IEEE HiPC'20

PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks

Vivek Kumar

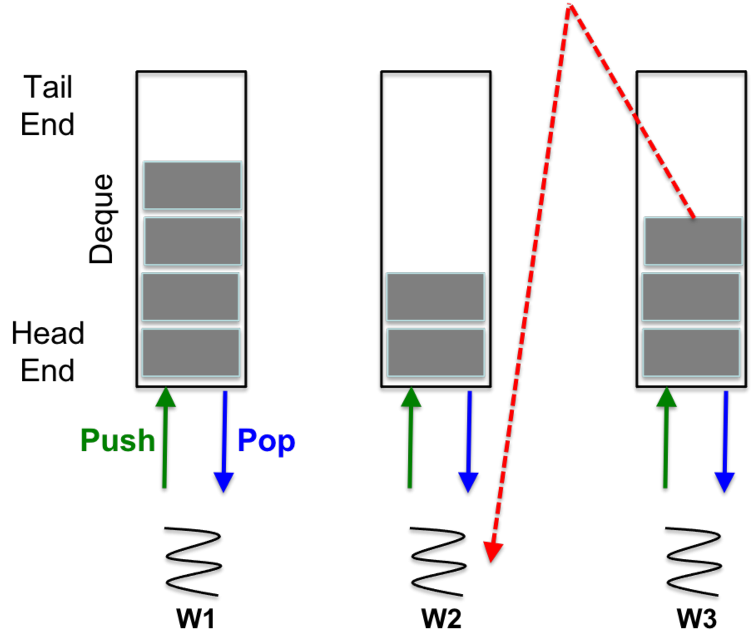
IIIT New Delhi, India

Outline

- Introduction
- Contributions
- Motivating analysis
- Insights and approach
- Implementation
- Experimental Evaluation
- Summary

Task Parallelism on Multicore Processors

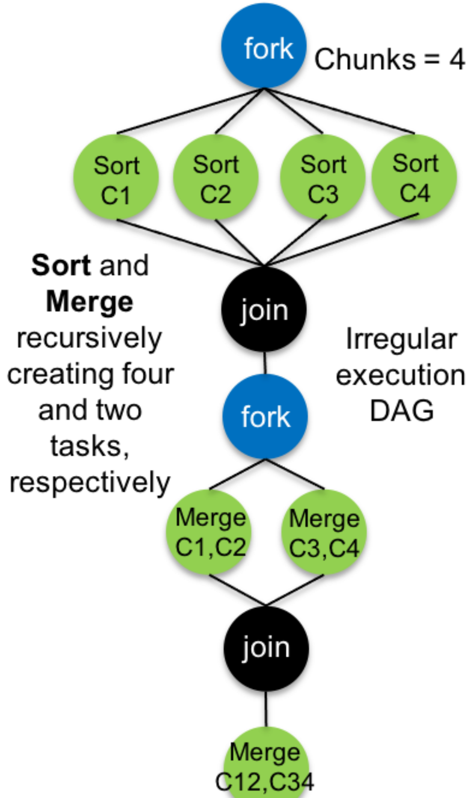
```
1. void foo() {
2.   finish {
3.     async S1; // Parallel Task-1
4.     async S2; // Parallel Task-2
5.   } // Synchronization point
6.   S3; // Starts after termination of Task-1 & Task-2
7. }
```



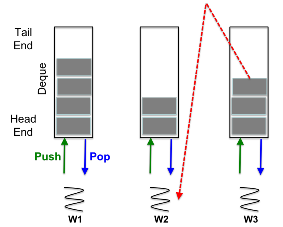
- Dynamic task parallelism using async-finish
  - async **fork** a new task that can run in parallel to other tasks inside finish
  - finish **joins** all async tasks created inside its scope
- High **productivity** due to serial-elision
  - Removing all async and finish constructs results in a valid sequential program
- High **performance** from work-stealing runtime
  - Each worker (**victim**) **push** and **pop** async on its **deque**
  - Idle worker (**thief**) **randomly** chooses a victim to **steal** a task

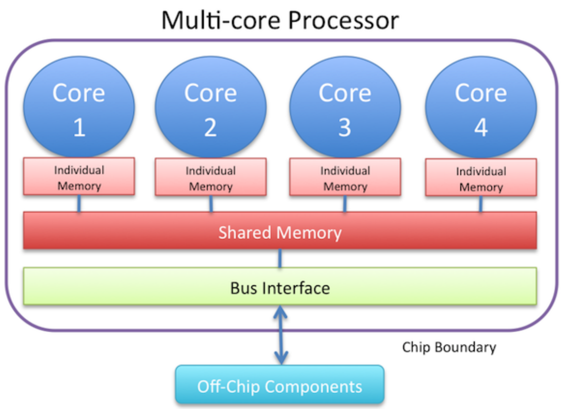
Merge Sort on **UMA** Multicore Processor

```
int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
```




```
} /* End of code */
```



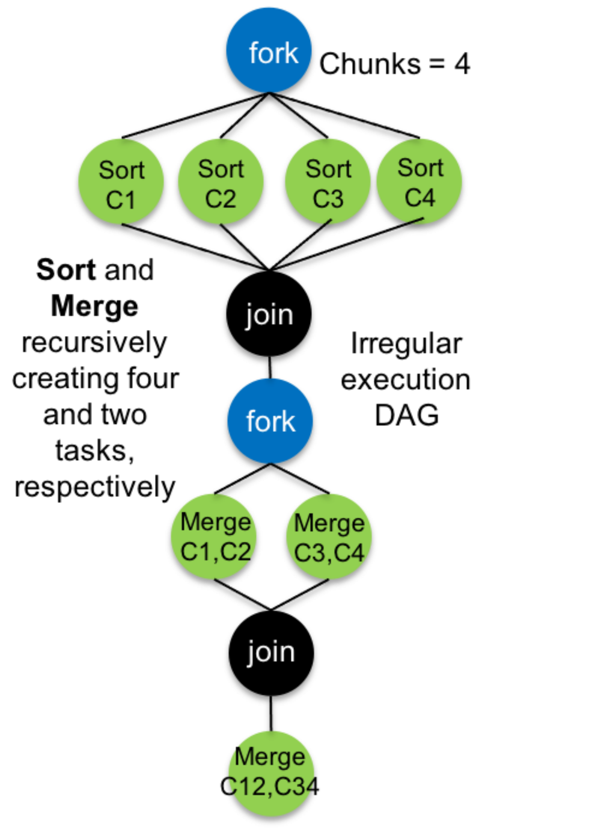


- Multicore processor with Uniform Memory Access (UMA)
  - High performance
    - Same latency to access a memory location by all cores

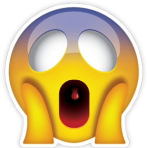
Multicore processor figure source: https://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/fig1.png

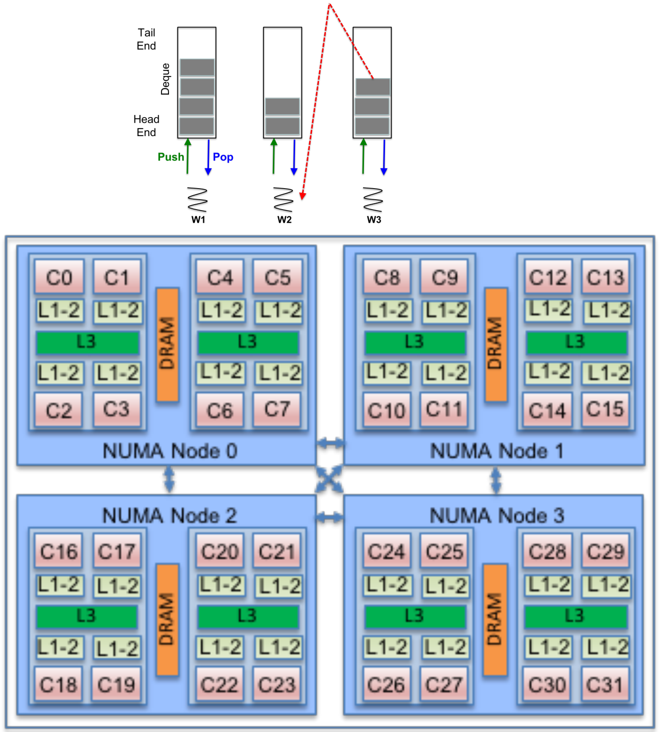
Merge Sort on **NUMA** Multicore Processor

```
int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
```



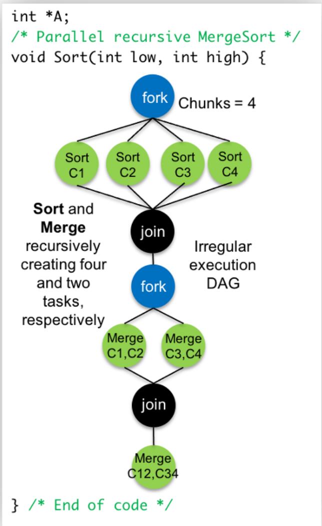

```
} /* End of code */
```

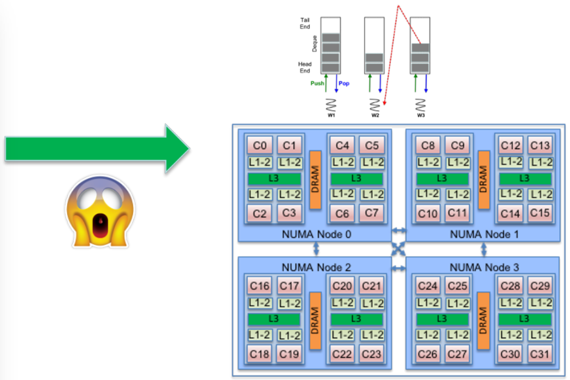




- Multicore processor with Non-UMA (NUMA)
  - Low performance
    - Random work-stealing disrupts the locality
      - Task and its data may not be on the same NUMA node
      - Thief doesn't prioritize local steal over remote steal

Work-Stealing in a Recursive Application with Irregular Execution DAG





- How to schedule a task on a NUMA node that has the task's data
  - Programmer based task mapping
    - Program modification
    - Breaks serial elision
- How to prioritize local steal over remote steals
  - Hierarchical work-stealing
    - Remote steal breaks locality
    - Not stealing from remote node can starve workers within a node

Contributions

PufferFish programming model

For NUMA-aware task parallelism that uses data-affinity hints and *almost* supports serial elision

Lightweight work-stealing implementation

That integrates data-affinity hints with a hierarchical work-stealing library without causing starvation

Locality preserving hierarchical elastic tasks

That improves locality by reducing context switches at task creation by increasing or decreasing its parallelism

Detailed performance study

Using both micro and real-world benchmarks on a 32-core NUMA processor

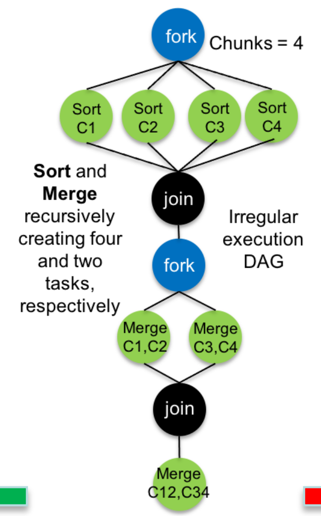
Motivating Analysis

Merge Sort using Hierarchical Place Trees (HPT [1])

```
int *A;
void Sort(int low, int high) {
  if((high-low)<LIMIT) return SeqSort(low, high);
  int Chunks=(high-low)/4;
  finish {
    async Sort(/*Chunk C1*/);
    async Sort(/*Chunk C2*/);
    async Sort(/*Chunk C3*/);
    async Sort(/*Chunk C4*/);
  }
  finish {
    async Merge(/*Chunk C1*/, /*Chunk C2*/);
    async Merge(/*Chunk C3*/, /*Chunk C4*/);
  }
  Merge(/*Chunk C12*/, /*Chunk C34*/);
}
void kernel() {
  A = new int[N];
  initialize();
  Sort(0, N);
  delete(A)
}
```

async-finish for UMA

```
int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
```

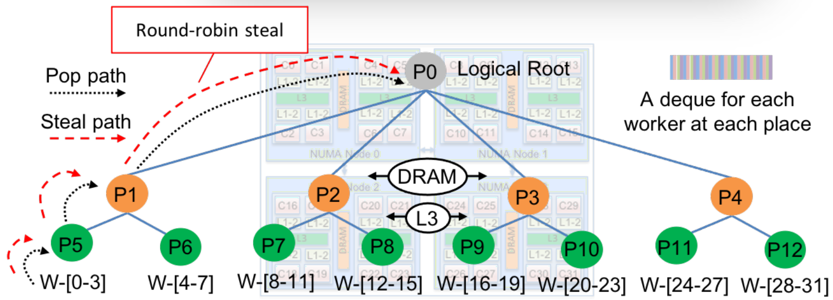



```
} /* End of code */
```

```
int *A;
void Sort(int low, int high) {
  .....
  Place* Parent = get_parent_place();
  if(Parent == NULL) {
    finish {
      async_at_hpt(P1) Sort(...);
      async_at_hpt(P2) Sort(...);
      async_at_hpt(P3) Sort(...);
      async_at_hpt(P4) Sort(...);
    }
    finish {
      async_at_hpt(P1) Merge(...);
      async_at_hpt(P4) Merge(...);
    }
    Merge(...);
  } else {
    /* Original async-finish blocks
     * by replacing each of the async
     * with async_at_hpt(Parent)
     */
  }
}
...
```

HPT for NUMA

- HPT implementation in HClib [2]
  - Top-level task partitioning by programmer
    - Required at each finish scopes
      - Breaks serial elision property of async-finish
  - Hierarchical work-stealing
    - Worker W0 attempts to pop task from P5, P1, P0, and then attempts to steal also in same order if pop failed
      - Starvation at NUMA places P2 and P3 during Merge



[1] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A portable abstraction for task parallelism and data movement", LCPC'10

[2] http://habanero-rice.github.io/hclib/

Insights and Approach

- Preserve serial elision in async-finish programming over NUMA processor
  - PufferFish programming model for integrating data-affinity hints in an async

async_hinted	numa_alloc_block_cyclic	numa_alloc_interleaved	numa_free
Assign data-affinity hints with an async task	Block cyclic allocation of physical pages on NUMA nodes	Round-robin allocation of physical pages over NUMA nodes	Deallocate the physical pages

- Hierarchical work-stealing should neither break the task locality, nor it should induce starvation
  - Automatically calculate place to push async_hinted
  - If there is no load imbalance at a worker's leaf place, let it directly execute the task
    - Avoids context switch at task creation and improves locality

Merge Sort using PufferFish Programming Model

```
int *A;
void Sort(int low, int high) {
  if((high-low)<LIMIT) return SeqSort(low, high);
  int Chunks=(high-low)/4;
  finish {
    async Sort(/*Chunk C1*/);
    async Sort(/*Chunk C2*/);
    async Sort(/*Chunk C3*/);
    async Sort(/*Chunk C4*/);
  }
  finish {
    async Merge(/*Chunk C1*/, /*Chunk C2*/);
    async Merge(/*Chunk C3*/, /*Chunk C4*/);
  }
  Merge(/*Chunk C12*/, /*Chunk C34*/);
}
void kernel() {
  A = new int[N];
  initialize();
  Sort(0, N);
  delete(A)
}
```

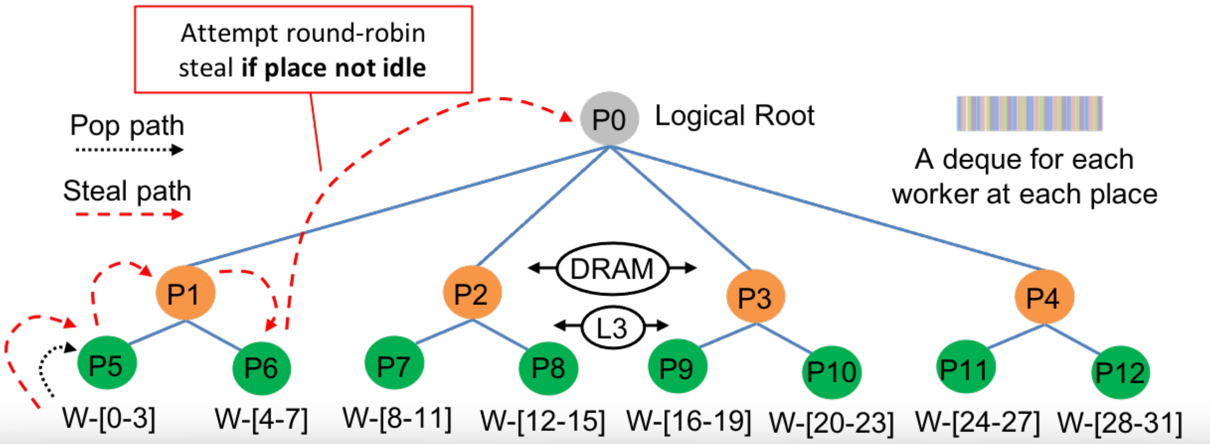
async-finish for UMA

```
int *A;
void Sort(int low, int high) {
  if((high-low)<LIMIT) return SeqSort(low, high);
  int Chunks=(high-low)/4;
  finish {
    async_hinted (A, C1_start, C1_end) Sort(/*Chunk C1*/);
    async_hinted (A, C2_start, C2_end) Sort(/*Chunk C2*/);
    async_hinted (A, C3_start, C3_end) Sort(/*Chunk C3*/);
    async_hinted (A, C4_start, C4_end) Sort(/*Chunk C4*/);
  }
  finish {
    async_hinted (A, C1_start, C2_end) Merge(/*Chunk C1*/, /*Chunk C2*/);
    async_hinted (A, C3_start, C4_end) Merge(/*Chunk C3*/, /*Chunk C4*/);
  }
  Merge(/*Chunk C12*/, /*Chunk C34*/);
}
void kernel() {
  A = numa_alloc_blockcyclic<int>(N);
  initialize();
  Sort(0, N);
  numa_free(A)
}
```

async_hinted-finish for NUMA

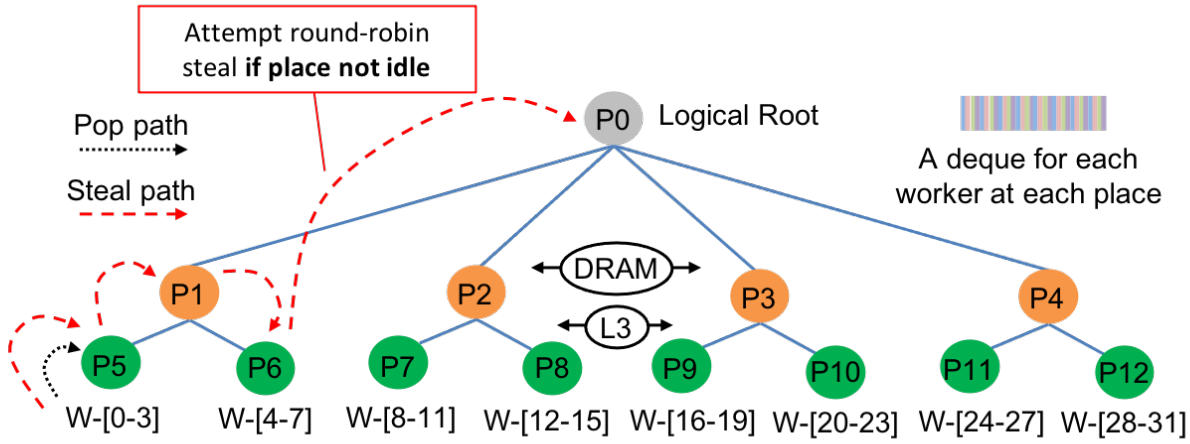
- PufferFish programming model
  - Implemented over HPT implementation in HClib
  - Assigns data-affinity hints to async tasks instead of place affinity
    - No program modification based on NUMA architecture
    - Supports serial elision
      - Except for two NUMA memory allocation/deallocation APIs

Hierarchical Elastic Tasks



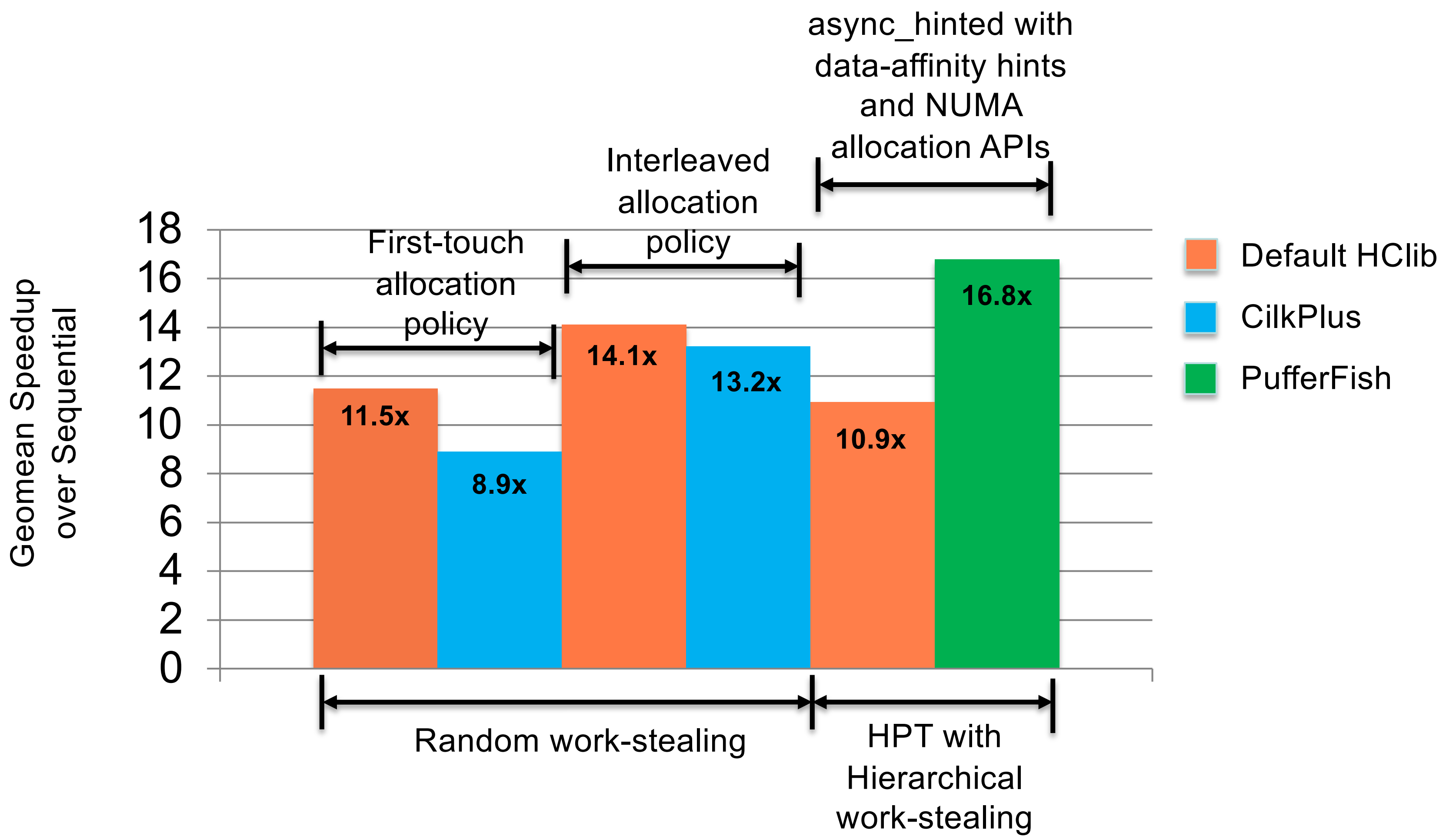

```
void async_hinted (void* Array, int start_index, int end_index, task_t* task) {
    Place* DRAM_place = get_place_with_physicalPages(Array, start_index, end_index);
    if (/* Physical pages map to multiple DRAM places */) {
        async_at_hpt (P0, task);
    } else {
        if (/* Current worker is under same DRAM_place */) {
            if (/*No failed steals at current worker's leaf place */) {
                direct_execution (task); /* Avoids a context switch with push */
            } else {
                async_at_hpt (/* Current worker's leaf place */, task);
            }
        } else {
            async_at_hpt (DRAM_place, task);
        }
    }
}
```

Hierarchical Work-Stealing



- Modifications to HPT in HClib
  - Worker can **pop** only from its leaf place
  - Hierarchical **steals** within a NUMA domain and then from logical root
    - W0 at place P5 steal from all deques at places P5, P1, P6, and P0, respectively until successful
      - Strict locality without worker starvation

Performance Analysis on AMD EPYC 7551



Executing summary for seven recursive benchmarks with regular/irregular DAG on a 32-core processor with four NUMA nodes

Summary and Conclusion

- Mapping async-finish to NUMA node in recursive applications
  - Breaks serial elision
  - Create starvation
- PufferFish
  - async-finish programming model with data-affinity hints instead of NUMA place hints
    - Almost serial elision
    - No program modifications for different NUMA configurations
  - Hierarchical work-stealing with strict locality and hierarchical elastic tasks
    - Improves locality without starvation

Artifact

- Open sourced on Github
  - https://github.com/hipec/pufferFish/archive/v1.0.zip
- Author information
  - http://vivkumar.github.io/