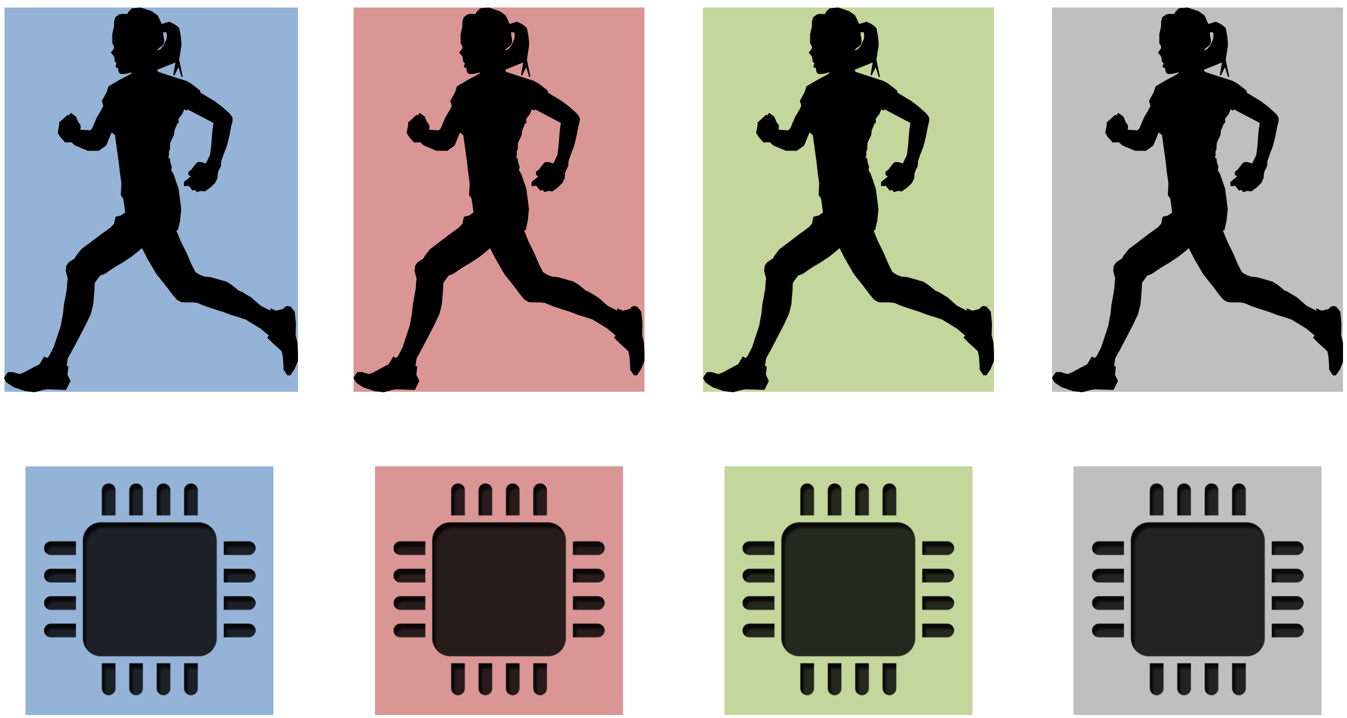
Runtime Support for Scalable Task-parallel Programs

Sriram Krishnamoorthy
Pacific Northwest National Lab
xSIG workshop
May 2018

http://hpc.pnl.gov/people/sriram/

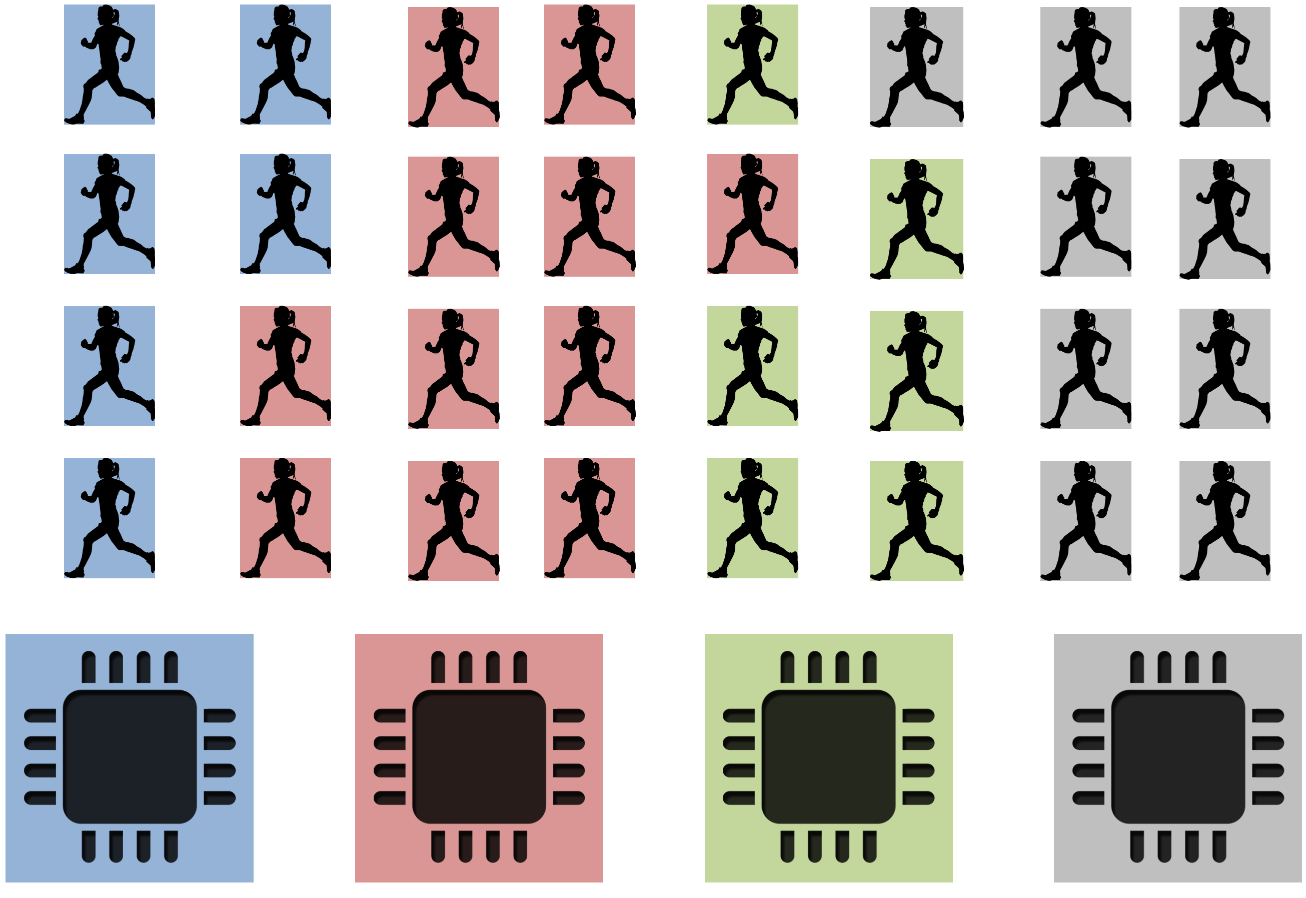
Single Program Multiple Data

```
int main () {
   ...
}
```



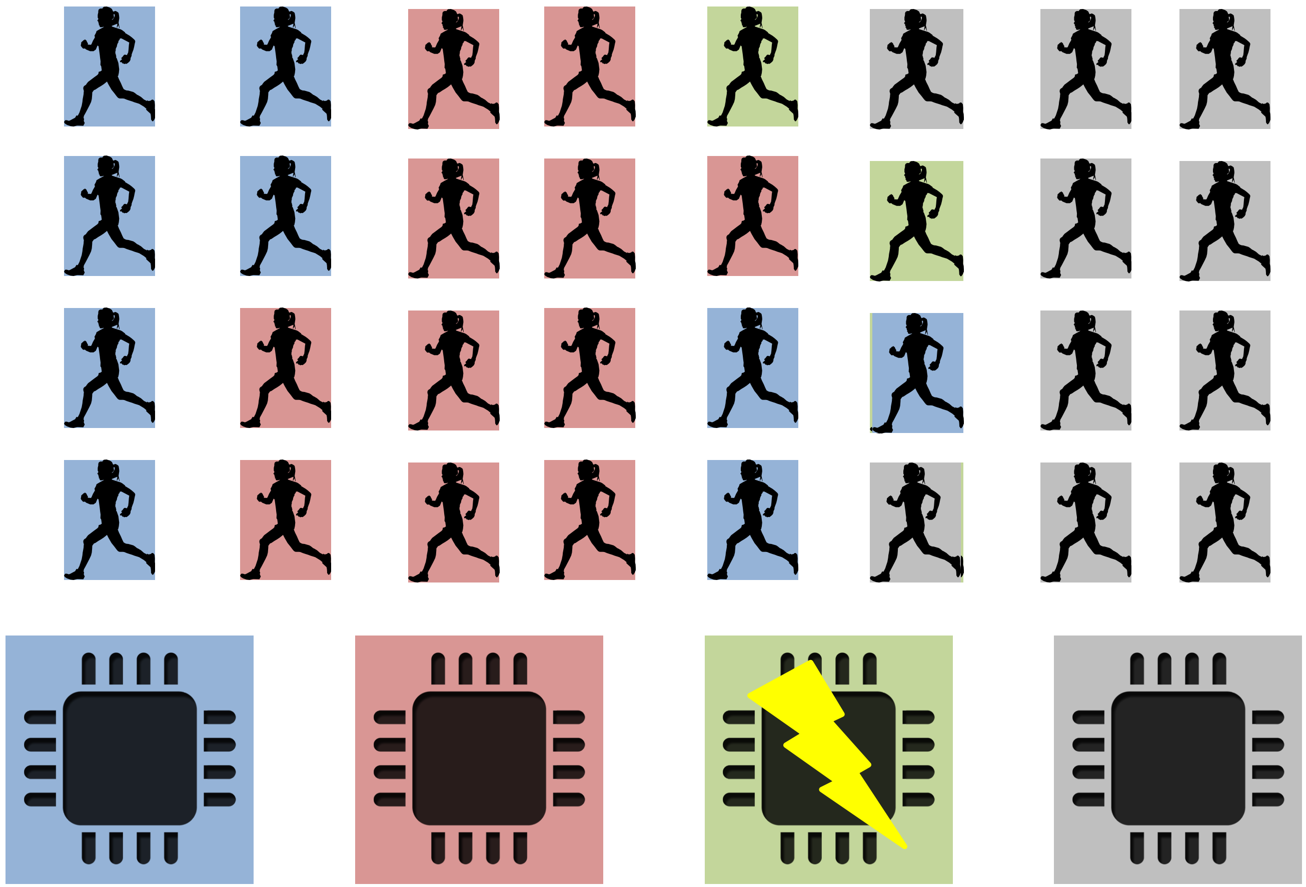
Task Parallelism

```
int main () {
   ...
}
```



Task Parallelism

```
int main () {
   ...
}
```



Sriram Krishnamoorthy

Task-parallel Abstractions

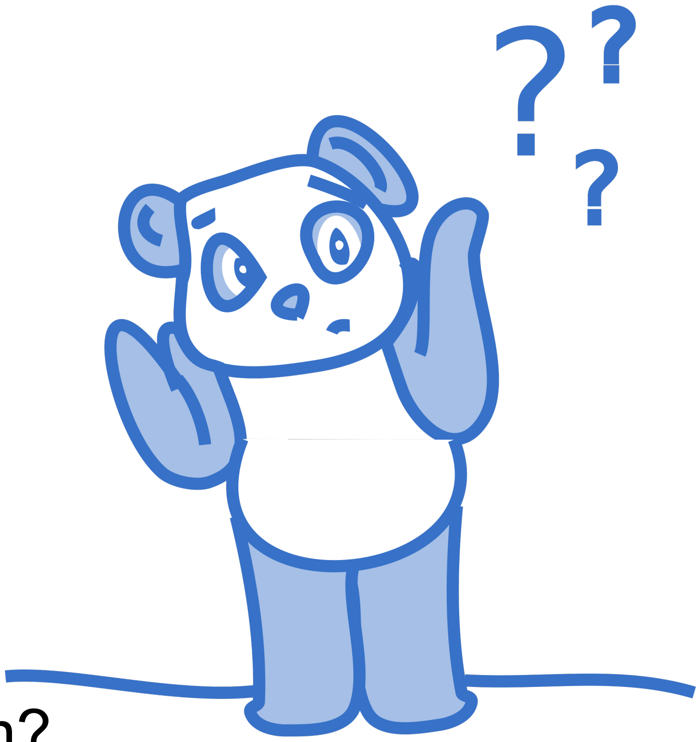
- Finer specification of concurrency, data locality, and dependences
  - Convey more application information to compiler and runtime
- Adaptive runtime system to manage tasks
- Application writer specifies the computation
  - Writes optimizable code
- Tools to transform code to generate an efficient implementation

The Promise

- Application writer specifies the computation
- Computation mapped to specific execution environment by the software stack
- We are transferring some of the burden away from the programmer

The Challenge

- We are transferring some of the burden to the software stack
- Handling million MPI processes is supposed to be hard; how about billions of tasks?
- What about the software ecosystem?



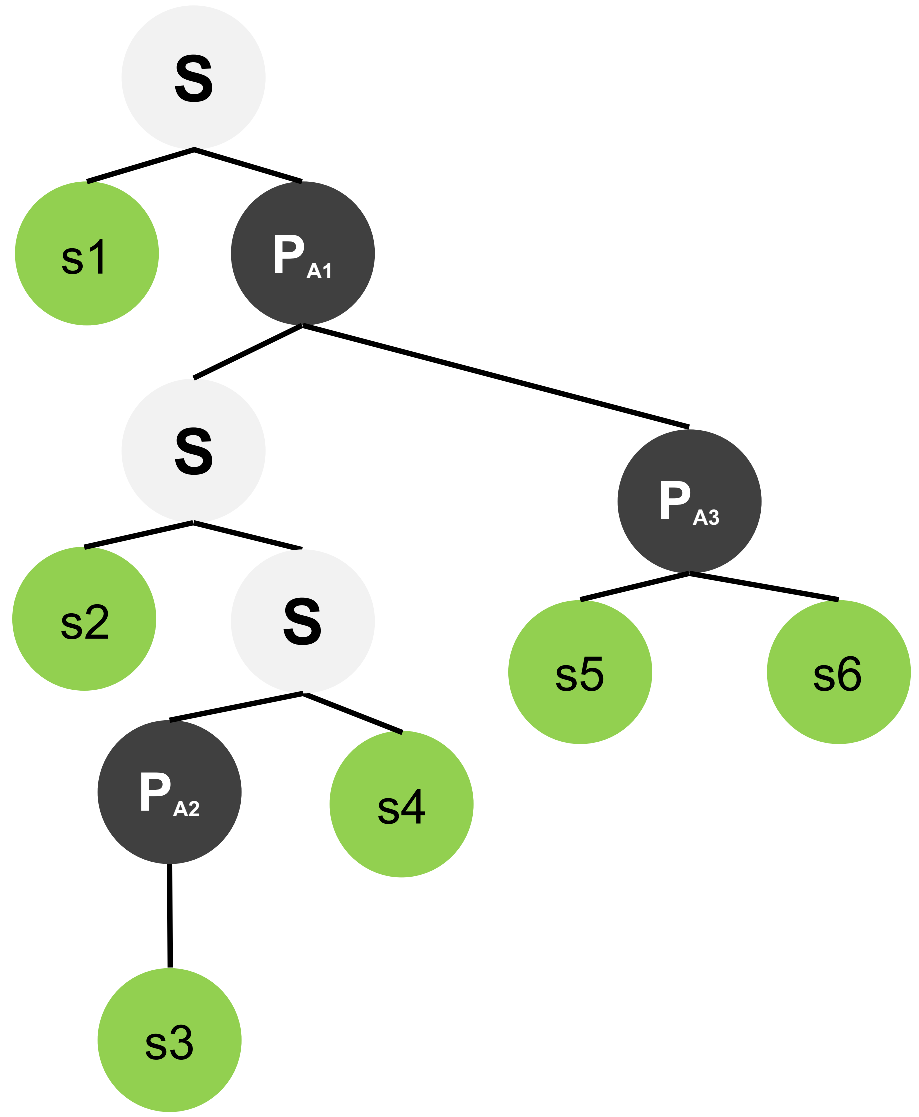
Tracing and Constraining Work Stealing Schedulers

Research Directions

- Concurrency management and tracing
- Dynamic load balancing
- Data locality optimization
- Task granularity selection
- Data race detection

Recursive Task Parallelism

```
fn() {
  s1;
  async { /*A1*/
    s2;
    finish async s3;//A2
    s4;
  }
  async s5; //A3
  s6;
}
```



Sriram Krishnamoorthy

Work Stealing

- A worker begins with one/few tasks
  - Tasks spawn more tasks
  - When a worker is out of tasks, it steals from another worker

- A popular scheduling strategy for recursive parallel programs
  - Well-studied load balancing strategy
  - Provably efficient scheduling
  - Understandable space and time bounds

Objective

- Trace execution under work stealing
- Exploit information from trace to perform various optimizations
- Constrain the scheduler to obtain desired behavior

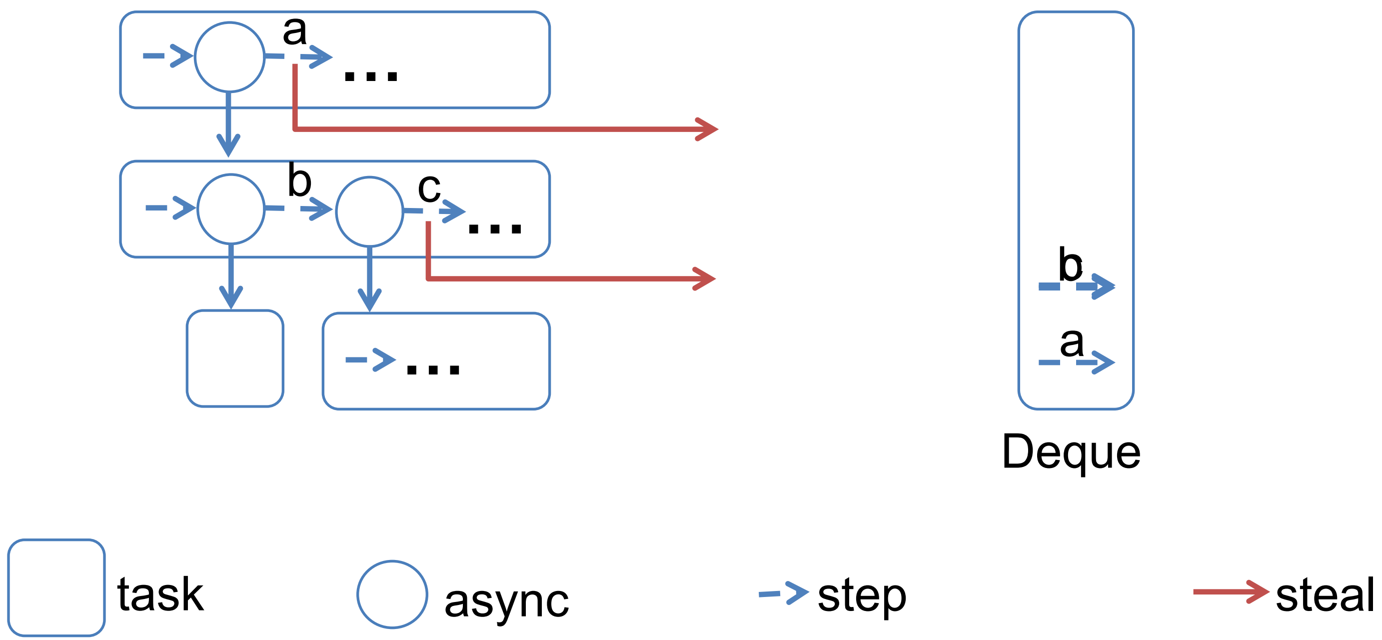
Tracing

Steal tree: low-overhead tracing of work stealing schedulers.
PLDI'13 http://dl.acm.org/citation.cfm?id=2462193

Tracing Work Stealing

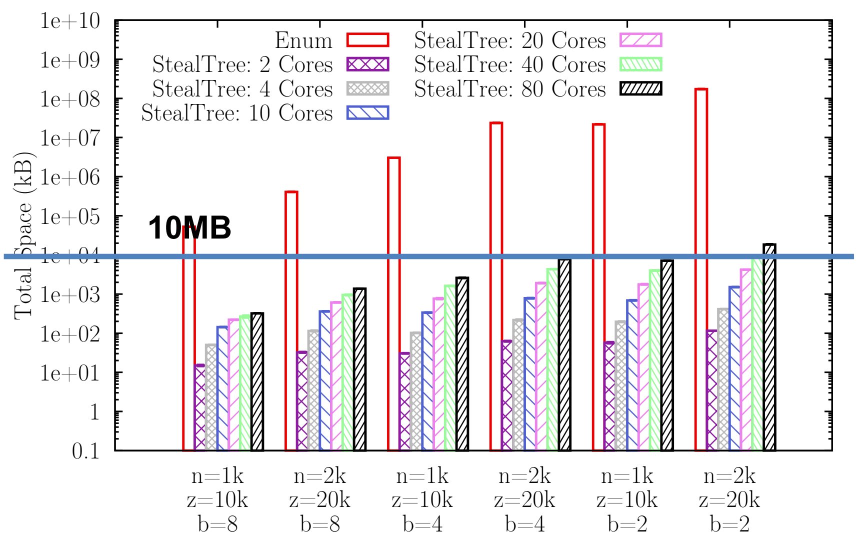
- *When* and *where* each task executed
- Captures the order of events for online and offline analysis
- Challenges
  - Sheer size of the trace
  - Application perturbation might make it impractical

Tracing Approach: Illustration

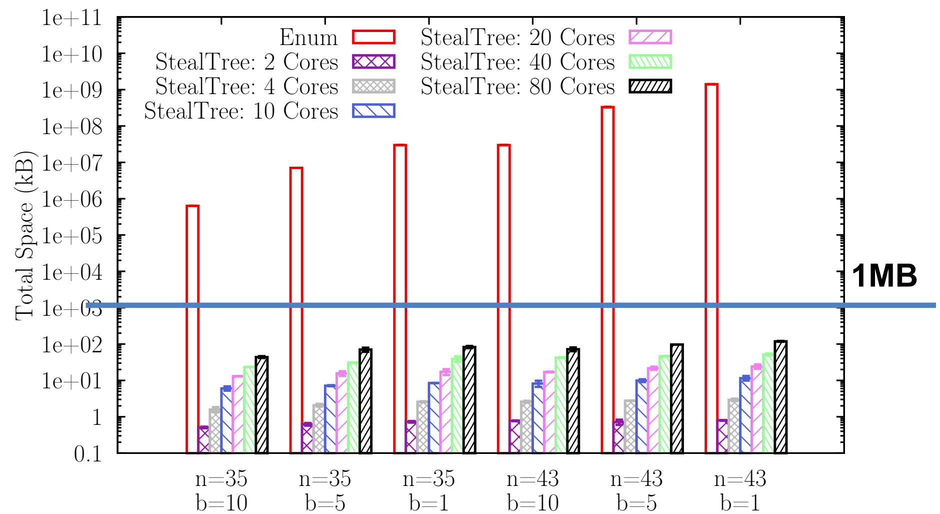


- Steals in order of levels
- Almost one steal per level

Space Overhead: Shared Memory



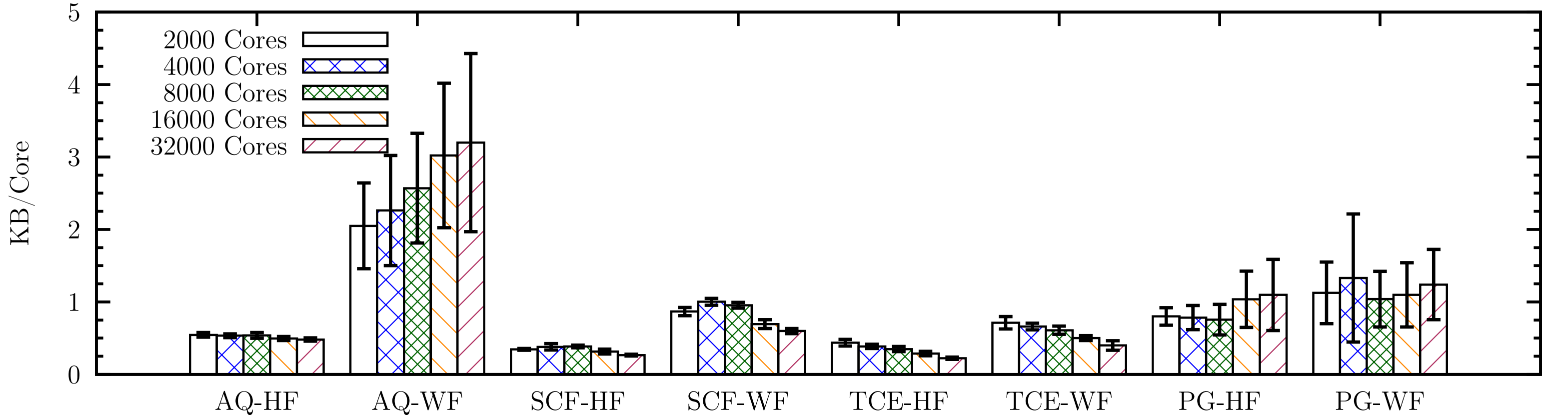
Cholesky



Fib

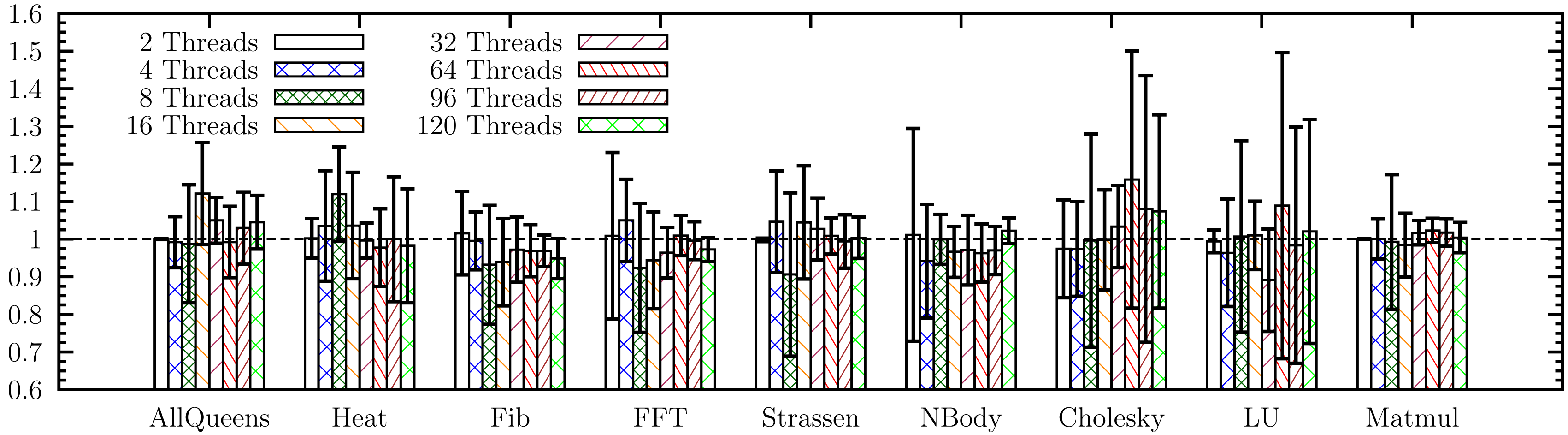
Small trace sizes, less affected by core count or problem size

Space Overhead: Distributed Memory



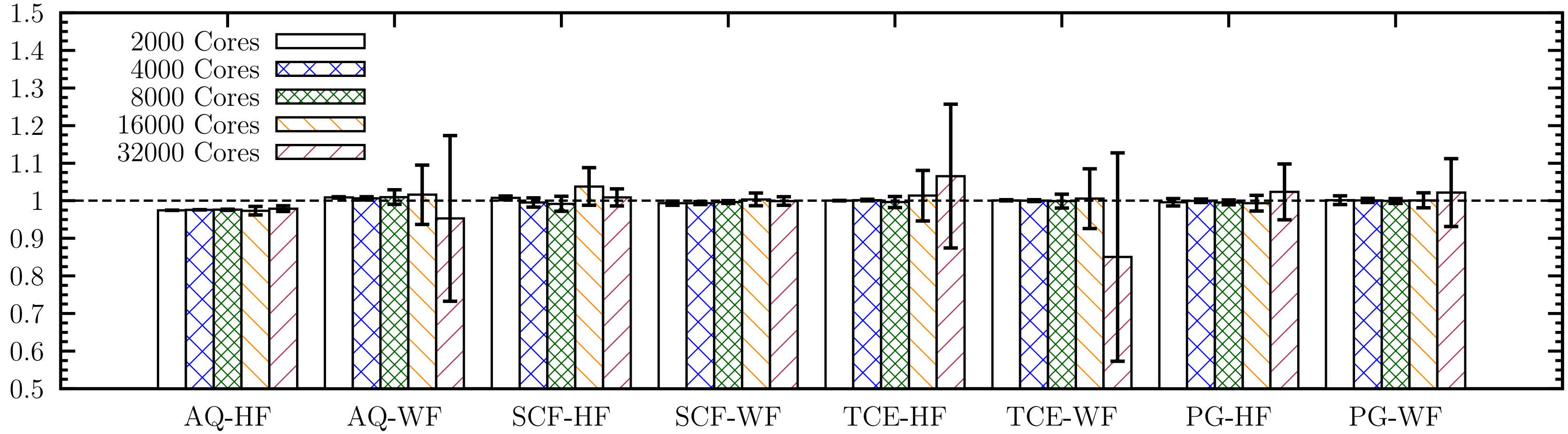
Still less than 160MB in total on 32000 cores

Time Overhead: Shared Memory



Time overhead within variation in execution time

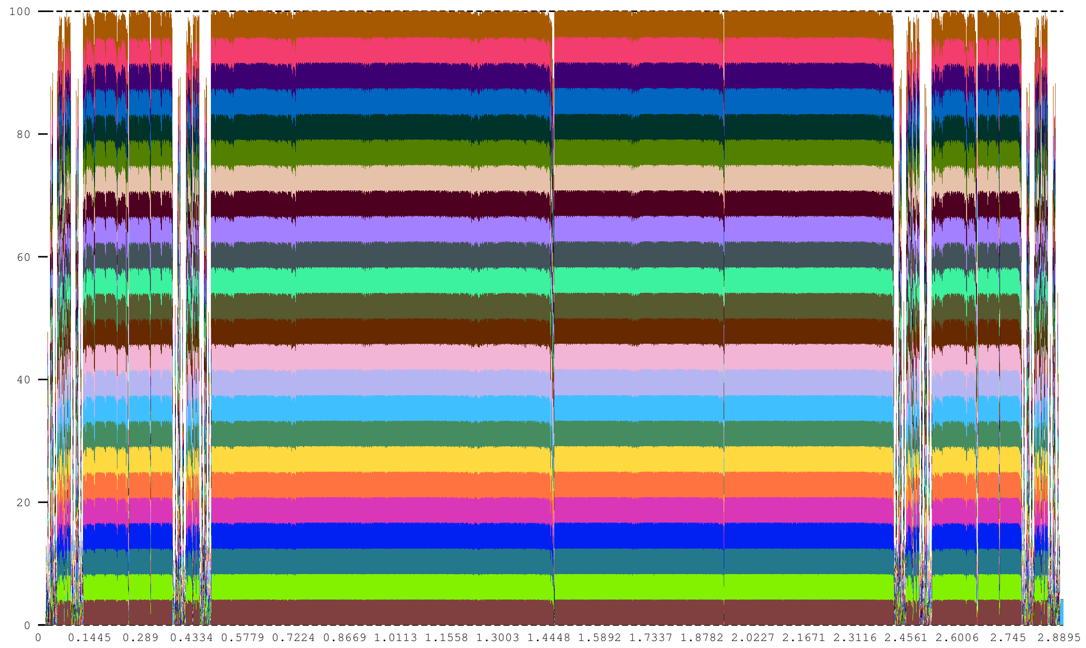
Time Overhead: Distributed Memory



Time overhead within variation in execution time

What can we do with a steal tree?

Visualization



- Core utilization plot over time
- Cilk LU benchmark on 24 cores
- Trace size <100KB

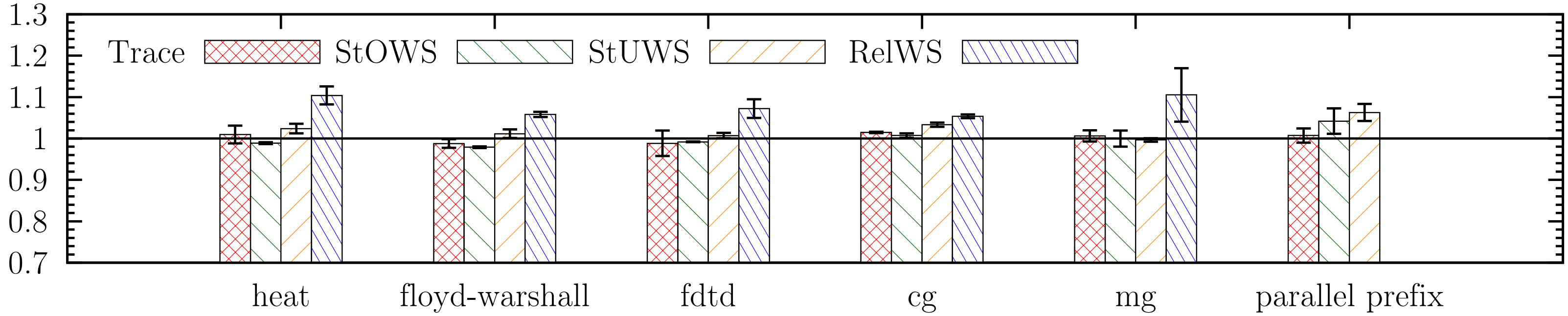
Replay

Optimizing data locality for fork/join programs using constrained work stealing.
SC'14. http://dl.acm.org/citation.cfm?id=2683687

Replay Schedulers

- Strict, ordered replay (StOWS)
  - Exactly reproduce the template schedule
  - Donation of continuations to be stolen
- Strict, unordered replay (StUWS)
  - Reproduce the template schedule, but allow the order to deviate (respecting the application's dependencies)
- Relaxed work-stealing replay (RelWS)
  - Reproduce the template schedule as much as possible, but allow workers to deviate when they are idle, by further stealing work

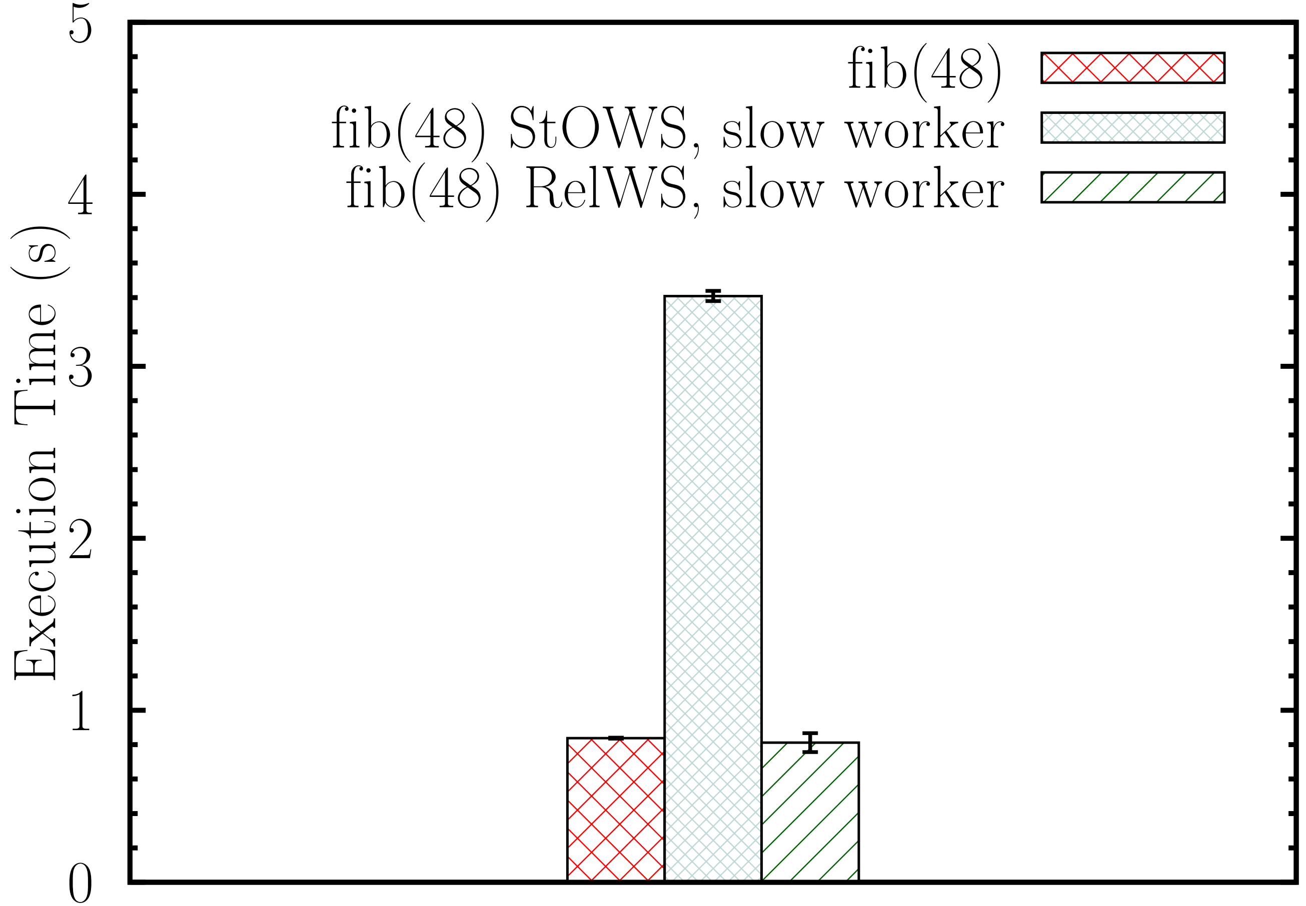
How good are the schedulers?



Relaxed work stealing incurs some overhead because it combines replay and work stealing

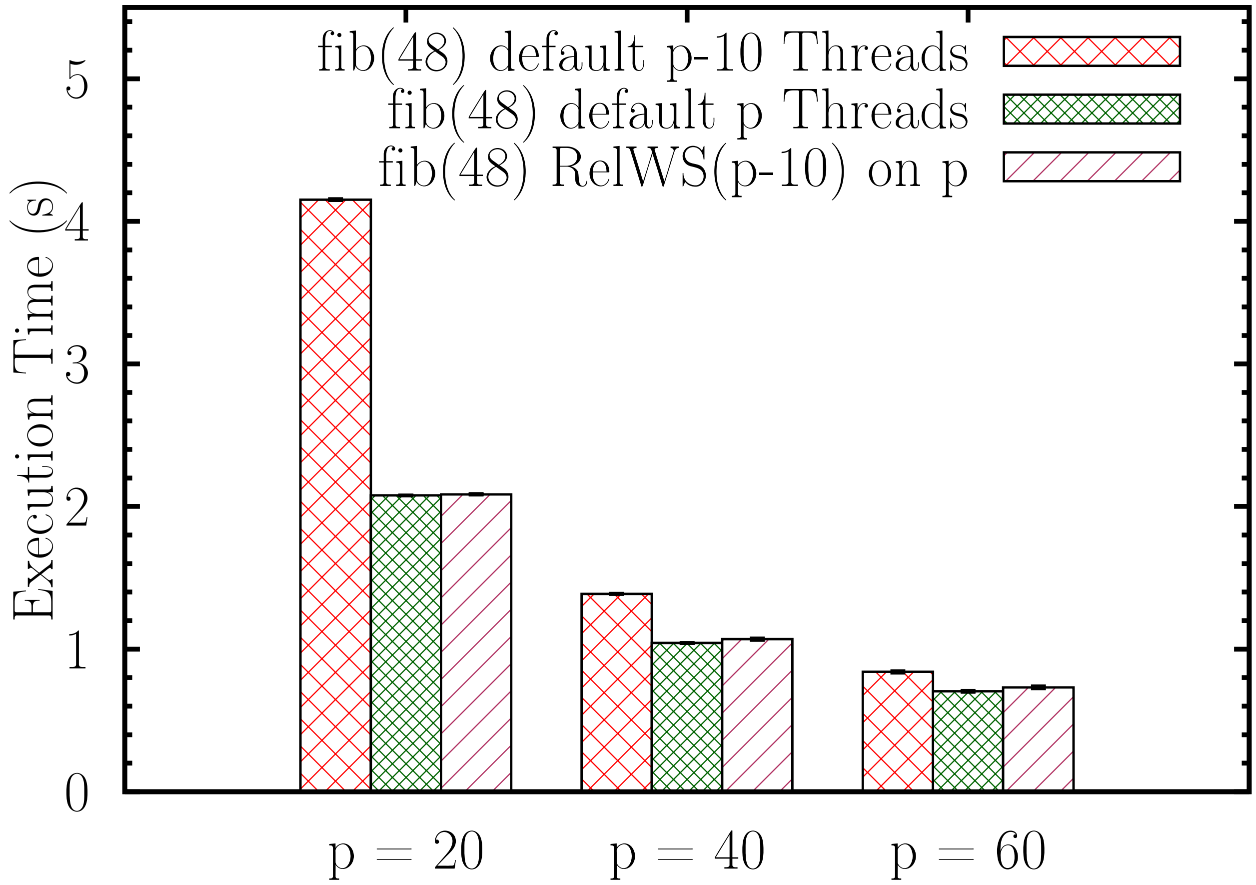
Relaxed Work Stealing: Adaptability I

Slow down one out of 80 workers 4 times



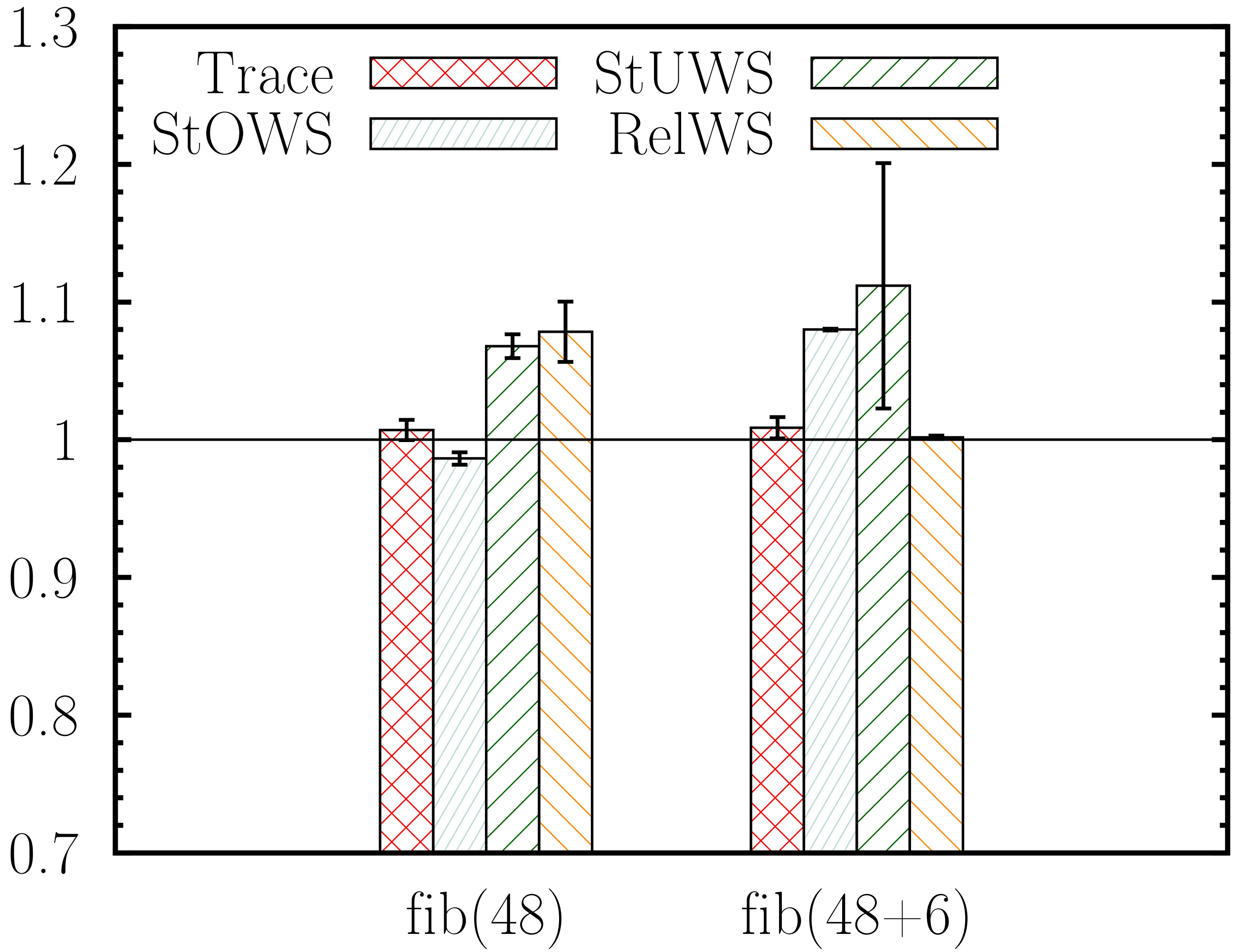
Relaxed Work Stealing: Adaptability II

Relaxed replay of schedule from (p-10) workers on p workers

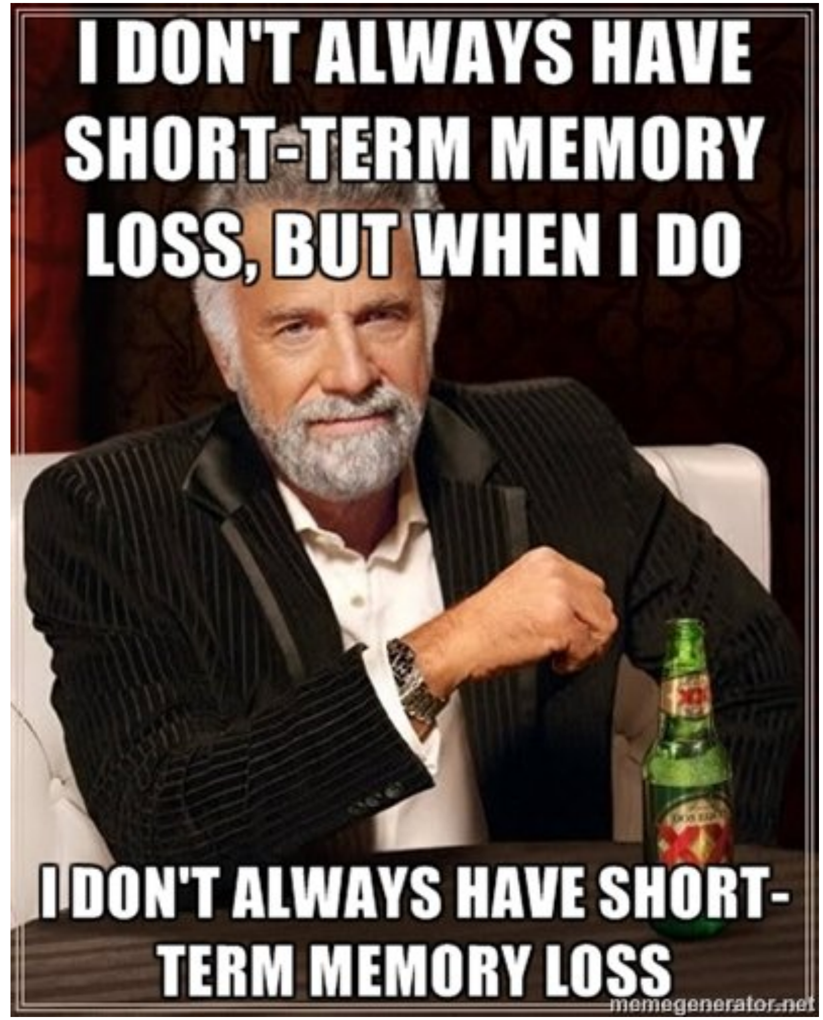


Relaxed Work Stealing: Adaptability III

Relaxed work stealing of fib(54) with a schedule from fib(48)



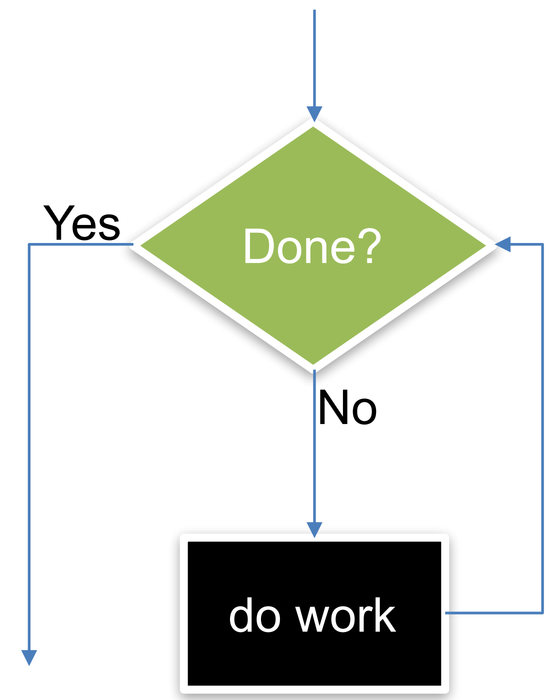
Retentive Stealing



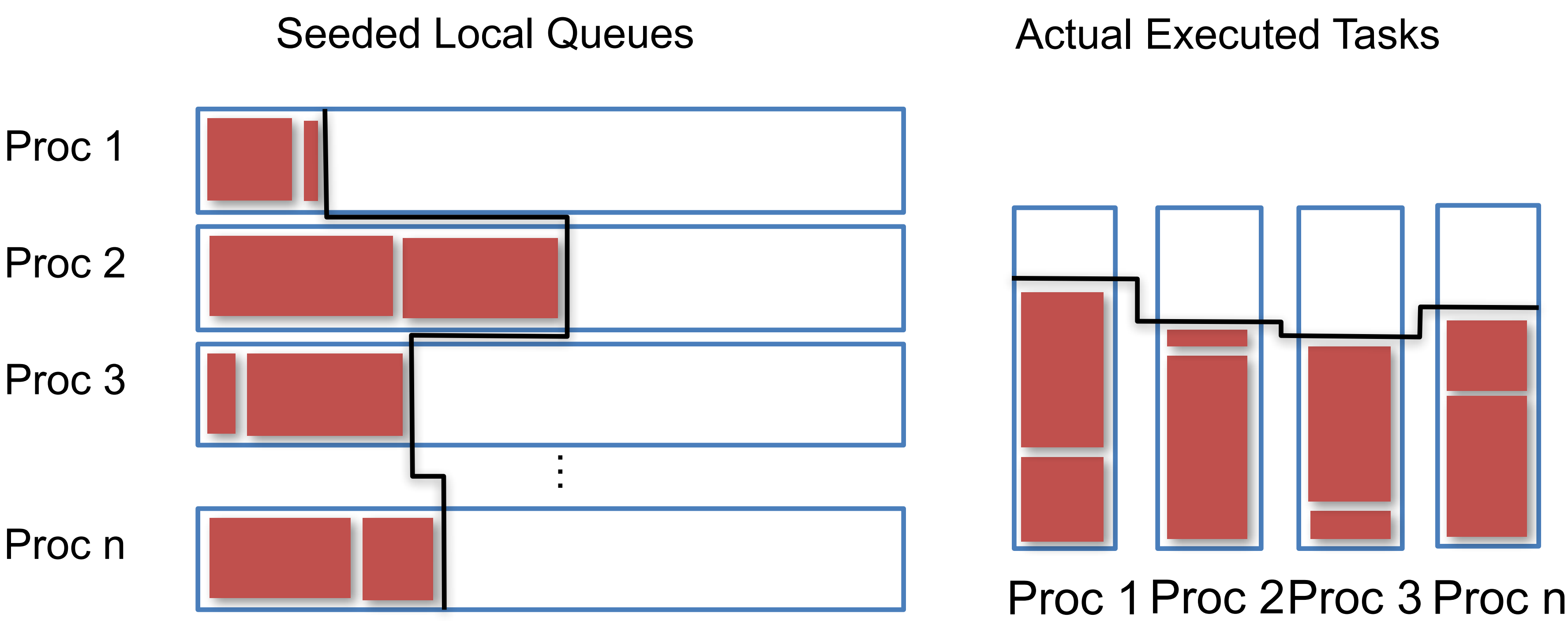
Work stealing and persistence-based load balancers for iterative overdecomposed applications. HPDC'12 http://dl.acm.org/citation.cfm?id=2287103

Iterative Applications

- Applications repeatedly executing the same computation
  - Many scientific applications are iterative
- Static or slowly evolving execution characteristics
- Execution characteristics preclude static balancing
  - Application characteristics (comm. pattern, sparsity,…)
  - Execution environment (topology, asymmetry, …)



Retentive Work Stealing



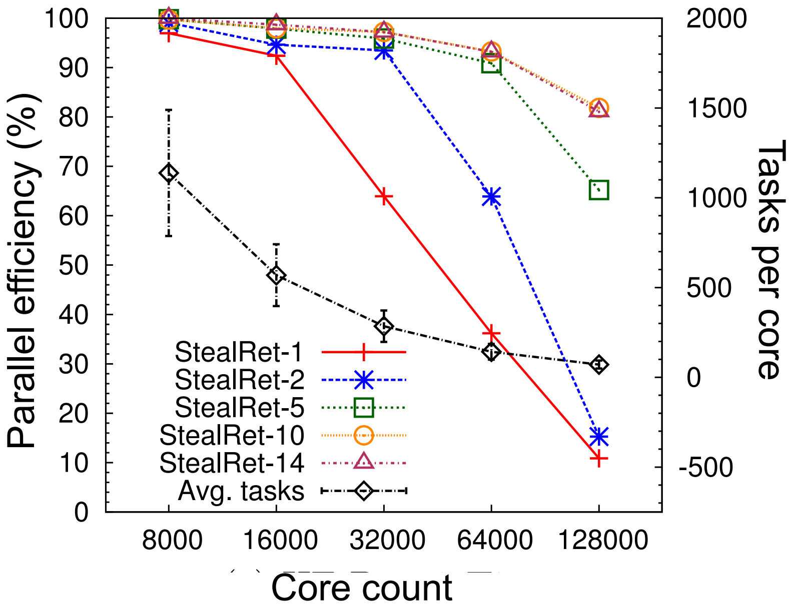
Intuition: Stealing indicates poor initial balance

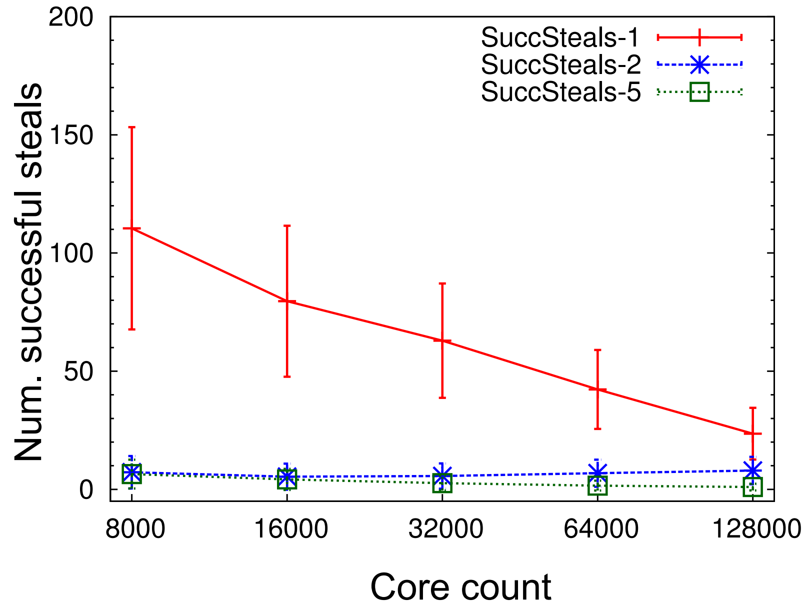
Retentive stealing

- Use work stealing to load balance within each phase
  - Persistence-based load balancers only rebalance across phases

- Begin next iteration with a trace of the previous iteration’s schedule

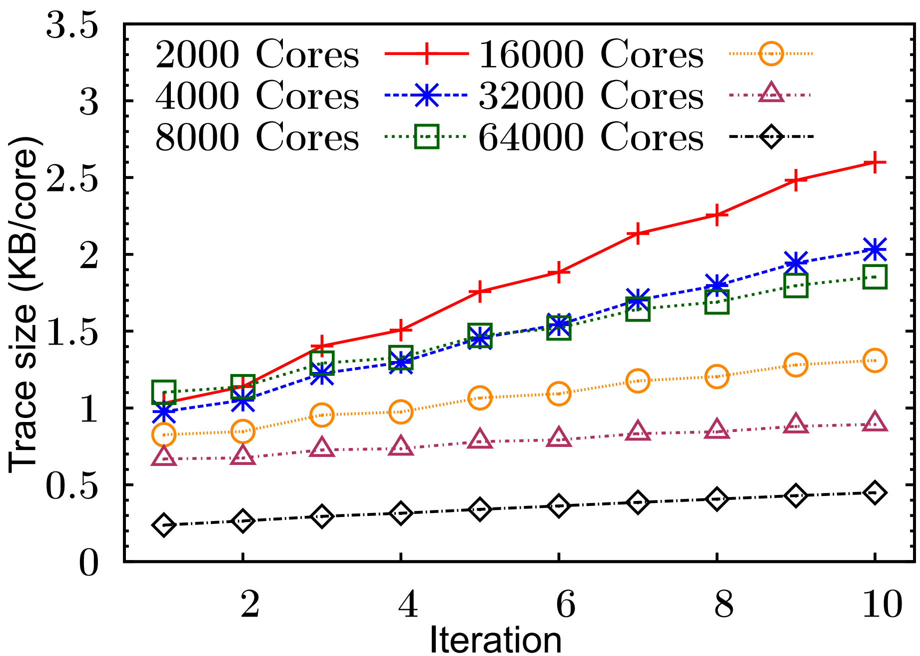
Retentive stealing results





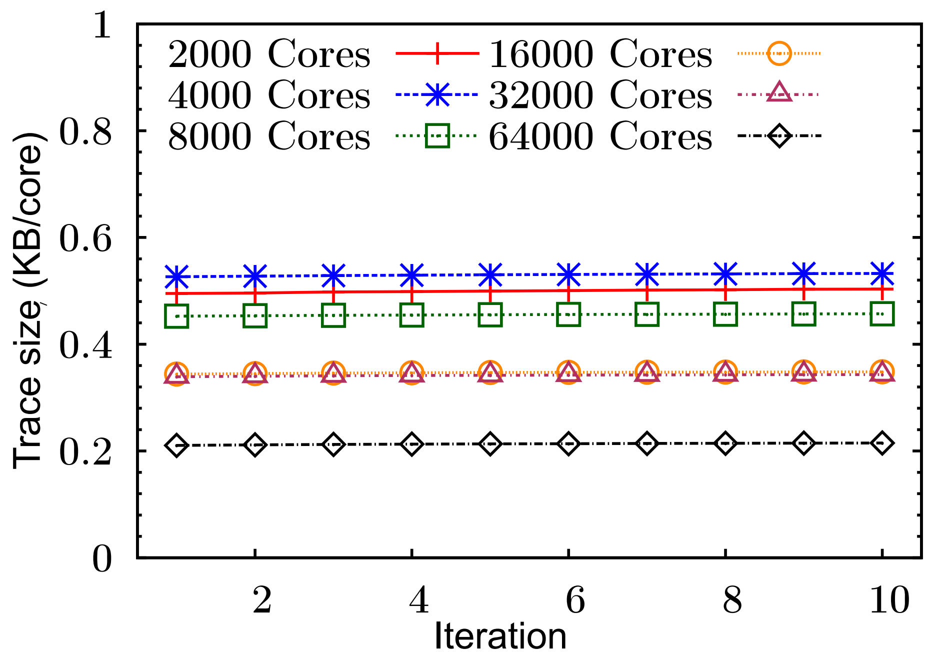
Retentive stealing stabilizes stealing costs

Retentive Stealing Space Overhead: HF



- Execution on Titan
- Space overhead increase but still same manageable across iterations

Retentive Stealing Space Overhead: TCE



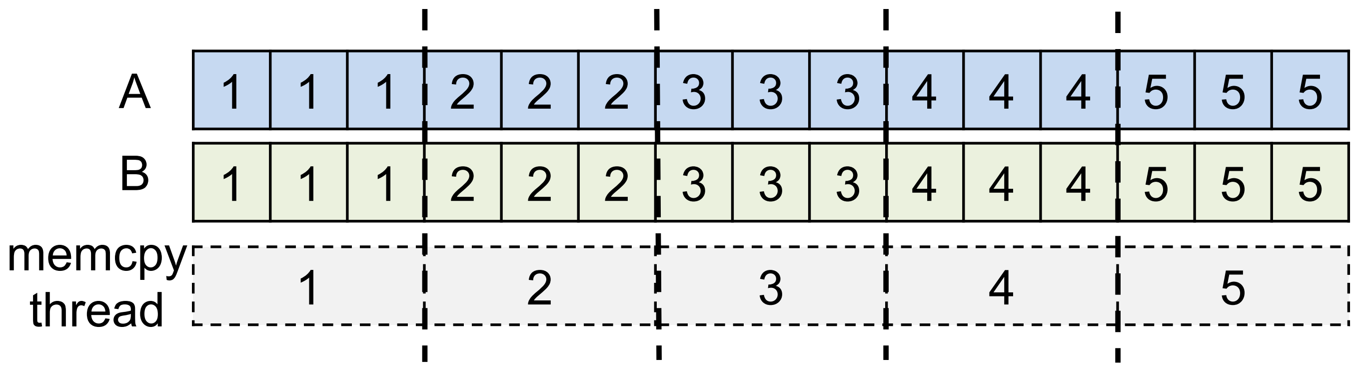
- Execution on Titan
- Space overhead stays the same across iterations

Data Locality Optimization: NUMA Locality

Optimizing data locality for fork/join programs using constrained work stealing. SC'14. http://dl.acm.org/citation.cfm?id=2683687

Constrained Schedules in OpenMP

```
#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    A[i] = B[i] = 0; //init
#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    B[i] = A[i]; //memcpy
```

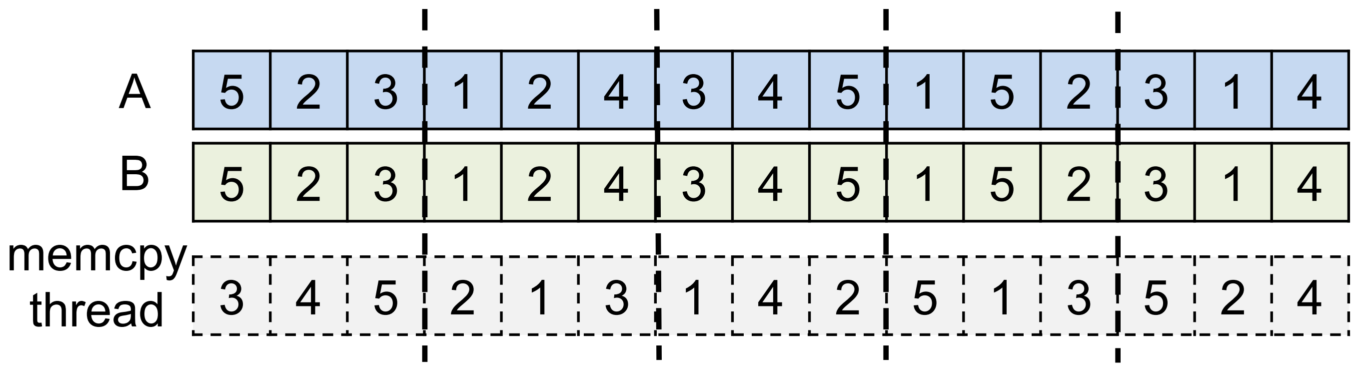


Empirical study

- Parallel memory copy of 8GB of data, using OpenMP schedule static
- 80-core system with eight NUMA domains, first-touch policy
- Execution time: 169ms

Cilk Scheduling

```
cilk_for (i = 0; i < size; i++)
    A[i] = B[i] = 0; //init
cilk_for (i = 0; i < size; i++)
    B[i] = A[i]; //memcpy
```

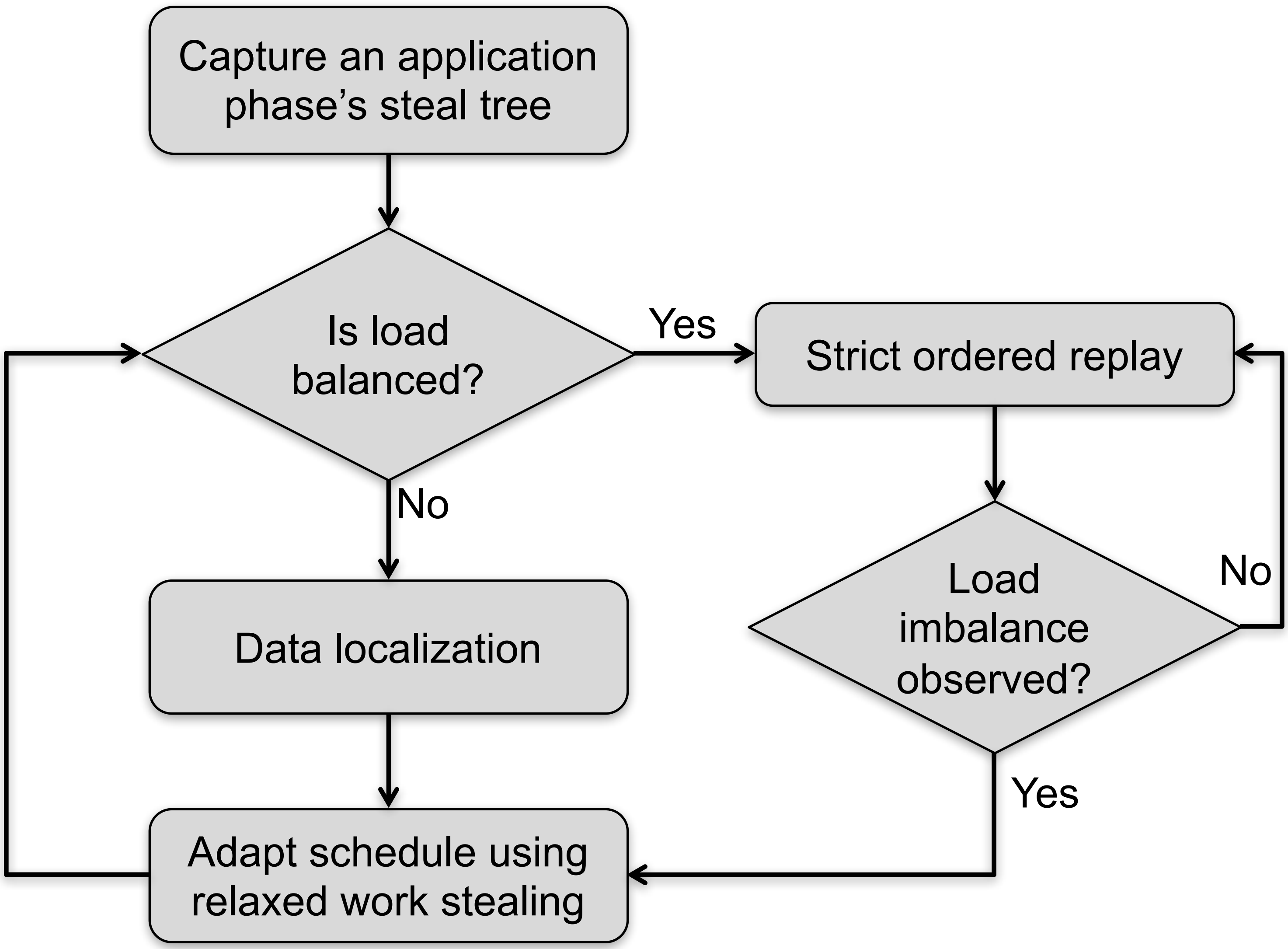


Empirical study

- Parallel memory copy of 8GB of data, using MIT Cilk or OpenMP 3.0 tasks
- Execution time: 436ms (Cilk/OMP task) vs 169ms (OMP static)

Can we constrain the scheduler to improve NUMA locality?

Solution: Evolve a Schedule

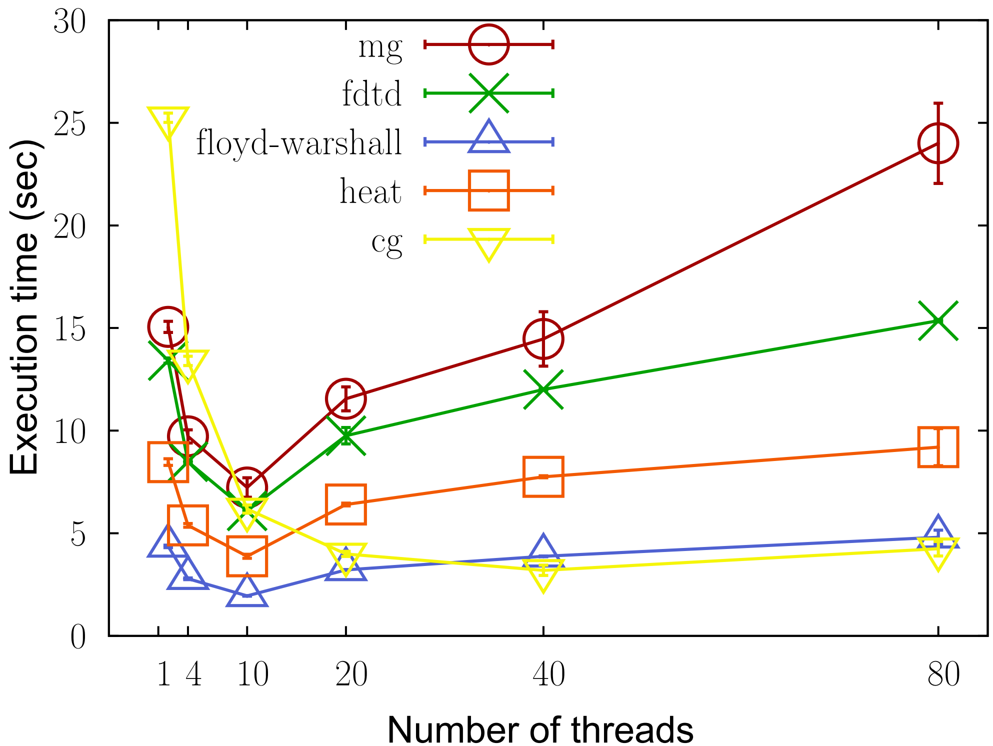


Alternative Strategy: Manual Steal Tree Construction

- Explicit markup of steal tree in the user program
- Useful in non-iterative applications

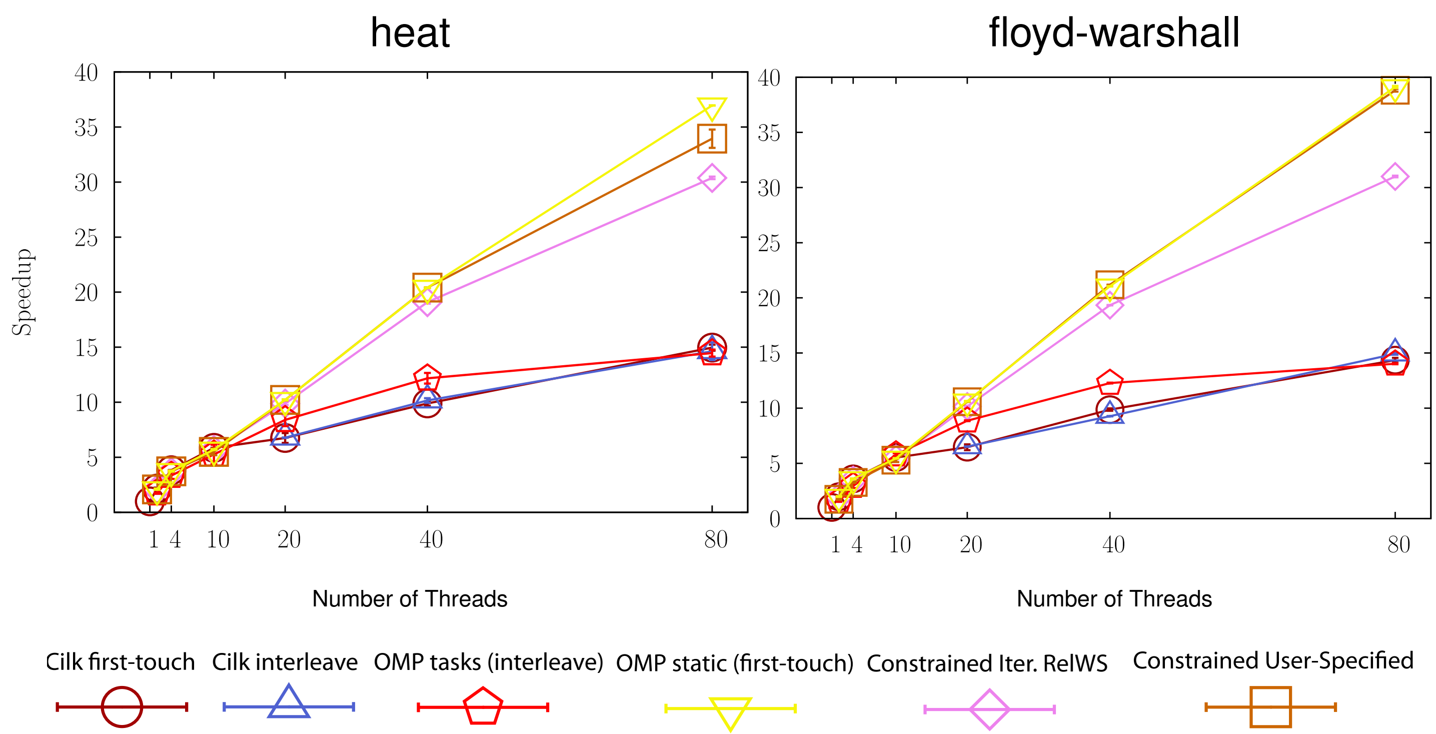
Data Redistribution Cost

First few iterations, data is redistributed (copied) to match a given schedule



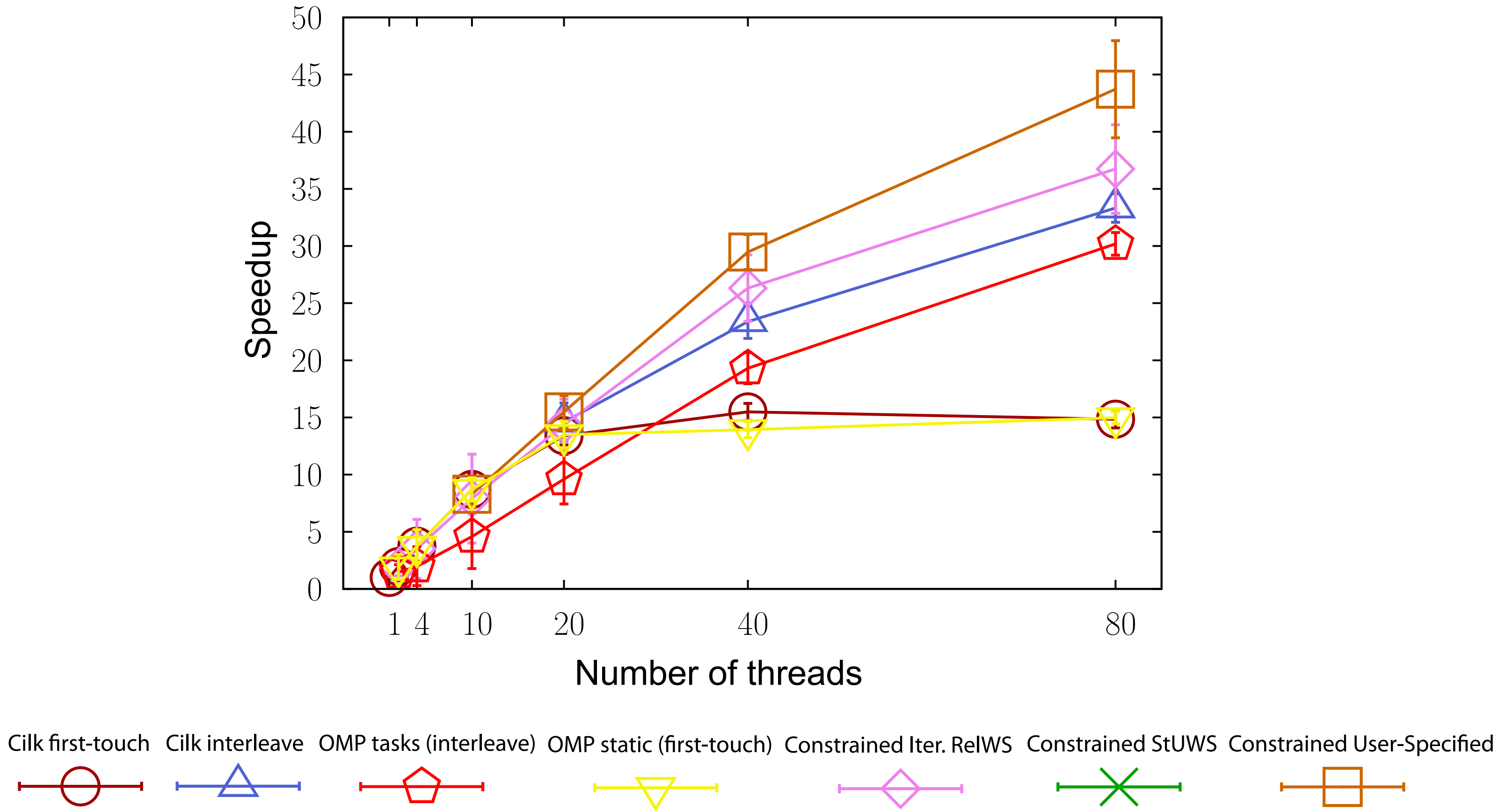
Sriram Krishnamoorthy

Benchmarks: Iterative Matching Structure



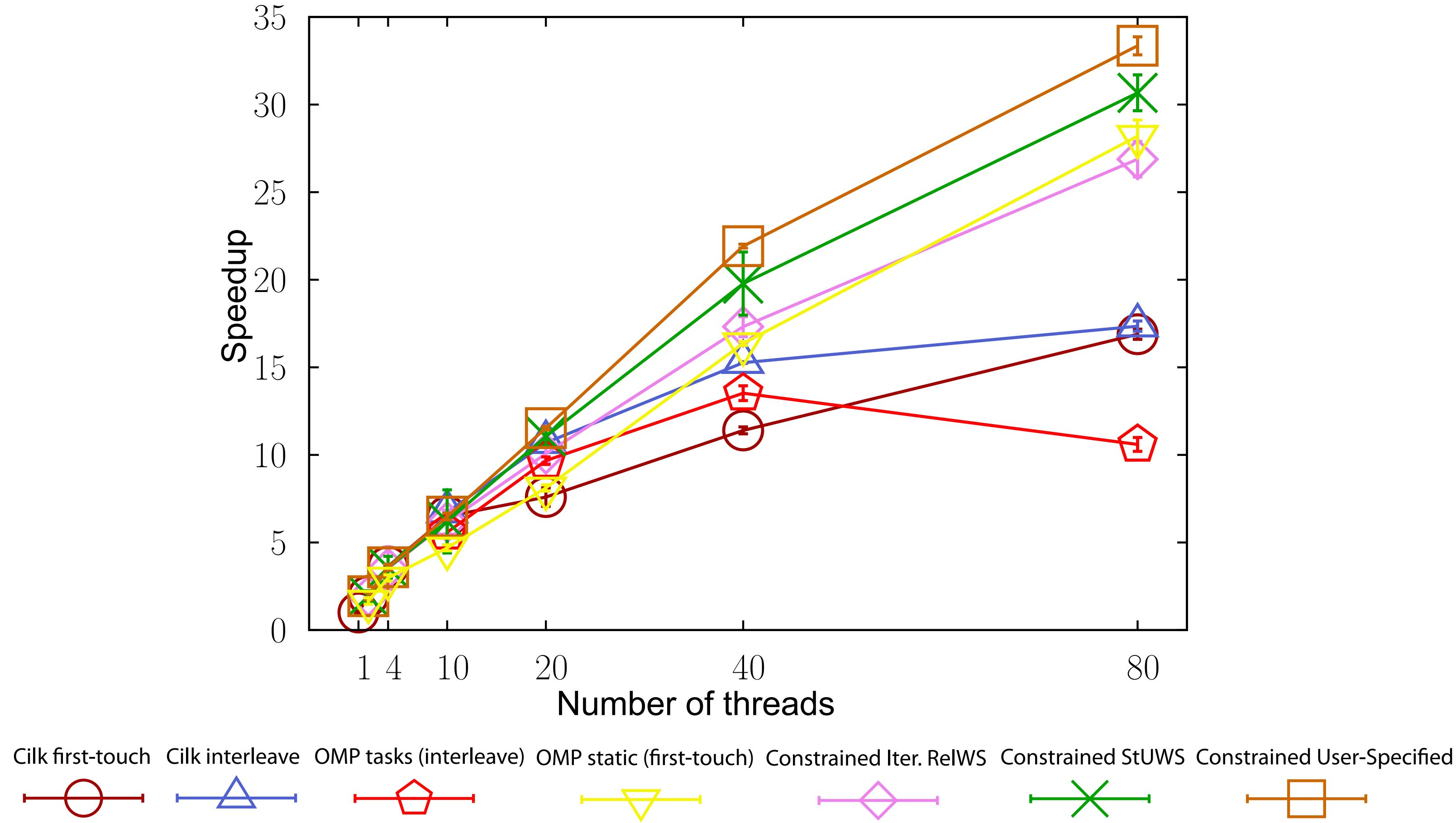
Extract template schedule, apply RelWS for five iterations until convergence, then use StOWS

Benchmarks: Iterative Differing Structure



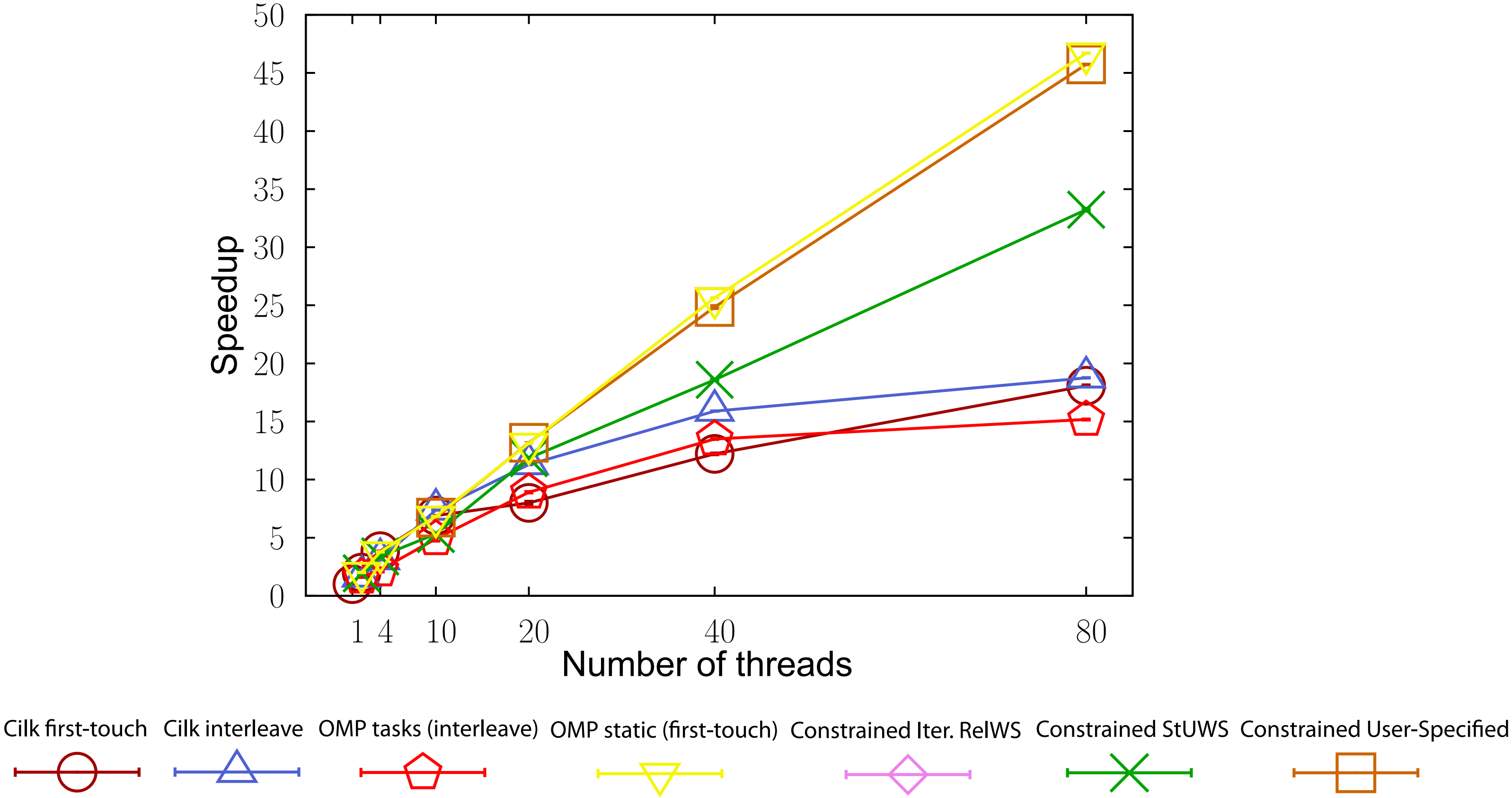
Start with random work stealing on kernel, refine with RelWS until convergence, then use StOWS

Benchmarks: Iterative Multiple Structures



We evaluate two approaches: using the same schedule across all kernels, and using a different schedule for each kernel

Benchmarks: Non-iterative Matching Structure



Reuse schedule from initialization for other phases with StUWS

Task Granularity Selection

Optimizing data locality for fork/join programs using constrained work stealing. SC'14. http://dl.acm.org/citation.cfm?id=2683687

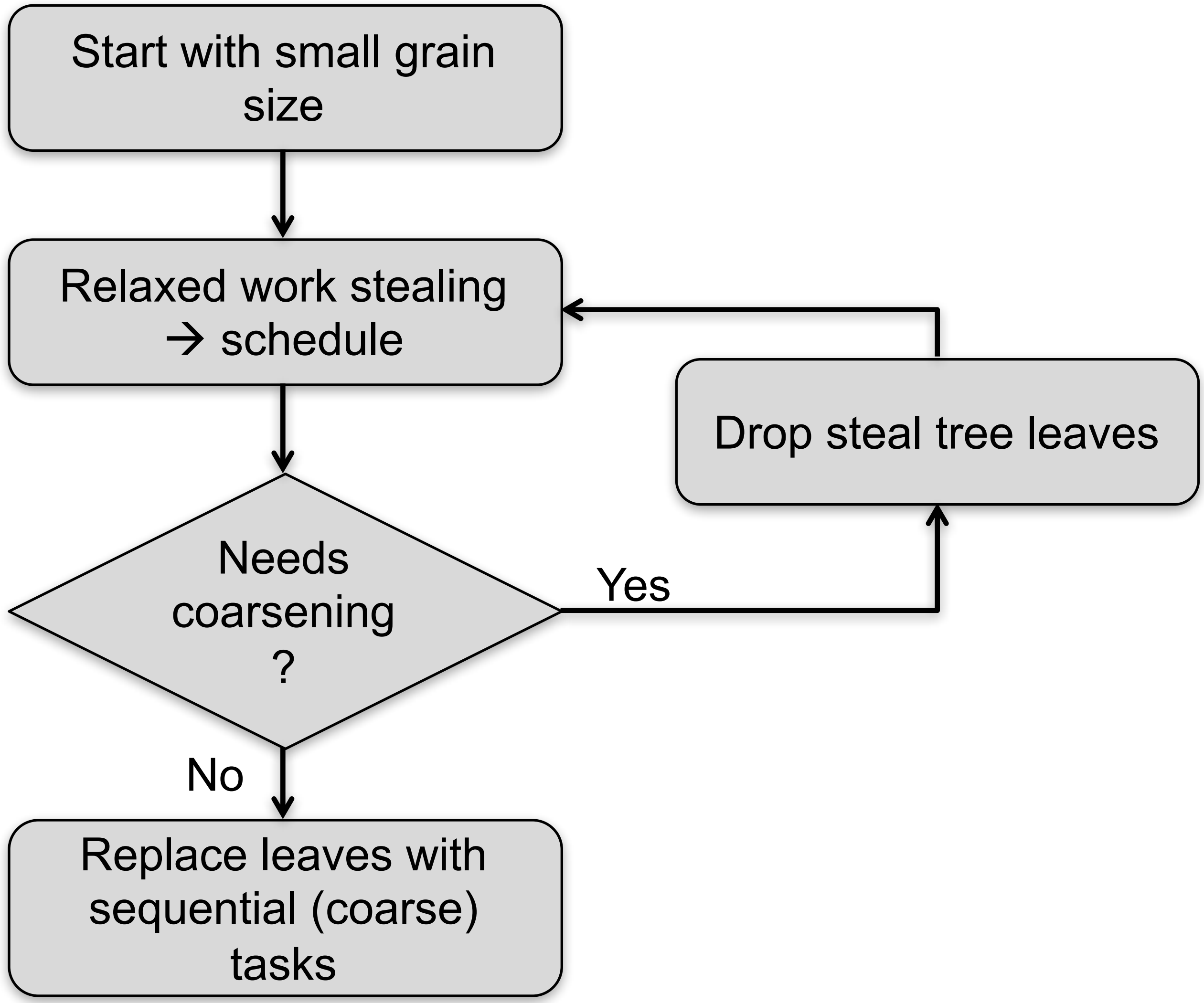
Task granularity selection

- A key challenge for task-parallel programs
- Trade-off
  - Expose more concurrency
  - Achieve good sequential performance with a coarse grain size

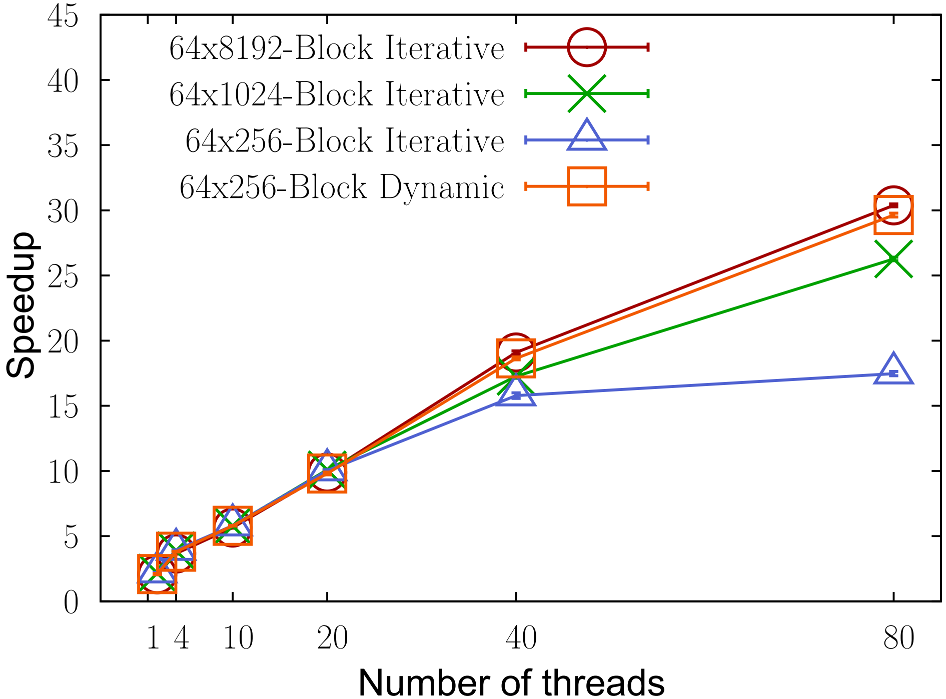
Observation

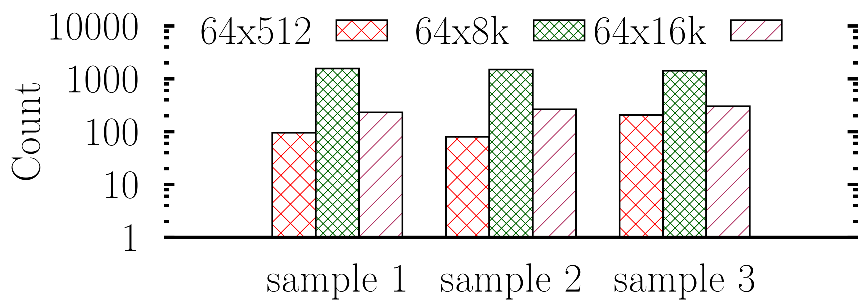
- Concurrency only need to be exposed to achieve load balance
  - Once load is balanced, exposed concurrency can be “turned off”
- We can coax the scheduler to select coarser grained work units

Iterative Granularity Selection



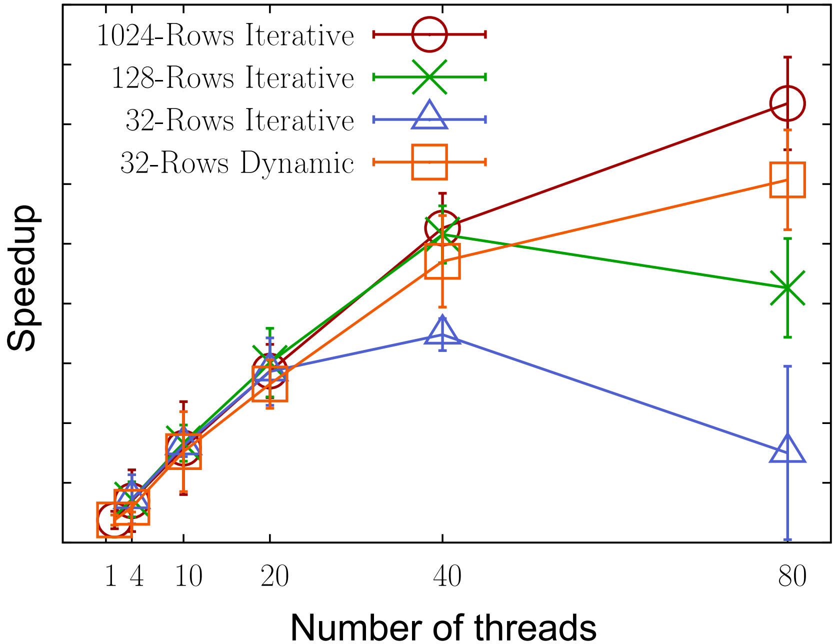
Dynamic Granularity Selection: heat

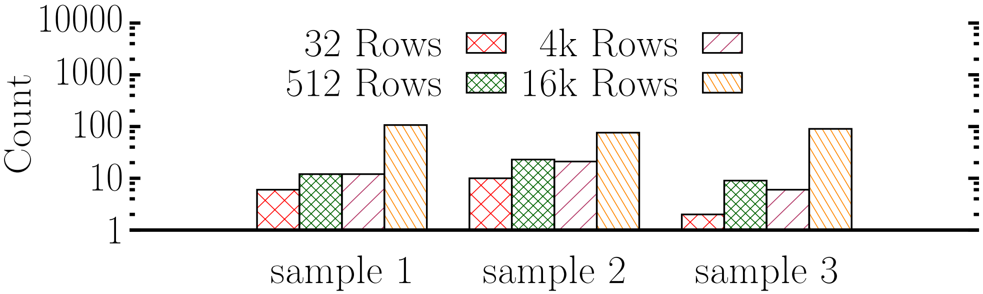




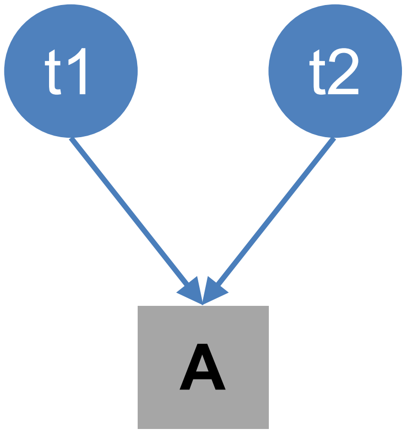
Iterative locality optimization with grain size selection

Dynamic Granularity Selection: cg





Data Race Detection



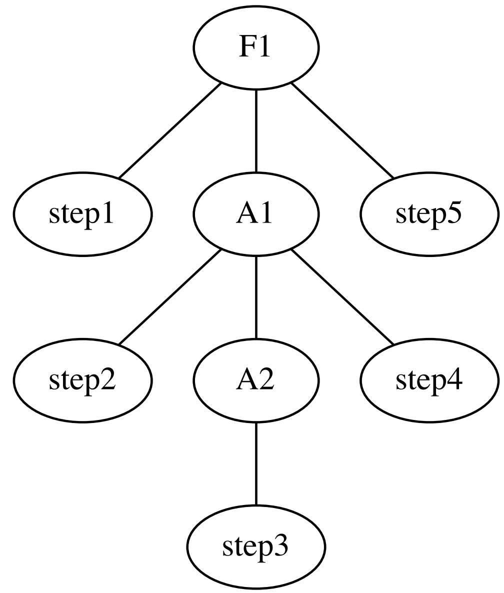
Steal tree: low-overhead tracing of work stealing schedulers.
PLDI'13 http://dl.acm.org/citation.cfm?id=2462193

Data Race Detection

- Detect conflicting operations in a fork/join program
- Key check:
  - Determine if two memory operation can execute in parallel
  - For any possible schedule

Dynamic Program Structure Tree (DPST)

```
finish {      //F1
  step1;
  async {     //A1
    step2;
    async { //A2
      step3;
    }
    step4;
  }
  step5;
}
```



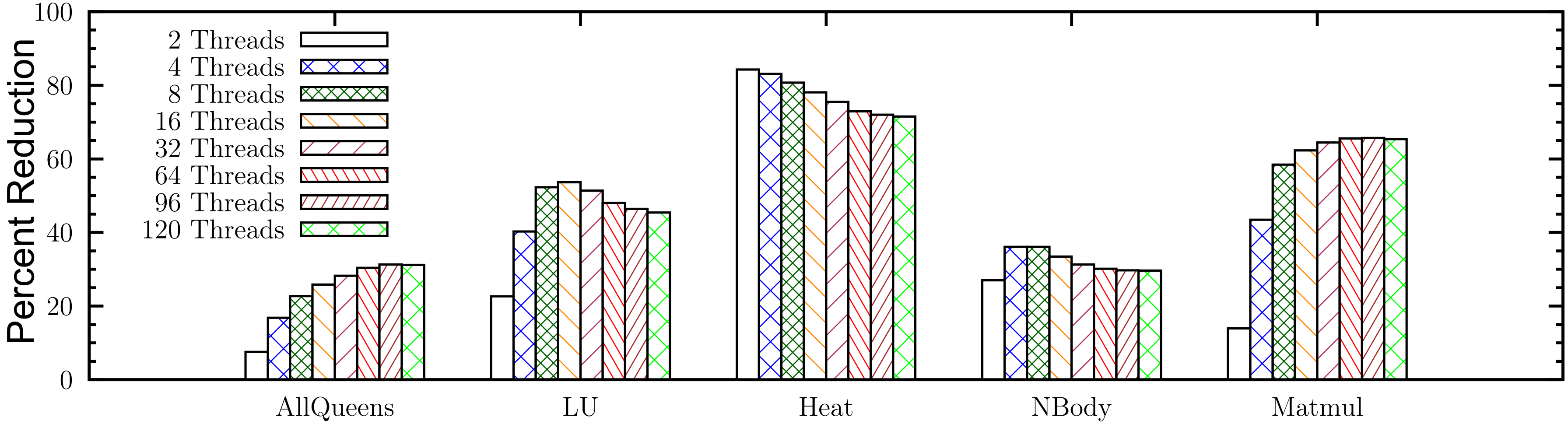
- Two steps s1 and s2 may execute in parallel if:
  - l1 is least common ancestor (LCA) of s1 and s2 in DPST
  - c1 is ancestor of s1 and immediate child of l1
  - c1 is an async node

Steal-Tree Aided LCA Computation

- The nodes of the DPST tree can be annotated with the nodes of steal tree they belong to
- Data race detection involves multiple walks of the DPST for each memory access checked

```
lca(s1, s2):
  if (s1.st_node == s2.st_node)
    return dpst_lca(s1,s2); //dpst walk
  if (s1.st_node.level > s2.st_node.level)
    return lca(s1.st_node.victim, s2)
  return lca(s1, s2.st_node.victim)
```

Application: Data Race Detection



Significant reduction in the number of DPST edges traversed

Other Results

- Locality-aware task graph scheduling
  - Color-based constraints on work stealing schedulers
- Cache locality optimization
  - Effect-based splicing of concurrent tasks to improve cache locality
- Speculative work stealing
  - Expose greater concurrency
- Localized parallel failure recovery

Lessons Learned

- Random work stealing with ability to constrain its behavior can bring several benefits

- Steal trees can be useful in a variety of contexts
  - Retentive stealing
  - Data locality optimization
  - Task granularity selection
  - Data race detection
  - …

- Need to design interfaces to programmatically extract and use work stealing schedules

Continuing Research Challenges

- Recursive program specification
- Enabling user to express high level intent and properties
- Compiler analysis and transformation
- Runtime techniques
  - Scheduling and load balancing
  - Fault tolerance
  - Power/energy efficiency
  - Data locality
- Correctness and performance tools
- Architectural and other low-level support for such abstractions

Conclusions

- Abstractions supporting task parallelism can meet performance and programmability challenges
- Runtime systems can adapt productively
  - Changing the load balancer or adding fault tolerance involved no change in the user code
- Maturing an execution paradigm requires lots of research and experience

Thank You!