Accelerating Irregular Applications via Efficient Synchronization and Data Access Techniques

Christina Giannoula
SAFARI Live Seminar
09 November 2022

SAFARI ETH Zürich
Irregular Applications

- Graph Analytics
- Databases
- Medical Imaging

How can we accelerate the **irregular** applications?

- Neural Networks
- Bioinformatics
- Economic Modeling
Characteristic 1: Inherent Imbalance

- The objects involved are not of equal size
Characteristic 2: Random Memory Accesses

- Not sequential
- Not strided
- Input-driven

Diagonal Matrix  |  Highly Sparse Matrix  |  Highly Skewed Matrix
Characteristic 3: Low Operational Intensity

• High bottleneck by the memory subsystem
Challenge 1: Excessive Synchronization

- Inherent Imbalance
- Random Memory Accesses

**SpMV**

Coarse-Grained Approach

Diagonal Matrix

Highly Skewed Matrix

Fine-Grained Approach
Challenge 1: Excessive Synchronization

• Inherent Imbalance

• Random Memory Accesses

A large amount of processors’ cycles is spent on synchronization
Challenge 2: High Memory Intensity

- Random Memory Accesses
- Low Operational Intensity

The SpMV Execution
Challenge 2: High Memory Intensity

- Random Memory Accesses
- Low Operational Intensity

A large amount of processors’ cycles is spent on data accesses
Challenge 2: High Memory Intensity

- Random Memory Accesses
- Low Operational Intensity

The Era of Heterogeneity

Uniform Systems

Non-Uniform / Heterogeneous Systems
Our Approach

Synchronization of Threads

+ Management of Data

The two major priorities in the execution of irregular applications
Our Approach

Efficient Synchronization

• High load balance
• Low-cost inter-thread communication
• High levels of parallelism

Efficient Data Management

• Low-cost data accesses
• High memory bandwidth

The two major priorities in the execution of irregular applications
Low-overhead synchronization approaches in cooperation with well-crafted data access techniques can significantly improve performance and energy efficiency of emerging irregular applications.
Our Goal

CPU System (processor-centric)
- Processors
- Cache
- Main Memory

Processing-In-Memory (PIM) System (memory-centric)
- Host CPU
  - Cache
- PIM-Enabled Memory
  - Processor
  - Memory Arrays

Irregular Applications: important yet difficult

- Graph Analytics
- Databases
- Bioinformatics
Core Contributions

1. **Graph Processing**
   - **ColorTM (ISC’18, SRC PACT’18)**
     - High-Performance Graph Coloring for CPU Systems

2. **Pointer-Chasing**
   - **SmartPQ (CF’19)**
     - An Adaptive Priority Queue for NUMA CPU Systems

3. **Irregular Workloads**
   - **SynCron (HPCA’21)**
     - A Lightweight Synchronization Mechanism for PIM Systems

4. **Sparse Linear Algebra**
   - **SparseP (Sigmetrics’22)**
     - A Library of Efficient Sparse Matrix Vector Multiplication Kernels for Real PIM Systems
Core Contributions

1. **Graph Processing**

   ColorTM (ISC’18, SRC PACT’18)
   High-Performance Graph Coloring for CPU Systems

---

**CPU System**

- Processors
- Cache
- Main Memory

**PIM System**

- Host CPU
- Cache
- PIM-Enabled Memory
  - Processor
  - Memory Arrays

---

Trade-off between using synchronization with lower data access costs
Graph Coloring

Applications: PageRank, Community Detection, Resource Allocation ...

The Problem

How can we accelerate the graph coloring kernel?

Chromatic Scheduling

ColorTM  SmartPQ  SynCron  SparseP  Future Work
Prior Parallel Algorithms

1. Parallel Graph Coloring - No Synchronization
Prior Parallel Algorithms

1. Parallel Graph Coloring - No Synchronization

2. Detect Coloring Conflicts
Prior Parallel Algorithms

Sequential Solving (*SeqSolve* [Gebr. +’00])

1. Parallel Graph Coloring - No Synchronization
2. Detect Coloring Conflicts
3. Resolve Coloring Conflicts *Sequentially*
Prior Parallel Algorithms

Iterative Solving (\textit{IterSlv} [Boman.+’05], \textit{IterSlvR} [Rokos.+’15])

1. Parallel Graph Coloring - No Synchronization
2. Detect Coloring Conflicts
3. Repeat Steps 1 + 2 Multithreaded
Prior Parallel Algorithms

Lazy Iterative Coloring (e.g., SeqSlv, IterSlv, IterSlvR)
Prior Parallel Algorithms

Lazy Iterative Coloring (e.g., SeqSlv, IterSlv, IterSlvR)
- At least 2 iterations on the whole graph
- Lazy coloring conflict detection + resolution

<table>
<thead>
<tr>
<th></th>
<th>SeqSlv</th>
<th>IterSlv</th>
<th>IterSlvR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td>++</td>
<td>+++</td>
<td>+++</td>
</tr>
<tr>
<td>Synchronization</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
</tr>
<tr>
<td>Data Accesses</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
**ColorTM [ISC’18, SRC PACT’18]**

**Eager Iterative Coloring**
- **Eager coloring** conflict detection + resolution
- **Speculative** computation + synchronization

<table>
<thead>
<tr>
<th></th>
<th>SeqSlv</th>
<th>IterSlv</th>
<th>IterSlvR</th>
<th>ColorTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td>+</td>
<td>+ +</td>
<td>+ +</td>
<td>+</td>
</tr>
<tr>
<td>Synchronization</td>
<td>+ + +</td>
<td>+ + +</td>
<td>+ + +</td>
<td>+</td>
</tr>
<tr>
<td>Data Accesses</td>
<td>—</td>
<td>— — —</td>
<td>— — —</td>
<td>+ + +</td>
</tr>
</tbody>
</table>
**ColorTM: Key Idea 1**

**Eager Conflict Detection + Resolution**

- Iterate on **each vertex** until a valid coloring is found

- **Low Data Access Costs**
- **Low Latency**
ColorTM: Key Idea 2

Speculative Synchronization + Computation

✓ Employ hardware transactional memory
✓ Perform most computations speculatively - outside the critical section

✓ Low Synchronization Costs
✓ High Amount of Parallelism

Iterate on each vertex

2
begin_HTM();
3
validate_color();
4
end_HTM();
Performance Analysis

Speedup

Real-World Graphs

ColorTM
SmartPQ
SynCron
SparseP
Future Work

2.84x
**Balanced ColorTM**

**Imbalanced** Chromatic Scheduling

**Balanced** Chromatic Scheduling

1. 91x faster than prior works using 56 threads

**Community Detection:** 1.12x faster over the imbalanced variant using 56 threads

Low Resource Utilization

High Resource Utilization

---

**Future Work**

**ColorTM** SmartPQ SynCron SparseP Future Work
Core Contributions

- High contention $\rightarrow$ low data access costs
- Low contention $\rightarrow$ lightweight synchronization

CPU System

PIM System

SmartPQ (CF’19)
An Adaptive Priority Queue for NUMA CPU Systems
Motivation

• Priority Queues (PQs) are widely used in graph processing kernels, discrete event simulations ...

• Key Observations:
  1. PQs exhibit medium contention

```
insert()
Low Contetion
```

```
deleteMin()
High Contetion
```
Motivation

- Priority Queues (PQs) are widely used in graph processing kernels, discrete event simulations ...

- Key Observations:
  1. PQs exhibit medium contention

Can we design an ‘intelligent’ PQ to always perform best?
Key Contributions

1. A black-box approach to provide high-performance NUMA-aware Concurrent Data Structures (CDSs)

   Framework
   NUMA-Oblivious CDSs

   Nuddle: NUMA-Aware CDSs

2. An adaptive PQ to perform best under various contention workloads

   SmartPQ:
   NUMA-Aware PQ
   NUMA-Oblivious PQ
1. NUMA Node Delegation (Nuddle)

A generic framework to design NUMA-aware CDSs

NUMA Node 0

Processor (server thread) Processor (server thread) Processor (server thread)

Cache

Main Memory

Data Structure

NUMA Node 1

Processor (client thread) Processor (client thread) Processor (client thread)

Cache

NUMA Node 2

Processor (client thread) Processor (client thread) Processor (client thread)

Cache

Future Work
1. NUMA Node Delegation (Nuddle)

A **generic** framework to design **NUMA-aware CDSs**

**NUMA Node 0**
- Processor (server thread)
- Processor (server thread)
- Processor (server thread)

**Main Memory**

**Data Structure**

**Cache**

**NUMA Node 1**
- Processor (client thread)
- Processor (client thread)
- Processor (client thread)

**Cache**

**NUMA Node 2**
- Processor (client thread)
- Processor (client thread)
- Processor (client thread)

**Cache**

**response msg (up 15 clients) [Roghanchi+ SOSP’17]:** 1 cache line

**request msg (1 client) [Roghanchi+ SOSP’17]:** 1 cache line
1. NUMA Node Delegation (Nuddle)

A generic framework to design NUMA-aware CDSs

- Any NUMA-oblivious → NUMA-aware CDS
- Minimizes the memory traffic
- Low-overhead communication protocol

- Needs synchronization
- Parallelization is limited to the number of server threads
2. SmartPQ

An adaptive PQ that switches between two algorithmic modes whenever it is needed.
2. SmartPQ

An adaptive PQ that switches between two algorithmic modes whenever it is needed

Key Challenges:
1. How to switch between the two modes with low synchronization overheads?
2. SmartPQ

An adaptive PQ that switches between two algorithmic modes whenever it is needed.

Key Challenges:

1. How to switch between the two modes with low synchronization overheads?
2. SmartPQ

An adaptive PQ that switches between two algorithmic modes whenever it is needed

Key Challenges:

1. How to switch between the two modes with low synchronization overheads?

2. When to switch from the one to the other mode?
   
   Decision Tree Classifier
   
   NUMA-Aware, NUMA-Oblivious, Neutral
2. SmartPQ

An adaptive PQ that switches between two algorithmic modes whenever it is needed. 

**Low Cost?**

2-4 ms traversal time with 180 nodes and a very low tree depth of 8

**Key Challenges:**
1. How to switch between the two modes with low synchronization overheads?
2. When to switch from the one to the other mode?

**87.9% accuracy** in a test set of 10K different contention workloads

**Decision Tree Classifier**

NUMA-Aware, NUMA-Oblivious, Neutral
Throughput Evaluation

40% - 60% \text{insert()} - \text{deleteMin()} Ratio

Nuddle

80% - 20% \text{insert()} - \text{deleteMin()} Ratio

\text{alistarh\_herlihy}\]

\text{NUMA-Oblivious}

\text{NUMA-Aware}

\begin{center}
\begin{tabular}{llllllllllllll}
\textbf{Number of Threads} & 2 & 4 & 8 & 15 & 22 & 29 & 36 & 50 & 57 & 106 & 155 \\
\hline
\textbf{Throughput (MOps/s)} \\
\end{tabular}
\end{center}
Throughput with Varying Contention

Throughput (MOps/s) vs. Execution Time (s)

- alistarh_herlihy
- Nuddle
- SmartPQ
Throughput with Varying Contention

Throughput (MOps/s)

Execution Time (s)

1.87x 1.38x
Core Contributions

3. Irregular Workloads

SynCron (HPCA’21)
A Lightweight Synchronization Mechanism for PIM Systems

Enabling very low synchronization costs
Synchronization challenges in PIM systems:

1. Lack of hardware cache coherence support
2. Lack of a shared level of cache memory
3. Expensive communication across PIM units
PIM Synchronization Solution Space

(1) Shared Memory
(2) Message-passing
PIM Synchronization Solution Space

SynCron’s Key Techniques:
1. Hardware support for synchronization acceleration
2. Direct buffering of synchronization variables
3. Hierarchical message-passing communication
4. Integrated hardware-only overflow management
1. Hardware Synchronization Support

- No Complex Cache Coherence Protocols
- No Expensive Atomic Operations
- Low Hardware Cost
2. Direct Buffering of Variables

PIM Unit 0
- PIM Core 0
- PIM Core 1
- Synchronization Engine 0

Main Memory

Synchronization Processing Unit
- Synchronization Table
- Indexing Counters

PIM Unit 1
- PIM Core 0
- PIM Core 1
- Synchronization Engine 1

Main Memory

Address
---
---
---
---
---

ColorTM | SmartPQ | SynCron | SparseP | Future Work
2. Direct Buffering of Variables

PIM Unit 0

PIM Core 0

PIM Core 1

Synchronization Engine 0

Main Memory

Synchronization Processing Unit

Synchronization Table

Indexing Counters

Local lock acquire

PIM Unit 1

PIM Core 0

PIM Core 1

Synchronization Engine 1

Main Memory

Address

<table>
<thead>
<tr>
<th>Address</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x33A9</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
2. Direct Buffering of Variables

- No Costly Memory Accesses
- Low Latency
3. Hierarchical Communication
3. Hierarchical Communication

PIM Unit 0
- PIM Core 0
- PIM Core 1
- Synchronization Engine 0
- Main Memory

PIM Unit 1
- PIM Core 0
- PIM Core 1
- Synchronization Engine 1
- Main Memory
- syncronVar

PIM Unit 2
- PIM Core 0
- PIM Core 1
- Synchronization Engine 2
- Main Memory

PIM Unit 3
- PIM Core 0
- PIM Core 1
- Synchronization Engine 3
- Main Memory

Local lock acquire
Master
3. Hierarchical Communication

Minimize Expensive Network Traffic

Global lock acquire
4. Integrated Overflow Management

- Low Performance Degradation
- High Programming Ease
Throughput of Pointer Chasing

- Stack - 100K
- Hash Table - 1K
- Linked List - 20K

Number of PIM Cores

Operations / µs

High Contention
Small #Variables

Medium Contention
Medium #Variables

Low Contention
High #Variables
Throughput of Pointer Chasing

Central  Hier  SynCron  Ideal

**Stack - 100K**

- High Contention: 1.26x, 1.18x
- Medium Contention: 1.59x
- Low Contention

**Hash Table - 1K**

- High Contention: 1.68x, 1.19x
- Medium Contention: 1.78x
- Low Contention

**Linked List - 20K**

- High Contention: 1.18x, 1.26x
- Medium Contention: 1.59x
- Low Contention

ColorTM  SmartPQ  SynCron  SparseP  Future Work
Performance in Data Analytics

- Central
- Hier
- SynCron
- Ideal

Speedup

-9.5%
1.23x

Algorithms:
- bfs.sl
- cc.sx
- sssp.co
- pr.wk
- tf.sl
- tc.sx
- ts.air
- ts.pow
- AVG

Future Work

色色
SmartPQ
SynCron
SparseP
Future Work
System Energy in Data Analytics

Energy Breakdown

- Cache
- Network
- Memory

- bfs.sl
- cc.sx
- sssp.co
- pr.wk
- tf.sl
- tc.sx
- ts.air
- ts.pow
- AVG

ColorTM  SmartPQ  SynCron  SparseP  Future Work
Core Contributions

Across multiple PIM cores → low data transfer costs
Across multiple threads → lightweight synchronization

SparseP (Sigmetrics’22)
A Library of Efficient Sparse Matrix Vector Multiplication Kernels for Real PIM Systems
Motivation

• Sparse Matrix Vector Multiplication (SpMV):
  • Widely used in machine learning, graph analytics, scientific computing...
  • A highly bandwidth-bound kernel

Roofline Model

[Diagram showing the roofline model with SpMV and peak memory bandwidth highlighted]
Motivation

- Sparse Matrix Vector Multiplication (SpMV):
  - Widely used in machine learning, graph analytics, scientific computing...
  - A highly bandwidth-bound kernel
- Real Near-Bank Processing-In-Memory (PIM) Systems:
  - High levels of parallelism
  - Large aggregate memory bandwidth
Key Contributions

1. Design **efficient SpMV kernels** for current and future PIM systems
   - SparseP = 25 SpMV kernels

SparseP is Open-Source

SparseP: [https://github.com/CMU-SAFARI/SparseP](https://github.com/CMU-SAFARI/SparseP)

2. Provide a **comprehensive analysis** of SpMV on the first commercially-available real PIM system
   - 26 sparse matrices
   - Comparisons to state-of-the-art **CPU** and **GPU** systems
   - **Recommendations** for software, system and hardware designers

Recommendations for Architects and Programmers

SpMV Execution on a PIM System

1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results

PIM-Enabled Memory

Main Memory

Host CPU

ColorTM
SmartPQ
SynCron
SparseP
Future Work
SparseP supports two types of data partitioning techniques:

1D Partitioning (**compute-aware**)  
- $4 \times 1$ input vector  
- $1 \times 4$ output vector  
- Core 1  
  - Core 2  
  - Core 3  
  - Core 4  
- perform the **complete** SpMV computation only on PIM cores

2D Partitioning (**data-aware**)  
- $2 \times 2$ input vector  
- $2 \times 2$ output vector  
- Core 1  
  - Core 2  
  - Core 3  
  - Core 4  
- trade-off computation vs data transfer costs
Compute-Aware vs Data-Aware SpMV

From 64 up to 2528 PIM Cores, 32-bit float

1D (compute-aware) 2D (data-aware)

1.45x 1.31x

Speedup

1.8 1.6 1.4 1.2 1.0 0.8 0.6 0.4 0.2 0.0

regular scale-free

hgc mc2 pfm rtn rjt ash del tdk mem amz fth wbg ldr psb bns wbs in pkx cmb sxw skt ask GM (1) GM (2)

ColorTM SmartPQ SynCron SparseP Future Work

66
Compute-Aware vs Data-Aware SpMV

From 64 up to 2528 PIM Cores, 32-bit float

- 1D (compute-aware)
- 2D (data-aware)

>1100 Idle Cores
>2200 Idle Cores

Best-performing SpMV execution: trades off computation with lower data transfer costs

Regular
Scale-free

ColorTM SmartPQ SynCron SparseP Future Work
Compute-Aware vs Data-Aware SpMV

From 64 up to 2528 PIM Cores, 32-bit float

- 1D (compute-aware)
- 2D (data-aware)

Recommendation

Improve Data Transfer Costs:
- Provide **hardware support** to effectively **overlap** computation with data transfers in the PIM system.
- **Integrate** PIM-enabled memory as the main **memory** of the system.
- Design **high-speed communication channels** and **optimized libraries** in data transfers to/from PIM-enabled memory.
SpMV Execution on a PIM System

1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results

Main Memory

PIM-Enabled Memory

Host CPU

ColorTM  SmartPQ  SynCron  SparseP  Future Work
Synchronization Approaches

Multithreaded PIM Core:

Coarse-Grained \((lb-cg)\):

- Thread 1
- Thread 2
- Thread 3

Fine-Grained \((lb-fg)\):

- Thread 1
- Thread 2
- Thread 3

Lock-Free \((lf)\):

- Thread 1
- Thread 2
- Thread 3

Partial results:

\[\text{Thread 1} + \text{Thread 2} = \text{output vector}\]
Performance of Synchronization Schemes

16 threads, 32-bit integer

![Bar chart showing speedup for different schemes](chart.png)

- **lb-cg**: Delaunay_n13, Wing_nodal, Raefsky4, PKUSTK08
- **lb-fg**: Delaunay_n13, Wing_nodal, Raefsky4, PKUSTK08
- **lf**: Delaunay_n13, Wing_nodal, Raefsky4, PKUSTK08

Fine-grained locking (lb-fg) does **not improve** performance over coarse-grained locking (lb-cg)

Diagram showing PIM Core architecture with:
- **24 KB Instr. Mem.**
- **64 KB Data Mem.**
- **64 MB DRAM Bank**
- **DMA Engine**
Performance of Synchronization Schemes

16 threads, 32-bit integer

Fine-Grained Locking: memory accesses to the local DRAM bank are serialized in the DMA engine of the UPMEM PIM hardware.
Improving Synchronization in Multithreaded PIM Systems

- Provide low-cost synchronization mechanisms for a multithreaded PIM core.
- Design hardware support to enable concurrent memory accesses to the local DRAM bank.
- Integrate multiple DRAM banks per PIM core to increase execution parallelism.

## Recommendation

- **Multithreaded PIM Core**
- **DMA Engine**
- **32 MB DRAM Bank**

---

**Speedup**

- **lb-cg**
- **lb-fg**
- **lf**

**Future Work**

- **ColorTM**
- **SmartPQ**
- **SynCron**
- **SparseP**
SpMV Execution on Various Systems

**CPU System**

1. Execute the kernel

**Real PIM System**

1. Load the input vector

**PIM-Enabled Memory**

2. Execute the kernel

**GPU System**

2. Execute the kernel

1. Load the input vector

3. Retrieve the final vector

4. Merge the partial results

**Host CPU**
CPU/GPU Comparisons

- **Kernel-Only (32-bit float):**
  - CPU = 0.51% of Peak Perf.
  - GPU = 0.21% of Peak Perf.
  - PIM (1D) = **50.7%** of Peak Perf.

- **Kernel-Energy (32-bit float):**
  - CPU = 0.247 J
  - GPU = **0.051 J**
  - PIM (1D) = 0.179 J

- **End-to-End (32-bit float):**
  - CPU = **4.08 GFlop/s**
  - GPU = 1.92 GFlop/s
  - PIM (1D) = 0.11 GFlop/s

<table>
<thead>
<tr>
<th>System</th>
<th>Peak Performance</th>
<th>Bandwidth</th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon Silver 4110</td>
<td>660 GFlops</td>
<td>23.1 GB/s</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100</td>
<td><strong>14.13 TFlops</strong></td>
<td>897 GB/s</td>
</tr>
<tr>
<td>PIM</td>
<td>UPMEM 1st Gen.</td>
<td>4.66 GFlops</td>
<td><strong>1.77 TB/s</strong></td>
</tr>
</tbody>
</table>
CPU/GPU Comparisons

• **Kernel-Only** (32-bit float):
  - CPU = 0.51% of Peak Perf.
  - GPU = 0.21% of Peak Perf.
  - **PIM (1D) = 50.7% of Peak Perf.**

• **Kernel-Energy** (32-bit float):
  - CPU = 0.247 J
  - GPU = 0.051 J
  - **PIM (1D) = 0.179 J**

• **End-to-End** (32-bit float):
  - CPU = 4.08 GFlop/s
  - GPU = 1.92 GFlop/s
  - **PIM (1D) = 0.11 GFlop/s**

**PIM: 1.38x higher energy efficiency over CPU**

<table>
<thead>
<tr>
<th>System</th>
<th>Peak Performance</th>
<th>Bandwidth</th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon Silver 4110</td>
<td>660 GFlops</td>
<td>23.1 GB/s</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100</td>
<td>14.13 TFlops</td>
<td>897 GB/s</td>
</tr>
<tr>
<td>PIM</td>
<td>UPMEM 1st Gen.</td>
<td>4.66 GFlops</td>
<td>1.77 TB/s</td>
</tr>
</tbody>
</table>

System Peak Performance Bandwidth TDP

**Processor-Centric**

**Memory-Centric**
Core Contributions

1. Graph Processing
   - ColorTM (ISC’18, SRC PACT’18)
     High-Performance Graph Coloring for CPU Systems

2. Pointer-Chasing
   - SmartPQ (CF’19)
     An Adaptive Priority Queue for NUMA CPU Systems

3. Irregular Workloads
   - SynCron (HPCA’21)
     A Lightweight Synchronization Mechanism for PIM Systems

4. Sparse Linear Algebra
   - SparseP (Sigmetrics’22)
     A Library of Efficient Sparse Matrix Vector Multiplication Kernels for Real PIM Systems

Research Statement:

Low-overhead synchronization approaches in cooperation with well-crafted data access techniques can significantly improve performance and energy efficiency of emerging irregular applications.
Talk Summary

• Irregular applications exhibit:
Inherent imbalance, random accesses, low operational intensity

• Key optimization opportunities:
Lightweight synchronization + well-crafted data access policies
Future Research Directions

• Designing new adaptive approaches for irregular applications to capture dynamic workload demands and contention:
  • Adaptive algorithmic designs
  • Adaptive runtime systems
  • Adaptive hardware mechanisms

• Extending the techniques that we propose to accelerate irregular applications in new/unconventional systems:
  • Hybrid/heterogeneous memory systems
  • Disaggregated memory systems

• Leveraging the key insights and recommendations that we provide to improve multiple aspects of CPU and PIM hardware and software
Accelerating Irregular Applications via Efficient Synchronization and Data Access Techniques

Christina Giannoula
SAFARI Live Seminar
09 November 2022

Thank you!