Hierarchical Work Stealing on Manycore Clusters

Seung-Jai Min[1], Costin Iancu[1], Katherine Yelick[1,2]

[1]Lawrence Berkeley National Laboratory and
[2]University of California at Berkeley

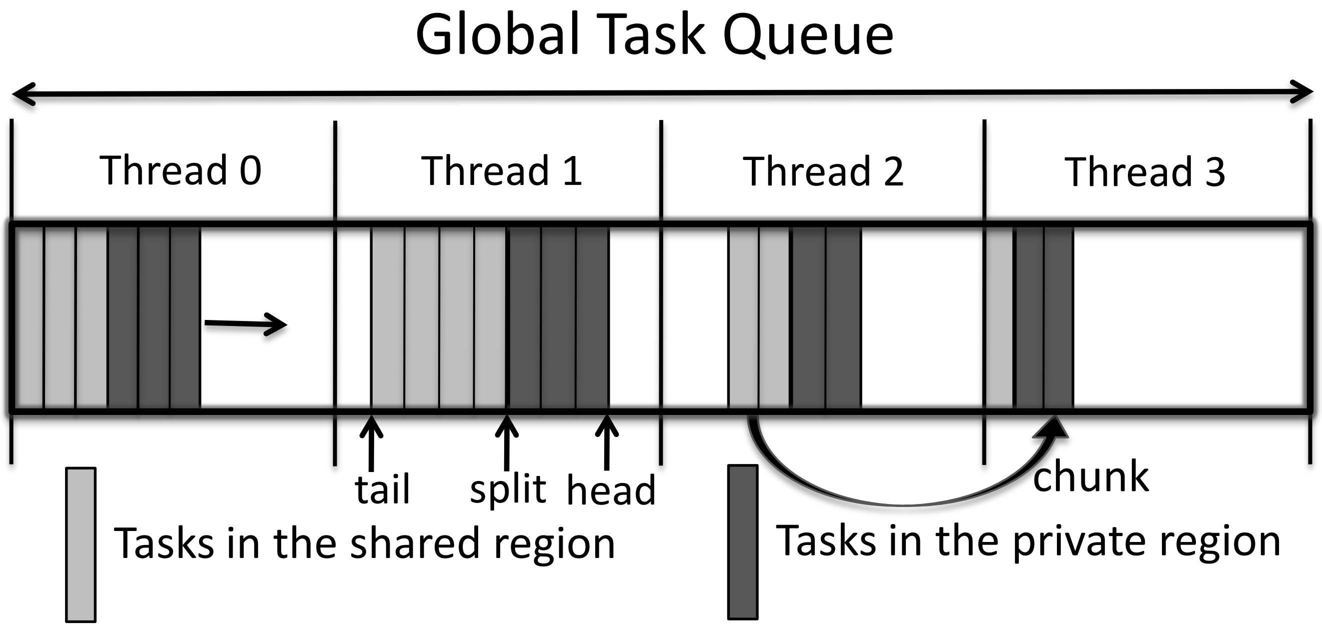
Motivation

- SPMD (Single-Program Multiple-Data) model in UPC
  - Fixed set of threads matches the underlying hardware
  - The global address space handles irregular data accesses
  - Irregular computational patterns:
    - Not statically load balanced (even with graph partitioning, etc.)
    - The work and parallelism unfold dynamically throughout the program execution
  - No direct support for applications with dynamic tasking
  - Some kind of dynamic load balancing needed with a task queue

HotSLAW

- One-sided data access mechanism to implement work-stealing efficiently on large scale systems
- Builds on prior work on dynamic tasking
  - "SLAW" by Guo et al. (Rice Univ.)
    - Scalable Locality-aware Adaptive Work-Stealing
    - Combines work-first and help-first with bounded memory usage
    - Allows stealing only within a place (a user defined locality domain)
  - "Scalable Work Stealing" by Dinan et al. (Ohio State Univ.)
    - Work-stealing for large scale distributed-memory systems
    - Steals a fixed ratio of work per event (HalfSteal)

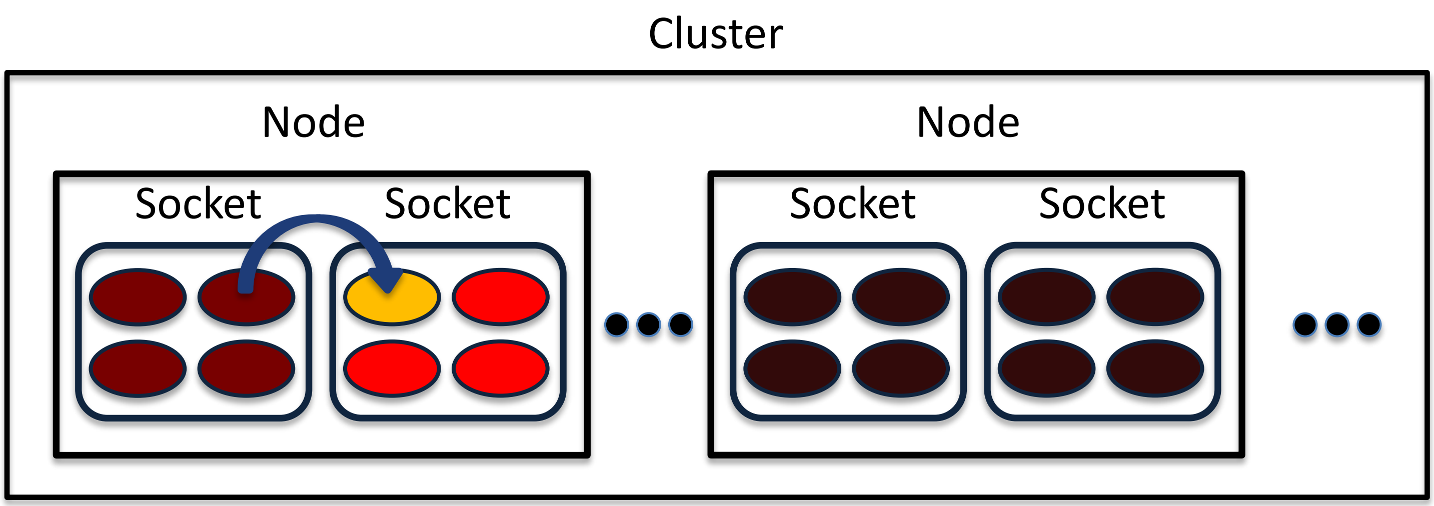
HotSLAW Implementation



- A global queue is stitched from per-thread local queue
- Per-thread Local queue = shared region + private region
  - Shared region: stealing from other threads is serialized through a lock
    - FIFO queue: the oldest task contains the largest amount of work in the task graph
  - Private region
    - LIFO stack: the most recently created task has a higher chance of exploiting $ locality

HotSLAW Implementation (cont.)

- Hierarchical Work Stealing
  - HVS (Hierarchical Victim Selection)
    - Determines from which thread a thief thread steals work
  - HCS (Hierarchical Chunk Selection)
    - Dictates how much work a thief thread teals from the victim



HVS (Hierarchical Victim Selection)

- RANDOM selection has been the state-of-the-art strategy in selecting victims for work-stealing in shared-memory domain
- SLAW limits work-stealing only within a place in SMP
  - Places provide for a two-level abstract view (local vs. non-local)
  - A place is defined as sharing an L2 cache in their study
- HotSLAW supports multi-level hierarchy
  - Provides API to control # of locality levels and # of CPUs per level
  - A thread first attempts to steal from the nearest neighbors, and gradually moves up the locality hierarchy
  - Number of steal attempts: # of cores for SMP, 4xlog(N) for cluster

HCS (Hierarchical Chunk Selection)

- Work stealing is sensitive to the # of tasks stolen. (this amount is referred to as chunk size)
- Fixed chunk policy
  - Steal one task from the tail of the victim's queue, hoping to maximize the probability of stealing the task with the max amount of work
- StealHalf policy
  - Thieves steal one half of the victim's (shared) queue.
  - StealHalf policy reduces the number of expensive inter-node stealing
- HCS (Hierarchical Chunk Selection) Policy
  - Based on the distance between the thief and the victim, HCS steals a fixed-sized chunk for lower hierarchy levels and uses StealHalf at the topmost level, e.g. inter-node.

UPC Task Library API

- High-level API:
  - Concise and expressive
  - abstracts concurrent task management details
- Task
  - Function granularity with a signature containing pointers to input and out

```
void my_func(void *input, void *output);
```

Input and output are contiguous memory
Input is copied into the library space and travels with the task on migration

```
void FIB( int *n, int *out ) {
    int n1 = *n-1;
    int n2 = *n-2;
    int x, y;
    if (*n < 2){  /* CUTOFF */
      *out = *n;
      return;
    }
    taskq_put(taskq, FIB, &n1, &x);
    taskq_put(taskq, FIB, &n2, &y);
    taskq_wait(taskq);
    *out = x + y;
}
```

UPC Task Library API (cont.)

```
// allocates a global task queue; it is a collective function
taskq_t * taskq_all_alloc(int, …);

// frees a global task queue; it is a collective function
void taskq_all_free(taskq_t *);

// creates a task using the input arguments and puts it into the task queue
int taskq_put(taskq_t *, void *func, void *in, void *out);

// removes a task from the top of the local task queue and executes it
int taskq_execute(taskq_t *);

// attempts to steal tasks from random victim threads
int taskq_steal(taskq_t *);

// waits tasks that are spawned before it to complete; a blocking operation
void taskq_wait(taskq_t *);

// returns 1 if the task queue is globally empty; it is a collective function
int  taskq_all_isEmpty(bupc_taskq_t *);
```

*This list shows the main APIs. It is not a complete list.

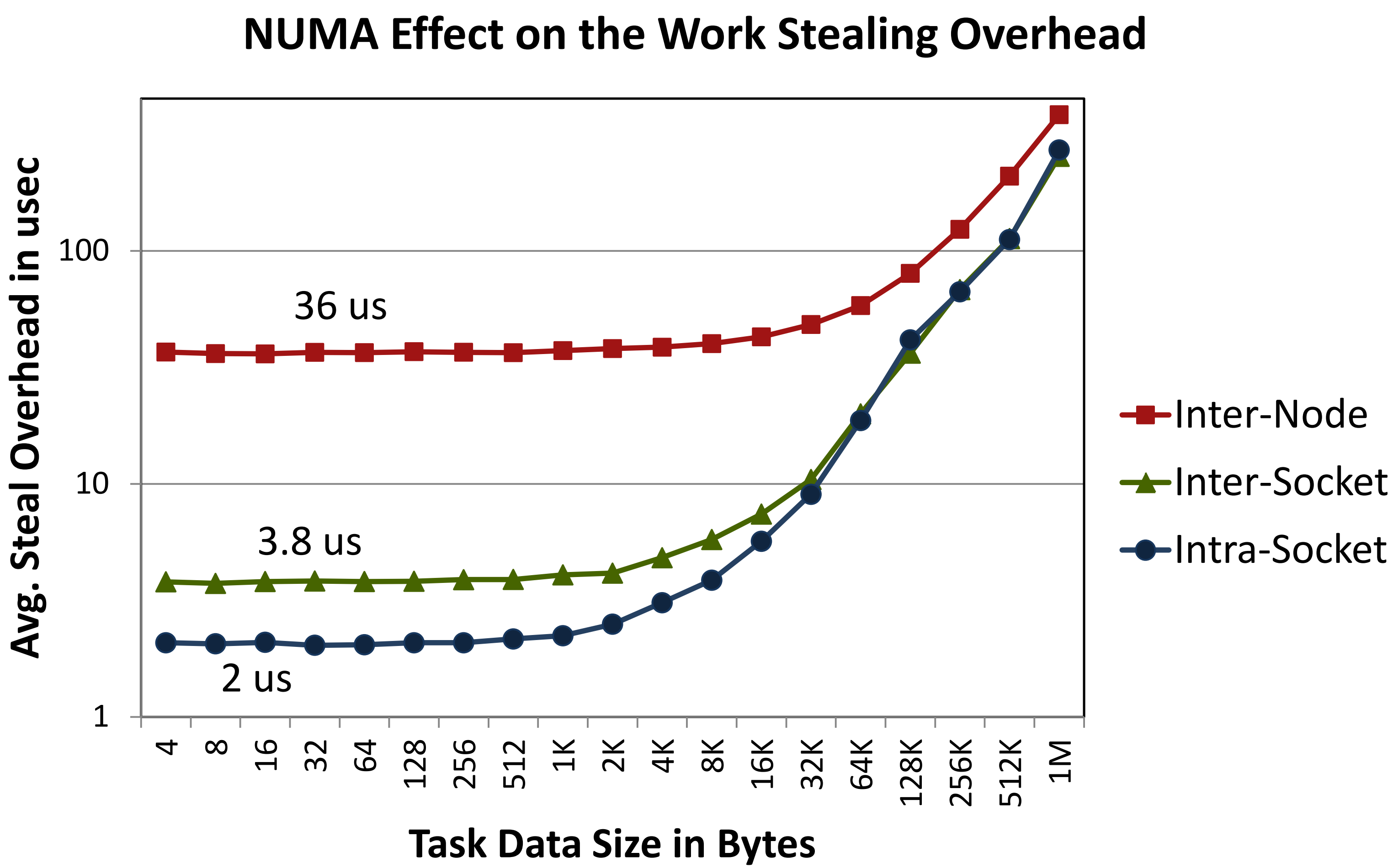
Evaluation Setup

- System
  - Shared-memory machine
    - Two-socket Quad-core Intel Xeon 5530 (Nehalem) 2.4GHz
  - Carver: IBM iDataPlex Distributed-memory system
    - Two Quad-core Intel Xeon 5500 (Nehalem) 2.67 GHz
    - A total of 8 cores per node, connected by 4X QDR InfiniBand

Evaluation Setup (cont.)

- Benchmarks
  - Fibonacci: recursively creates a Fibonacci sequence
  - N-Queens: place N Queens on a NxN chess board
  - Unbalanced Tree Search (UTS): counts nodes in a tree
  - SparseLU: computes LU matrix factorization
- Developed UPC versions using the UPC Task library
- OpenMP implementations
  - BOTS (Barcelona OpenMP Task Suites): Fib, NQ, SparseLU
  - UTS from UTS1-1 distribution website

Work Stealing Overhead

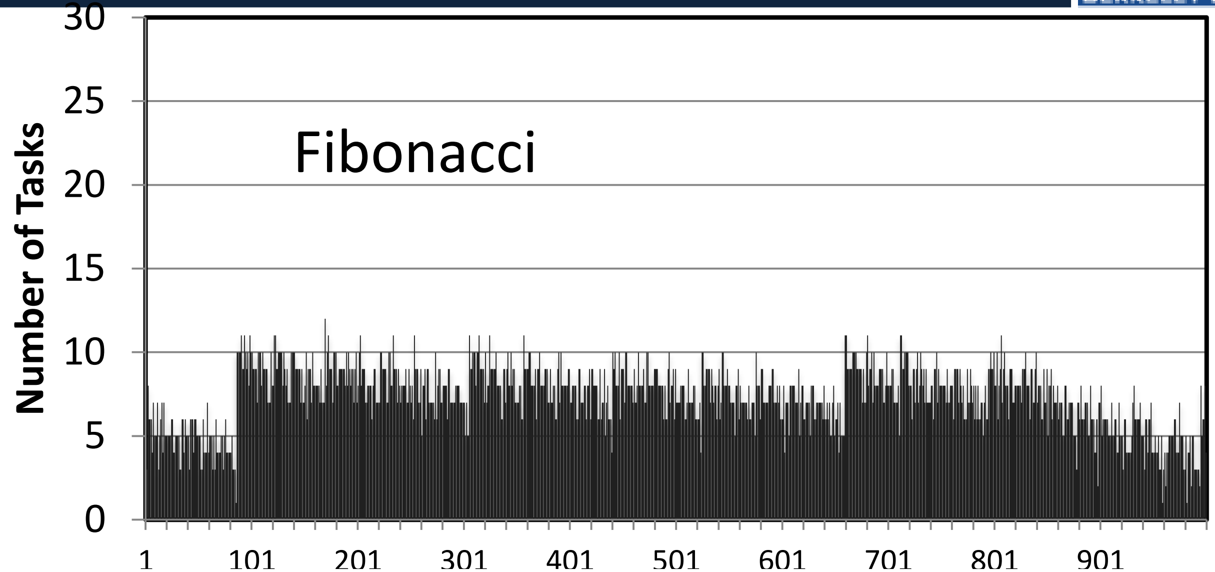


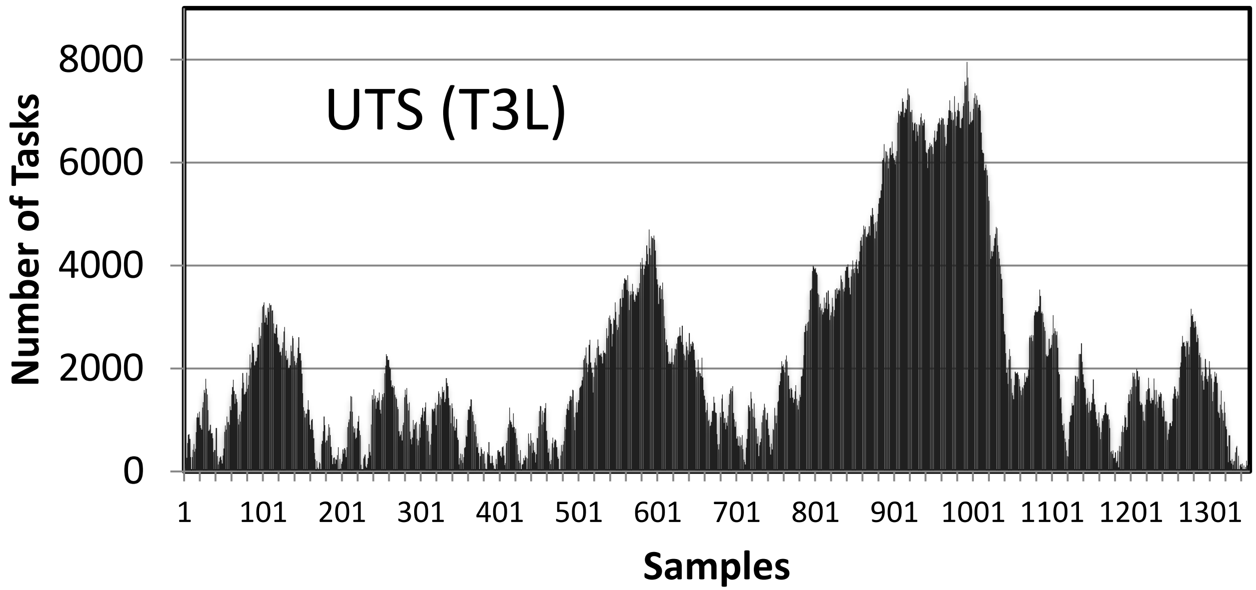
Average time to steal an empty task with varying input argument size on IBM iDataPlex

Task Queue Behavior

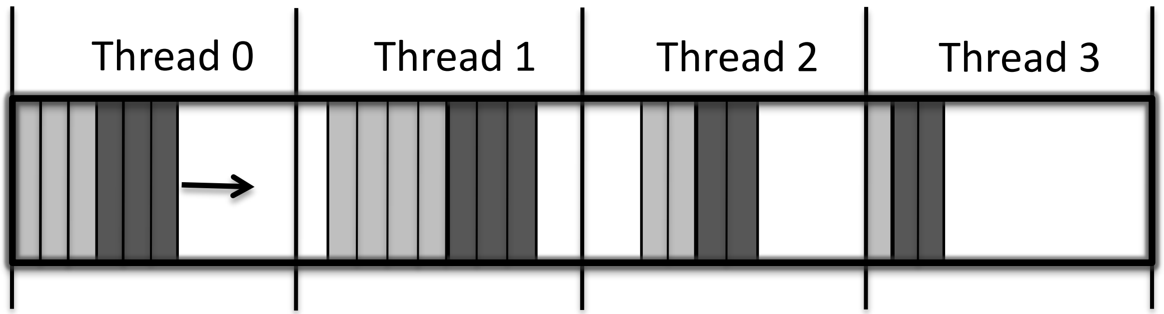
Unbounded task queue with help-first

Sampled a random task queue every 1000 taskq_put



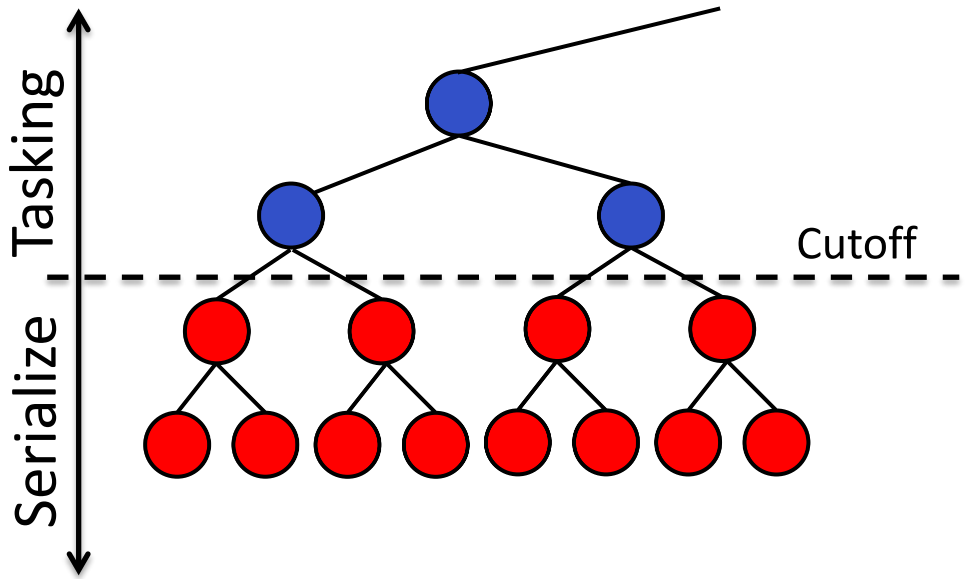


Bounded Queue



- Static memory allocation for task queue management
- Simple implementation and guaranteed memory bound
- This approach fits well with practical optimization goal:
  - Generating work and parallelism at application startup using help-first, then switching to work-first and executing tasks inline to avoid task creation and manipulation overhead

Tree-Depth Cutoff Serialization



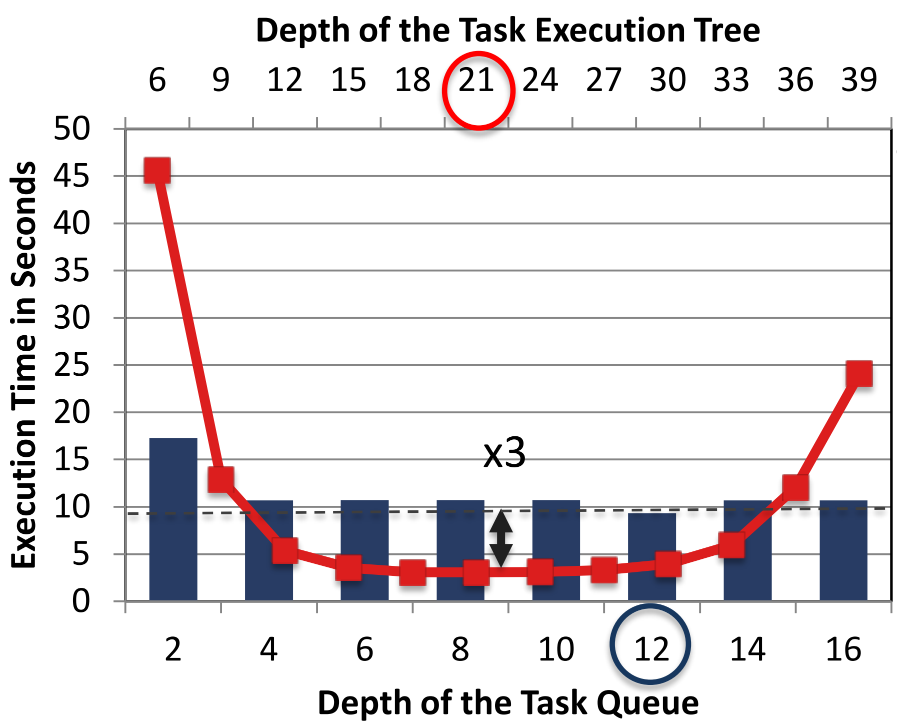
+ Good for structured task tree

- Works only for recursion tree style, but not for parallel-for style parallelism

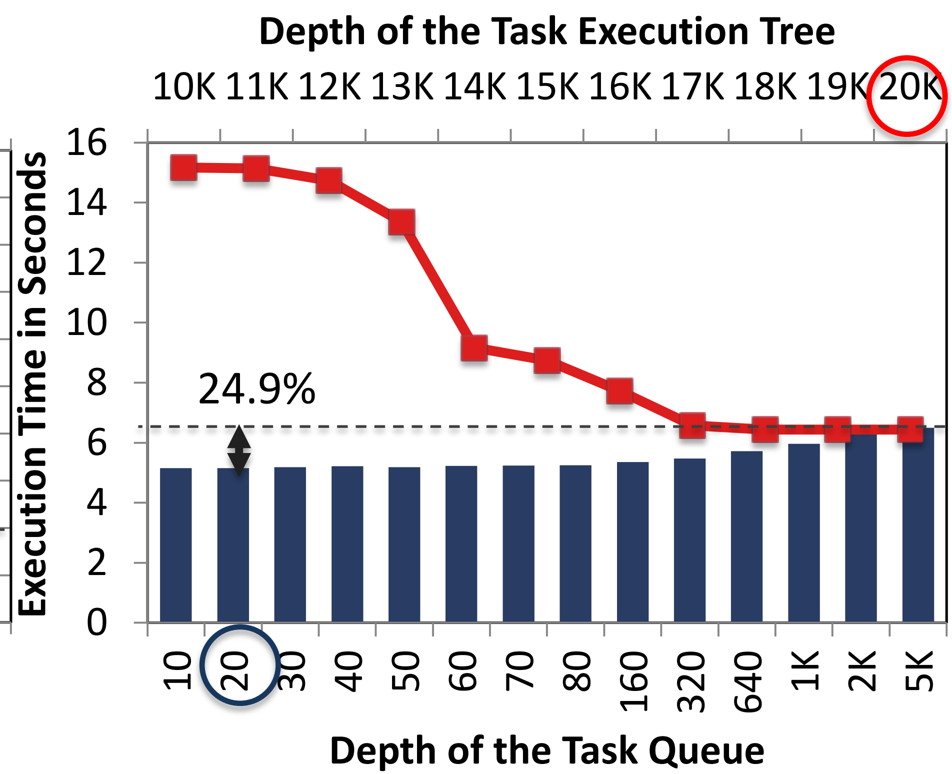
- Can prematurely serialize a large sub-tree

Cut-off Serialization

Fibonacci

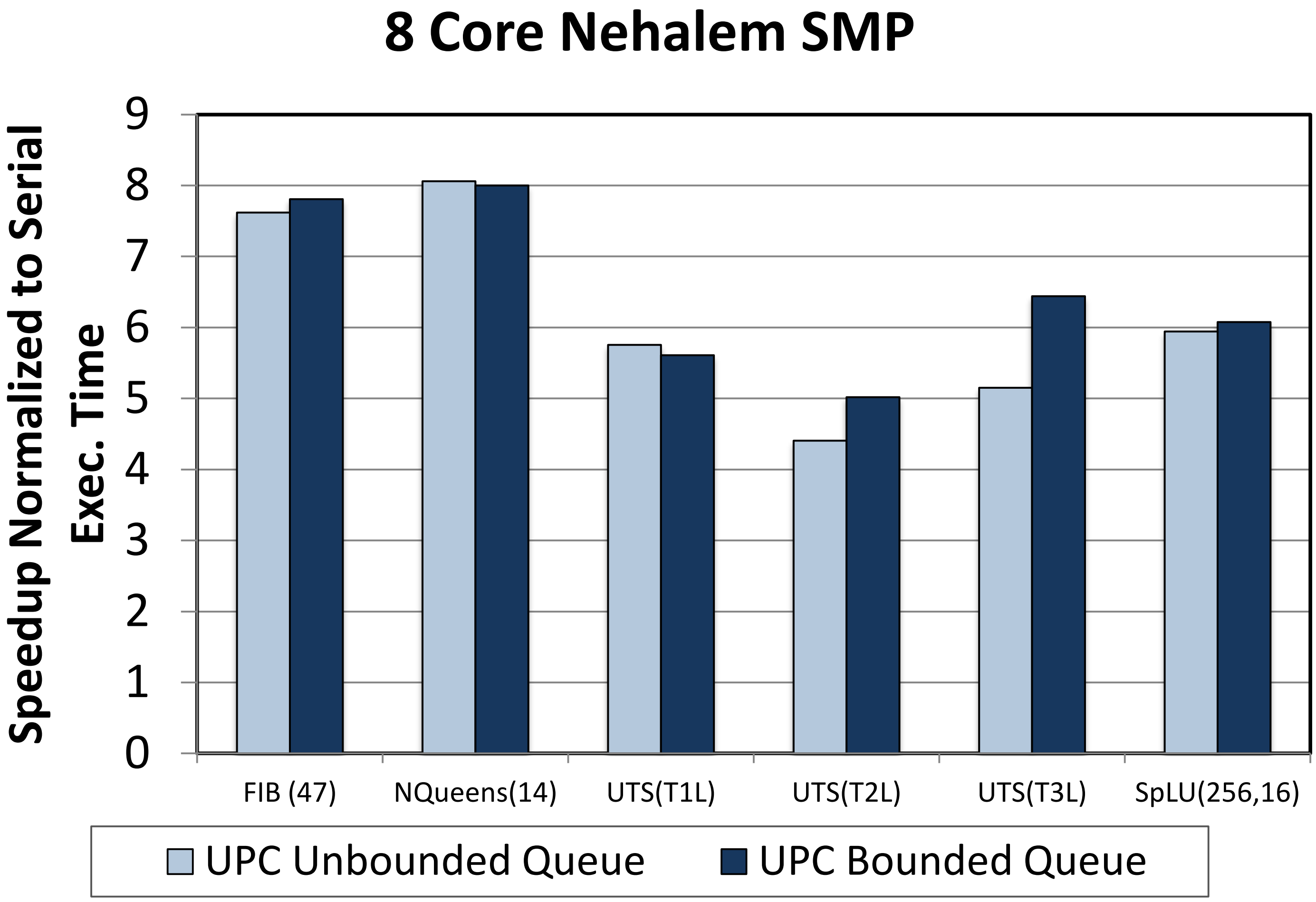


UTS (T3L)



Tree-Depth Cutoff   Bounded Queue

Performance of Bounded-Queue

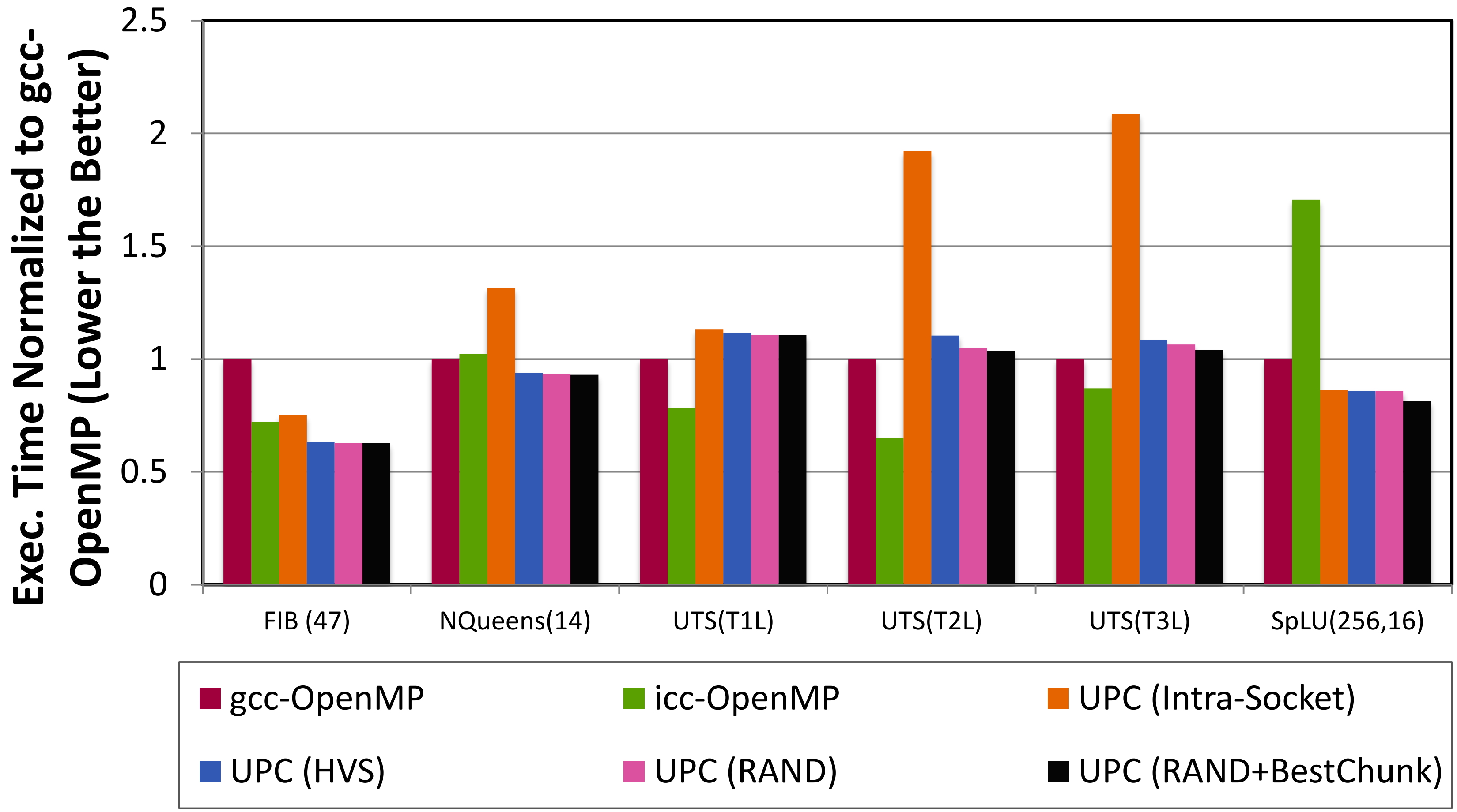


Both UPC versions are optimized with tree-depth cutoff serialization (except SparserLU)
Bounding the queues provides additional performance improvements up to 18%

Benchmark Characteristics

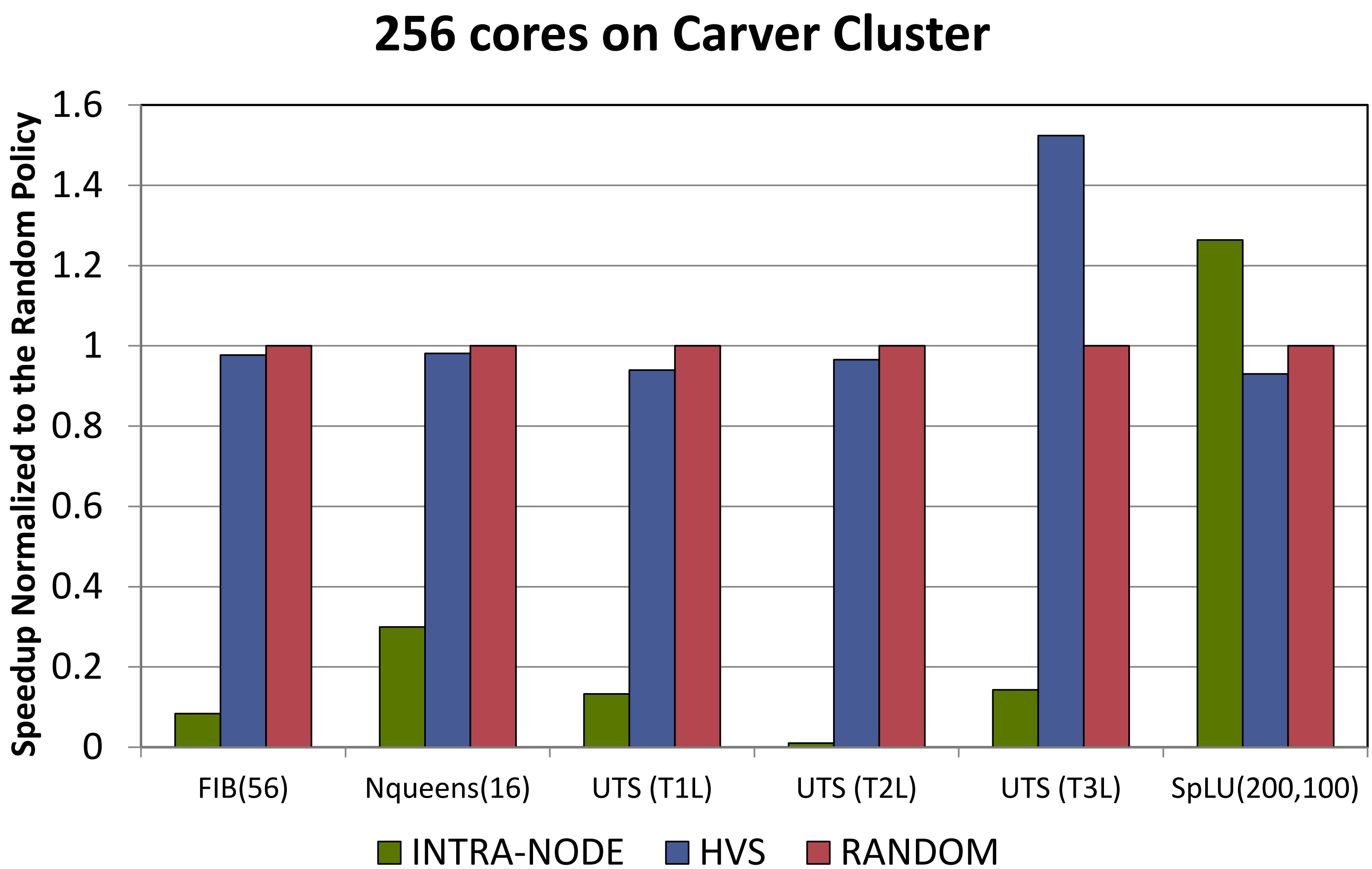
Benchmark	Tasks Created	Avg. Task Time	Input / Output Size (bytes)	Task Creation Ovhd	Steal count	Tasks Serialized
Fibonacci	2,692,537	1.163 us	4 / 8	0.172 us	95	258,928 (8.7%)
N-Queens	306,719	23.270 us	80 / 4	0.174 us	47	129,012 (29.6%)
UTS (T1L)	102,181,082	0.089 us	32/ 0	0.162 us	485	93,553,030 (47.8%)
UTS (T2L)	96,793,510	0.114 us	32/ 0	0.161 us	378	82,249,556 (45.9%)
UTS (T3L)	111,345,631	0.075 us	32/ 0	0.159 us	46703	108,983,482 (49.4%)
SparseLU	1,430,912	6.281 us	16,16,24/ 0	0.166 us	2320	1,344,733 (48.4%)

Victim Selection Policies on SMP

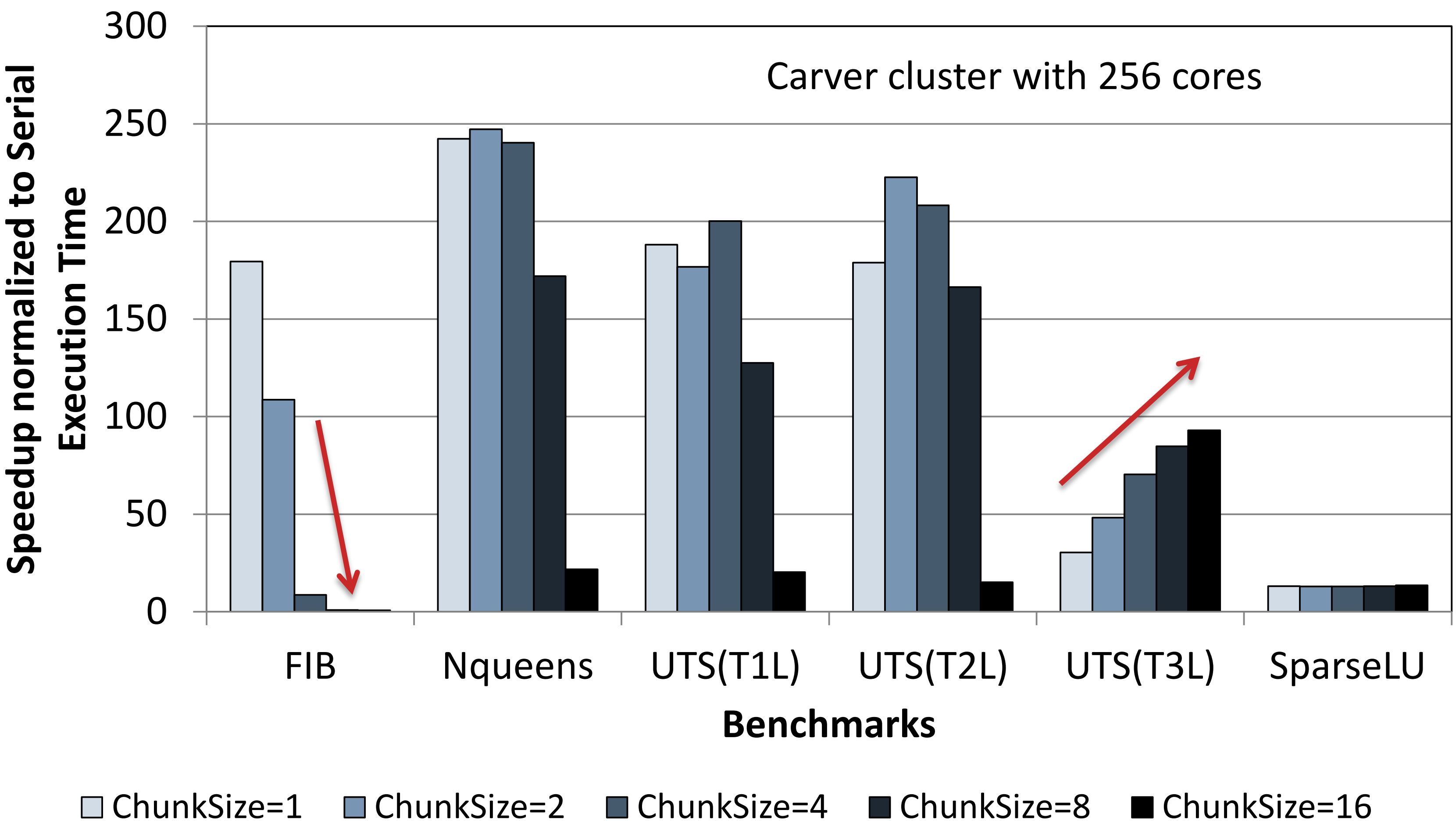


All UPC versions use fixed chunk size of 1, except the UPC (RAND+BestChunk) uses the best fixed-chunk sizes searched

Victim Selection Policies

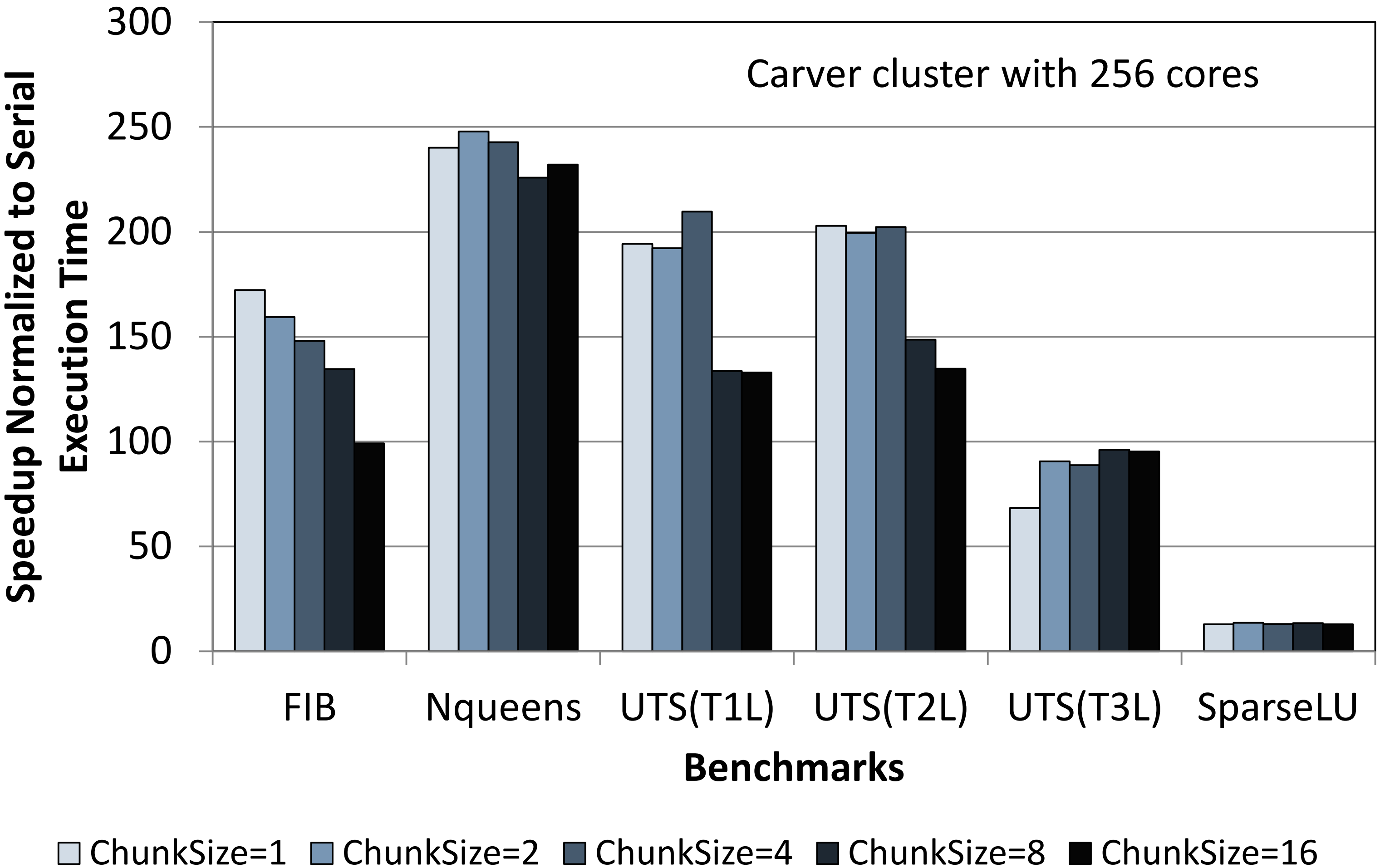


Fixed Chunk Selection



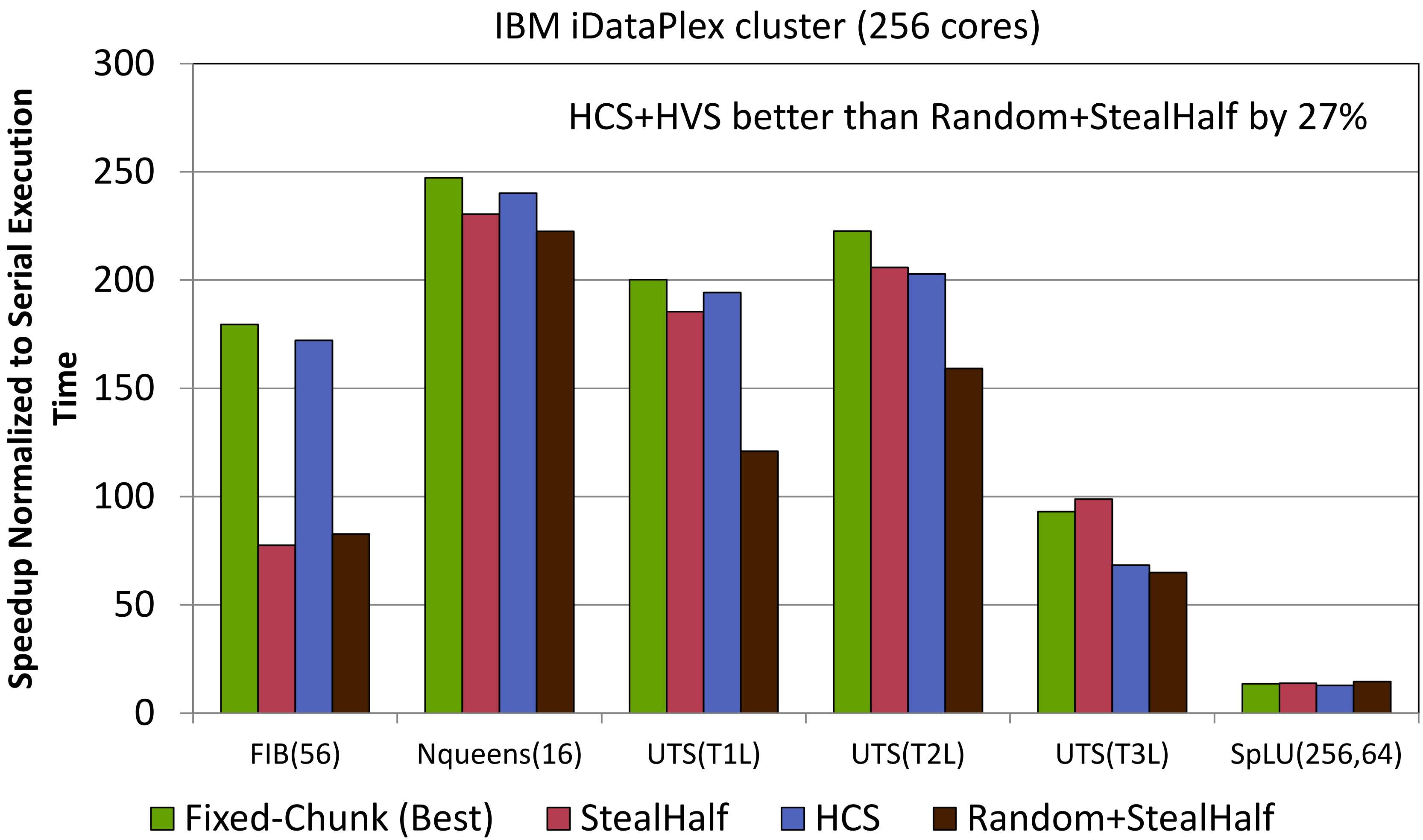
Performance drops drastically except UTS (T3L) and SparseLU

Hierarchical Chunk Selection



Robust performance on the chunk sizes variations

Chunk Selection Policy



Fixed-Chunk (Best), StealHalf, and HCS use HVS, while Random+StealHalf uses random

Fixed-Chunk (Best): FIB, Nqueens, UTS(T1L), and UTS(T2L) StealHalf: UTS (T3L)

Conclusion

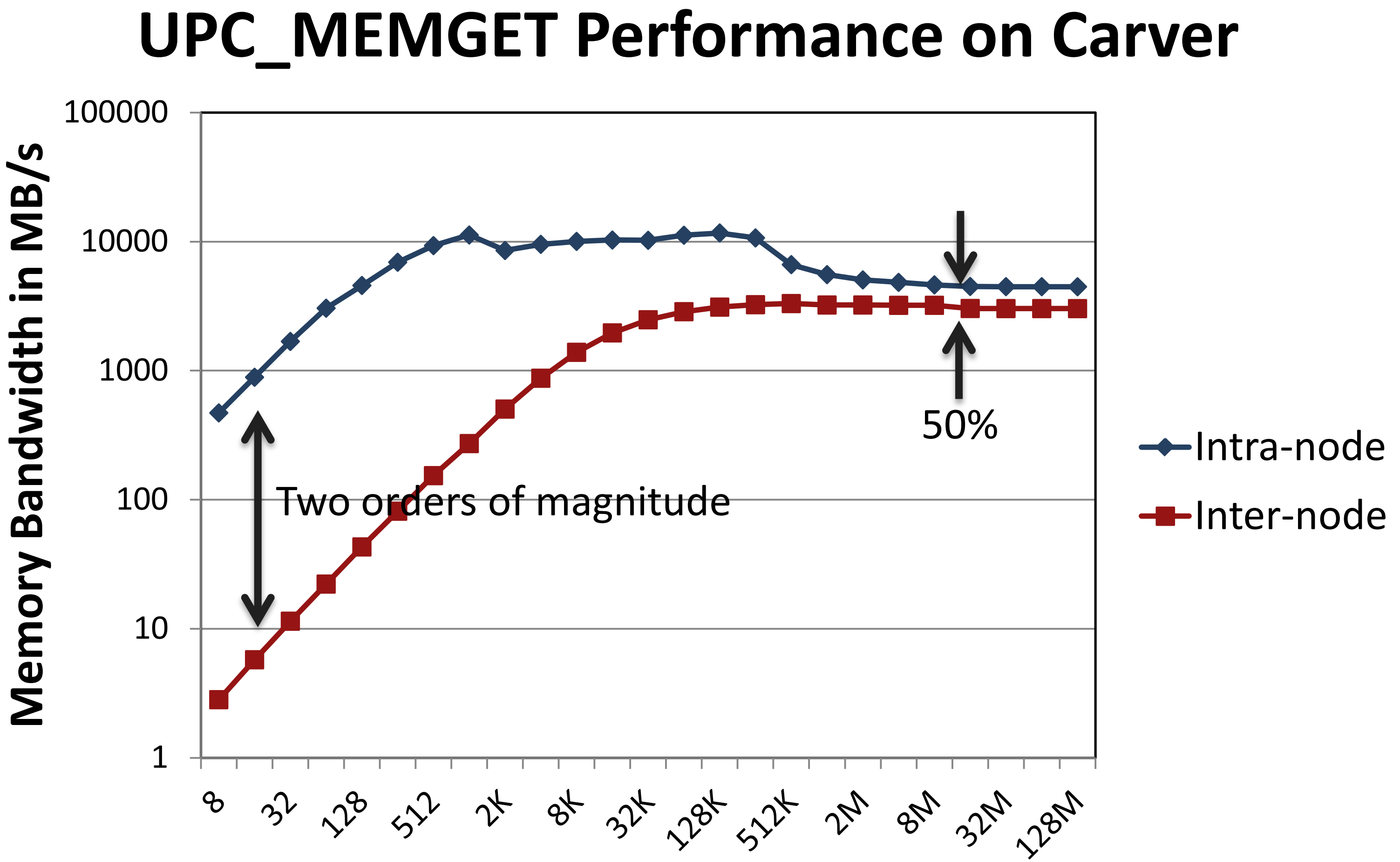
- HotSLAW: a dynamic tasking library for the Unified Parallel C (UPC) programming language.
- HotSLAW provides a simple and effective way of adding task parallelism to SPMD programs
- HotSLAW implements Bounded Queue
- To exploit locality, we presented two hierarchical work-stealing optimization techniques: HVS and HCS
- *Hierarchical victim selection (HVS)* steals work from the nearest available victims to preserve locality
- *Hierarchical chunk selection (HCS)* dynamically determines the amount of work to steal based on the locality of the victim thread

Conclusion (cont.)

- We evaluated HotSLAW performance on both shared- and distributed-memory architectures
- On shared-memory systems, HotSLAW provides performance comparable to manually optimized OpenMP implementations
- On distributed-memory systems:
  - HVS improves performance by up to 52% when compared to the default random selection
  - HCS improves performance by up to 122% compared to the StealHalf method
  - The combination of HVS and HCS enables HotSLAW to achieve 27% better performance than the state-of-the-art approach using random victim selection and HalfSteal strategy

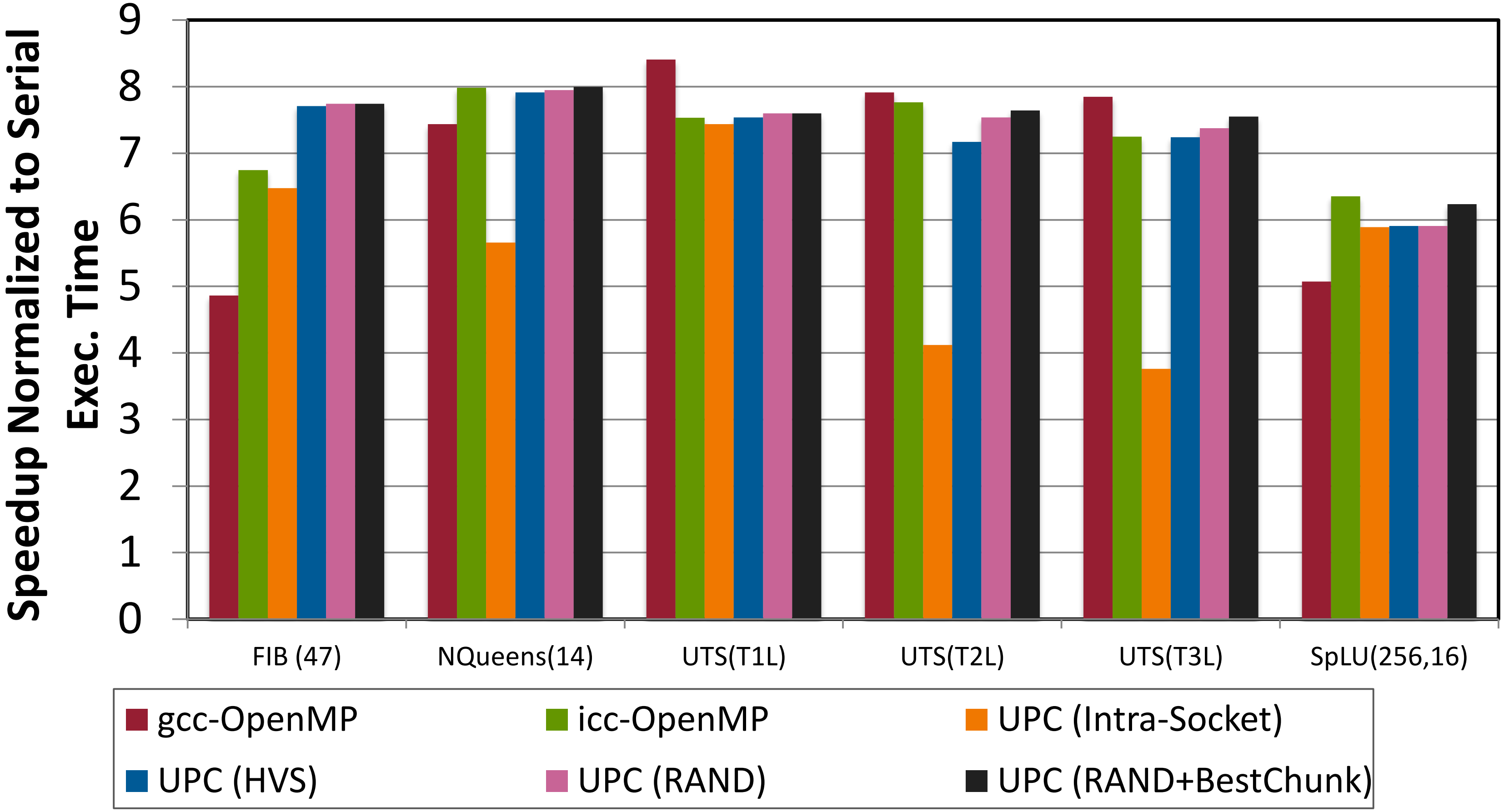
Thank You

Work Stealing Overhead (cont.)



Memory bandwidth on the IBM iDataPlex cluster. Intra-node measures the inter-socket bandwidth and inter-node measures the InifiniBand bandwidth

Victim Selection Policies on SMP



All UPC versions use fixed chunk size of 1, except the UPC (RAND+BestChunk) uses the best chunk sizes searched