Scheduling Task Parallelism on Multi-Socket Multicore Systems

Stephen Olivier, UNC Chapel Hill
Allan Porterfield, RENCI
Kyle Wheeler, Sandia National Labs
Jan Prins, UNC Chapel Hill

The University of North Carolina at Chapel Hill

Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks

Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

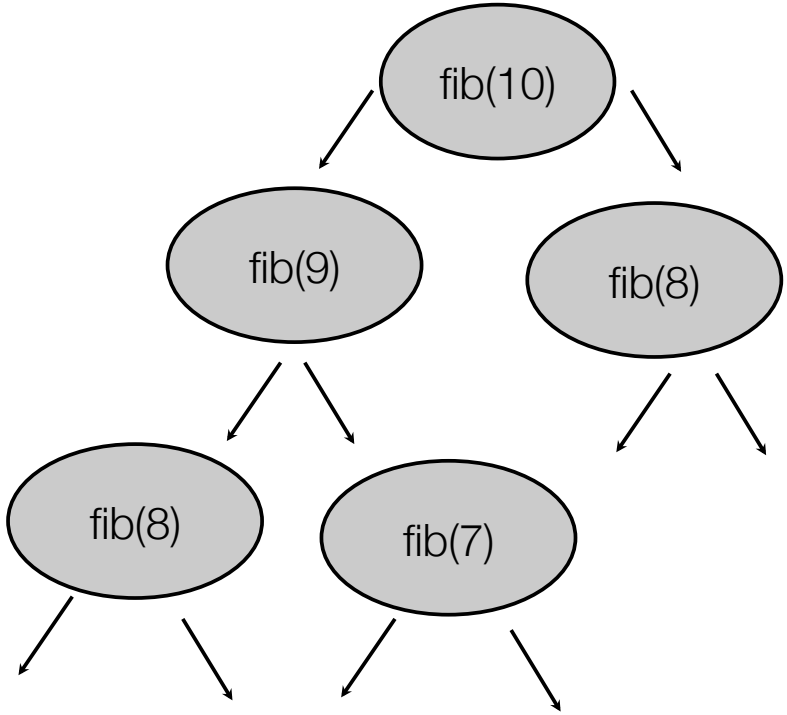
Closing Remarks

Task Parallel Programming in a Nutshell

- A task consists of executable code and associated data context, with some bookkeeping metadata for scheduling and synchronization.
- Tasks are significantly more lightweight than threads.
  - Dynamically generated and terminated at run time
  - Scheduled onto threads for execution
- Used in Cilk, TBB, X10, Chapel, and other languages
  - Our work is on the recent tasking constructs in OpenMP 3.0.

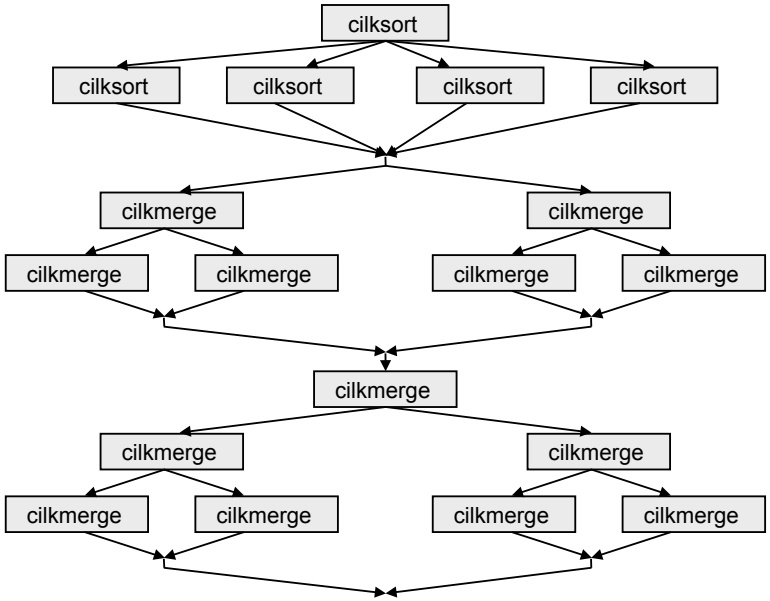
Simple Task Parallel OpenMP Program: Fibonacci

```
int fib(int n)
{
   int x, y;
   if (n < 2) return n;

   #pragma omp task
      x = fib(n - 1);
   #pragma omp task
      y = fib(n - 2);

   #pragma omp taskwait

   return x + y;
}
```



Useful Applications

- Recursive algorithms
  - E.g. Mergesort
  - List and tree traversal
- Irregular computations
  - E.g., Adaptive Fast Multipole
- Parallelization of while loops



- Situations where programmers might otherwise write a difficult-to-debug low-level task pool implementation in pthreads

Goals for Task Parallelism Support

- Programmability
  - Expressiveness for applications
  - Ease of use
- Performance & Scalability
  - Lack thereof is a serious barrier to adoption
  - Must improve software run time systems

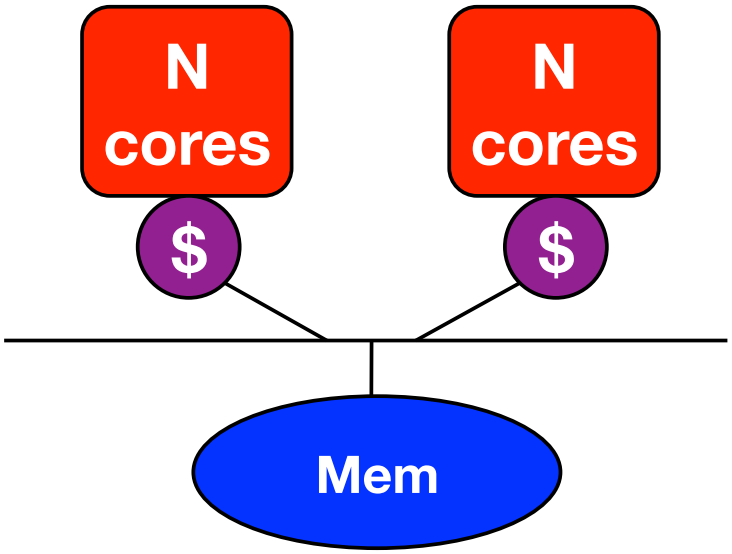
Issues in Task Scheduling

- Load Imbalance
  - Uneven distribution of tasks among threads
- Overhead costs
  - Time spent creating, scheduling, synchronizing, and load balancing tasks, rather than doing the actual computational work
- Locality
  - Task execution time depends on the time required to access data used by the task

The Current Hardware Environment

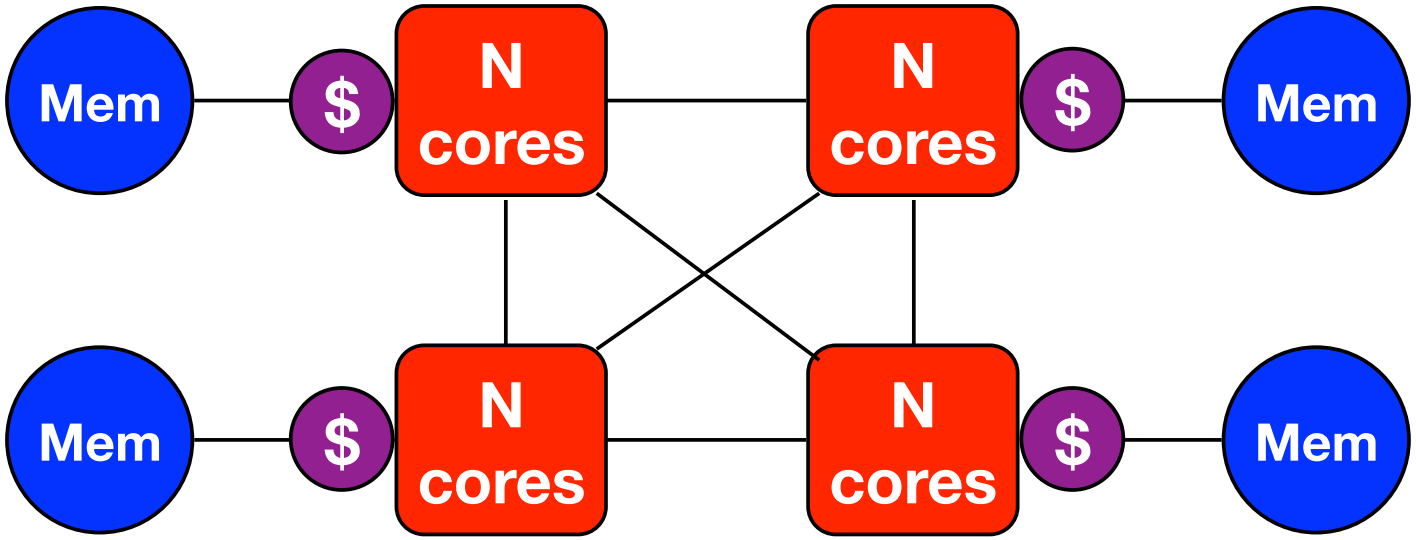
- Shared Memory is not a free lunch.
  - Data can be accessed without explicitly programmed messages as in MPI, but not always at equal cost.
- However, OpenMP has traditionally been agnostic toward affinity of data and computation.
  - Most vendors have (often non-portable) extensions for thread layout and binding.
  - First-touch traditionally used to distribute data across memories on many systems.

Example UMA System



- Incarnations include Intel server configurations prior to Nehalem and the Sun Niagara systems
- Shared bus to memory

Example Target NUMA System



- Incarnations include Intel Nehalem/Westmere processors using QPI and AMD Opterons using HyperTransport.
- Remote memory accesses are typically higher latency than local accesses, and contention may exacerbate this.

Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks

Work Stealing

- Studied and implemented in Cilk by Blumofe et al. at MIT
- Now used in many task-parallel run time implementations
- Allows dynamic load balancing with low critical path overheads since idle threads steal work from busy threads
- Tasks are enqueued and dequeued LIFO and stolen FIFO for exploitation of local caches
- Challenges
  - Not well suited to shared caches now common in multicore chips
  - Expensive off-chip steals in NUMA systems

PDFS (Parallel Depth-First Schedule)

- Studied by Blelloch et al. at CMU
- Basic idea: Schedule tasks in an order close to serial order
- If sequential execution has good cache locality, PDFS should as well.
- Implemented most easily as a shared LIFO queue
- Shown to make good use of shared caches
- Challenges
  - Contention for the shared queue
  - Long queue access times across chips in NUMA systems

Our Hierarchical Scheduler

- Basic idea: Combine benefits of work stealing and PDFS for multi-socket multicore NUMA systems
- Intra-chip shared LIFO queue to exploit shared L3 cache and provide natural load balancing among local cores
- FIFO work stealing between chips for further low overhead load balancing while maintaining L3 cache locality
  - Only one thief thread per chip performs work stealing when the on-chip queue is empty
  - Thief thread steals enough tasks, if available, for all cores sharing the on-chip queue

Implementation

- We implemented our scheduler, as well as other schedulers (e.g., work stealing, centralized queue), in extensions to Sandia's Qthreads multithreading library.
- We use the ROSE source-to-source compiler to accept OpenMP programs and generate transformed code with XOMP outlined functions for OpenMP directives and run time calls.
- Our Qthreads extensions implement the XOMP functions.
- ROSE-transformed application programs are compiled and executed with the Qthreads library.

Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks

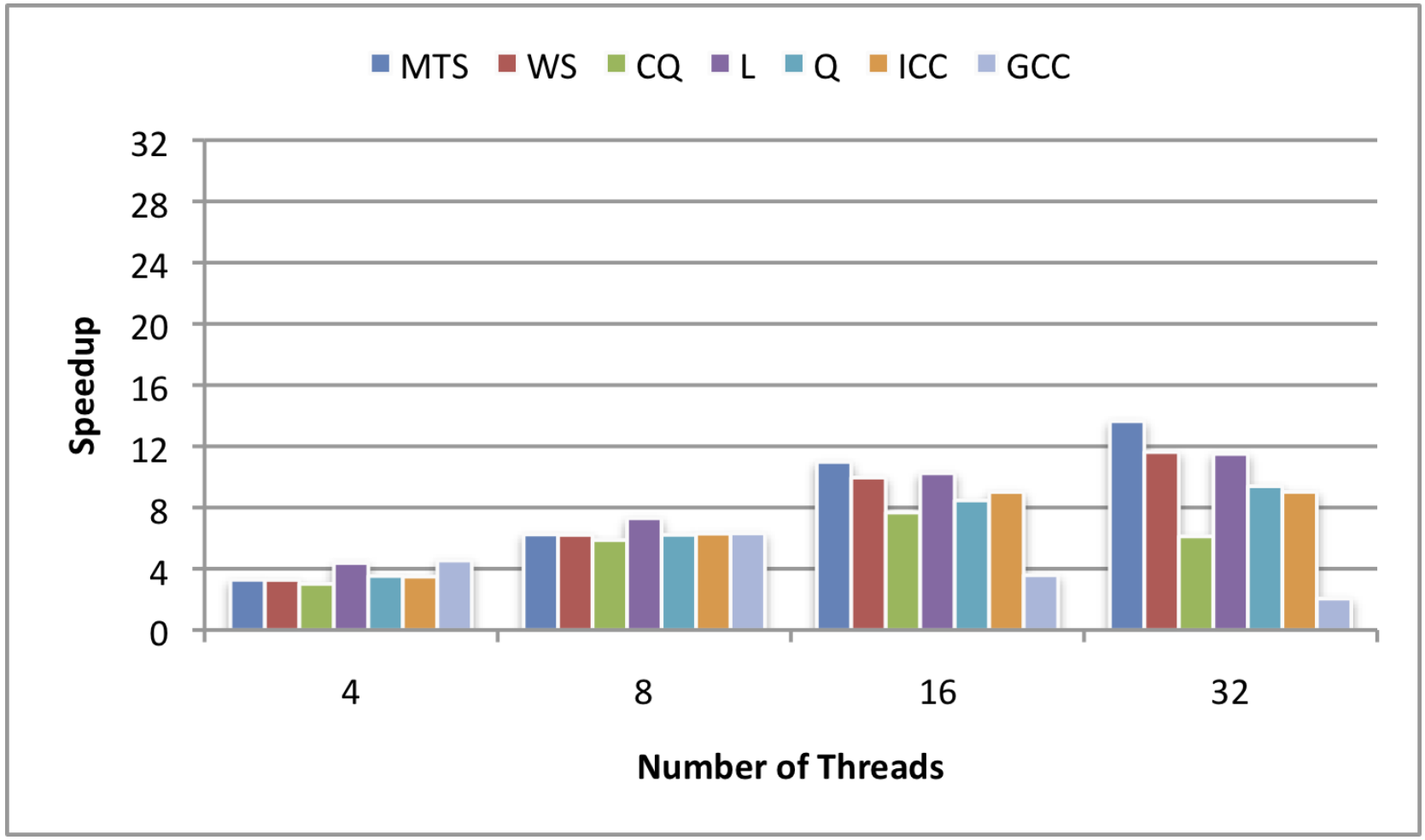
Evaluation Setup

- Hardware: Shared memory NUMA system
  - Four 8-core Intel x7550 chips fully connected by QPI
- Compiler and Run time systems: ICC, GCC, Qthreads
  - Five Qthreads implementations
    - Q: Per-core FIFO queues *with round robin task placement*
    - L: Per-core LIFO queues *with round robin task placement*
    - CQ: Centralized queue
    - WS: Per-core LIFO queues *with FIFO work stealing*
    - MTS: Per-chip LIFO queues *with FIFO work stealing*

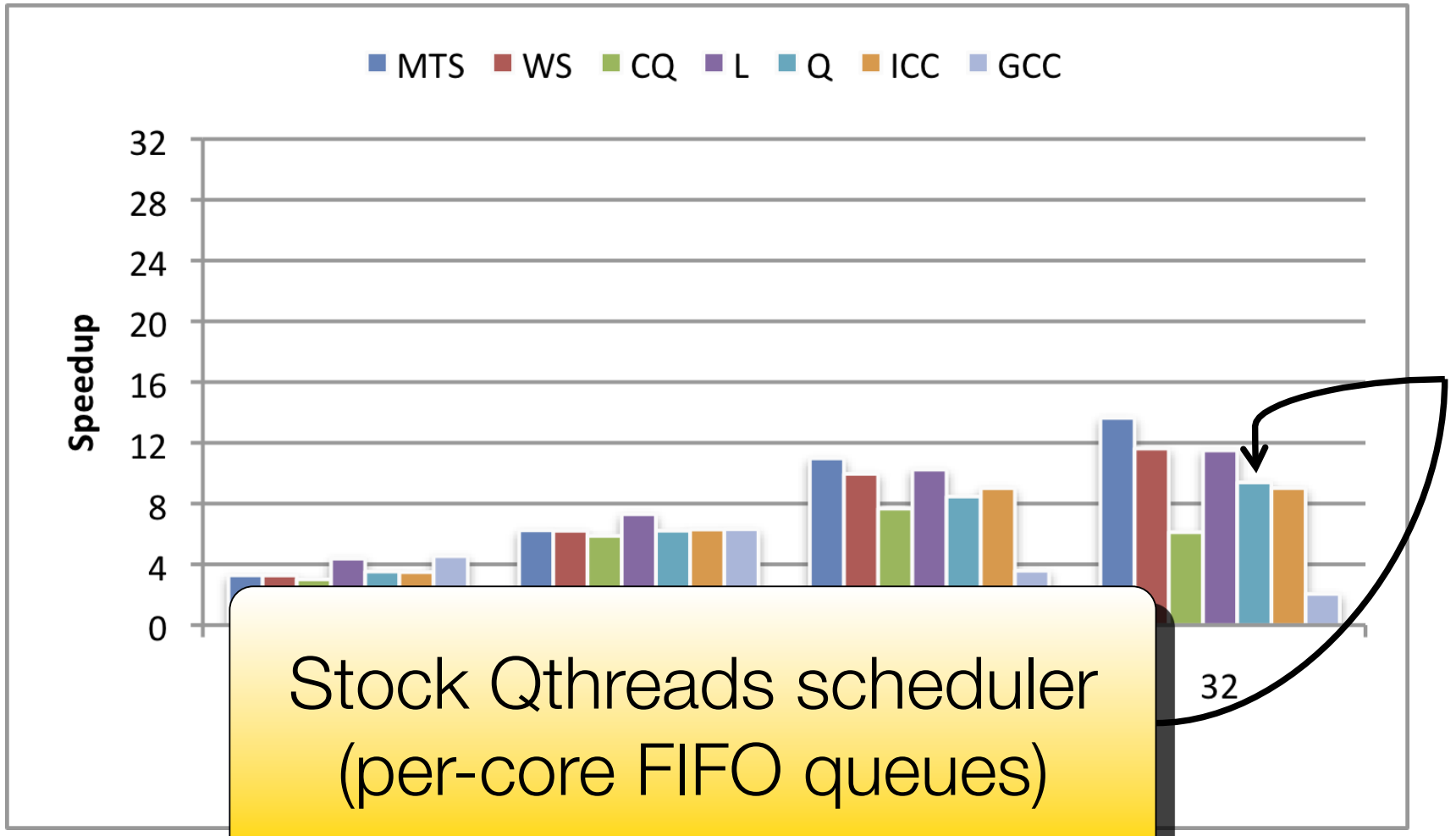
Evaluation Programs

- From the Barcelona OpenMP Tasks Suite (BOTS)
  - Described in ICPP ‘09 paper by Duran et al.
  - Available for download online
- Several of the programs have cut-off thresholds
  - No further tasks created beyond a certain depth in the computation tree

Health Simulation Performance

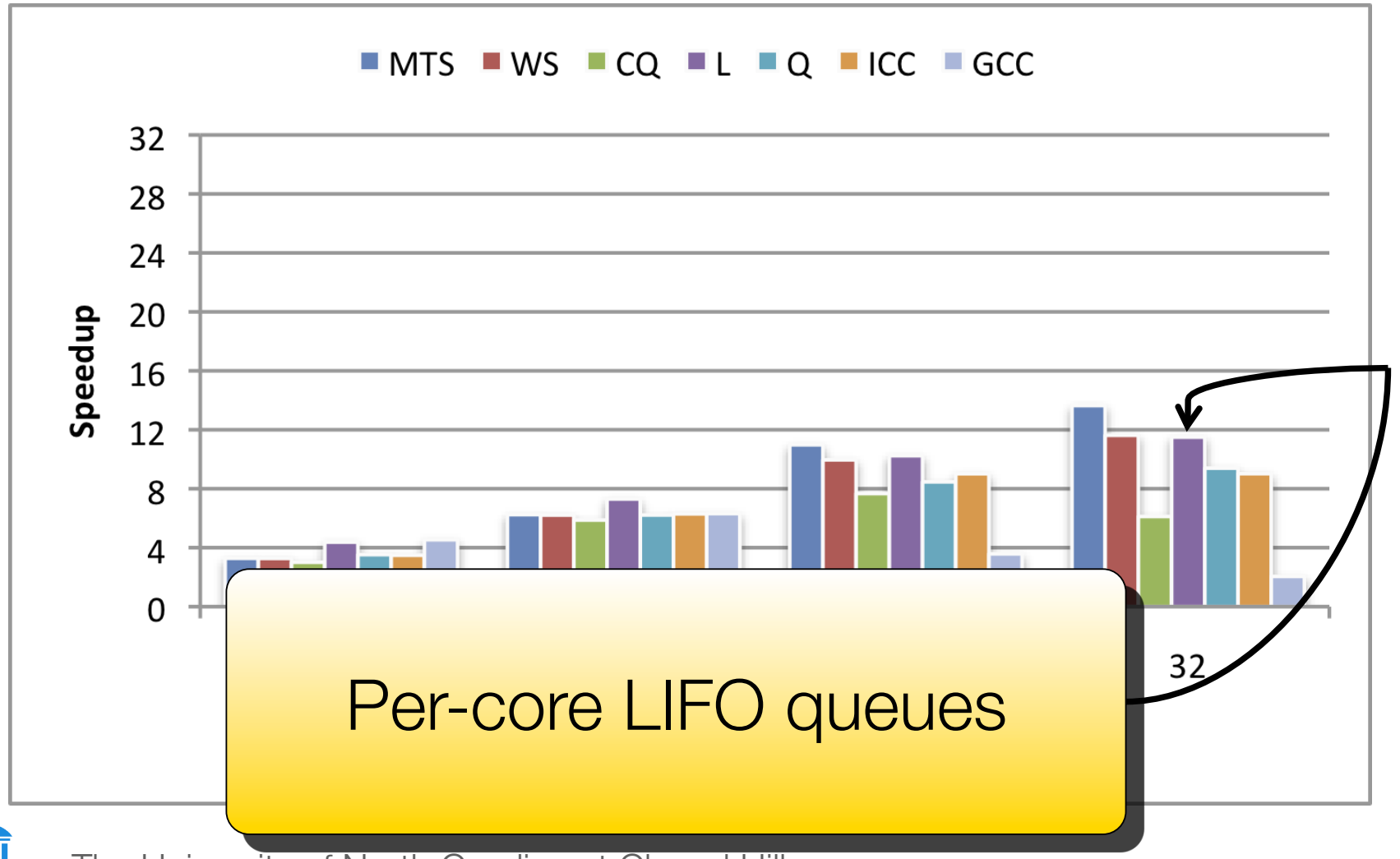


Health Simulation Performance

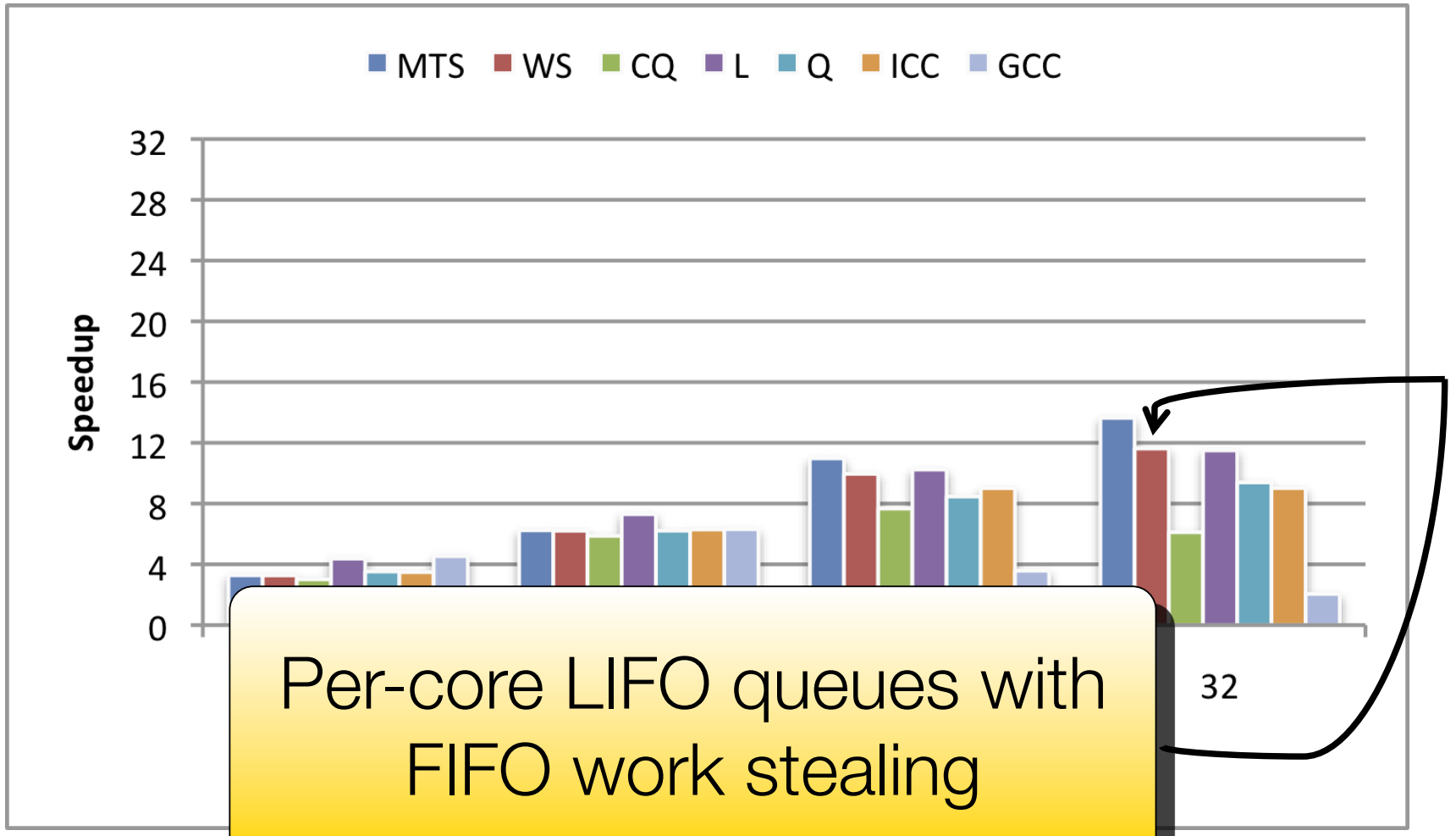


Stock Qthreads scheduler (per-core FIFO queues)

Health Simulation Performance

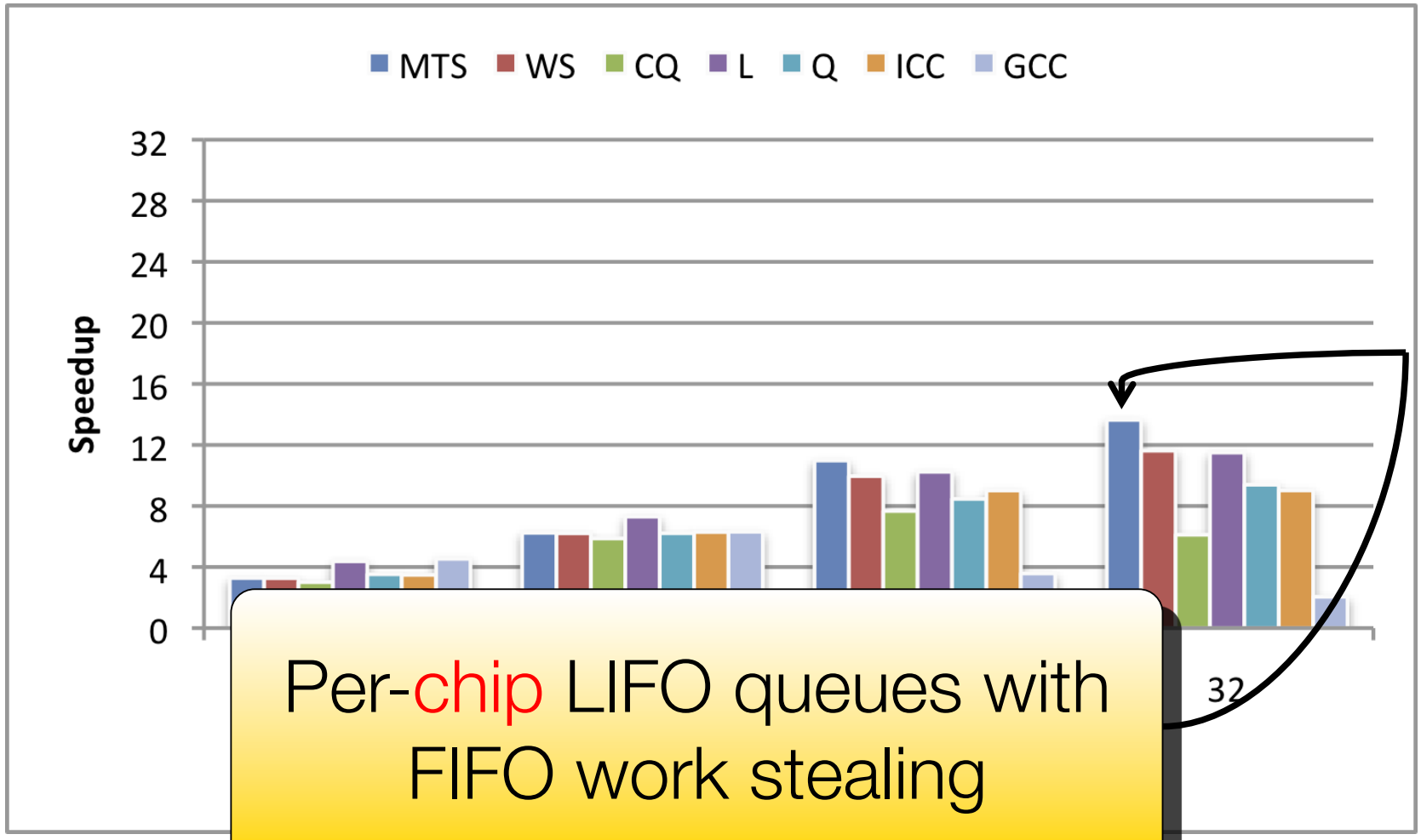


Health Simulation Performance



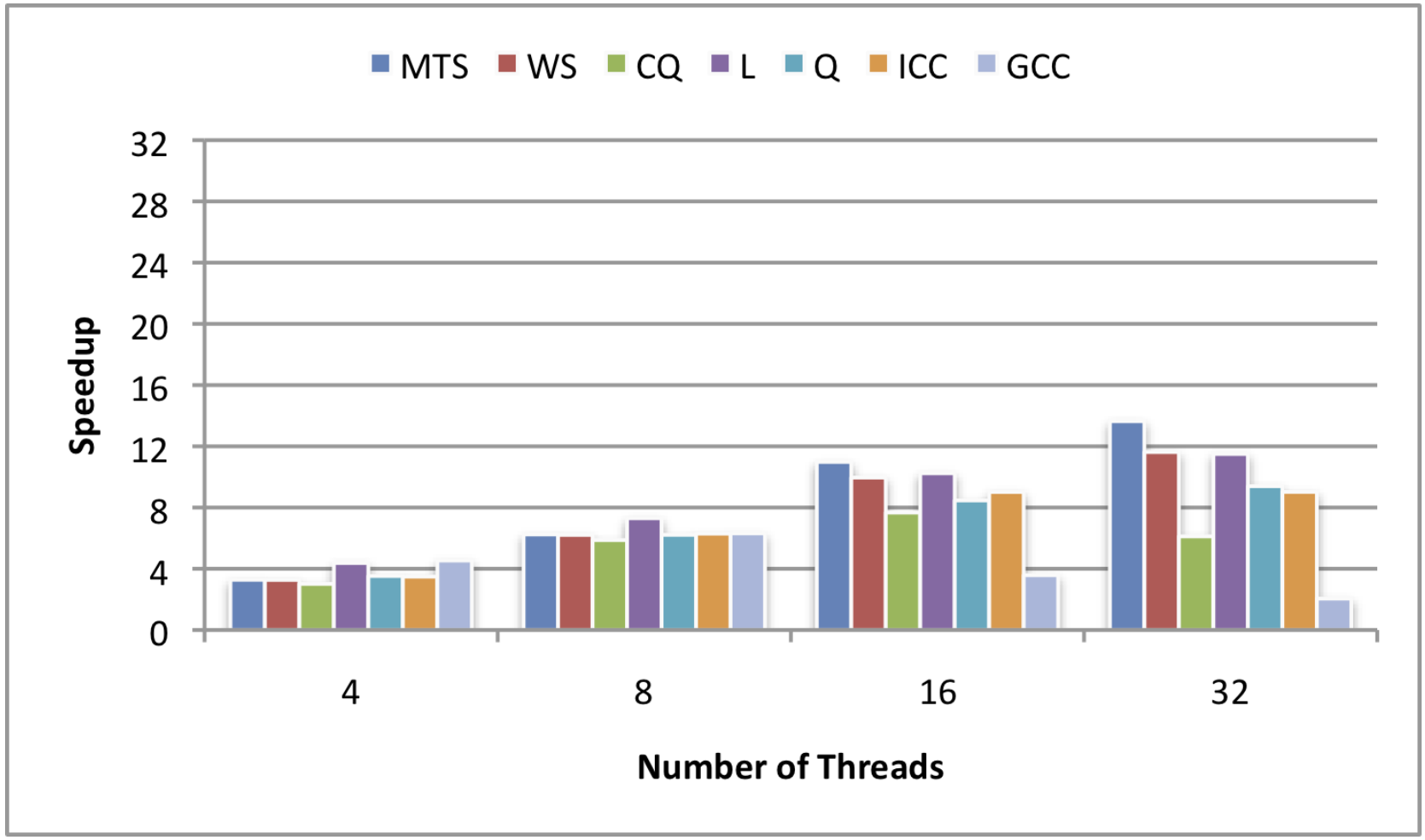
Per-core LIFO queues with FIFO work stealing

Health Simulation Performance

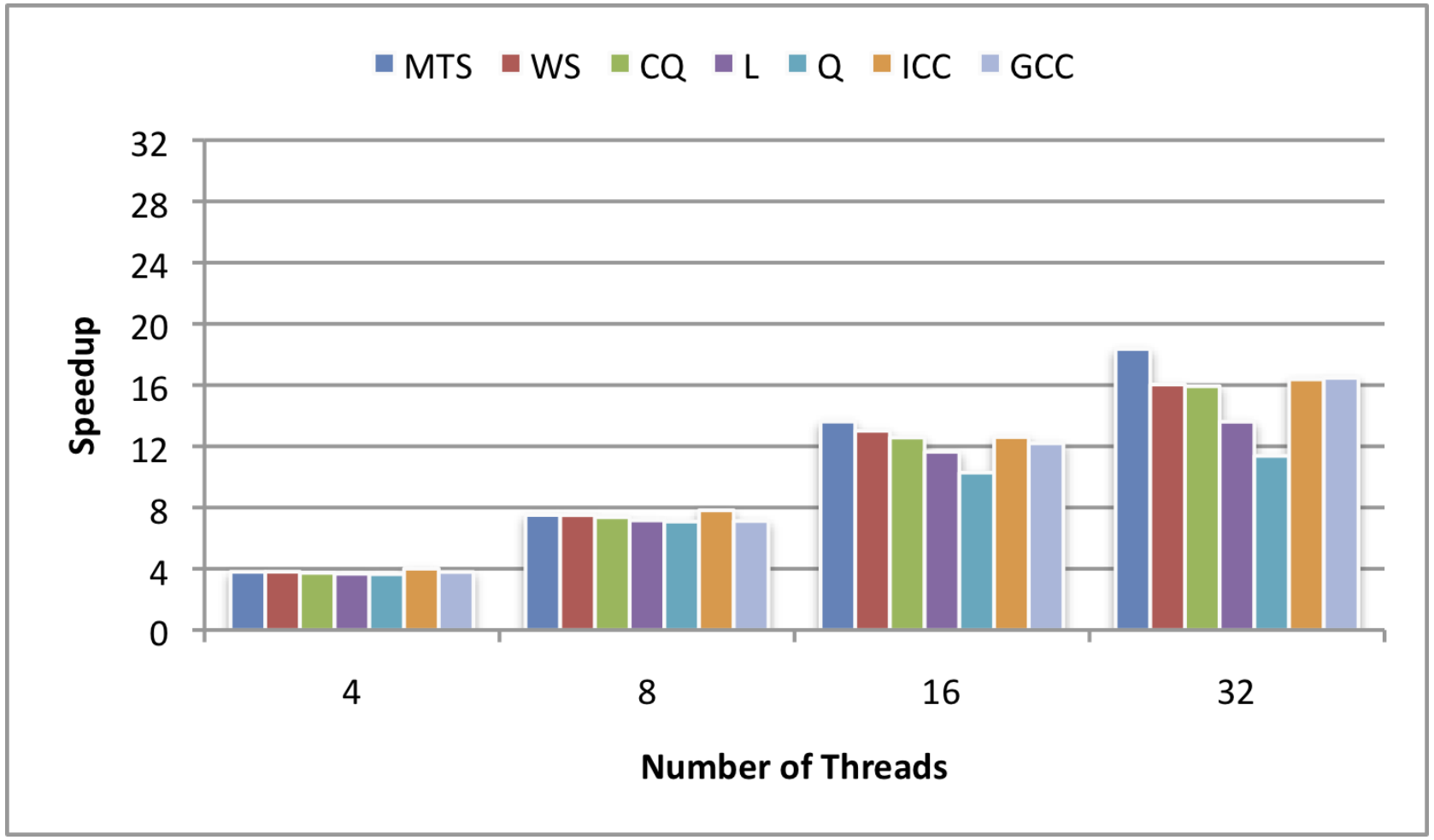


Per-chip LIFO queues with FIFO work stealing

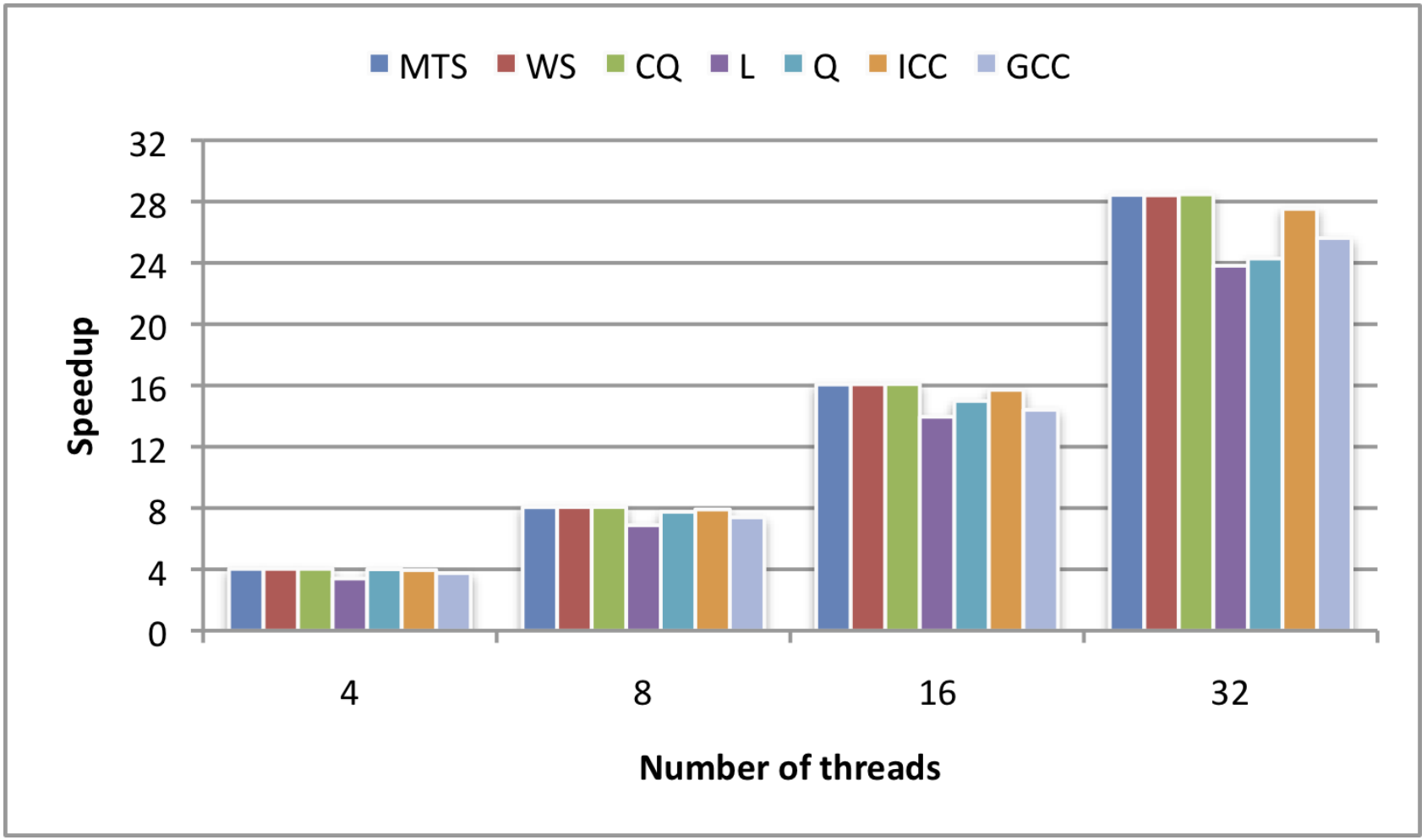
Health Simulation Performance



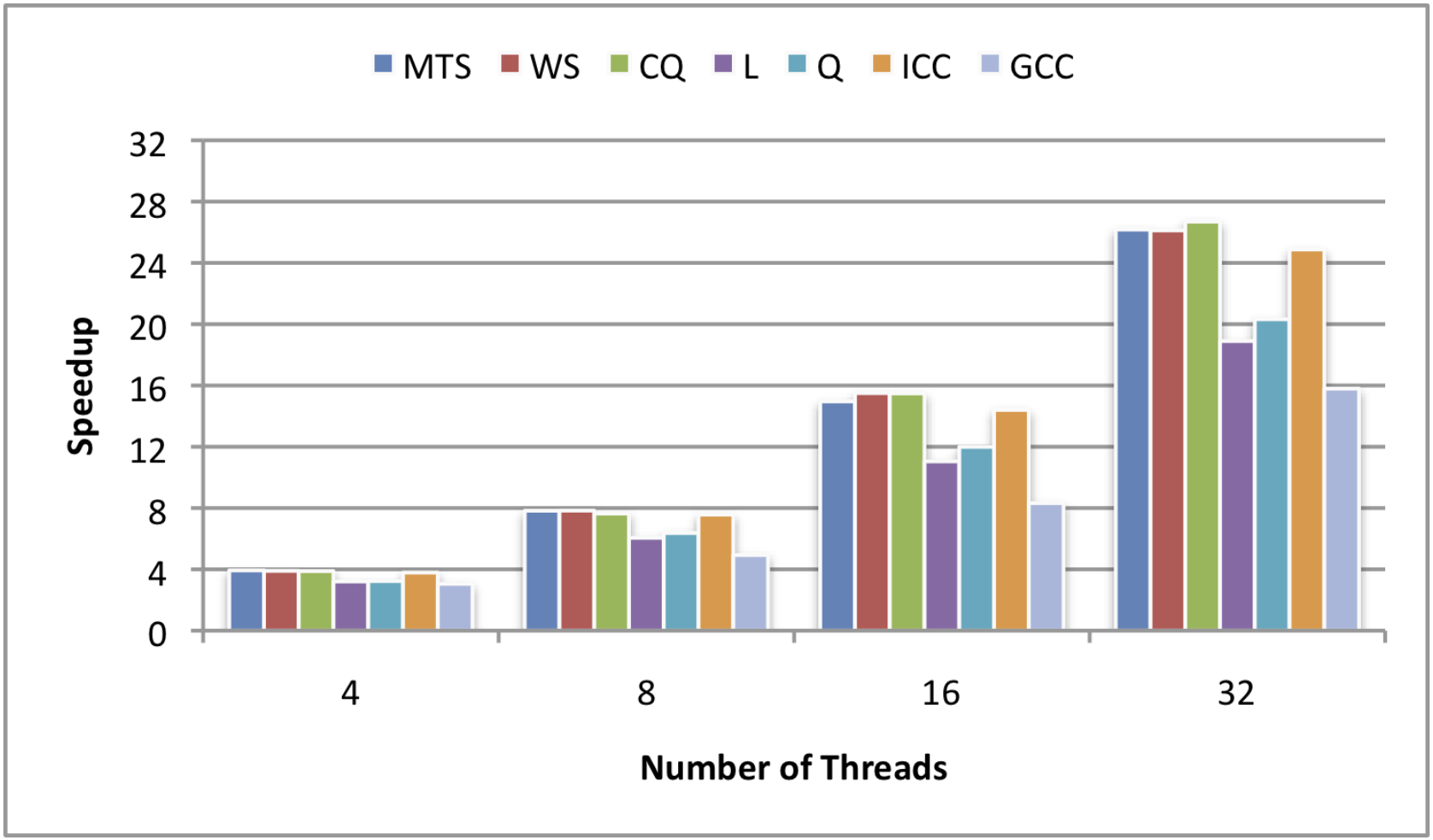
Sort Benchmark



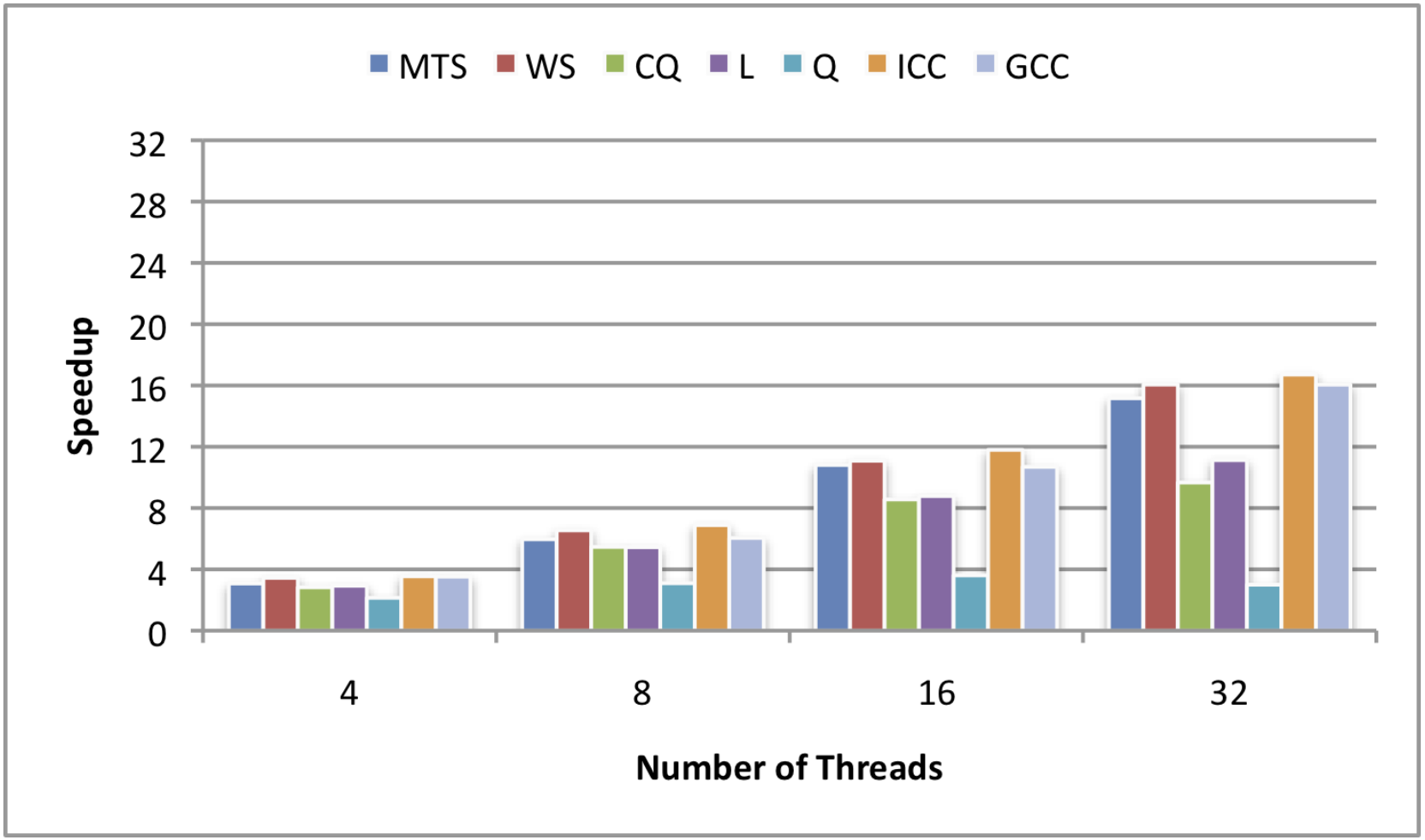
NQueens Problem



Fibonacci

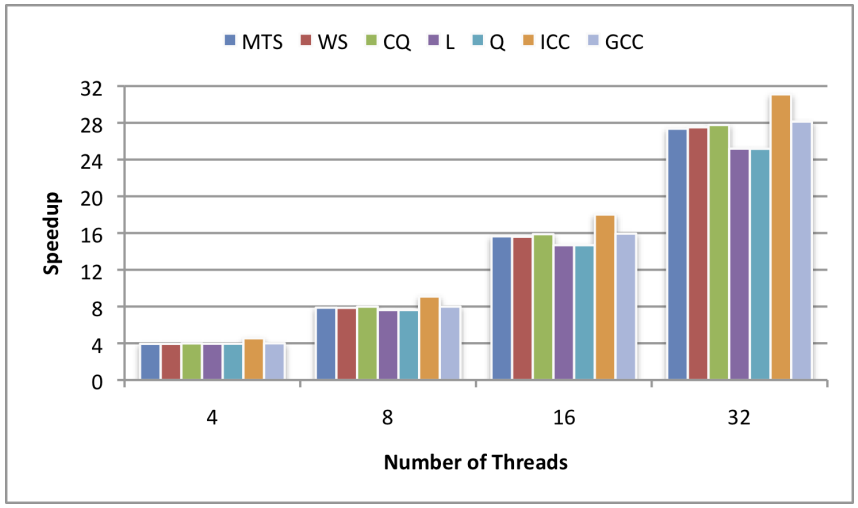

MTS
WS
CQ
L
Q
ICC
GCC
Speedup
32
28
24
20
16
12
8
4
0
4
8
16
32
Number of Threads

Strassen Multiply

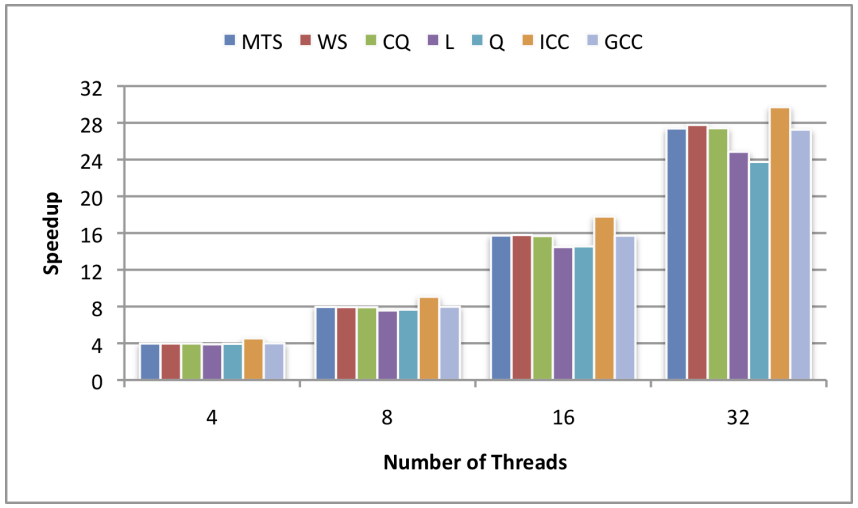


Protein Alignment

Single task startup

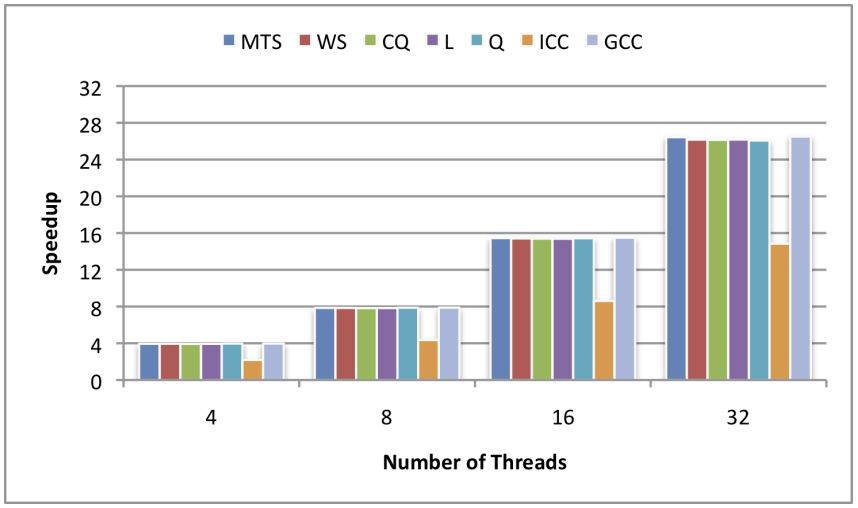


For loop startup

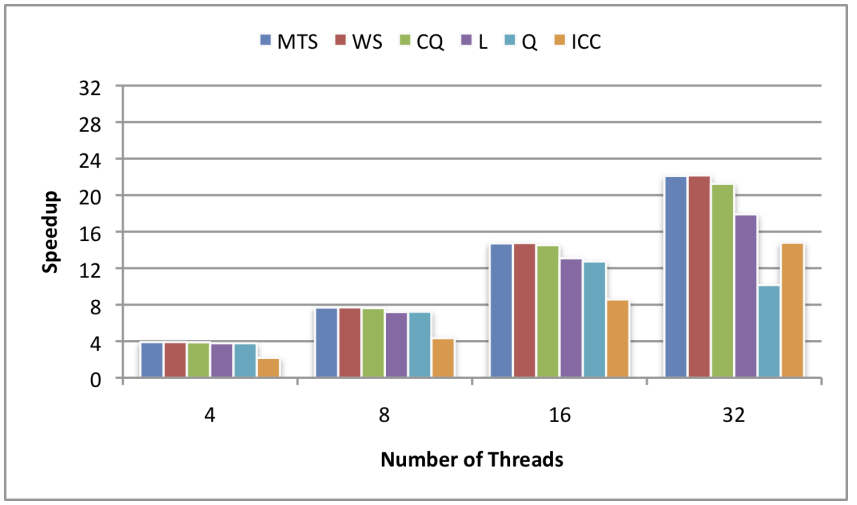


Sparse LU Decomposition

Single task startup



For loop startup



Per-Core Work Stealing vs. Hierarchical Scheduling

- Per-core work stealing exhibits lower variability in performance on most benchmarks
  - Both per-core work stealing and hierarchical scheduling Qthreads implementations had smaller standard deviations than ICC on almost all benchmarks

Configuration	Alignment (`single`)	Alignment (`for`)	Fib	Health	NQueens	Sort	SparseLU (`single`)	SparseLU (`for`)	Strassen
ICC 32 threads	4.4	2.0	3.7	2.0	3.2	4.0	1.1	3.9	1.8
GCC 32 threads	0.11	0.34	2.8	0.35	0.77	1.8	0.49	N/A	1.4
Qthreads MTS 32 workers	0.28	1.5	3.3	1.3	0.78	1.9	0.15	0.16	1.9
Qthreads WS 32 shepherds	0.035	1.8	2.0	0.29	0.60	0.90	0.060	0.24	3.0

Standard deviation as a percent of the fastest time

Per-Core Work Stealing vs. Hierarchical Scheduling

- Hierarchical scheduling benefits
  - Significantly fewer remote steals observed on almost all programs

Benchmark	MTS		WS	
	Steals	Failed	Steals	Failed
Alignment (`single`)	1016	88	3695	255
Alignment (`for`)	109	122	1431	286
Fib	633	331	467	984
Health	28948	10323	295637	47538
NQueens	102	141	1428	389
Sort	1134	404	19330	3283
SparseLU (`single`)	18045	8133	68927	24506
SparseLU (`for`)	13486	11889	68099	32205
Strassen	227	157	14042	823

Per-Core Work Stealing vs. Hierarchical Scheduling

- Hierarchical scheduling benefits
  - Lower L3 misses, QPI traffic, and fewer memory accesses as measured by HW performance counters on *health*, *sort*

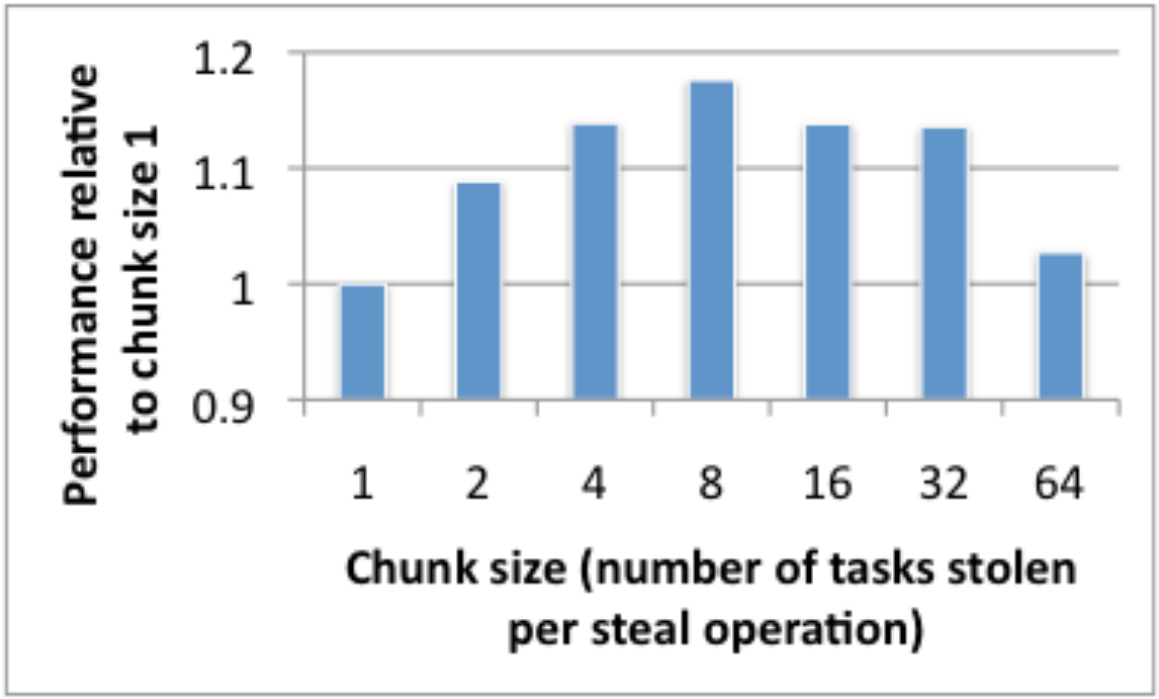
Health

Metric	MTS	WS	%Diff
L3 Misses	1.16e+06	2.58e+06	38
Bytes from Memory	8.23e+09	9.21e+09	5.6
Bytes on QPI	2.63e+10	2.98e+10	6.2

Sort

Metric	MTS	WS	%Diff
L3 Misses	1.03e+7	3.42e+07	54
Bytes from Memory	2.27e+10	2.53e+10	5.5
Bytes on QPI	4.35e+10	4.87e+10	5.6

Stealing Multiple Tasks



Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks

Looking Ahead

- Our prototype Qthreads run time is competitive with and on some applications outperforms ICC and GCC.
  - Implementing non-blocking task queues could further improve performance.
- Hierarchical scheduling shows potential for scheduling on hierarchical shared memory architectures.
  - System complexity is likely to increase rather than decrease with hardware generations.

Thanks.

- Stephen Olivier, UNC Chapel Hill
- Allan Porterfield, RENCI
- Kyle Wheeler, Sandia National Labs
- Jan Prins, UNC Chapel Hill