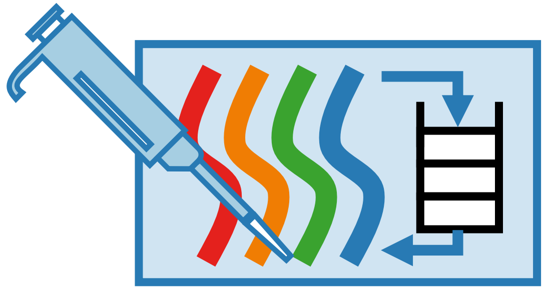
- Time-division multiplex many stages onto same core or processing element
  - General-purpose core: compute, loads/stores, control flow
  - Decoupled communication between stages on different cores or same cores
  - Core dynamically selects stages to execute
- **Tradeoffs for efficient stages fundamentally changes**



Making efficient dynamic *temporal* pipelines

- Decouple all long-latency events
  - **spatial: decoupling limited by load imbalance**
- Use dynamic temporal features of hardware to load balance
  - **spatial: requires known communication rates**
    (StreamIt [CC'02])
- Result: many small stages
  - **spatial: few large stages**
    (DSWP [PACT'04,MICRO'05], PS-DSWP [CGO'10], ...)
- **Phloem makes these first-class considerations**

A representative architecture: Pipette [MICRO'20]



- Implements irregular applications as multithreaded programs
- Each stage runs on a thread of a multithreaded core
- Architectural support for cheap, fast inter-thread communication
  - enq(queue, value)
  - deq(queue)
- Reuse simultaneous multithreading to achieve load balance
- Reference accelerators further decouple memory accesses
- Change control flow through special values

Pipelines built manually

```
def bfs(src):
  …
  for v in current frontier:
    start, end = offsets[v], offsets[v+1]
    for ngh in neighbors[start:end]:
      dist = distances[ngh]
      if dist is not set:
        set distance; add to next frontier
  …
```

- Process current frontier
- Enumerate neighbors
- Visit neighbors
- Update data, next frontier

Pipelines built manually

```
void bfs(Graph* g, int* cur_frontier, int* next_frontier,
         int root, int* distances) {
    int cur_frontier_idx = 0, next_frontier_idx = 0;
    int cur_dist = 0;
    // Add root to frontier
    cur_frontier[cur_frontier_idx++] = root;
    distances[root] = 0;
    while (cur_frontier_idx != 0) {
        cur_dist++;
        // Process current frontier
        for (int i = 0; i < cur_frontier_idx; i++) {
            int v = cur_frontier[i];
            // Enumerate neighbors
            int edge_start = g->nodes[v];
            int edge_end = g->nodes[v+1];
            for (int e = edge_start; e < edge_end; e++) {
                // Visit neighbor
                int ngh = g->edges[e];
                // If dist decreases, update it,
                // add ngh to next frontier
                int old_dist = distances[ngh];
                if (cur_dist < old_dist) {
                    distances[ngh] = cur_dist;
                    next_frontier[next_frontier_idx++] = ngh;
                }
            }
        }
        swap(&cur_frontier, &next_frontier);
        cur_frontier_idx = next_frontier_idx;
        next_frontier_idx = 0;
    }
}
```

```
void bfs_stage1(Graph* g, int* cur_frontier, int*
next_frontier,
    int root, int* distances) {
  int cur_frontier_idx = 0;
  int cur_dist = 0;
  // Add root to frontier
  cur_frontier[cur_frontier_idx++] = root;
  distances[root] = 0;
  while (cur_frontier_idx != 0) {
    cur_dist++;
    // Process current frontier
    for (int i = 0; i < cur_frontier_idx; i++) {
      int v = cur_frontier[i];
      enq(1, v);
      enq(1, v+1);
    }
    enq_ctrl(1, NEXT);
    swap(&cur_frontier, &next_frontier);
    cur_frontier_idx = deq(5);
  }
  enq_ctrl(1, LAST);
}
```

```
void bfs_stage2(Graph* g, int* cur_frontier, int*
next_frontier,
    int root, int* distances) {
  setup_reference_accelerator(1, INDIRECT, g->nodes);
  setup_control_value_handler(1, &&q1_handle_ctrl);
  while (true) {
    while (true) {
      // Enumerate neighbors
      int edge_start = deq(1);
      int edge_end = deq(1);
      for (int e = edge_start; e < edge_end; e++) {
        enq(2, e);
      }
    }
```

```
void bfs_stage3(Graph* g, int* cur_fr
next_frontier,
    int root, int* distances) {
  setup_reference_accelerator(2, INDI
  setup_control_value_handler(2, &&q2
  while (true) {
    while (true) {
      // Visit neighbor
      int ngh = deq(2);
      enq(3, ngh);
      enq(4, ngh);
    }
q2_handle_ctrl:
    if (deq(2) == LAST) {
      enq_ctrl(3, LAST);
      break;
    }
    enq_ctrl(3, NEXT);
  }
}
```

```
void bfs_stage4(Graph* g, int* cur_fr
next_frontier,
    int root, int* distances) {
  int next_frontier_idx = 0;
  int cur_dist = 0;
  setup_reference_accelerator(4, INDI
  setup_control_value_handler(3, &&q3
  while (true) {
    cur_dist++;
    while (true) {
      int ngh = deq(3);
      // If dist decreases, update i
      // add ngh to next frontier
      int old_dist = deq(4);
      if (cur_dist < old_dist) {
        distances[ngh] = cur_dist;
```

Agenda

Intro → Background → Phloem → Evaluation

```
for ngh in g->nodes[edge_start:edge_end]:
```

```
...
for (i = 0; i < cur_frontier_idx; i++) {
  v = cur_frontier[i];
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    ngh = g->edges[e];
    ...
```

Decouple | Add queues | Recompute | Accelerate accesses | Use control values | Use control handlers

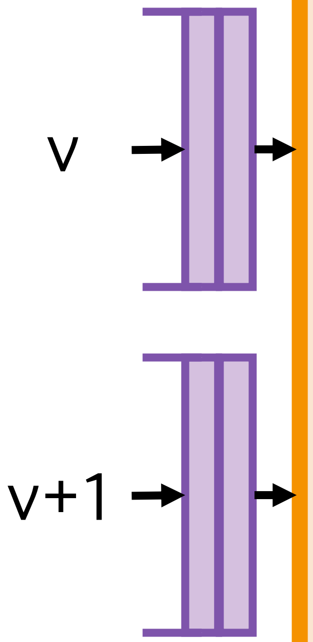
```
...
for (i = 0; i < cur_frontier_idx; i++) {
  v = cur_frontier[i];
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    ngh = g->edges[e];
    ...
```

```
...
for (i = 0; i < cur_frontier_idx; i++) {

  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {

    ...
```

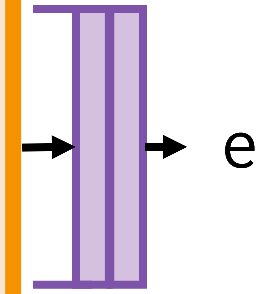
Decouple | Add queues | Recompute | Accelerate accesses | Use control values | Use control handlers

```
...
for (i = 0; i < cur_frontier_idx; i++) {

  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {

    ...
```

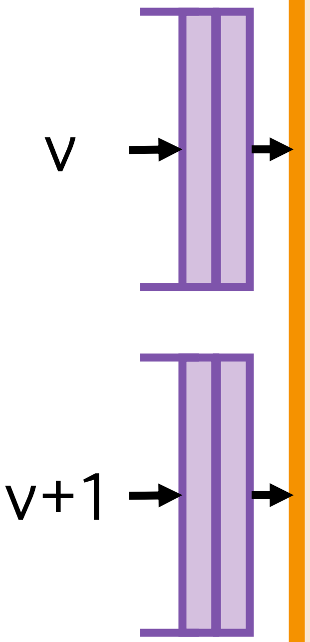
Decouple → Add queues → Recompute → Accelerate accesses → Use control values → Use control handlers



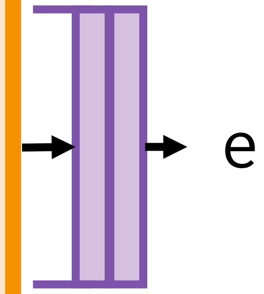

```
...
for (i = 0; i < cur_frontier_idx; i++) {
  v = deq(); v_plus_1 = deq();
  edge_start = g->nodes[v];
  edge_end = g->nodes[v_plus_1];
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
    ...
```



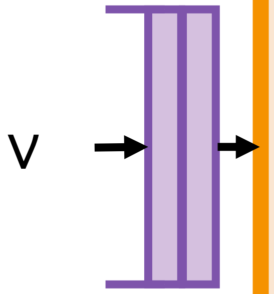
Decouple → Add queues → Recompute → Accelerate accesses → Use control values → Use control handlers



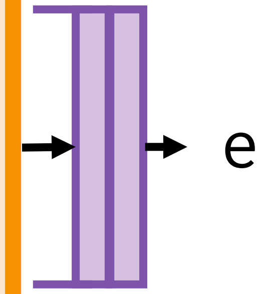

```
...
for (i = 0; i < cur_frontier_idx; i++) {
  v = deq(); v_plus_1 = deq();
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
    ...
```



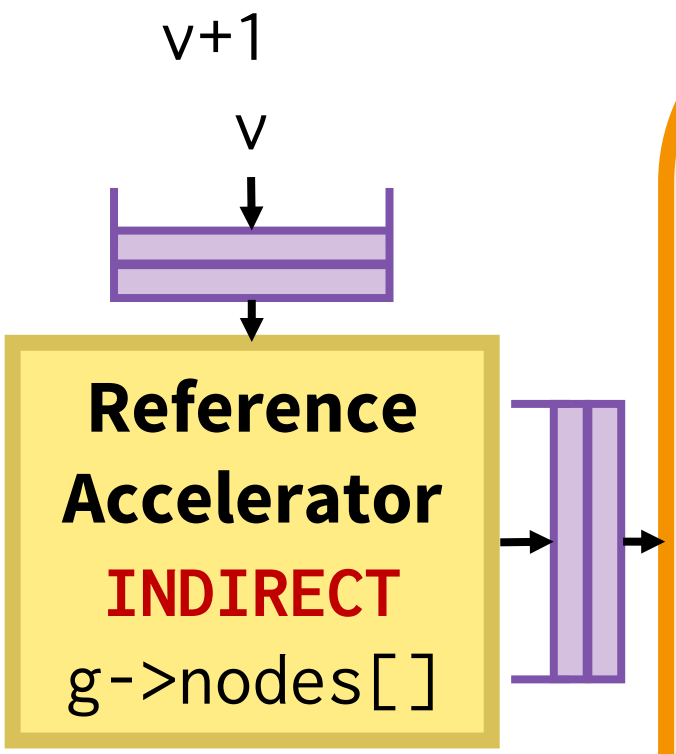
Decouple → Add queues → Recompute → Accelerate accesses → Use control values → Use control handlers




```
...
for (i = 0; i < cur_frontier_idx; i++) {
  v = deq();
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
    ...
```

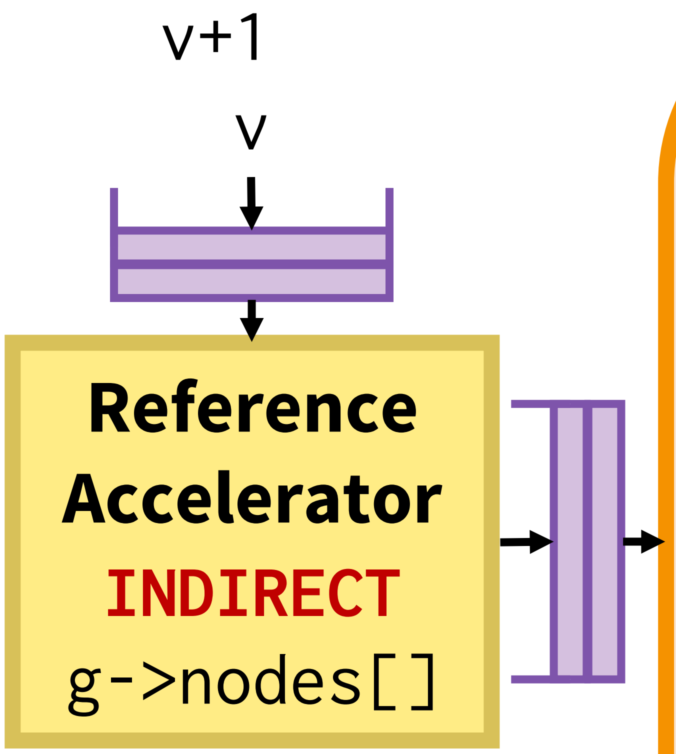


Decouple → Add queues → Recompute → Accelerate accesses → Use control values → Use control handlers



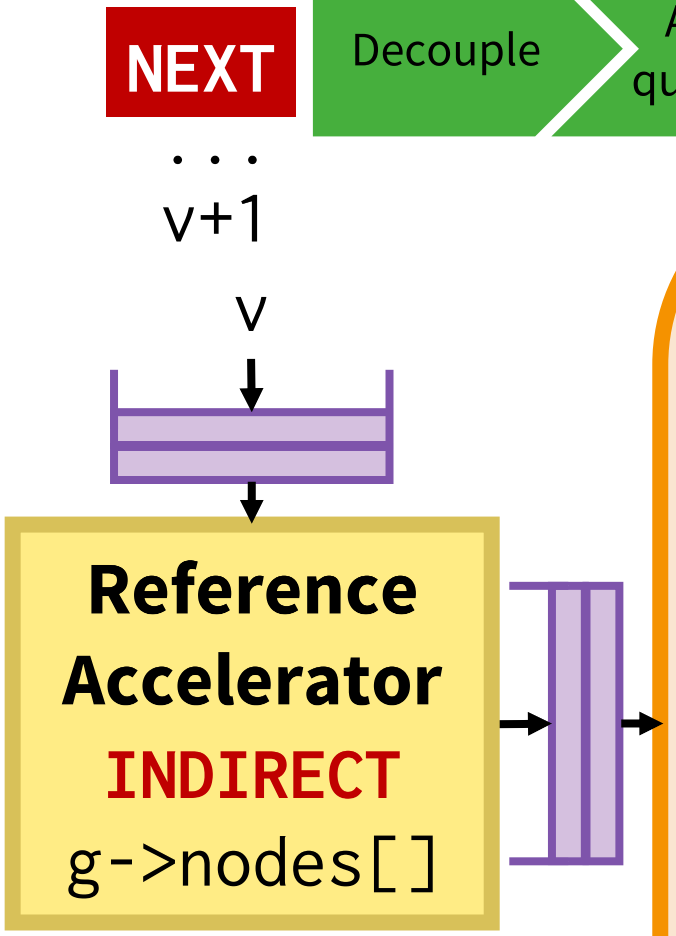

```
setup_reference_accelerator();
for (i = 0; i < cur_frontier_idx; i++) {
  v = deq();
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
    ...
```

Decouple | Add queues | Recompute | **Accelerate accesses** | Use control values | Use control handlers



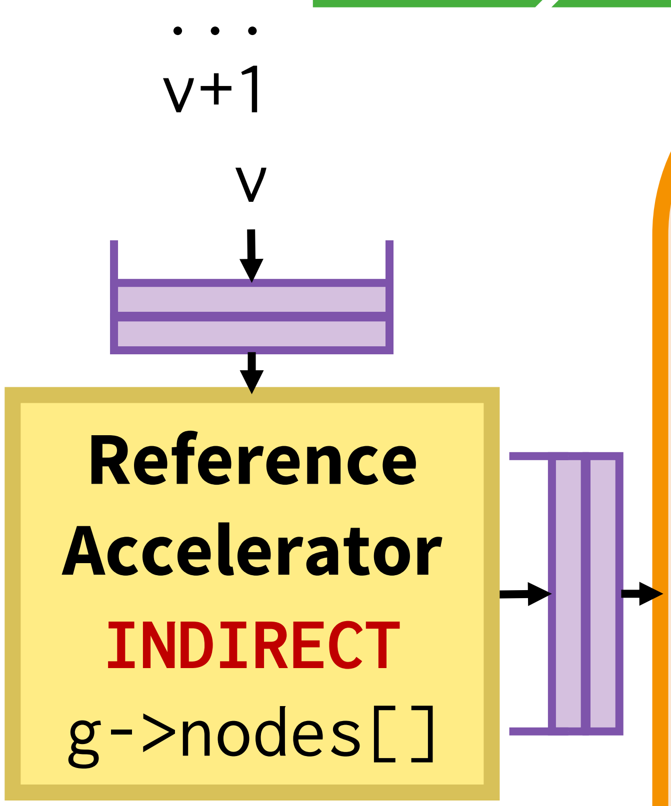

```
setup_reference_accelerator();
for (i = 0; i < cur_frontier_idx; i++) {


  edge_start = deq();
  edge_end = deq();
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
    ...
```

Decouple → Add queues → Recompute → Accelerate accesses → Use control values → Use control handlers




```
setup_reference_accelerator();
while (true) {                    {


  edge_start = deq();
  edge_end = deq();
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
} } enq_ctrl(NEXT);
```

NEXT | Decouple | Add queues | Recompute | Accelerate accesses | Use control values | Use control handlers



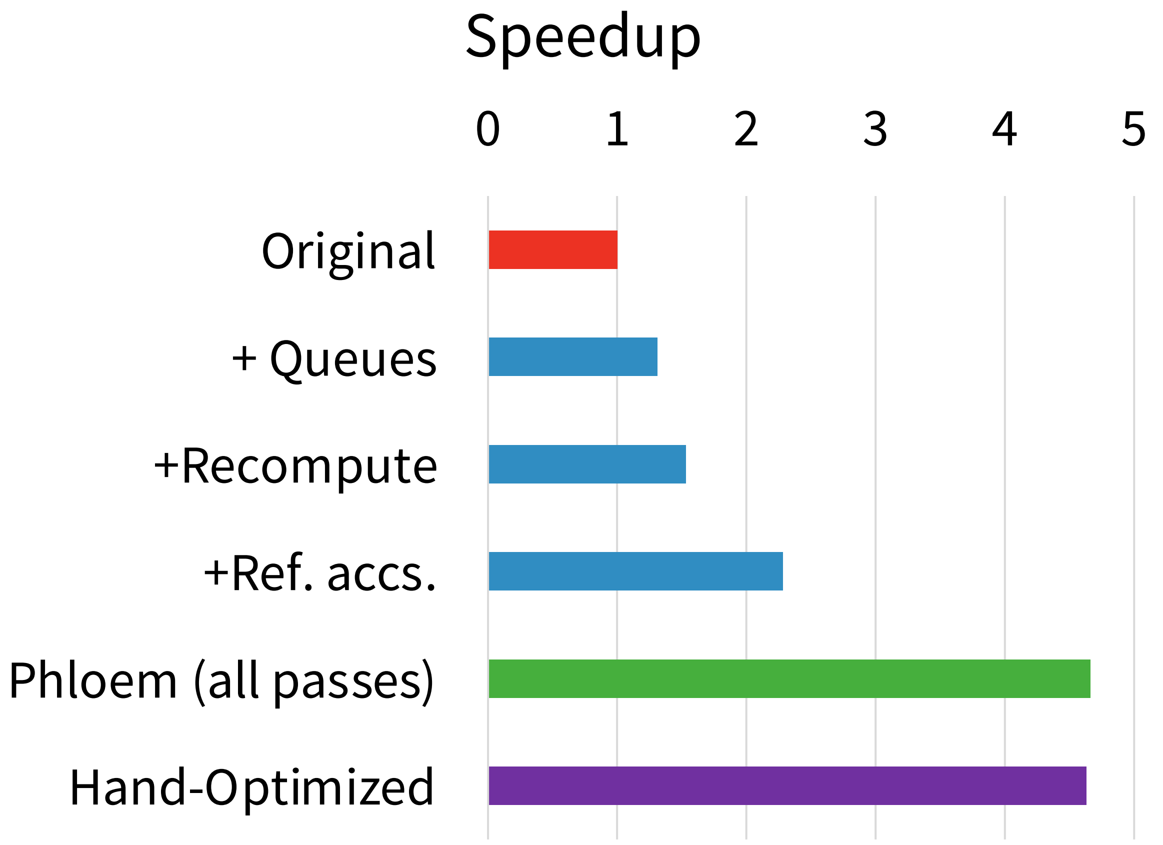

```
setup_reference_accelerator();
setup_control_value_handler(&&handle_ctrl);
while (true) {
  edge_start = deq();
  edge_end = deq();
  for (e = edge_start; e < edge_end; e++) {
    enq(e);
  }
}
handle_ctrl:
deq();
enq_ctrl(NEXT);
```

Review of Phloem transformations

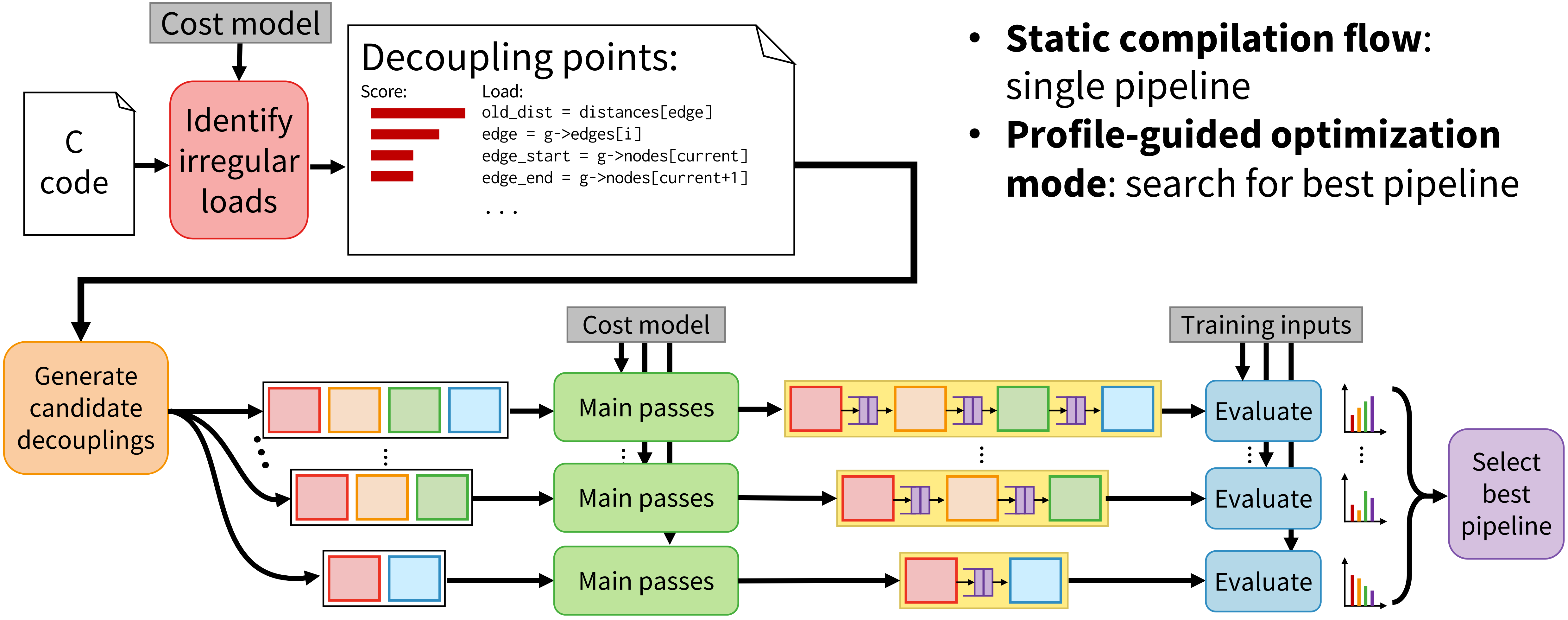
1. Add queues
2. Recompute values
3. Accelerate memory accesses
4. Use control values
5. Use control handlers
6. Inter-stage dead code elimination

(Handling races and aliasing)

(Reducing unnecessary communication)



Phloem pipeline pipeline



- **Static compilation flow**: single pipeline
- **Profile-guided optimization mode**: search for best pipeline

See paper for more

- Handling race conditions and aliasing
- Automatically offloading chains of memory accesses
- Static and profile-guided cost models
- Energy results and breakdowns
- Sweeps on pipeline length

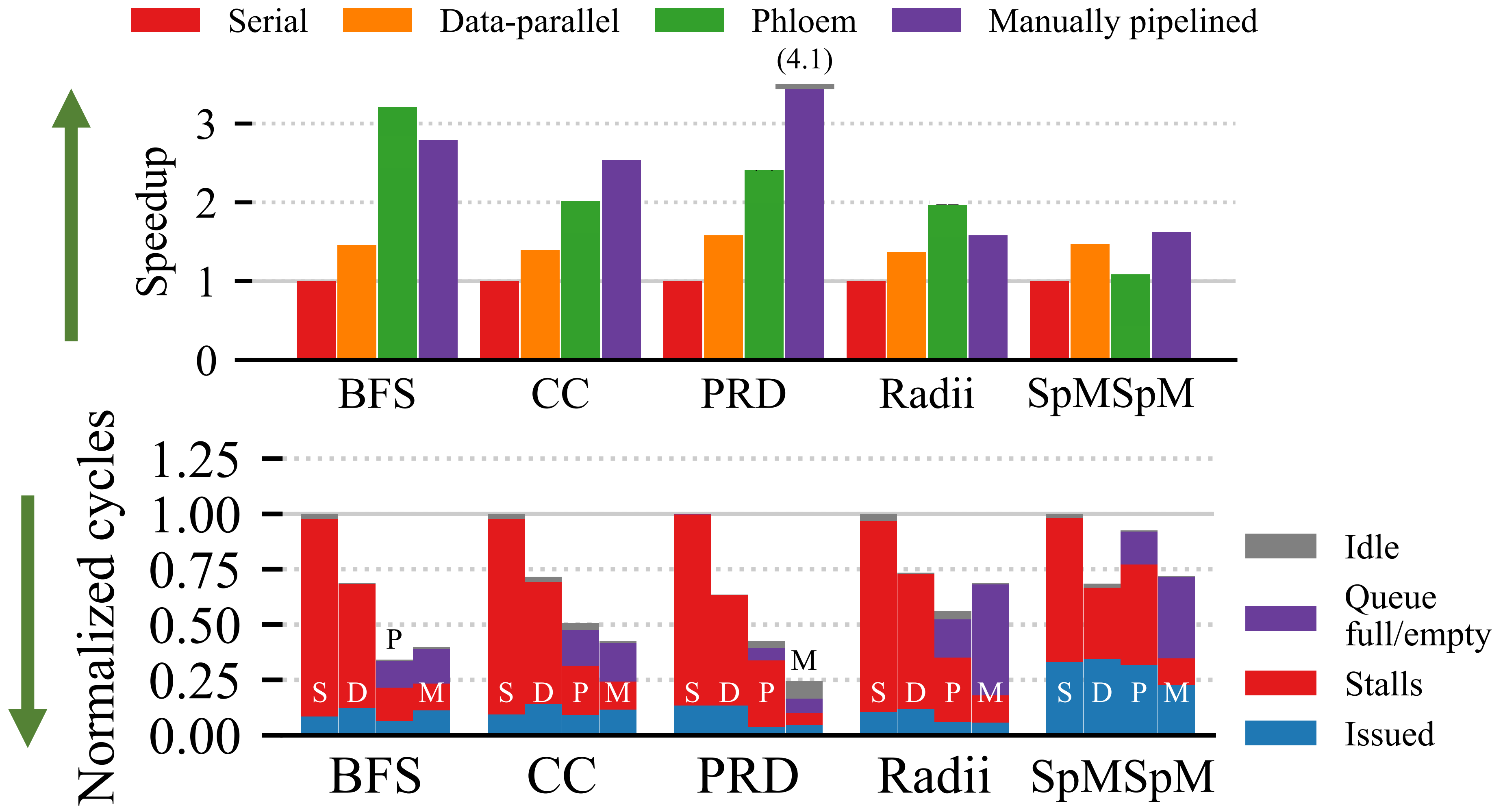
Agenda

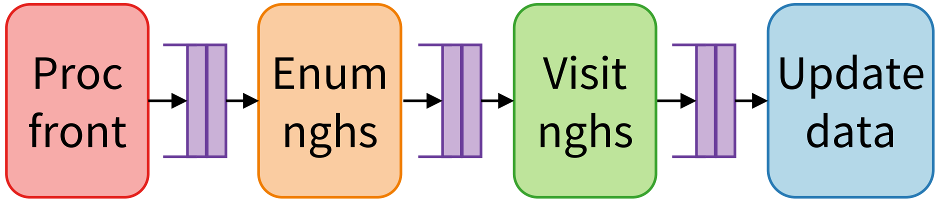
Intro → Background → Phloem → Evaluation

Methodology

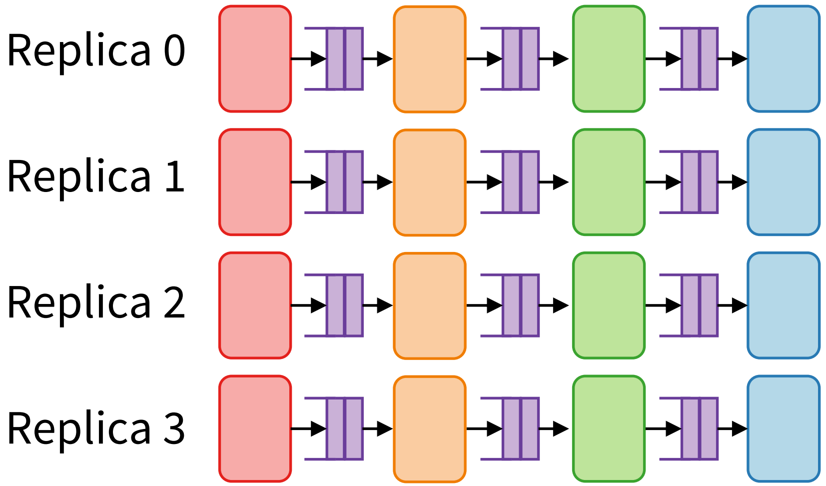
- Event-driven cycle simulation based on ZSim
- 4-way simultaneous multithreaded OOO core with 6-wide issue (similar to Intel Skylake)
- Comparison systems:
  - Baseline: serial OOO core
  - Data-parallel 4-way multithreaded
  - Manually pipelined version
- Applications evaluated: BFS, Connected Components, PageRank-Delta, Radii estimation, SpMSpM

Performance similar to hand-tuned code



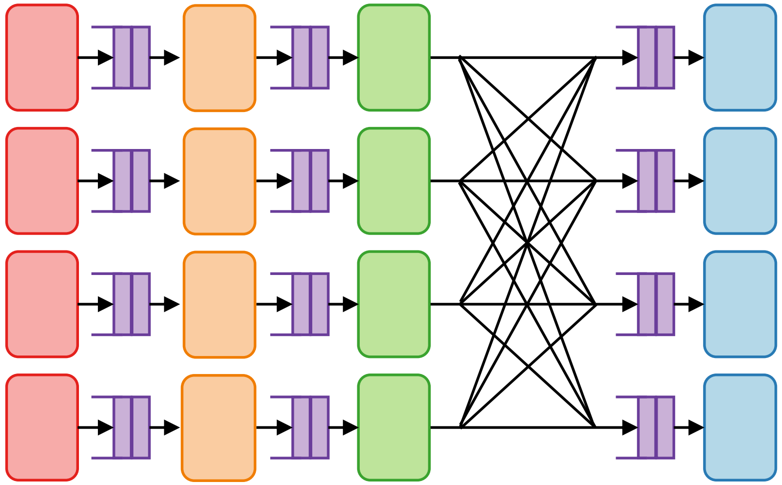



```
#pragma phloem
```



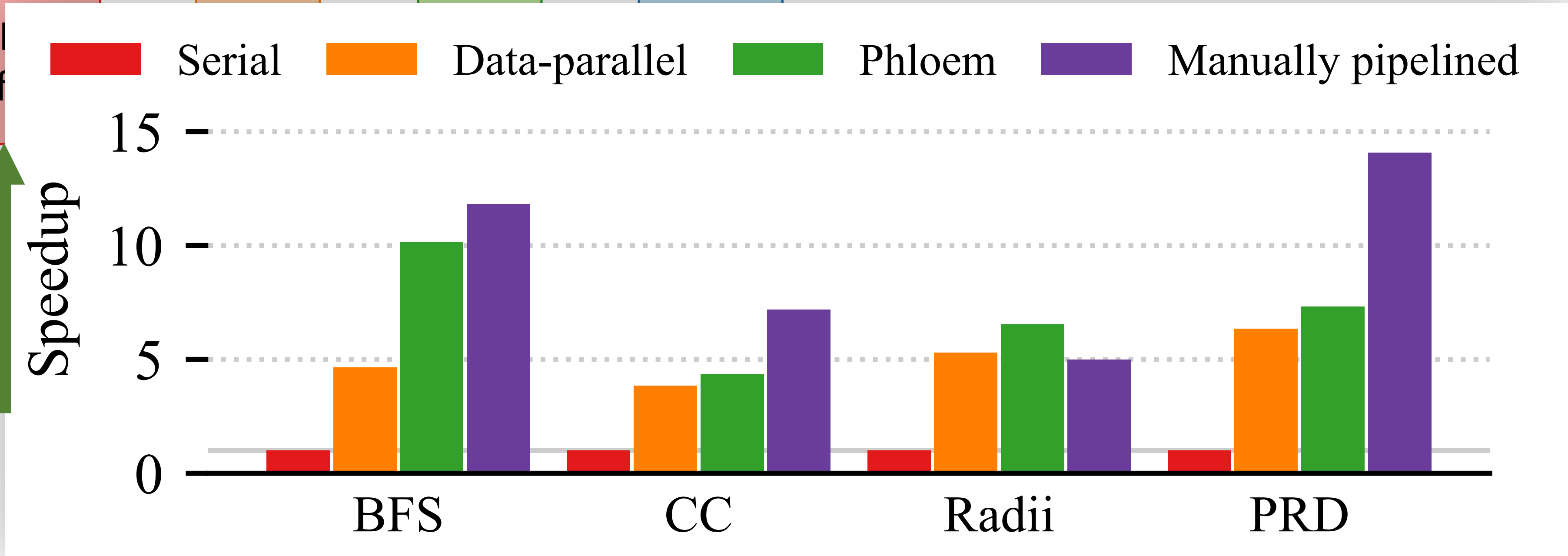

```
// set up replica 0 frontier
// set up replica 1 frontier
  ...
```

```
#pragma phloem replicate
```

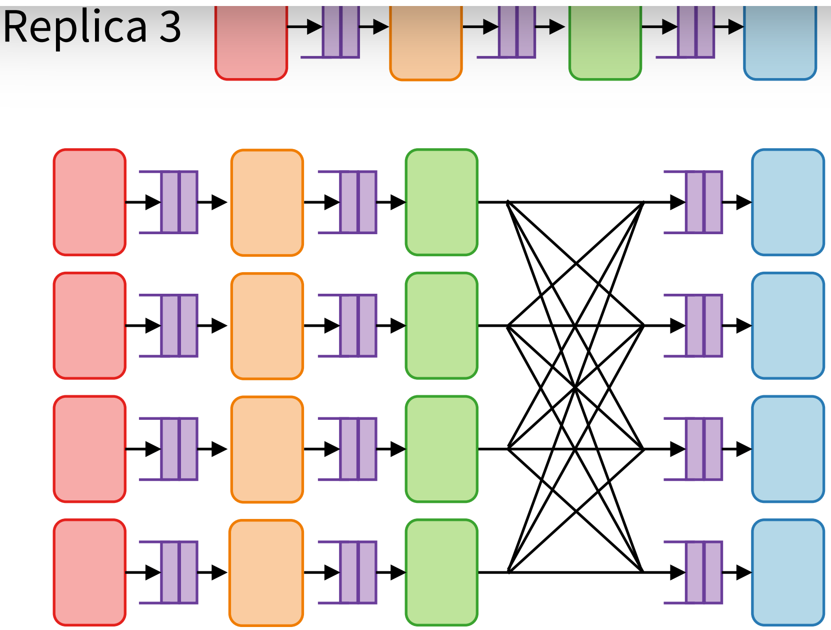


```
if (ngh & 0x3 == 0)
  // send to replica 0
else if (ngh & 0x3 == 1)
  // send to replica 1
  ...
```

```
...
int ngh = g->edges[e];
#pragma phloem distribute
int old_dist = distances[ngh];
...
```



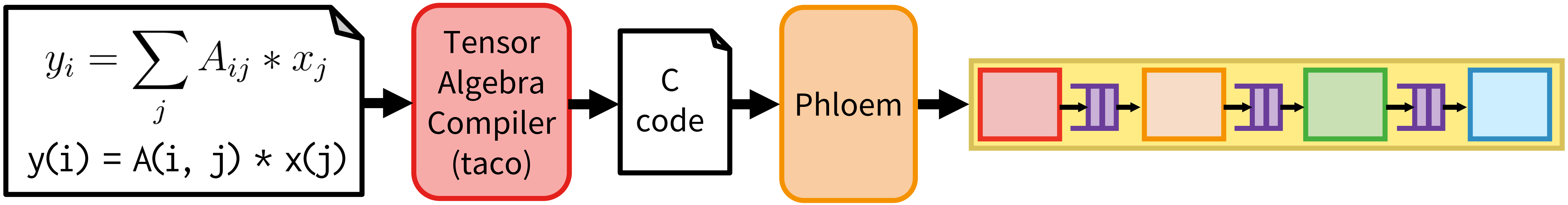

```
// set up replica 0 frontier
// set up replica 1 frontier
 ...
```

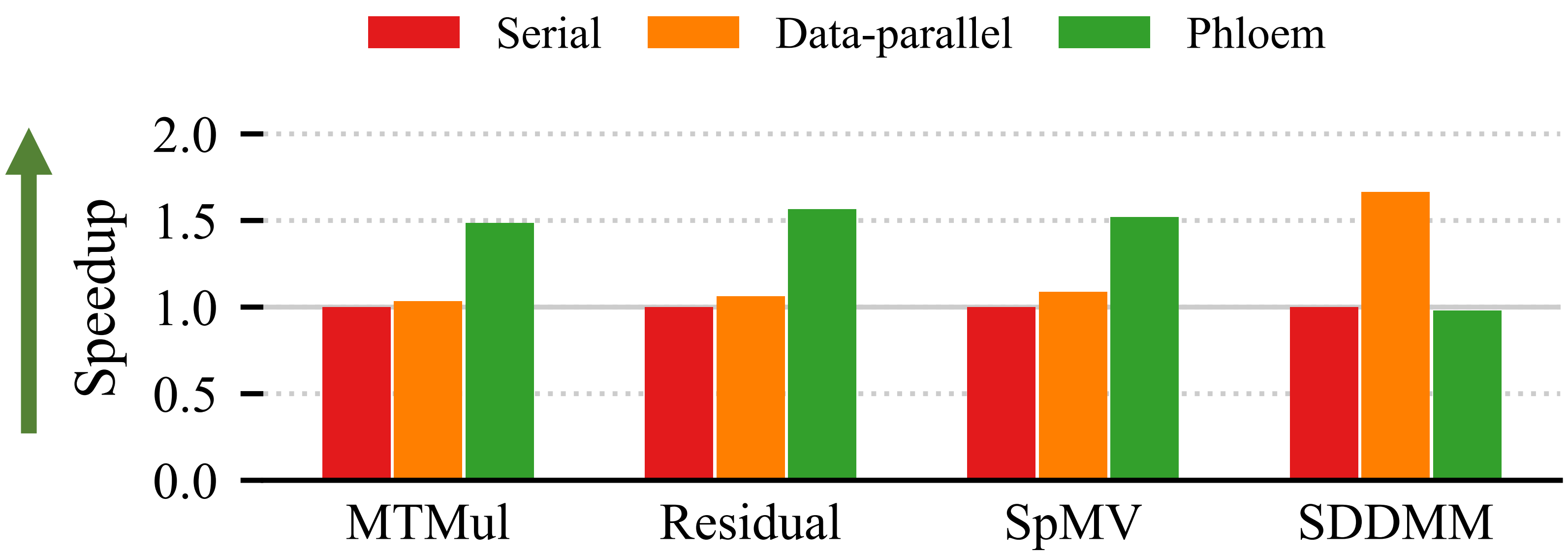



```
if (ngh & 0x3 == 0)
  // send to replica 0
else if (ngh & 0x3 == 1)
  // send to replica 1
  ...
```

```
...
int ngh = g->edges[e];
#pragma phloem distribute
int old_dist = distances[ngh];
...
```

Extending domain-specific languages





Summary and conclusion

- Emerging hardware support for building fine-grain pipelines from irregular applications changes the tradeoffs for efficient pipelines
- Phloem systematizes compiling irregular applications into pipelines
- Fast static mode and comprehensive profile-guided mode
- Achieves gmean 1.7x speedup, 80% of manual performance
- Makes state-of-the-art hardware support accessible to all

Thank you!

Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism

Quan M. Nguyen and Daniel Sanchez

qmn@csail.mit.edu and sanchez@csail.mit.edu

HPCA 2023

Wednesday, 1 March 2023