PHI: ARCHITECTURAL SUPPORT FOR SYNCHRONIZATION- AND BANDWIDTH-EFFICIENT COMMUTATIVE SCATTER UPDATES

Anurag Mukkara, Nathan Beckmann, Daniel Sanchez

MICRO 2019
Scatter updates are common but inefficient
Scatter updates are common in sparse algorithms

- e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors
Scatter updates are common but inefficient

- Scatter updates are common in sparse algorithms
  - e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

- Current memory hierarchies are optimized for reads
  - Scatter updates suffer from high synchronization and high memory bandwidth
Scatter updates are common but inefficient

- Scatter updates are common in sparse algorithms
  - e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

- Current memory hierarchies are optimized for reads
  - Scatter updates suffer from high synchronization and high memory bandwidth
Scatter updates are common but inefficient

- Scatter updates are common in sparse algorithms
  - e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

- Current memory hierarchies are optimized for reads
  - Scatter updates suffer from high synchronization and high memory bandwidth

- **Key insight:** Many scatter updates are **commutative** and can be reordered for performance
Scatter updates are common but inefficient

- Scatter updates are common in sparse algorithms
  - e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

- Current memory hierarchies are optimized for reads
  - Scatter updates suffer from high synchronization and high memory bandwidth

- **Key insight:** Many scatter updates are commutative and can be reordered for performance

- PHI extends the cache hierarchy to exploit temporal and spatial locality of commutative scatter updates
Scatter updates are common but inefficient

- Scatter updates are common in sparse algorithms
  - e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

- Current memory hierarchies are optimized for reads
  - Scatter updates suffer from high synchronization and high memory bandwidth

- **Key insight**: Many scatter updates are commutative and can be reordered for performance

- PHI extends the cache hierarchy to exploit temporal and spatial locality of commutative scatter updates
Scatter updates are common in sparse algorithms
- e.g., in push graph algorithms, vertices scatter updates to outgoing neighbors

Current memory hierarchies are optimized for reads
- Scatter updates suffer from high synchronization and high memory bandwidth

Key insight: Many scatter updates are commutative and can be reordered for performance

PHI extends the cache hierarchy to exploit temporal and spatial locality of commutative scatter updates
PHI gives large benefits
PHI gives large benefits

- PageRank algorithm on UK web graph
- 16-core processor with 32MB cache, 4 memory controllers
PHI gives large benefits

- PageRank algorithm on UK web graph
- 16-core processor with 32MB cache, 4 memory controllers

Memory traffic
PHI gives large benefits

- PageRank algorithm on UK web graph
- 16-core processor with 32MB cache, 4 memory controllers

Memory traffic

Performance
Agenda

- Background
- PHI Design
- Evaluation
Scatter updates are important
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**
Sparse algorithms perform push or pull-based indirect accesses

- Push mode: Indirect accesses are **scatter updates**

```python
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**

```python
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```
Sparse algorithms perform push or pull-based indirect accesses

Push mode: Indirect accesses are **scatter updates**

```python
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**

```python
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**
  
  ```python
  for src in vertices:
      for dst in outNeighbors(src):
          vertex(dst) += vertex(src)
  ```

- Pull mode: Indirect accesses are **gather reads**
  
  ```python
  for dst in vertices:
      for src in inNeighbors(dst):
          vertex(dst) += vertex(src)
  ```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**

```
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```

- Pull mode: Indirect accesses are **gather reads**

```
for dst in vertices:
    for src in inNeighbors(dst):
        vertex(dst) += vertex(src)
```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- **Push mode**: Indirect accesses are **scatter updates**
  
  ```python
  for src in vertices:
      for dst in outNeighbors(src):
          vertex(dst) += vertex(src)
  ```

- **Pull mode**: Indirect accesses are **gather reads**
  
  ```python
  for dst in vertices:
      for src in inNeighbors(dst):
          vertex(dst) += vertex(src)
  ```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**
  
  ```python
  for src in vertices:
      for dst in outNeighbors(src):
          vertex(dst) += vertex(src)
  ```

- Pull mode: Indirect accesses are **gather reads**
  
  ```python
  for dst in vertices:
      for src in inNeighbors(dst):
          vertex(dst) += vertex(src)
  ```
Sparse algorithms perform push or pull-based indirect accesses

- **Push mode:** Indirect accesses are **scatter updates**
  
  ```python
  for src in vertices:
      for dst in outNeighbors(src):
          vertex(dst) += vertex(src)
  ```

- **Pull mode:** Indirect accesses are **gather reads**
  
  ```python
  for dst in vertices:
      for src in inNeighbors(dst):
          vertex(dst) += vertex(src)
  ```
Scatter updates are important

- Sparse algorithms perform push or pull-based indirect accesses
- Push mode: Indirect accesses are **scatter updates**
  
  ```python
  for src in vertices:
      for dst in outNeighbors(src):
          vertex(dst) += vertex(src)
  ```

- Pull mode: Indirect accesses are **gather reads**
  
  ```python
  for dst in vertices:
      for src in inNeighbors(dst):
          vertex(dst) += vertex(src)
  ```

- Important to support scatter updates efficiently
  - Push mode performs less work when few vertices are active
  - Some algorithms do not admit a pull implementation
Scatter updates are inefficient on conventional hierarchies.
Scatter updates are inefficient on conventional hierarchies.

- Poor temporal and spatial locality when inputs do not fit in cache
- Wasteful data transfers from main memory
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies.

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies:

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging

---

Push PageRank on uk-2005 graph
Scatter updates are inefficient on conventional hierarchies

- Poor temporal and spatial locality when inputs do not fit in cache
  - Wasteful data transfers from main memory

- Multiple threads update the same vertex
  - Cache line ping-ponging

93% of traffic due to scatter updates
10x more traffic than compulsory

Push PageRank on uk-2005 graph
Prior hardware support for scatter updates
Prior hardware support for scatter updates

Remote memory operations (RMOs) send and perform update operations at a fixed location (e.g., shared cache banks)
Prior hardware support for scatter updates

- Remote memory operations (RMOs) send and perform update operations at a fixed location (e.g., shared cache banks)
  - Avoids cache-line ping ponging
Prior hardware support for scatter updates

- Remote memory operations (RMOs) send and perform update operations at a fixed location (e.g., shared cache banks)
  - Avoids cache-line ping ponging

- COUP [MICRO’15] modifies the coherence protocol to perform commutative operations in a distributed fashion
Prior hardware support for scatter updates

- Remote memory operations (RMOs) send and perform update operations at a fixed location (e.g., shared cache banks)
  - Avoids cache-line ping ponging

- COUP [MICRO’15] modifies the coherence protocol to perform commutative operations in a distributed fashion

- Both RMOs and COUP do not improve locality
Prior hardware support for scatter updates

- Remote memory operations (RMOs) send and perform update operations at a fixed location (e.g., shared cache banks)
  - Avoids cache-line ping-pong

- COUP [MICRO’15] modifies the coherence protocol to perform commutative operations in a distributed fashion

- Both RMOs and COUP do not improve locality
  - Bottlenecked by memory traffic with large inputs
PHI builds on Update Batching (UB)

Propagation Blocking [*IPDPS’17*, MILK [*PACT’16*]]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution

Propagation Blocking [IPDPS’17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution

Propagation Blocking [*IPDPS’17*, MILK [*PACT’16*]]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution

Propagation Blocking [IPDPS'17], MILK [PACT'16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution

---

**Source Vertices**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Destination Ids**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>11</th>
<th></th>
<th>0</th>
<th>7</th>
<th>9</th>
<th></th>
<th>4</th>
<th>6</th>
<th></th>
<th>3</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Destination Vertices**

- Cache fitting slice

- Propagation Blocking [IPDPS’17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices

Propagation Blocking [*IPDPS’17*, MILK [*PACT’16*]]
**PHI builds on Update Batching (UB)**

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices

![Binning Phase Diagram]

Propagation Blocking [IPDPS’17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices

Propagation Blocking [IPDPS'17], MILK [PACT'16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices

Propagation Blocking [IPDPS’17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices
- **Accumulation phase**: Reads and applies logged updates bin-by-bin

Propagation Blocking [IPDPS’17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase:** Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices
- **Accumulation phase:** Reads and applies logged updates bin-by-bin

Propagation Blocking [IPDPS'17], MILK [PACT’16]
PHI builds on Update Batching (UB)

- Maximizes spatial locality of memory transfers using two-phase execution
- **Binning phase**: Logs updates to memory, dividing them into cache-fitting slices (bins) of vertices
- **Accumulation phase**: Reads and applies logged updates bin-by-bin

Propagation Blocking [IPDPS’17], MILK [PACT’16]
Update Batching tradeoffs
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
- Compulsory memory traffic for all data structures
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
  - Compulsory memory traffic for all data structures

- Binning phase **ignores temporal locality**
  - Generates large stream of updates even with structured inputs
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
  - Compulsory memory traffic for all data structures

- **Binning phase** ignores temporal locality
  - Generates large stream of updates even with structured inputs

Push PageRank on uk-2005 graph

Unstructured input
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
  - Compulsory memory traffic for all data structures

- **Binning phase** ignores **temporal locality**
  - Generates large stream of updates even with structured inputs

Push PageRank on uk-2005 graph
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
  - Compulsory memory traffic for all data structures

- **Binning phase ignores temporal locality**
  - Generates large stream of updates even with structured inputs

Push PageRank on uk-2005 graph

Unstructured input

<table>
<thead>
<tr>
<th>Push</th>
<th>UB</th>
<th>PHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>1.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Update Batching tradeoffs

- **Perfect spatial locality** for all main memory transfers
  - Compulsory memory traffic for all data structures
- **Binning phase** ignores temporal locality
  - Generates large stream of updates even with structured inputs

Push PageRank on uk-2005 graph

**Unstructured input**

**Structured input**

- **Updates**
- **Destination**
- **Source**
- **CSR**
Key techniques of PHI
Key techniques of PHI

- In-cache update buffering and coalescing
  - Exploits temporal locality
Key techniques of PHI

- In-cache update buffering and coalescing
  - Exploits temporal locality

- Selective update batching
  - Achieves high spatial locality
Key techniques of PHI

- In-cache update buffering and coalescing
  - Exploits temporal locality

- Selective update batching
  - Achieves high spatial locality

Bandwidth efficient
Key techniques of PHI

- In-cache update buffering and coalescing
  - Exploits temporal locality

- Selective update batching
  - Achieves high spatial locality

- Hierarchical buffering and coalescing
  - Enables update parallelism
  - Eliminates synchronization overheads
In-cache buffering and coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
Buffer updates in cache without ever accessing main memory

Treat cache as a large coalescing buffer for updates
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
In-cache buffering and coalescing

- Buffer updates in cache without ever accessing main memory
- Treat cache as a large coalescing buffer for updates
- Reduction ALU in cache bank performs coalescing
Handling cache evictions
Handling cache evictions

- PHI adapts to the amount of spatial locality in the evicted line
Handling cache evictions

- PHI adapts to the amount of spatial locality in the evicted line

- Cache controller performs update batching **selectively**
  - Achieves good spatial locality in all cases
Handling cache evictions

- PHI adapts to the amount of spatial locality in the evicted line

- Cache controller performs update batching **selectively**
  - Achieves good spatial locality in all cases

- **Key insight:** Update batching is a good tradeoff only when the evicted line has poor spatial locality
Case 1: Evicted line has few updates
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Memory

Cache

0x10: F00 4
0xA4: 0 0 7 0
0xF8: 0 3 0 0

INV
Invalid line

0xF0: 0 4 0 0
Buffered-updates line

0x10: F00 4 A48 7
Line with batched updates
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Evict 0xA4

Cache

0x10: F00 4 0
0xA4: 0 0 7 0
0xF8: 0 3 0 0

Memory

INV

Invalid line

0xF0: 0 4 0 0
Buffered-updates line

0x10: F00 4 A48 7
Line with batched updates
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Evict 0xA4

Memory

Cache

0x10: F00 4
INV
0xF8: 0 3 0 0

0xF0: 0 4 0 0
Invalid line
Buffered-updates line

0x10: F00 4 A48 7
Line with batched updates
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Evict 0xA4

Memory

Cache

Evicted line

Buffered-updates line

Line with batched updates
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

![Diagram showing cache, memory, and buffer states](image-url)

- Evict 0xF8
- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

```
Evict 0xF8
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |

Evict 0x10
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |
       |   |   |   |   |

Cache

0x10: F00 4 A48 7
INV

0xF8: 0 3 0 0

Memory

INV

Invalid line

0xF0: 0 4 0 0

Buffered-updates line

0x10: F00 4 A48 7

Line with batched updates
```
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Evict 0x10

Evict 0xF8
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

![Diagram showing cache and memory operations.](image-url)
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

- Evict 0x08
- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

```
Memory
0x10: F00 4 A48 7
```

```
Cache
INV
INV
0xF8: 0 3 0 0
```

```
INV
0xF0: 0 4 0 0
```

```
Invalid line
Buffered-updates line
Line with batched updates
```
Case 1: Evicted line has few updates

- Log updates to temporary buffers (stored in cache)
- These buffers are later evicted to memory when full

Memory

```
0x10: F00 4 A48 7
```

Cache

```
0x11: F84 3
INV
INV
```

Invalid line

```
0xF0: 0 4 0 0
```

Buffered-updates line

```
0x10: F00 4 A48 7
```

Line with batched updates
Case 1: Evicted line has many valid updates
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0</td>
<td>1 2 1 7</td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0</td>
<td>4 6 3 0</td>
</tr>
<tr>
<td>0xDF</td>
<td>0 7 9 2</td>
</tr>
<tr>
<td>0xBC</td>
<td>5 6 1 8</td>
</tr>
</tbody>
</table>

**Invalid line**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0</td>
<td>0 4 0 0</td>
</tr>
</tbody>
</table>
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>0xF0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>1 2 1 7</td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Address</th>
<th>0xF0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>4 6 3 0</td>
</tr>
<tr>
<td>0xDF:</td>
<td>0 7 9 2</td>
</tr>
<tr>
<td>0xBC:</td>
<td>5 6 1 8</td>
</tr>
</tbody>
</table>

Evict 0xF0

**INV**

Invalid line

<table>
<thead>
<tr>
<th>Address</th>
<th>0xF0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>0 4 0 0</td>
</tr>
</tbody>
</table>

Buffered-updates line
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

![Memory](Memory.png)

![Cache](Cache.png)

- Evict 0xF0
- Invalid line
- Buffered-updates line
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

![Diagram showing memory and cache updates]

- Evict 0xF0

- Memory:
  - 0xF0: 1 2 1 7

- Cache:
  - 0xF0: 4 6 3 0
  - 0xDF: 0 7 9 2
  - 0xBC: 5 6 1 8

- Invalid line: 0xF0
- Buffered-updates line: 0xF0: 0 4 0 0
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

Memory

0xF0: 1 2 1 7

Cache

Evict 0xF0

0xF0: 4 6 3 0
0xDF: 0 7 9 2
0xBC: 5 6 1 8

GET 0xF0

DATA

MERGE

INVALID

Invalid line

0xF0: 0 4 0 0

Buffered-updates line
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

**Memory**

| 0xF0: | 1 | 2 | 1 | 7 |

**Cache**

- Evict 0xF0
  - 0xF0: 4 6 3 0
  - 0xDF: 0 7 9 2
  - 0xBC: 5 6 1 8

**Invalid line**

- 0xF0: 0 4 0 0

**Buffered-updates line**

- INV: [Invalid line]
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

![Diagram showing memory and cache states]

- Memory:
  - 0x FO: 5 8 4 7
  - Effective: E
  - Invalid: IN

- Cache:
  - 0x FO: 4 6 3 0
  - 0x DF: 0 7 9 2
  - 0x BC: 5 6 1 8

- Evict 0x FO:
  - Buffered-updates line
  - 0x FO: 0 4 0 0
Case 1: Evicted line has many valid updates

- Fetch line from main memory and merge updates

**Memory**

0xF0: 5 8 4 7

**Cache**

INV: 

0xDF: 0 7 9 2

0xBC: 5 6 1 8

**Evict 0xF0**

0xF0: 0 4 0 0

**Invalid line**

**Buffered-updates line**
PHI avoids synchronization costs
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
  - No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
  - No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
  - No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
- No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
  - No need for a coherence protocol
PHI avoids synchronization costs

- Private caches buffer and coalesce updates locally, push them to shared cache on evictions
  - No need for a coherence protocol

- Private caches do not perform update batching
  - Simply evict buffered-update lines to shared cache
PHI has minimal hardware costs
PHI has minimal hardware costs

- Per-line buffered updates bit
  - 0.17% additional storage with 64-byte lines
PHI has minimal hardware costs

- Per-line buffered updates bit
  - 0.17% additional storage with 64-byte lines

- Reduction unit for each cache bank
  - Supports 64-bit floating-point and integer additions, logical operations
  - 0.06% of chip area in a 16-core system (0.09mm$^2$ in 45 nm)
Agenda

- Background
- PHI Design
- Evaluation
Evaluation methodology
Evaluation methodology

- Event-driven simulation using ZSim
Evaluation methodology

- Event-driven simulation using ZSim
- 16-core processor
  - Haswell-like OOO cores
  - 32 MB L3 cache
  - 4 memory controllers
Evaluation methodology

- Event-driven simulation using ZSim
- 16-core processor
  - Haswell-like OOO cores
  - 32 MB L3 cache
  - 4 memory controllers

Graph applications
- PageRank, PageRank Delta, Connected Components, Radii Estimation
- Degree Counting (No Pull)
- SpMV

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Cache</td>
<td></td>
</tr>
<tr>
<td>Private Cache 0</td>
<td>......</td>
</tr>
<tr>
<td>Core0</td>
<td>Core15</td>
</tr>
<tr>
<td>Private Cache 15</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation methodology

- Event-driven simulation using ZSim
- 16-core processor
  - Haswell-like OOO cores
  - 32 MB L3 cache
  - 4 memory controllers

- Graph applications
  - PageRank, PageRank Delta, Connected Components, Radii Estimation
  - Degree Counting (No Pull)
  - SpMV

- Large real world inputs
  - Up to 100 million vertices
  - Up to 1 billion edges
PHI improves performance significantly
PHI improves performance significantly

<table>
<thead>
<tr>
<th></th>
<th>Push</th>
<th>Pull</th>
<th>UB</th>
<th>Push-RMO</th>
<th>PHI</th>
</tr>
</thead>
</table>

20
PHI improves performance significantly
PHI improves performance significantly

- Pull and UB show mixed results
PHI improves performance significantly

- Pull and UB show mixed results
- Push-RMO improves performance by avoiding synchronization costs
PHI improves performance significantly

- Pull and UB show mixed results
- Push-RMO improves performance by avoiding synchronization costs
- PHI consistently outperforms other schemes
PHI reduces memory traffic
PHI reduces memory traffic
PHI reduces memory traffic
Pull incurs higher memory traffic for non-all-active algorithms (CC, RE)
Pull incurs higher memory traffic for non-all-active algorithms (CC, RE)
UB increases memory traffic when input has good locality
Pull incurs higher memory traffic for non-all-active algorithms (CC, RE)

- UB increases memory traffic when input has good locality
- PHI reduces memory traffic over UB by exploiting temporal locality
Conclusion
Conclusion

- Scatter updates are inefficient on conventional hierarchies
Conclusion

- Scatter updates are inefficient on conventional hierarchies
- PHI extends the cache hierarchy to make commutative scatter updates efficient
Conclusion

- Scatter updates are inefficient on conventional hierarchies
- PHI extends the cache hierarchy to make commutative scatter updates efficient
- Exploits both temporal and spatial locality
Conclusion

- Scatter updates are inefficient on conventional hierarchies
- PHI extends the cache hierarchy to make commutative scatter updates efficient
- Exploits both temporal and spatial locality
- Incurs low memory traffic and minimal synchronization
Conclusion

- Scatter updates are inefficient on conventional hierarchies
- PHI extends the cache hierarchy to make commutative scatter updates efficient
- Exploits both temporal and spatial locality
- Incurs low memory traffic and minimal synchronization

Thanks For Your Attention!
Questions Are Welcome!