Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing

Jonathan Lifflander*, Sriram Krishnamoorthy†, Laxmikant V. Kale*
{jliff12, kale}@illinois.edu, sriram@pnnl.gov

*University of Illinois Urbana-Champaign
†Pacific Northwest National Laboratory

November 20, 2014
Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated.
Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated

- Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk (gcc, icc), X10, Habanero Java
Motivation

- Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated
  - Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk (gcc, icc), X10, Habanero Java

- Work stealing is often used to schedule them
Motivation

- Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated
  - Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk (gcc, icc), X10, Habanero Java

- Work stealing is often used to schedule them
  - Well-studied dynamic load balancing strategy
Motivation

- Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated
  - Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk (gcc, icc), X10, Habanero Java

- Work stealing is often used to schedule them
  - Well-studied dynamic load balancing strategy
  - Provably efficient scheduling
Structured/task-based parallel programming (e.g. async-finish or spawn-sync) idioms have proliferated

- Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk (gcc, icc), X10, Habanero Java

Work stealing is often used to schedule them

- Well-studied dynamic load balancing strategy
- Provably efficient scheduling
- Understandable bounds on time and space
Exploring the Problem

- NUMA and Work Stealing

- Work stealing schedulers
Exploring the Problem

→ NUMA and Work Stealing

- Work stealing schedulers
  - A worker becomes a *thief* when it is idle
Exploring the Problem

→ NUMA and Work Stealing

- Work stealing schedulers
  - A worker becomes a *thief* when it is idle
  - Randomly selects a victim
Exploring the Problem

→ NUMA and Work Stealing

- Work stealing schedulers
  - A worker becomes a *thief* when it is idle
  - Randomly selects a victim
  - How might this degrade the performance in a NUMA environment?
Related work

- X10: locality-aware scheduling through explicit invocation of task execution at the location of data elements (Philippe, et al.)
- OpenMP: reuse schedules to improve memory affinity for looping constructs (Nikolopoulos, et al.)
Exploring the Problem

→ Related Work

- Related work
  - X10: locality-aware scheduling through explicit invocation of task execution at the location of data elements (Philippe, et al.)
Related work

- X10: locality-aware scheduling through explicit invocation of task execution at the location of data elements (Philippe, et al.)
- OpenMP: reuse schedules to improve memory affinity for looping constructs (Nikolopoulos, et al.)
Exploring the Problem

Related Work

- Related work
  - X10: locality-aware scheduling through explicit invocation of task execution at the location of data elements (Philippe, et al.)
  - OpenMP: reuse schedules to improve memory affinity for looping constructs (Nikolopoulos, et al.)
Can we construct a work-stealing schedule that maximizes data locality, while ensuring load balance?
Can we construct a work-stealing schedule that maximizes data locality, while ensuring load balance?

(with and **without** explicit programmer mapping?)
NUMA Policies

- First-touch
  - The *first time* memory is touched, the NUMA domain that the thread executes on determines the location of the page allocated

numactl --interleave=0,1,2,3,4,5,6,7
NUMA Policies

- **First-touch**
  - The *first time* memory is touched, the NUMA domain that the thread executes on determines the location of the page allocated

- **Interleaved**
  - Statically allocate pages in a round robin manner to the set of sockets specified

```
numactl --interleave=0,1,2,3,4,5,6,7
```
#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
# Motivating Example

→ Memory Copy: Adding Parallelism

```c
#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
```

A

1 1 1 2 2 2
3 3 3 4 4 4
5 5 5 5 5

B

1 1 1 2 2 2
3 3 3 4 4 4
5 5 5 5 5

memcpy

thread

1 2 3 4 5
Motivating Example
→ Memory Copy: Adding Parallelism

```c
#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
```

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Loops are naturally matched, leading to good performance.
Motivating Example

→ Memory Copy: Adding Parallelism

```c
#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

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

Empirical Study

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

Memory Copy: Adding Parallelism

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
Motivating Example

Memory Copy: Adding Parallelism

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

A

B

memcpy

thread

3 1 4 2 5 1 1 3 3 2 2 3 1 2 5 2
Motivating Example

→ Memory Copy: Adding Parallelism

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

Random work stealing mismatches the initialization and subsequent use, causing performance degradation.
Motivating Example

→ Memory Copy: Adding Parallelism

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, using MIT Cilk or OpenMP 3.0 Tasks
- Execution time: 436ms (Cilk/OMP task) vs. 169ms (OpenMP)
Our Approach: *Constrained Work Stealing*

1. Capture the schedule for a phase.
2. If iterative, evolve that schedule for phases with similar structure until convergence.
3. Re-use converged schedule.

OR

Build a user-specified schedule and constrain.
Our Approach: Constrained Work Stealing

(1) Capture the schedule for a phase.
Our Approach: *Constrained Work Stealing*

1. Capture the schedule for a phase.

2. If iterative, evolve that schedule for phases with similar structure until convergence.
Our Approach: *Constrained Work Stealing*

1. Capture the schedule for a phase.

2. If iterative, evolve that schedule for phases with similar structure until convergence.

3. Re-use converged schedule.
Our Approach: *Constrained Work Stealing*

1. Capture the schedule for a phase.

2. If iterative, evolve that schedule for phases with similar structure until convergence.

3. Re-use converged schedule.

OR

Build a user-specified schedule and constrain.
(1) Capturing a Work-Stealing Schedule

Using the theory in this paper, we can capture the work-stealing schedule.

Very low time and storage overhead.

Amount of information stored in practice is much smaller than $O(\text{number of tasks})$. 

Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing

Jonathan Lifflander
(1) Capturing a Work-Stealing Schedule


- Very low time and storage overhead
- Amount of information stored in practice is much smaller than \( O(\text{number of tasks}) \)
(1) Capturing a Work-Stealing Schedule


- Using the theory in this paper, we can capture the work-stealing schedule
Using the theory in this paper, we can capture the work-stealing schedule

Very low time and storage overhead
(1) Capturing a Work-Stealing Schedule


- Using the theory in this paper, we can capture the work-stealing schedule
- Very low time and storage overhead
- Amount of information stored in practice is much smaller than $\mathcal{O}(\text{number of tasks})$
(2) Evolving the Schedule

- Observations

  - The initialization phase and use phases may not match.
  - The use phases may traverse the data differently.
  - Hence, directly re-using a schedule may not be effective.

- Constrained work-stealing schedulers

  - Input is a template schedule.
  - Modify the template schedule when there is load imbalance.
  - Re-localize the data based on modified schedule.
  - Repeat this process until convergence.
(2) Evolving the Schedule

- Observations
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- Constrained work-stealing schedulers
  - Input is a template schedule
  - Modify the template schedule when there is load imbalance
  - Re-localize the data based on modified schedule
  - Repeat this process until convergence
(2) Evolving the Schedule

- Observations
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
(2) Evolving the Schedule

- **Observations**
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective
(2) Evolving the Schedule

- Observations
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- Constrained work-stealing schedulers
(2) Evolving the Schedule

- Observations
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- Constrained work-stealing schedulers
  - Input is a *template schedule*
Evolving the Schedule

- **Observations**
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- **Constrained work-stealing schedulers**
  - Input is a *template schedule*
  - Modify the template schedule when there is load imbalance
(2) Evolving the Schedule

- Observations
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- Constrained work-stealing schedulers
  - Input is a *template schedule*
  - Modify the template schedule when there is load imbalance
  - Re-localize the data based on modified schedule
(2) Evolving the Schedule

- **Observations**
  - The *initialization* phase and *use* phases may not match
  - The use phases may traverse the data differently
  - Hence, directly re-using a schedule may not be effective

- **Constrained work-stealing schedulers**
  - Input is a *template schedule*
  - Modify the template schedule when there is load imbalance
  - Re-localize the data based on modified schedule
  - Repeat this process until convergence
We have developed three schedulers:

- **Strict, ordered work stealing (STOWS)**
  - Exactly reproduce the template schedule

- **Strict, unordered work stealing (STUWS)**
  - Reproduce the template schedule, but allow the order to deviate (respecting the application's dependencies)

- **Relaxed work stealing (RELWS)**
  - Reproduce the template schedule as much as possible, but allow workers to deviate when they are idle, by further stealing work
(2) Evolving the Schedule

→ Constrained Work-Stealing Schedulers

- We have developed three schedulers:
  - Strict, ordered work stealing (STOWS)
    - Exactly reproduce the template schedule
  - Strict, unordered work stealing (STUWS)
    - Reproduce the template schedule, but allow the order to deviate (respecting the application’s dependencies)
  - Relaxed work stealing (RELWS)
    - Reproduce the template schedule as much as possible, but allow workers to deviate when they are idle, by further stealing work
We have developed three schedulers:

- **Strict, ordered work stealing (STOWS)***
  - Exactly reproduce the template schedule
- **Strict, unordered work stealing (STUWS)***
  - Reproduce the template schedule, but allow the order to deviate (respecting the application’s dependencies)
We have developed three schedulers:

- **Strict, ordered work stealing (STOWS)**
  - Exactly reproduce the template schedule
- **Strict, unordered work stealing (STUWS)**
  - Reproduce the template schedule, but allow the order to deviate (respecting the application’s dependencies)
- **Relaxed work stealing (RELWS)**
  - Reproduce the template schedule as much as possible, but allow workers to deviate when they are idle, by further stealing work
Experimental Setup

- Intel 80-core machine
  - Eight 2.27 GHz E7-8860 processors, each with 10 cores
  - Connected via Intel QPI 6.4 GT/s
  - 2 TB of DRAM
  - Compiled with GNU GCC version 4.3.4
  - We tried using OpenMP with ICC (Intel OpenMP implementation), but we found no significant scaling difference

- Machine runs Red Hat Linux version 4.4.7-3
  - Configured to use 4 KB pages
  - All of our codes set the affinity of threads
    - First 10 threads always go to a single socket
Experimental Setup

- Intel 80-core machine
  - Eight 2.27 GHz E7-8860 processors, each with 10 cores
  - Connected via Intel QPI 6.4 GT/s
  - 2 TB of DRAM
  - Compiled with GNU GCC version 4.3.4
  - MIT Cilk 5.4.6 translator or GCC and OpenMP 3.0 (version 200805)
    - We tried using OpenMP with ICC (Intel OpenMP implementation), but we found no significant scaling difference
  - Machine runs Red Hat Linux version 4.4.7-3
Experimental Setup

- Intel 80-core machine
  - Eight 2.27 GHz E7-8860 processors, each with 10 cores
  - Connected via Intel QPI 6.4 GT/s
  - 2 TB of DRAM
  - Compiled with GNU GCC version 4.3.4
  - MIT Cilk 5.4.6 translator or GCC and OpenMP 3.0 (version 200805)
    - We tried using OpenMP with ICC (Intel OpenMP implementation), but the we found no significant scaling difference
  - Machine runs Red Hat Linux version 4.4.7-3
  - Configured to use 4 KB pages
Experimental Setup

- Intel 80-core machine
  - Eight 2.27 GHz E7-8860 processors, each with 10 cores
  - Connected via Intel QPI 6.4 GT/s
  - 2 TB of DRAM
  - Compiled with GNU GCC version 4.3.4
  - MIT Cilk 5.4.6 translator or GCC and OpenMP 3.0 (version 200805)
    - We tried using OpenMP with ICC (Intel OpenMP implementation), but we found no significant scaling difference
  - Machine runs Red Hat Linux version 4.4.7-3
  - Configured to use 4 KB pages
  - All of our codes set the affinity of threads
    - First 10 threads always go to a single socket
(2) Evolving the Schedule

→ **RelWS**: How well does it work?
(2) Evolving the Schedule

→ RelWS: How well does it work?
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem</th>
<th>Configuration</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>heat</td>
<td>nx = ny = 32768</td>
<td>block = 64x8192</td>
<td>2k</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>n = 32768</td>
<td>block = 64x4096</td>
<td>4k</td>
</tr>
<tr>
<td>fdtd</td>
<td>ey = ex = hz = 32768</td>
<td>block = 64x8192</td>
<td>2k</td>
</tr>
<tr>
<td>NAS cg</td>
<td>NA=${2^{21}}$, NNZ=15</td>
<td>rows = 1024</td>
<td>2k</td>
</tr>
<tr>
<td>NAS mg</td>
<td>N{X,Y,Z}=1024,LM=11</td>
<td>block=16x16x4MB</td>
<td>64–4k</td>
</tr>
<tr>
<td>parallel prefix</td>
<td>N = 256 MB</td>
<td>block = 512</td>
<td>512</td>
</tr>
</tbody>
</table>
(3) Re-using the Schedule

→ Overhead of Constrained Work Stealing (on 80 Cores)

Mean normalized ratio (y-axis) compared to default Cilk implementation. Error bars are relative standard deviation with a sample size of 5.
Building a User-specified Schedule

- The user builds a mapping using an API we provide

API: designateAfterNextSpawn(int worker)

STUWS is used to schedule that mapping

The runtime builds a Steal Tree that is used as a template
Building a User-specified Schedule

- The user builds a mapping using an API we provide
  - API: `designateAfterNextSpawn(int worker)`

STUWS is used to schedule that mapping

The runtime builds a Steal Tree that is used as a template to schedule
Building a User-specified Schedule

- The user builds a mapping using an API we provide
  - API: `designateAfterNextSpawn(int worker)`
  - STUWS is used to schedule that mapping
Building a User-specified Schedule

- The user builds a mapping using an API we provide
  - API: `designateAfterNextSpawn(int worker)`
  - SUWS is used to schedule that mapping
  - The runtime builds a Steal Tree that is used as a template schedule
We have grouped the applications into several different categories

- Iterative, matching structure (heat, fdtd, floyd-warshall)
  - Extract template schedule, apply RELWS for five iterations until convergence, then use STOWS

- Iterative, differing structure (NAS cg)
  - Start with random work-stealing on kernel, refine with RELWS until convergence, then use STOWS

- Iterative, multiple structures (NAS mg)
  - We evaluate two approaches: using the same schedule across all kernels, and using a different schedule for each kernel

- Non-iterative, matching structure (parallel prefix)
  - Re-use schedule from initialization for other phases with STUWS
We have grouped the applications into several different categories

- Iterative, matching structure (heat, fdtd, floyd-warshall)
  - Extract template schedule, apply RELWS for five iterations until convergence, then use STOWS

- Iterative, differing structure (NAS cg)
  - Start with random work-stealing on kernel, refine with RELWS until convergence, then use STOWS

- Iterative, multiple structures (NAS mg)
  - We evaluate two approaches: using the same schedule across all kernels, and using a different schedule for each kernel

- Non-iterative, matching structure (parallel prefix)
  - Re-use schedule from initialization for other phases with STOWS
Whole Program Locality Optimization

- We have grouped the applications into several different categories
  - Iterative, matching structure (heat, fdtd, floyd-warshall)
    - Extract template schedule, apply RELWS for five iterations until convergence, then use STOWS
  - Iterative, differing structure (NAS cg)
    - Start with random work-stealing on kernel, refine with RELWS until convergence, then use STOWS
Whole Program Locality Optimization

- We have grouped the applications into several different categories
  - Iterative, matching structure (heat, fdtd, floyd-warshall)
    - Extract template schedule, apply RELWS for five iterations until convergence, then use STOWS
  - Iterative, differing structure (NAS cg)
    - Start with random work-stealing on kernel, refine with RELWS until convergence, then use STOWS
  - Iterative, multiple structures (NAS mg)
    - We evaluate two approaches: using the same schedule across all kernels, and using a different schedule for each kernel
We have grouped the applications into several different categories

- **Iterative, matching structure (heat, fdtd, floyd-warshall)**
  - Extract template schedule, apply RElWS for five iterations until convergence, then use STOWS
- **Iterative, differing structure (NAS cg)**
  - Start with random work-stealing on kernel, refine with RElWS until convergence, then use STOWS
- **Iterative, multiple structures (NAS mg)**
  - We evaluate two approaches: using the same schedule across all kernels, and using a different schedule for each kernel
- **Non-iterative, matching structure (parallel prefix)**
  - Re-use schedule from initialization for other phases with STUWS
Whole Program Locality Optimization

→ Data redistribution cost (for the first few iterations)
Whole Program Locality Optimization
→ Iterative, matching structure

Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing

Cilk first-touch  Cilk interleave  OMP tasks (interleave)  OMP static (first-touch)  Constrained Iter. RelWS  Constrained User-Specified

Heat

Floyd-Warshall
Whole Program Locality Optimization

→ Iterative, differing structure

![Graph showing speedup vs. number of threads for different optimization techniques: Cilk first-touch, Cilk interleave, OMP tasks (interleave), OMP static (first-touch), Constrained Iter. RelWS, and Constrained User-Specified.]
Whole Program Locality Optimization

→ Iterative, multiple structures
Whole Program Locality Optimization

→ Non-iterative, matching structure
Dynamic Coarsening

Finding the ideal grain size is difficult:

- Too large leads to load imbalance
- Too small increases runtime overheads

Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance.

- Steals higher in the Steal Tree correspond to large portions of work.

We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using $STUWS$.

Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size.

Details are in the paper.
Dynamic Coarsening

- Finding the ideal grain size is difficult
Dynamic Coarsening

- Finding the ideal grain size is difficult
  - Too large leads to load imbalance
Finding the ideal grain size is difficult
  - Too large leads to load imbalance
  - Too small increases runtime overheads
Dynamic Coarsening

- Finding the ideal grain size is difficult
  - Too large leads to load imbalance
  - Too small increases runtime overheads
  - Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance

- We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using STUWS

- Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size

- Details are in the paper
Dynamic Coarsening

- Finding the ideal grain size is difficult
  - Too large leads to load imbalance
  - Too small increases runtime overheads
  - Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance
  - Steals higher in the Steal Tree correspond to large portions of work

- We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using STUWS

- Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size

- Details are in the paper
Dynamic Coarsening

- Finding the ideal grain size is difficult
  - Too large leads to load imbalance
  - Too small increases runtime overheads
  - Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance
  - Steals higher in the Steal Tree correspond to large portions of work
  - We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using STUWS

Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size. Details are in the paper.
Finding the ideal grain size is difficult

- Too large leads to load imbalance
- Too small increases runtime overheads
- Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance
- Steals higher in the Steal Tree correspond to large portions of work
- We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using SΤUWS
- Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size
Dynamic Coarsening

- Finding the ideal grain size is difficult
  - Too large leads to load imbalance
  - Too small increases runtime overheads
  - Key observation: all parts of the Steal Tree do not equally contribute to locality and load balance
  - Steals higher in the Steal Tree correspond to large portions of work
  - We start with a fine-grained schedule and iteratively coarsen by pruning the Steal Tree and using \( STUWS \)
  - Using this technique we are able to achieve nearly the same performance as using the optimal chunk size, but starting with a much smaller chunk size
  - Details are in the paper
Conclusion

- We present a comprehensive approach to improving NUMA locality for work stealing:

  ▶ User-specified
  ▶ Automatic
  ▶ Up to 2.5x performance improvement on 80 cores compared to default Cilk!

- Future work
  ▶ Can we use static compiler analysis to better match phases and understand access patterns?
Conclusion

- We present a comprehensive approach to improving NUMA locality for work stealing:
  - User-specified
  - Automatic

Up to 2.5x performance improvement on 80 cores compared to default Cilk!

Future work
- Can we use static compiler analysis to better match phases and understand access patterns?
We present a comprehensive approach to improving NUMA locality for work stealing:

- User-specified
- Automatic
- Up to 2.5x performance improvement on 80 cores compared to default Cilk!
Conclusion

- We present a comprehensive approach to improving NUMA locality for work stealing:
  - User-specified
  - Automatic
  - Up to 2.5x performance improvement on 80 cores compared to default Cilk!

- Future work
  - Can we use static compiler analysis to better match phases and understand access patterns?
Questions?
Evolving the Schedule

Constrained Work-Stealing Schedulers

- Default scheduler
- StOWS scheduler
- StUWS scheduler
- ReIWS scheduler

Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing

Jonathan Lifflander