Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications

Jonathan Lifflander, Sriram Krishnamoorthy, Laxmikant V. Kale

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Presentation: Session 9: Thursday, June 21 @ 10:20am

Introduction

Applications iteratively executing identical or slowly evolving calculations require incremental rebalancing to improve load balance across iterations. The work to be performed is overdecomposed into *tasks*, enabling automatic rebalancing by the middleware.

We consider the design and evaluation of two traditionally disjoint approaches to rebalancing: work stealing and *persistence-based* load balancing. We apply the principle of persistence to work stealing to design an optimized *retentive* work stealing algorithm. We demonstrate high efficiencies on the full NERSC Hopper (146,400 cores) and ALCF Intrepid (163,840 cores) systems, and on up to 128,000 cores on OLCF Titan.

Challenges

- On distributed-memory, the cost of work stealing is significant due to communication latency and task migration costs.
- Previous rebalancing is ignored with traditional work stealing, incurring repeated costs across iterations.
- Work stealing algorithms that randomly redistribute work may disrupt locality or topology-optimized distributions.
- Persistence-based load balancers cannot adapt immediately to load imbalance: they require initial profiling and rebalancing only occurs at the end of a phase.
- Persistence-based load balancers incur periodic rebalancing overheads.

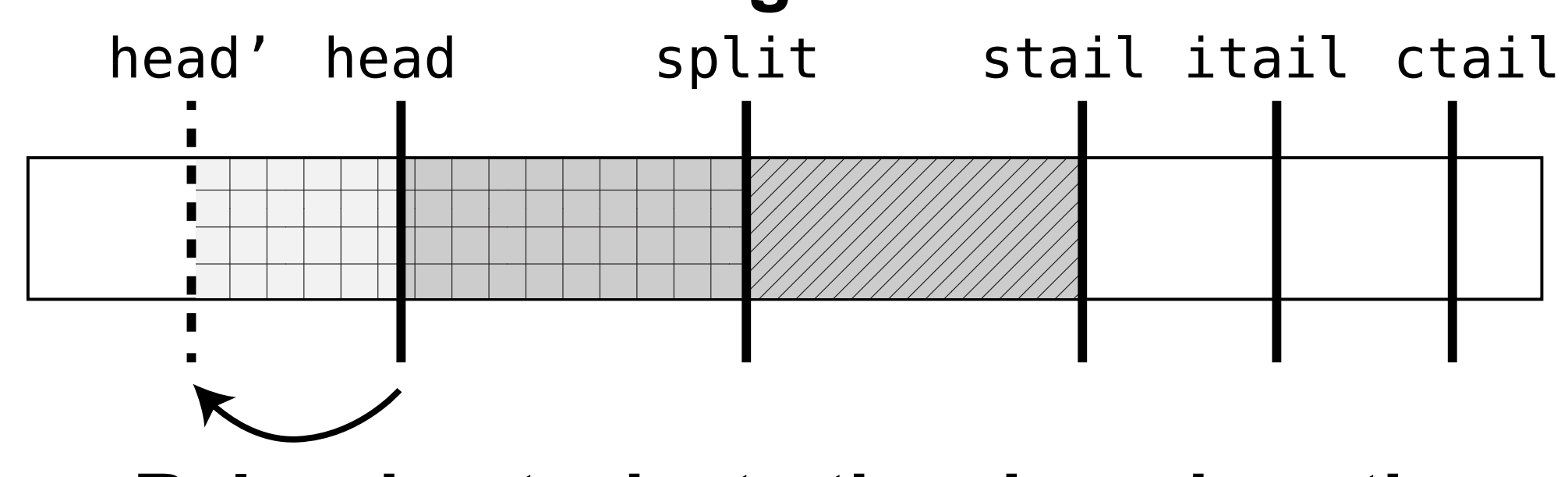
Retentive Work Stealing

In work stealing, every core executes tasks from a local queue until it is empty. It then becomes a thief that randomly searches for work until it finds additional work or termination is detected.

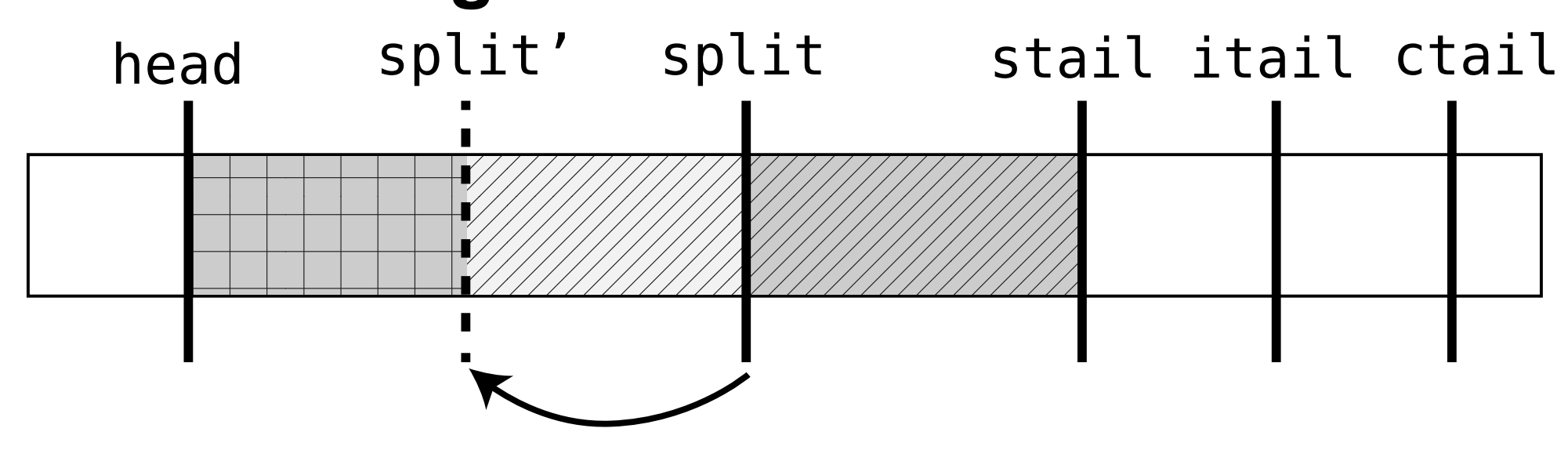
In our distributed-memory implementation, we use a bounded-buffer circular deque with a fixed-sized array to reduce data transfer costs and limit synchronization overhead.

The following figures illustrate the operations on the queue. A red dotted arrow indicates that a lock or atomic operation is involved.

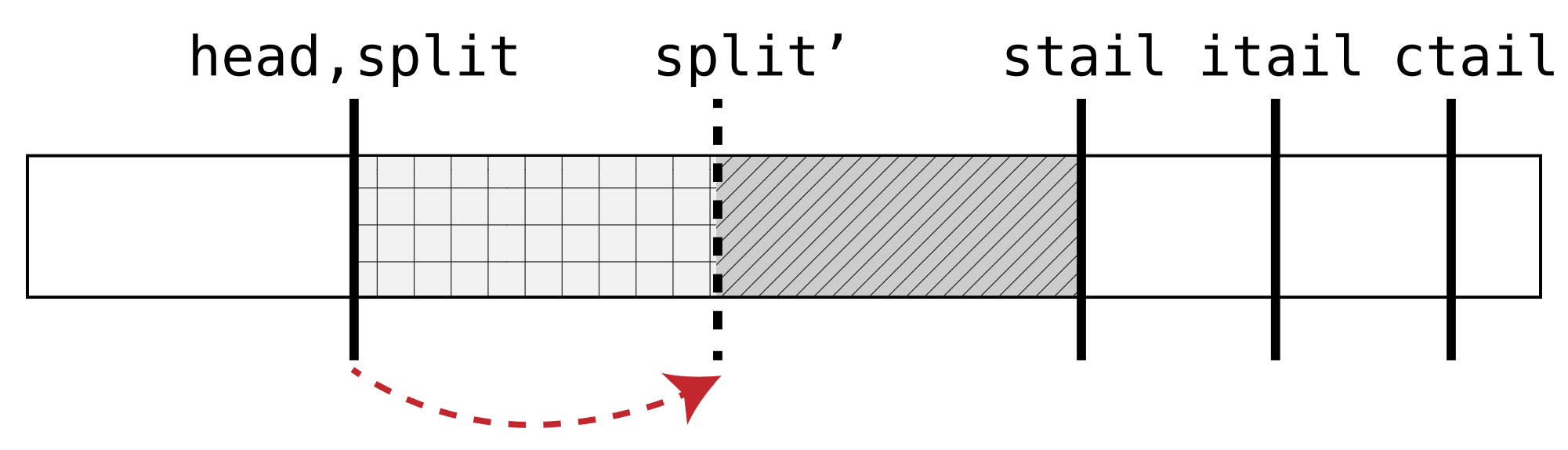
Adding a task



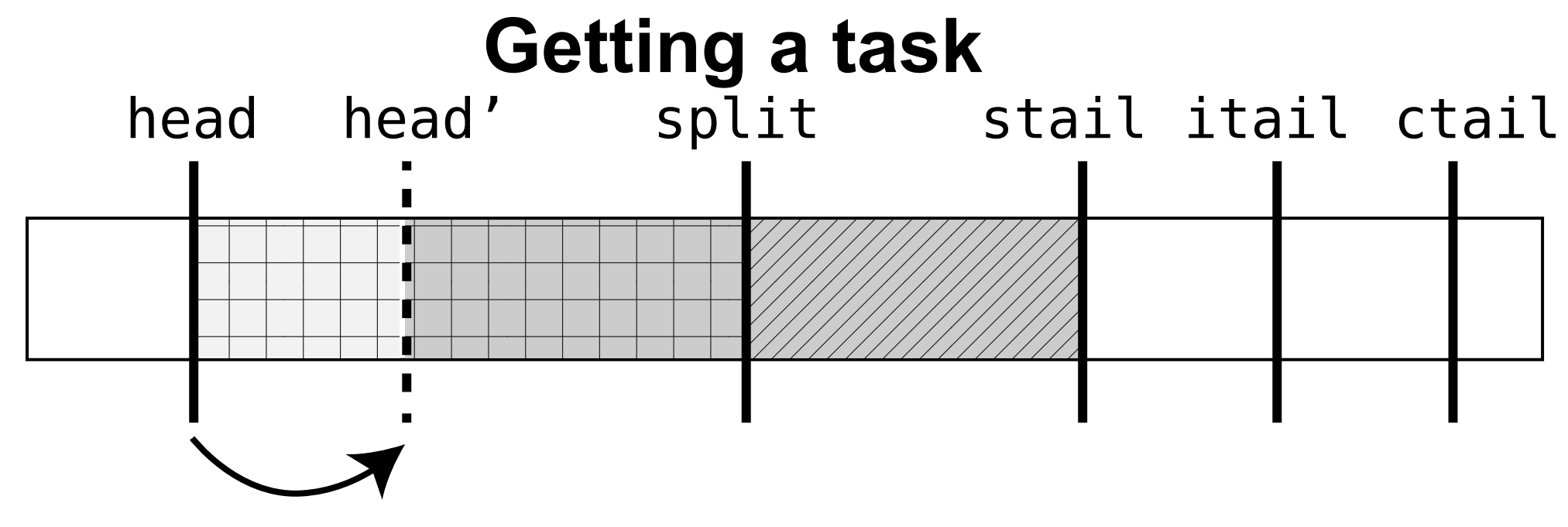
Releasing tasks to the shared section



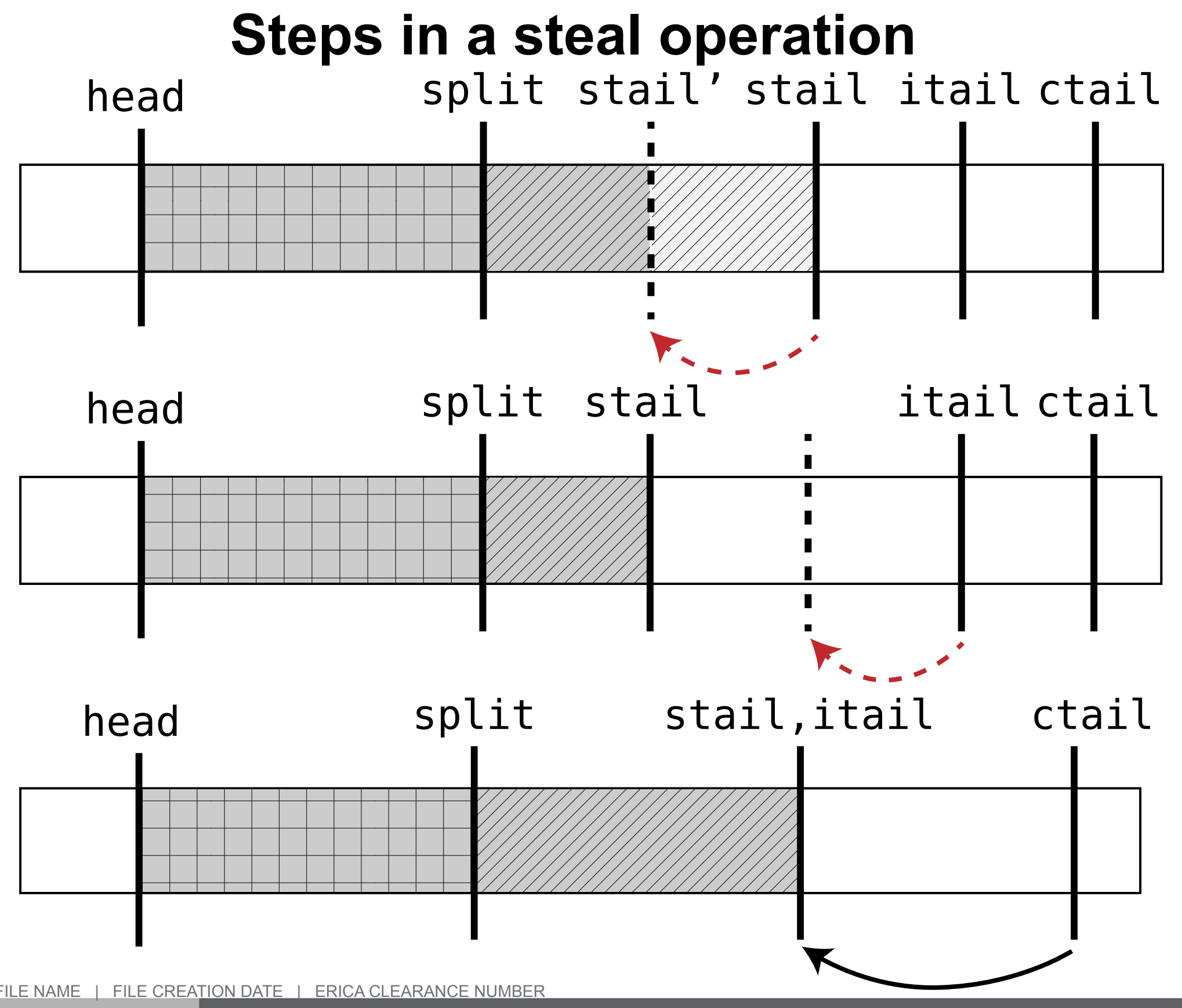
Acquiring tasks from the shared section



Getting a task

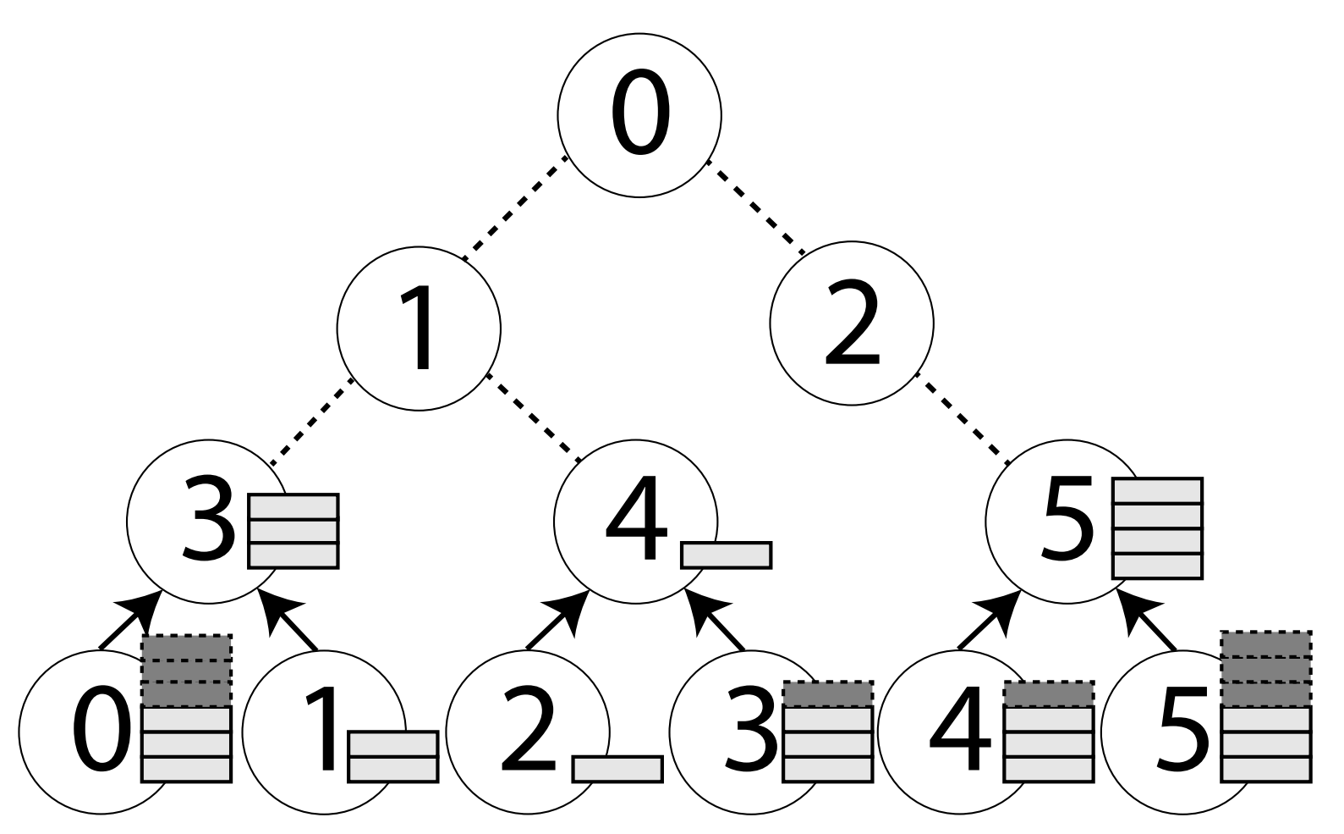


Steps in a steal operation

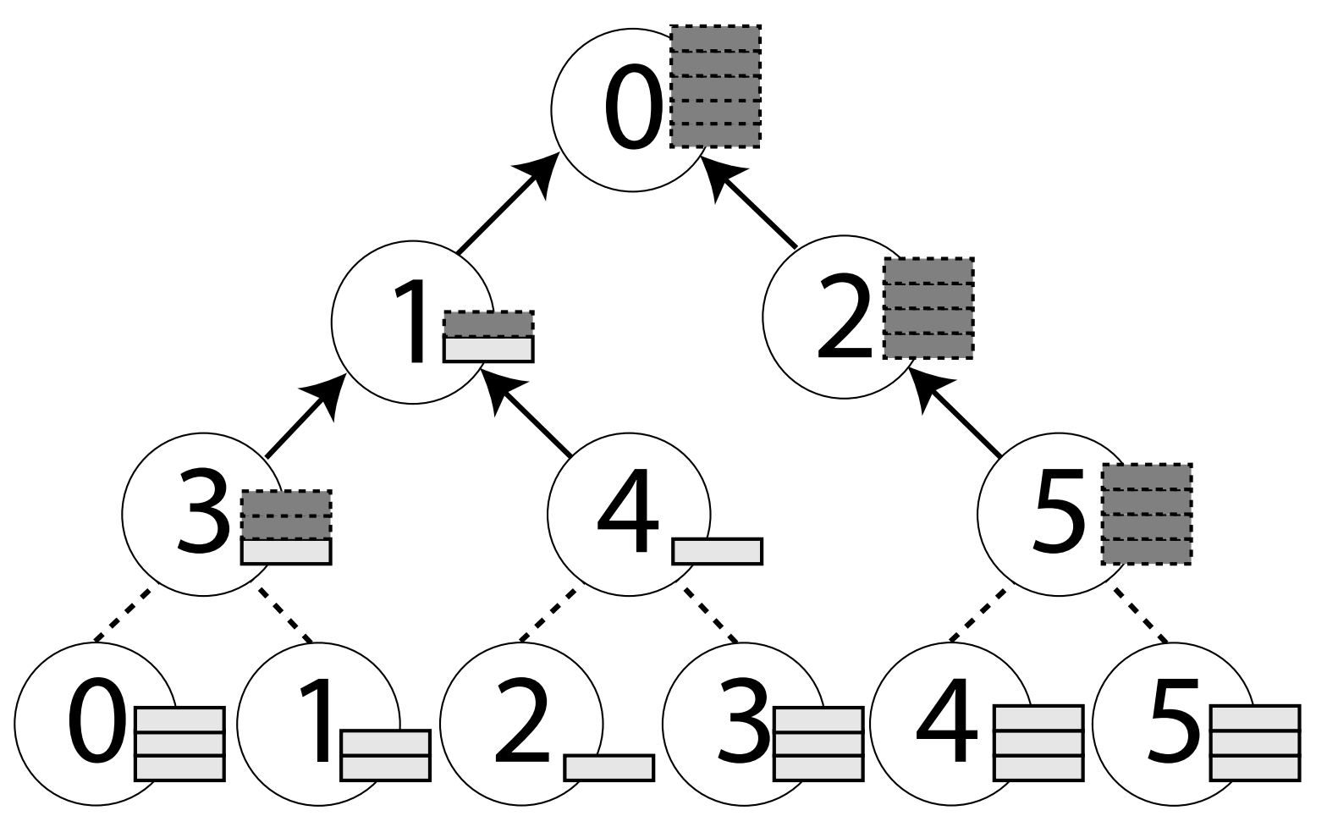


Persistence-based Load Balancing

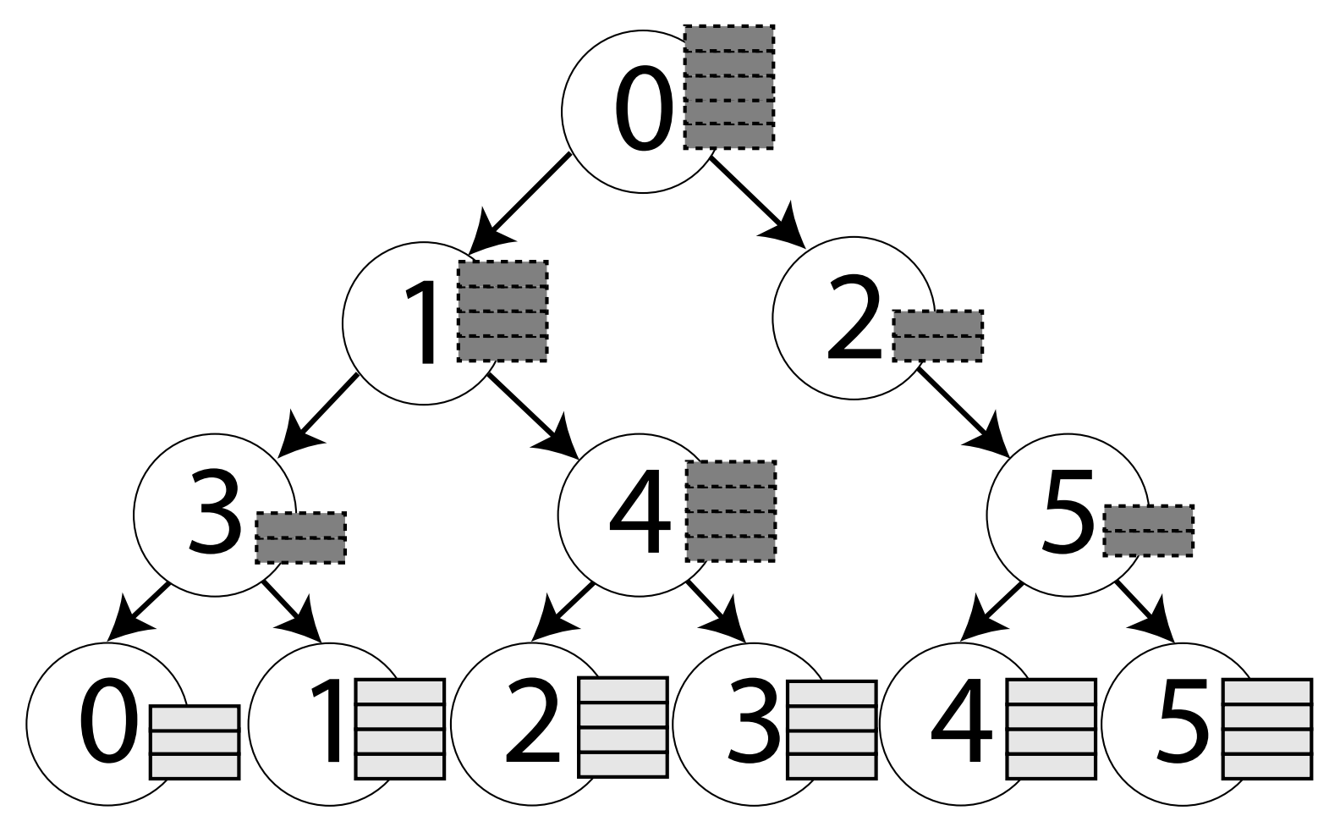
Applications that retain the computation balance over iterations, with gradual change, are said to adhere to the principle of persistence. This behavior can be exploited by measuring the performance profile in a previous iteration and using these measurements to rebalance the tasks. We present a scalable hierarchical persistence-based load balancing algorithm that greedily redistributes "excess" load to localize the rebalance operations and migration of tasks. In this algorithm, the cores are organized in a tree where every core is a leaf.



Leaves send their excess load (any load above a constant times the average load) to their parent, selecting the work units from shortest duration to longest.

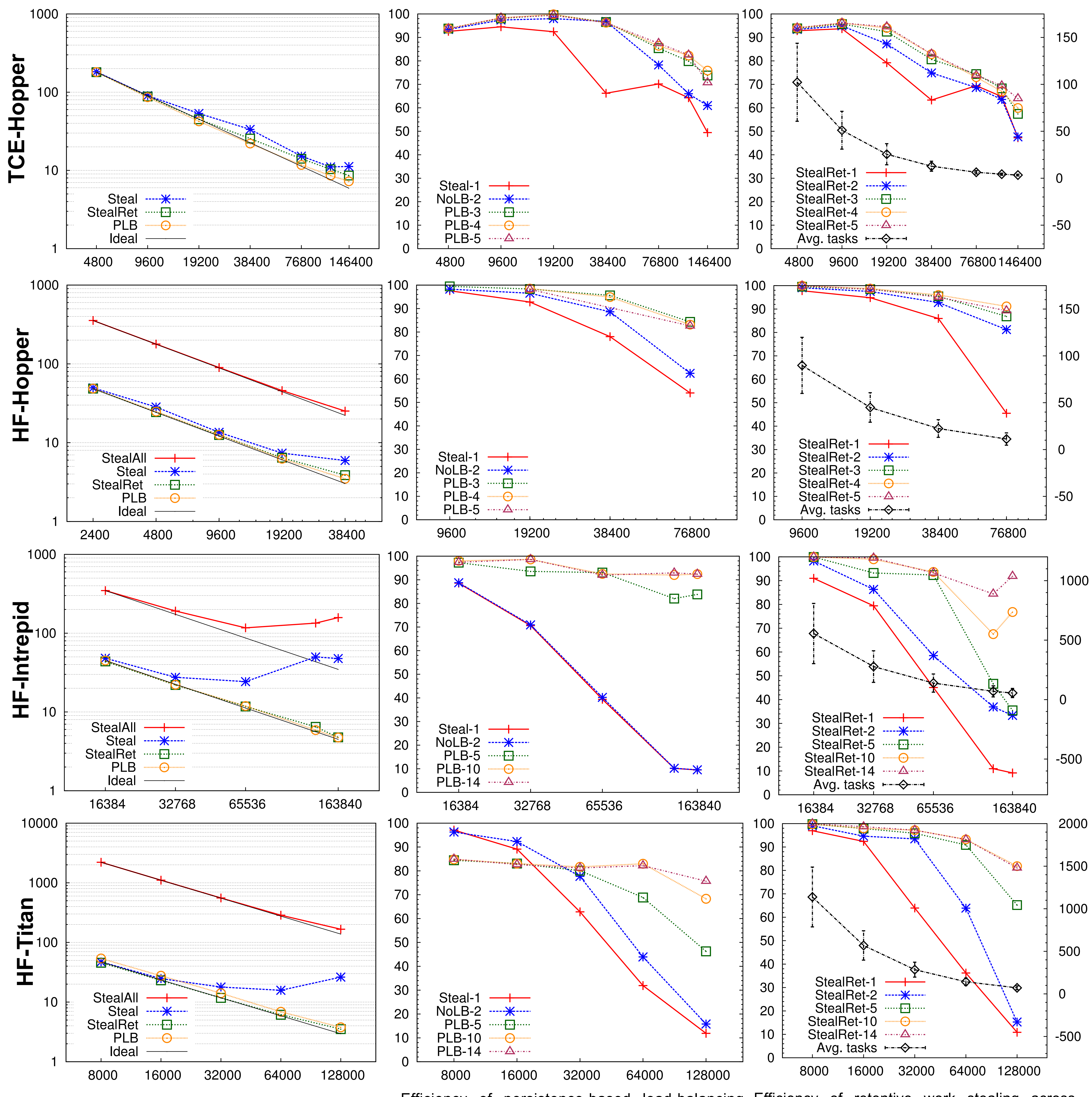


Each core receives excess load from its children and saves work for underloaded children until their load is above a constant times the average load. Remaining tasks are sent to the parent.



Starting at the root, the excess load is distributed to each child applying the centralized greedy algorithm: maximum duration task is assigned to minimum-loaded child.

Experimental Results



Execution times for first and last iteration. x-axis – number of cores; y-axis – execution time in seconds

Efficiency of persistence-based load-balancing across iterations for the three system sizes, relative to the ideal anticipated speedup. x-axis – number of cores; y-axis – efficiency

Efficiency of retentive work stealing across iterations relative to ideal anticipated speedup and tasks per core. x-axis – core count; left y-axis – efficiency; right y-axis – tasks per core (error bar: std. dev.)

Primary Contributions

- An active-message-based retentive work stealing algorithm that is highly optimized for distributed-memory
- A hierarchical persistence-based load balancing algorithm that performs greedy localized rebalancing
- Most scalable demonstration of work stealing — on up to 163,840 cores of BG/P, 146,400 cores of XE6, and 128,000 cores of XK6
- First comparative evaluation of two traditionally disjoint approaches, work stealing and persistence-based, for load balancing iterative scientific applications
- Demonstration of the benefits of retentive stealing in incrementally rebalancing iterative applications
- Experimental data demonstrating that the number of steals (successful or otherwise) does not grow substantially with scale
- A comparison of the execution time and quality of load balance between a centralized versus hierarchical persistence-based load balancer using a micro-benchmark

FILE NAME | FILE CREATION DATE | ERICA CLEARANCE NUMBER