Exploiting Ordered Parallelism in Fine-Grained Task-Parallel Programs

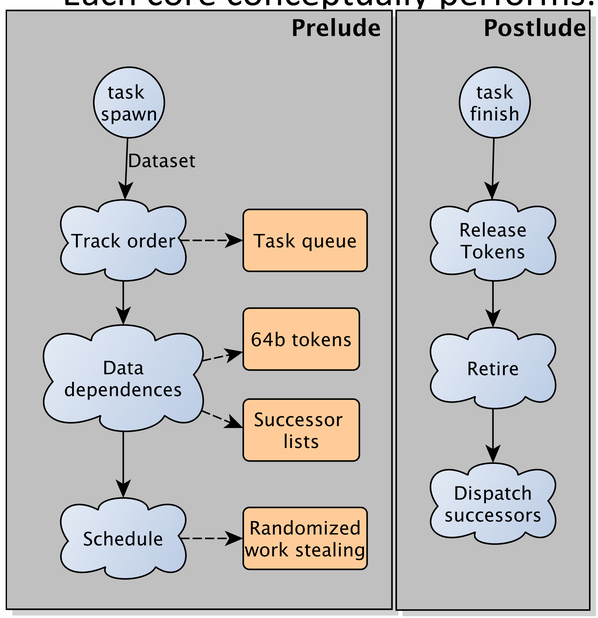
Aditya Venkataraman, Gagan Gupta, and Gurindar S. Sohi

1. Motivation

- Graphs exhibit *dynamic, irregular* parallelism
  - Fine-grained tasks ~ 1000 cycles
  - Sparse data-dependences
  - Dynamic task creation
  - Partial/total ordering among tasks
- State of the art: Runtime-engine orchestrates parallel execution, featuring
  - Unordered algorithms [Hassan et al, PPoPP'11]
  - Deterministic scheduling [Nguyen et al, ASPLOS'14]
  - Fully HW implementation [Jeffrey et al, MICRO'15]
- What is the ideal HW/SW co-design to minimize runtime overheads?

2. Parallelization Runtime

- Parakram [Gupta, UW Madison PhD Thesis'15]
  - Program-ordered parallel execution
  - Each core conceptually performs:

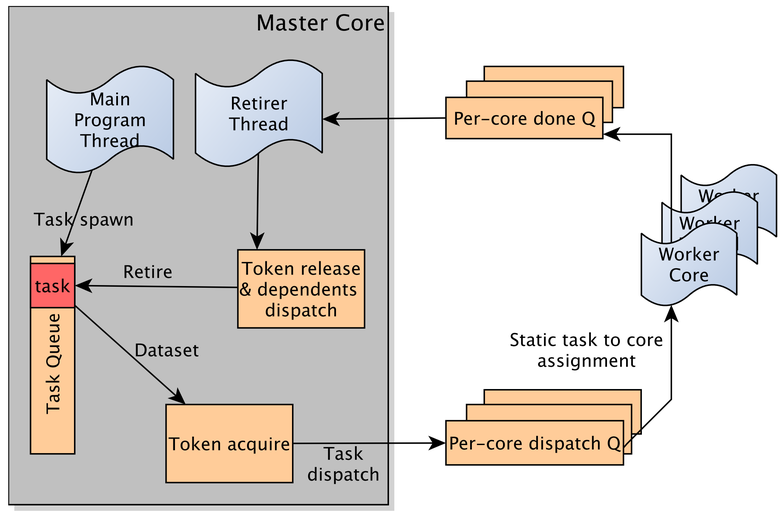


3. Runtime Overheads

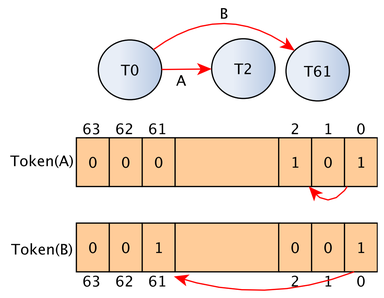
Following overheads reduce performance:

- Wait-free synchronization using Atomics
  - Global serialization (~80 cycles per MFENCE)
  - Needed for token acquire/release
- Work-stealing loses locality
- 'Pollution' of runtime data-structures
  - Accessed by any/all threads
- Linear successor lists to track dependences

4. New Architecture

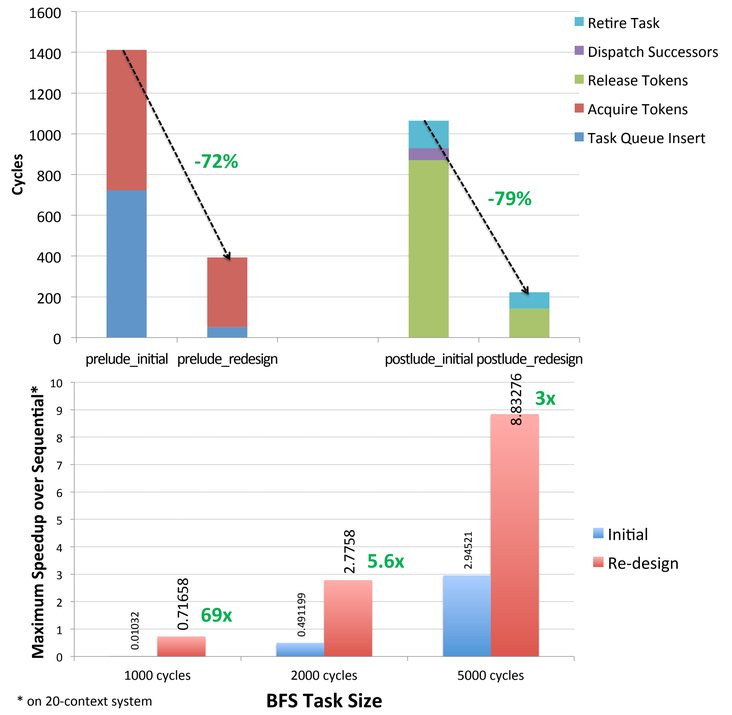


- Master-Slave approach
  - Communication through per-slave queues
- No atomics or MFENCEs.
- Task-distribution to enhance locality
- Token bitmaps that encode data-flow edges and owners
- Reduce runtime state



5. Results

- Prelude overheads reduced by ~72%
- Postlude overheads reduced by ~79%
- Unable to scale for irregular graphs with task sizes of 1000 cycles
  - 69x improvement over initial design
  - Speedups scale with increasing task-sizes



6. Future Work

- Explore micro-architectural support for:
  - Generic master-slave communication
  - Token bitmap manipulation
- Supporting globally ordered priority queues
- Port more applications to the new runtime