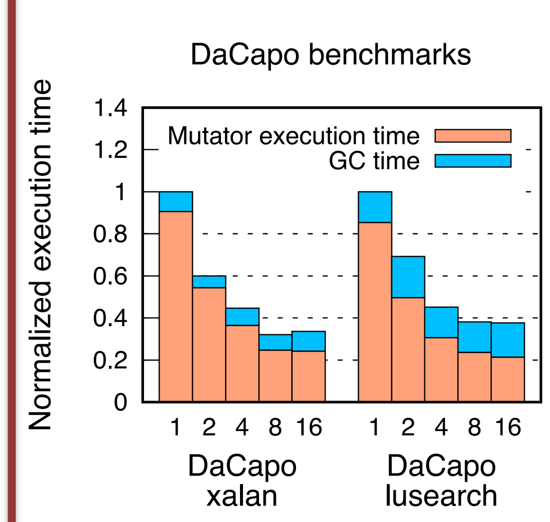
Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems

Kun Suo*, Jia Rao*, Hong Jiang*, Witawas Srisa-an#,

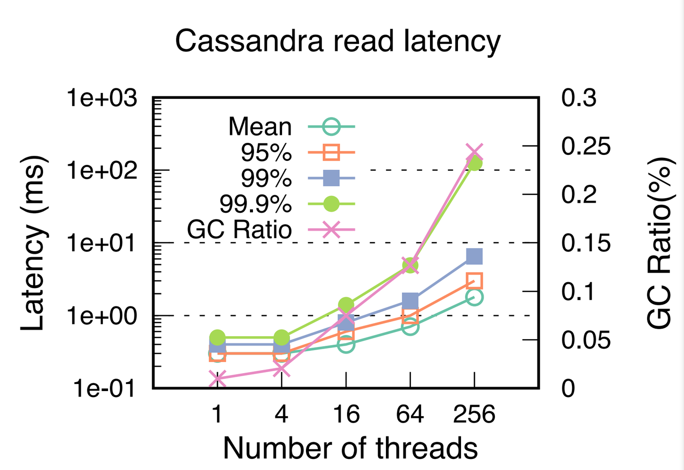
* The University of Texas at Arlington # The University of Nebraska–Lincoln

Motivation

- **Java is popular and widely adopted**
  - Ease of use, cross-platform portability, and wide-spread community support
  - Large-scale distributed systems are built on JVM, e.g., Cassandra, Hadoop, Spark, Kafka, etc.
- **GC causes suboptimal performance and poor scalability for applications on the multicore systems**
  - GC time becomes more significant in the overall time with increasing mutator threads or processing larger dataset
  - Request latency increases exponentially as the GC ratio goes up



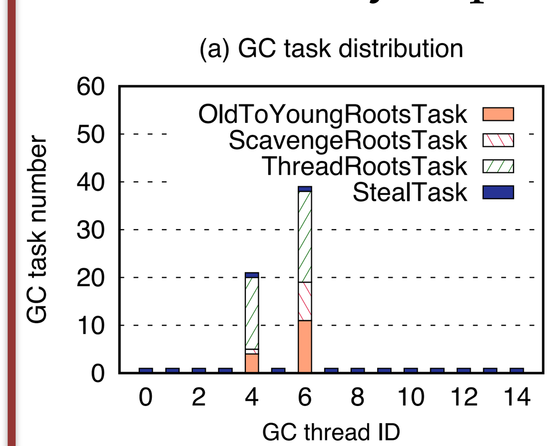
Execution time

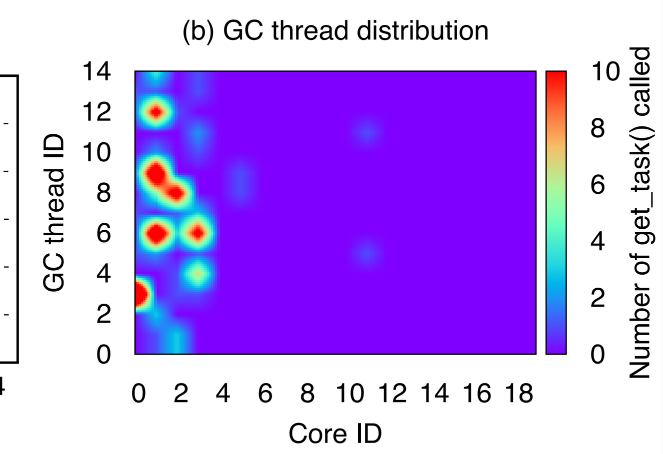


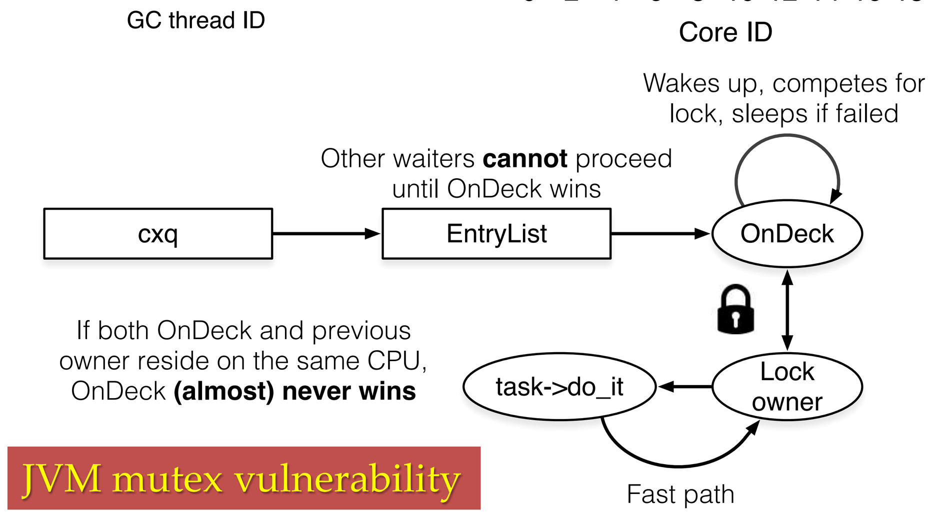
Tail latency

Load Imbalance

- **Task imbalance**
  - GC tasks are unevenly distributed among the GC threads and Parallel Scavenge fails to exploit the available parallelism
- **Thread imbalance**
  - Most GC threads are stacked on a few CPU cores and multicore parallelism is not fully exploited



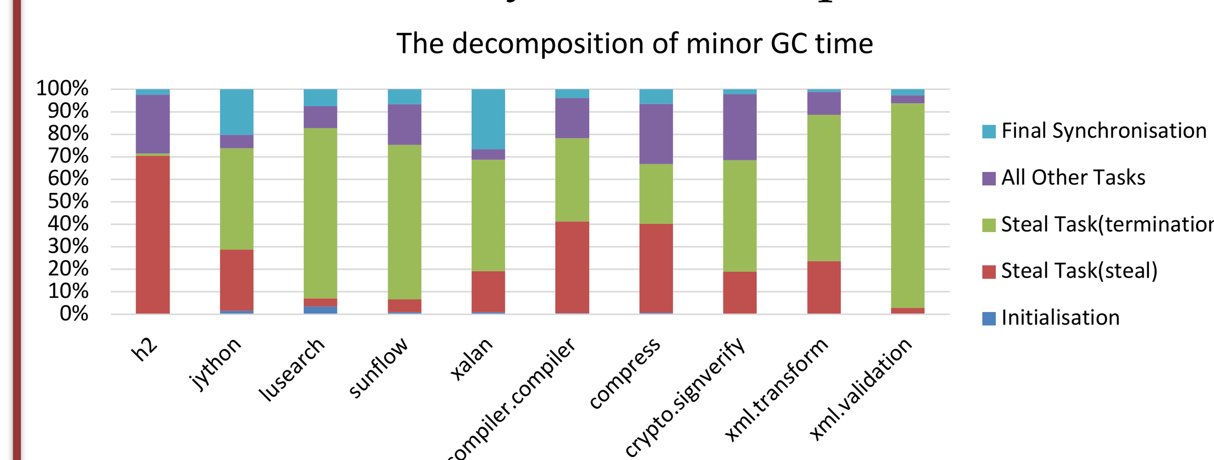




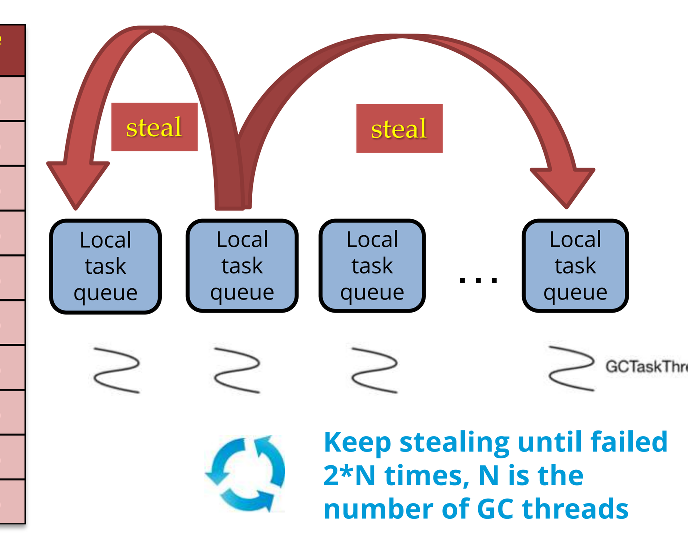
JVM mutex vulnerability

Ineffective Work Stealing

- **Steal tasks dominate the total GC time**
  - Work stealing is inefficient in addressing the GC load imbalance
  - The existing termination protocol wastes time in the steal termination phase
  - Applications suffer from high failure rate and too many steal attempts

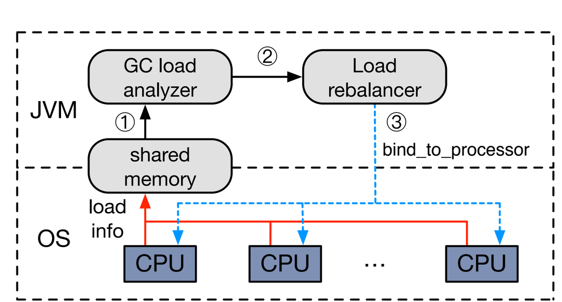


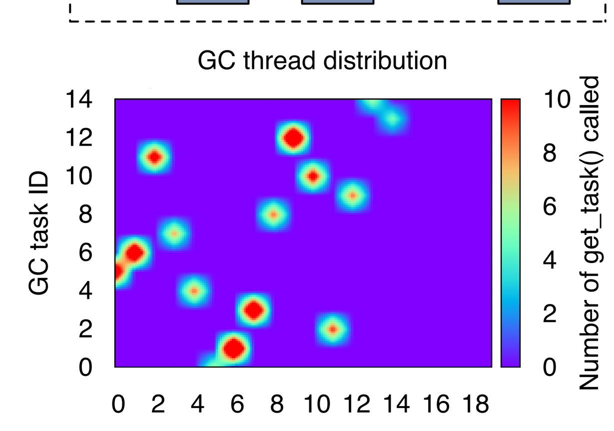
Benchmark	Total	Failure	Failure rate
h2	237983	166008	69.80%
jython	75036	66318	88.40%
lusearch	117383	108308	92.30%
sunflow	45648	34695	76.00%
xalan	22783	19303	84.70%
compiler.compiler	726920	274043	37.70%
compress	29348	26513	90.30%
crypto.signverify	64493	60388	93.60%
xml.transform	457198	341555	74.70%
xml.validation	651475	188189	28.90%

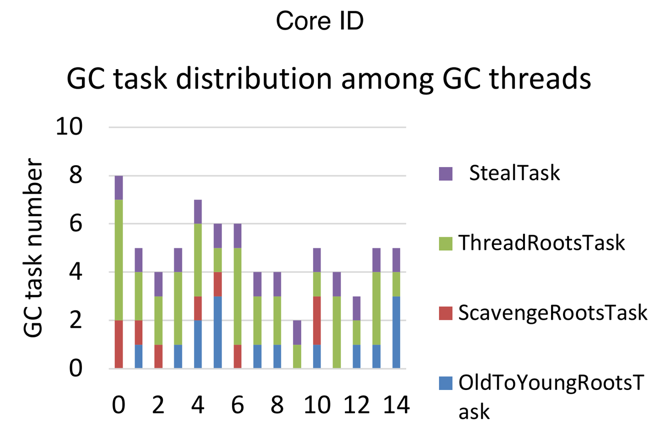


Proactive and Dynamic GC Load Balance

- **Addressing Load Imbalance**
  - We implement the backend function bind_to_processor() in JVM
  - JVM reads CPU load information, which is shared by memory into the /proc file system, and binds the GC thread to cores proactively based on the load
  - We add task affinity when GC threads get GC tasks from the task queue
- **Thread to Core and Task to Thread Balance**
  - GC threads are evenly distributed on multiple cores
  - All GC threads are able to fetch tasks from GCTaskQueue
  - GC thread and task affinity help mitigate load imbalance among GC threads. All GC threads are assigned with root tasks.

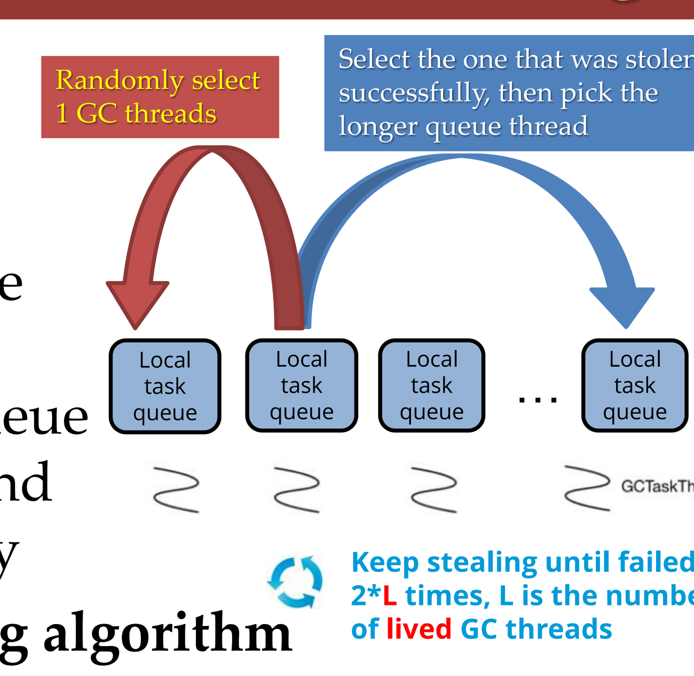


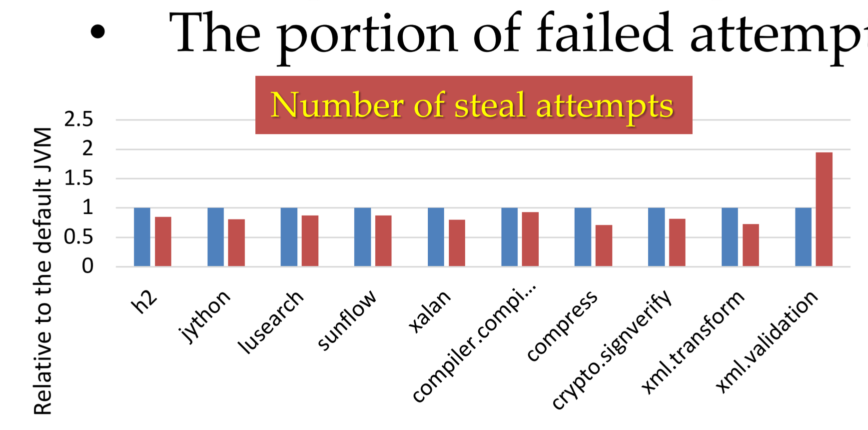




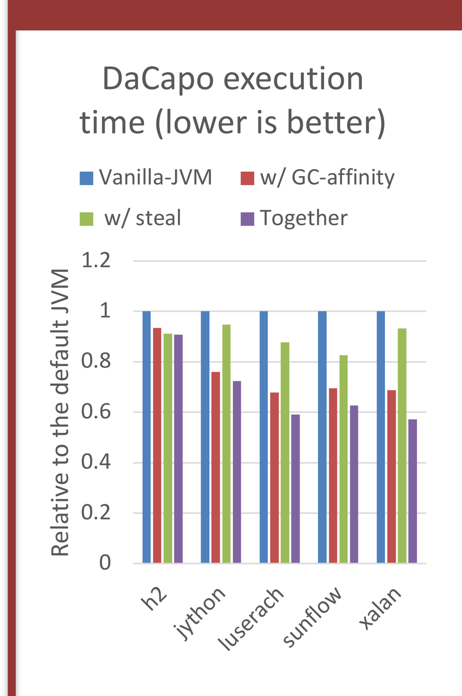
Adaptive and Semi-random Work Stealing

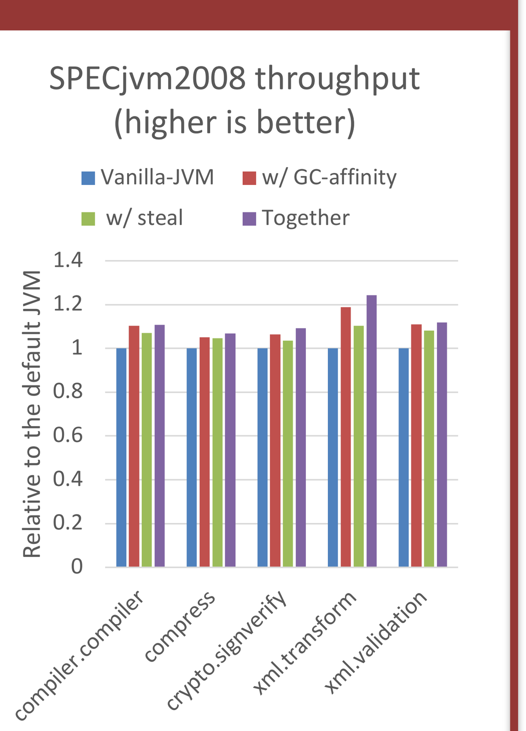
- **Addressing Inefficient Stealing**
  - We propose an adaptive and semi-random stealing algorithm
  - New termination protocol records the number of active GC threads
  - The algorithm memorizes the one queue from where the steal was a success and another queue is picked up randomly
- **Effectiveness of the optimized stealing algorithm**
  - The optimized stealing reduces the total number of steal attempt
  - The portion of failed attempts drops

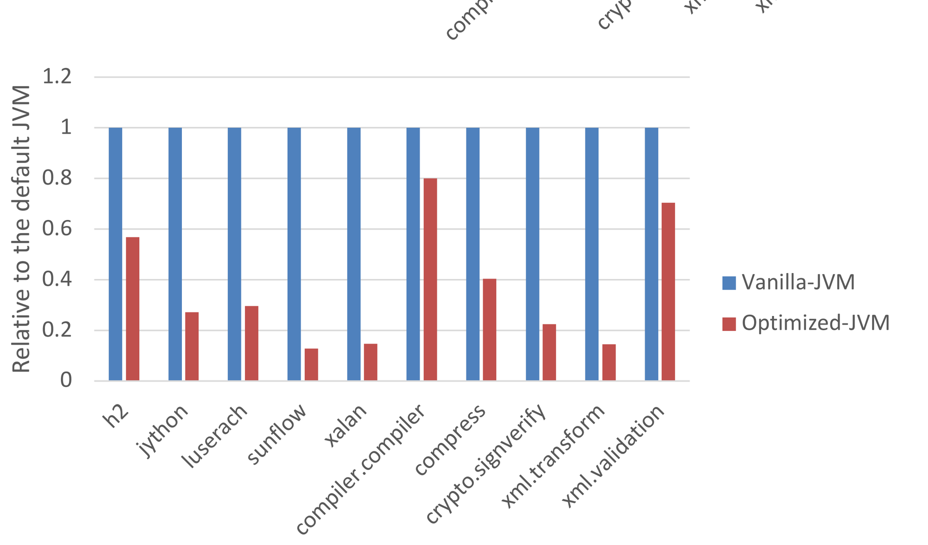




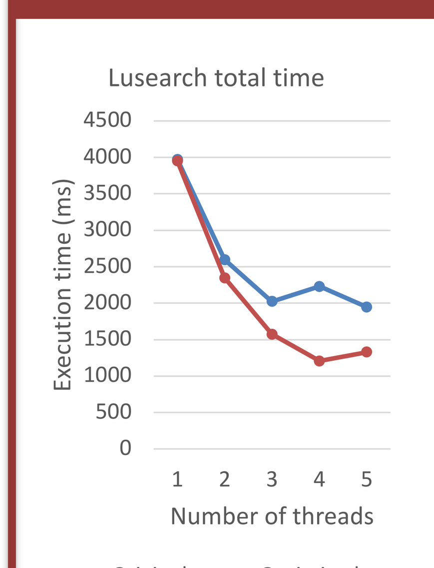
Overall & GC Performance

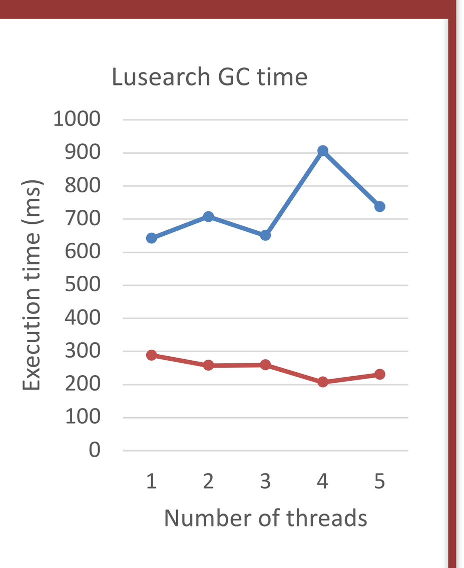


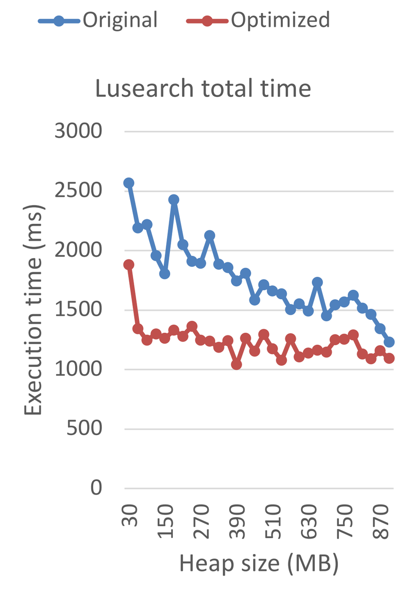


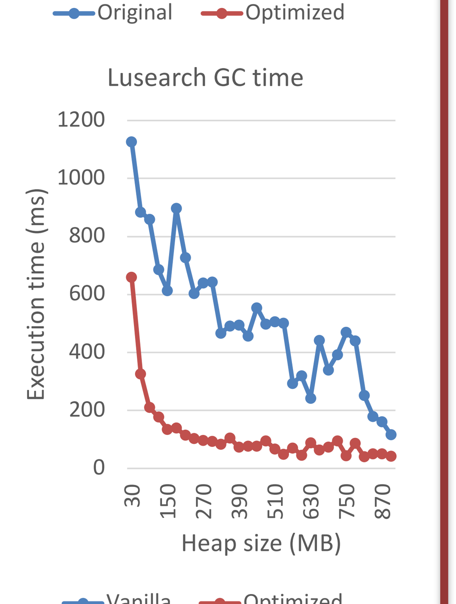


Scalability & Heap Size

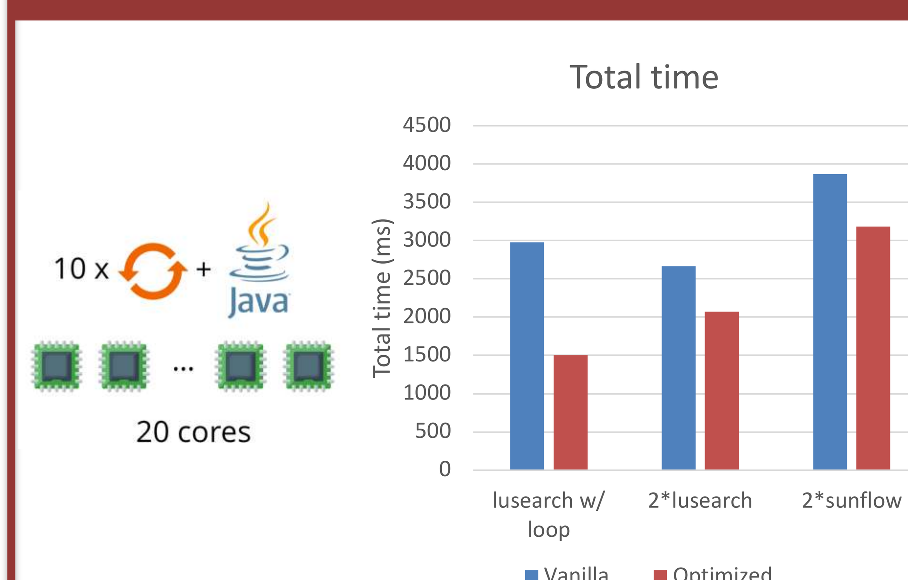


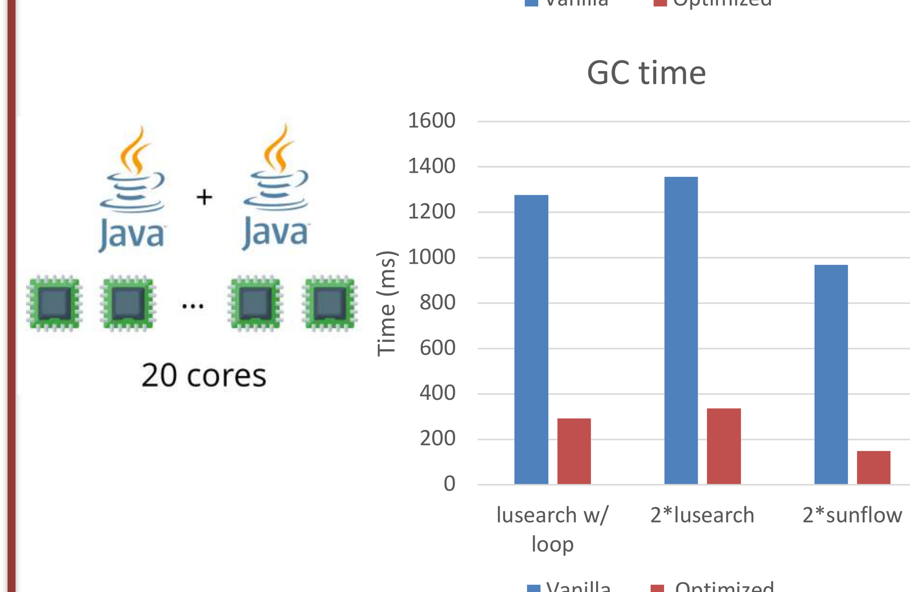






Constrained Resources





Conclusion

- Complex interplays among dynamic GC task assignment, unfair mutex locking, imperfect OS load balancing and less efficient stealing during the GC inflict loss of concurrency in parallel GC
- We propose an effective approach coordinating the JVM with the OS to address GC load imbalance and designed a more efficient work stealing algorithm
- This work is supported in part by U.S. NSF grants CNS-1649502 and IIS-1633753.