Almost Deterministic Work Stealing

SC'19

Shumpei Shiina, Kenjiro Taura

The University of Tokyo

11/20/2019

Overview

- **Work Stealing** is a popular scheduling algorithm for **Task Parallelism**
- However, data locality of **Work Stealing** is not good

$\longrightarrow$ We propose **Almost Deterministic Work Stealing (ADWS)** to solve this problem

Visualization of task mapping

- Simulation of 2D dambreaking
- Colors of cells represent ranks of workers (blue: 0 $\longrightarrow$ red: 63)

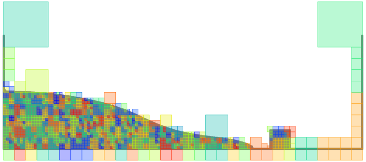


Fig. Random Work Stealing

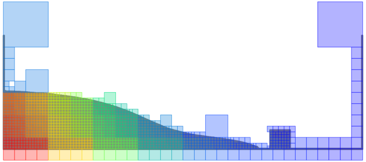


Fig. ADWS (no steal)

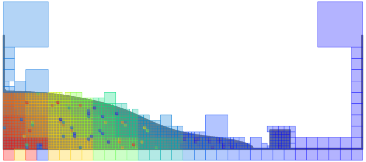


Fig. ADWS

Outline

Outline

Introduction

- Motivating Example: Calculation of Particle Interactions
- Task Parallelism and Work Stealing
- Data Locality in Work Stealing

Proposed Method: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

Evaluation

Related Work

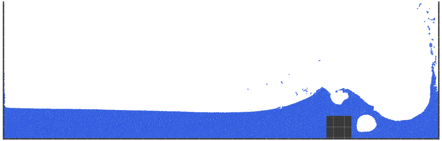
Conclusion

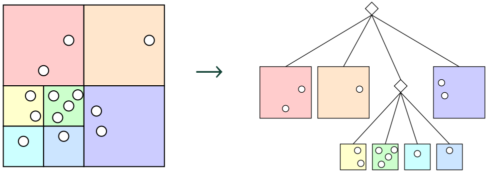
Outline

Motivating Example: Calculation of Particle Interactions

2D dambreaking simulation

- Smoothed Particle Hydrodynamics (SPH) which calculates short-range forces
- Particles are managed in a **quadtree** (an **octree** in 3D)
- The quadtree is usually **unbalanced**





Parallelization while Traversing the Tree

Sequential Code

```
particle_interaction(node) {
  if (node is leaf) {
    /* Calculate particle interactions
     * in leaf node */
  } else {
    for (child in node.children) {
      particle_interaction(child);
    }
  }
}
```

Task Parallel Code

```
particle_interaction(node) {
  if (node is leaf) {
    /* Calculate particle interactions
     * in leaf node */
  } else {
    task_group tg;
    for (child in node.children) {
      /* Spawn a child node as a task (fork) */
      tg.run([=]{ particle_interaction(child); });
    }
    /* Wait for completion of tasks (join) */
    tg.wait();
  }
}
```

Outline

Introduction

- Motivating Example: Calculation of Particle Interactions
- Task Parallelism and Work Stealing
- Data Locality in Work Stealing

Proposed Method: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

Evaluation

Related Work

Conclusion

Task Parallelism

- Parallel execution model by specifying dependencies between tasks
  - Directed Acyclic Graph (**DAG**)
- **Fork-join** pattern is frequently used
  - This paper focuses on **nested** fork-join programs

```
task_group tg;
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.wait();
```

Fig. TBB-like Task Group Notation

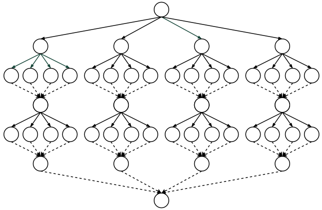
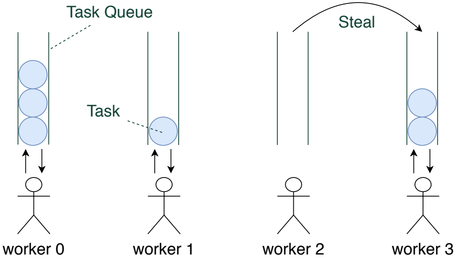


Fig. Directed Acyclic Graph (DAG)

Work Stealing[1]

- Frequently used strategy to schedule task parallel programs
- Each worker has its own **task queue**, and pushes/pops tasks to/from the queue
- If tasks are exhausted in its local queue, it tries to **steal** tasks from other workers
- Usually victims are chosen **randomly**
  - We call it random work stealing



1 R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," J. ACM, vol. 46, no. 5, pp. 720–748, 1999.

Outline

Introduction

- Motivating Example: Calculation of Particle Interactions
- Task Parallelism and Work Stealing
- Data Locality in Work Stealing

Proposed Method: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

Evaluation

Related Work

Conclusion

1. Data Locality in DAGs

- Close nodes in DAGs tend to touch close data
  - We want to schedule close nodes in close cores
- Much more important in hierarchical memory architectures
  - What if worker {0, 1}, {2, 3} are in the same NUMA nodes?

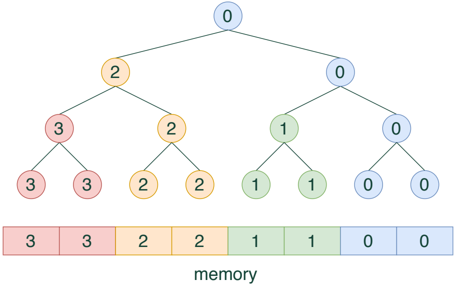


Fig. Good Data Locality

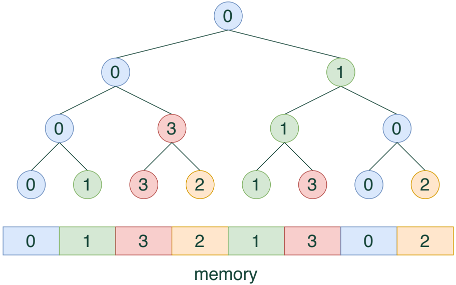
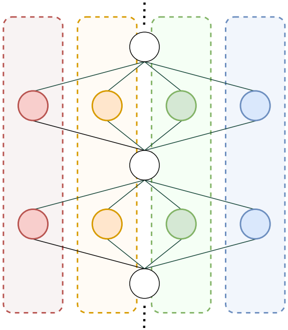


Fig. Bad Data Locality

2. Data Locality in Iterative Programs

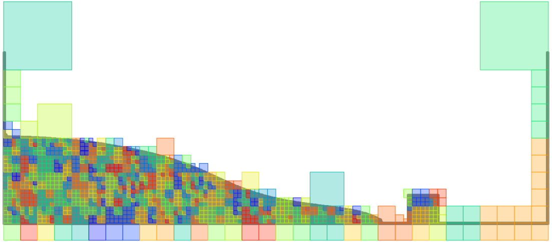
- Iterative programs have similar DAGs across iterations
  - e.g, programs that iterate an array over and over
- Data locality exists “vertically” in DAGs
- If scheduling is **deterministic**, data locality is good



Bad Data Locality in Random Work Stealing

Data locality is usually damaged by its **randomness**

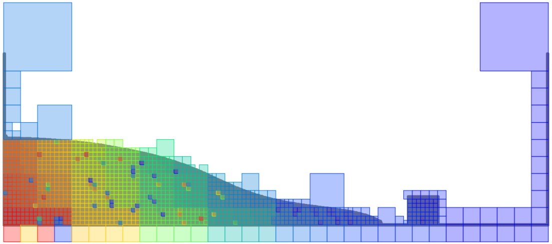
1. Data Locality in DAGs
   - Steal strategy is **unaware of memory hierarchy**
2. Data Locality in Iterative Programs
   - Scheduling is **not deterministic** across iterations



Good Data Locality in ADWS

Almost Deterministic Work Stealing (ADWS) improves both data locality

1. **Data Locality in DAGs**
   - Improved by task mapping that **matches task hierarchy with memory hierarchy**
2. **Data Locality in Iterative Programs**
   - Improved by **almost deterministic** scheduling across iterations
   - ADWS also does dynamic load balancing



Outline

Introduction

- Motivating Example: Calculation of Particle Interactions
- Task Parallelism and Work Stealing
- Data Locality in Work Stealing

Proposed Method: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

Evaluation

Related Work

Conclusion

ADWS Consists of Two Parts

1. **Deterministic Task Allocation**
   - Initial deterministic task mapping
   - Static partitioning for nested fork-join programs
2. **Hierarchical Localized Work Stealing**
   - Dynamic load balancing
   - Performs work stealing in a hierarchical manner

Task Hierarchy and Memory Hierarchy

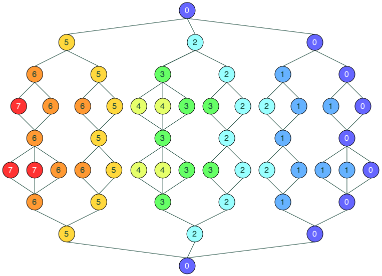


Fig. Desired Scheduling of a DAG

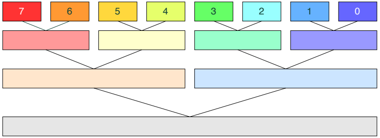


Fig. Example of Memory Hierarchy

- **Task mapping respects memory hierarchy**
  - Close workers touch close nodes in DAGs
- Without a priori knowledge, it seems impossible
  - What kind of information is needed?

Hints from Programmers

Programmers must specify **the amount of work for each task** explicitly

- It does not have to be absolute values; relative values are OK
  (ratio of `w_1, ..., w_4` to `w_all`)
- **Rough estimates** are acceptable thanks to dynamic load balancing at runtime
- It is usually **hardware-independent** and application-specific
  - e.g. the number of particles (next slide)

```
task_group tg(w_all);
tg.run([]{ ... }, w_1);
tg.run([]{ ... }, w_2);
tg.run([]{ ... }, w_3);
tg.run([]{ ... }, w_4);
tg.wait();
```

where `w_1 + w_2 + w_3 + w_4 == w_all`

Specifying Hints is Not So Hard

We can just use the number of particles in particle interactions

Particle Interactions in ADWS

```
particle_interaction(node) {
  if (node is leaf) {
    /* Calculate particle interactions in leaf node */
  } else {
    task_group tg(node.n_particles);
    for (child in node.children) {
      tg.run([=]{ particle_interaction(child); }, child.n_particles);
    }
    tg.wait();
  }
}
```

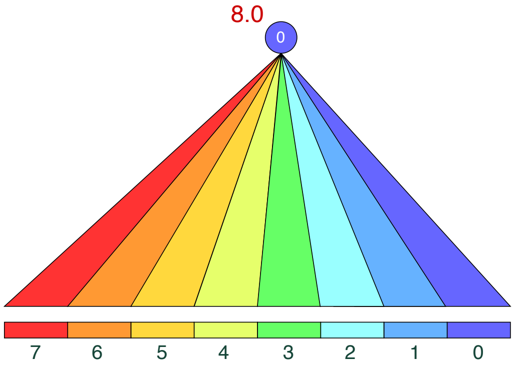
- The number of particles is a rough estimate

Outline

Overview of Deterministic Task Allocation (1/4)

- Circles: tasks
- Triangles: worker ranges
- Bottom rectangles: workers

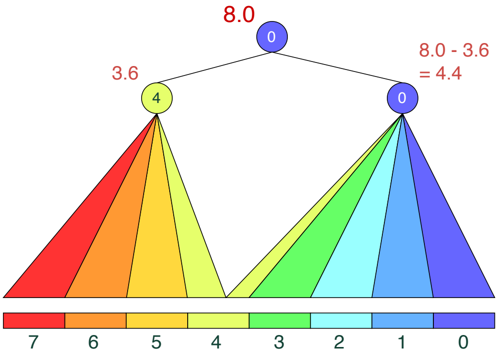
- Initially, there is only one task
- We want to distribute it to all workers



Overview of Deterministic Task Allocation (2/4)

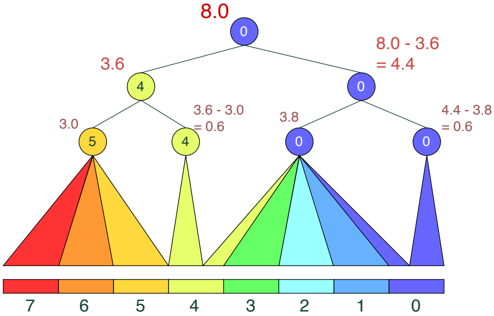
- Left task: the spawned task
- Right task: the continuation

- A new task is spawned
- Split the worker range into two parts based on the amount of work specified by programmers
- A task is executed by a worker whose rank is the smallest in its worker range



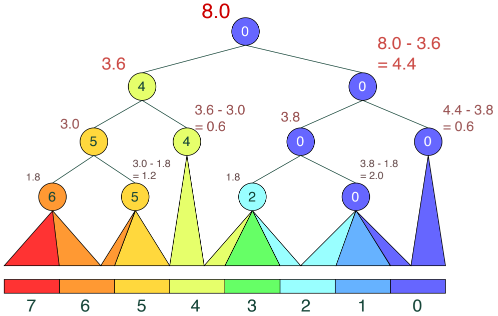
Overview of Deterministic Task Allocation (3/4)

- Continue to split worker ranges recursively and in parallel



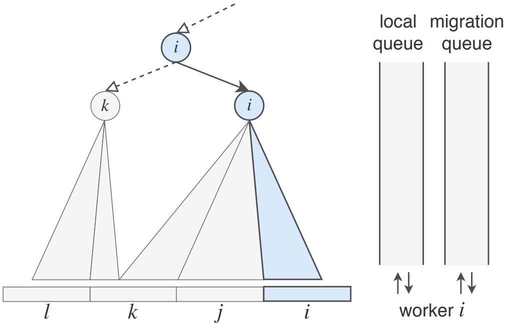
Overview of Deterministic Task Allocation (4/4)

- Task distribution proceeds while workers are executing actual tasks



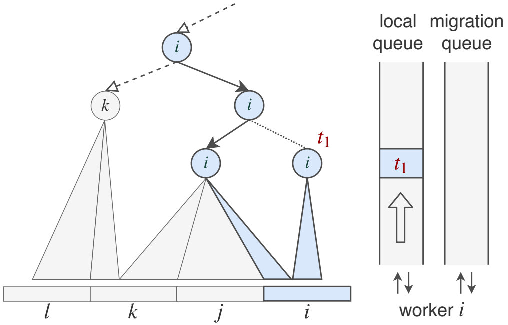
Algorithm of Deterministic Task Allocation (1/3)

- Workers **search** for left boundary of their **work region**
- If worker range is split at worker k
    - Worker i pushes a task to worker k



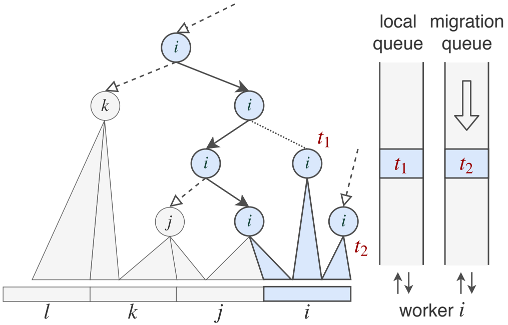
Algorithm of Deterministic Task Allocation (2/3)

- If worker range is split at worker i itself
  - Worker i pushes the continuation to **local queue**
  - Worker i executes the spawned task (left)



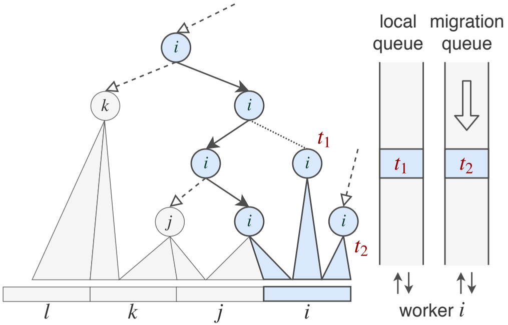
Algorithm of Deterministic Task Allocation (3/3)

- Tasks from other workers are pushed to **migration queue**



Characteristics of Deterministic Task Allocation

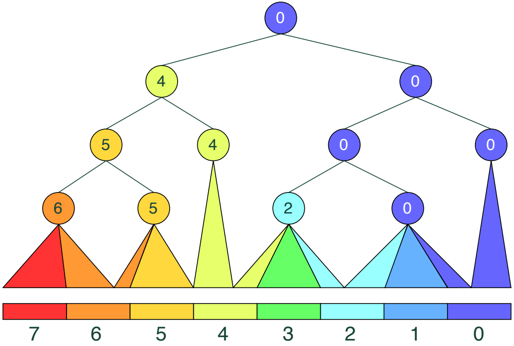
- Tasks are executed from left to right
  - The same order as serial execution
  - **Work-first** scheduling policy
- **Workers do not push tasks to a migration queue simultaneously**
  - No lock contention while searching
  - Please read the paper for more details



Outline

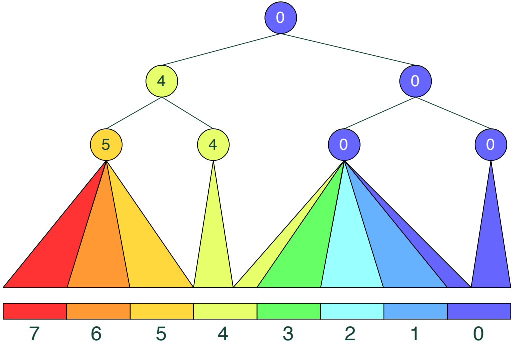
Hierarchical Localized Work Stealing (1/4)

- It depends on **Deterministic Task Allocation**
- Limit the range of steals to inside its "group"



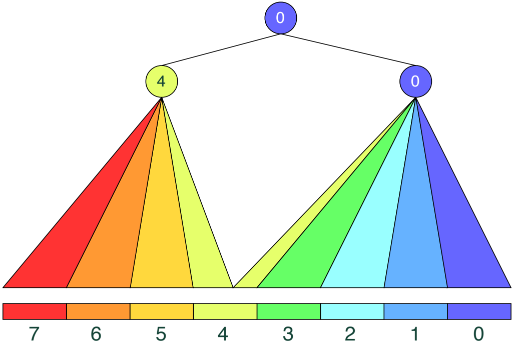
Hierarchical Localized Work Stealing (2/4)

- Move to its parent group when the current task group completes



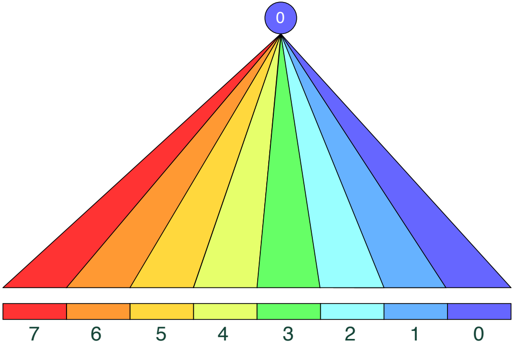
Hierarchical Localized Work Stealing (3/4)

- It follows partitioning of deterministic task allocation from bottom up



Hierarchical Localized Work Stealing (4/4)

- It becomes equivalent to random work stealing at last
- Ideally, few tasks are ready at this time



Outline

Experiment Environment

Implement ADWS on MassiveThreads[2], a lightweight threading library

- Skylake 6130 @ 2.1 GHz
- 4 sockets
- 4 NUMA nodes
- 16 x 4 = 64 cores

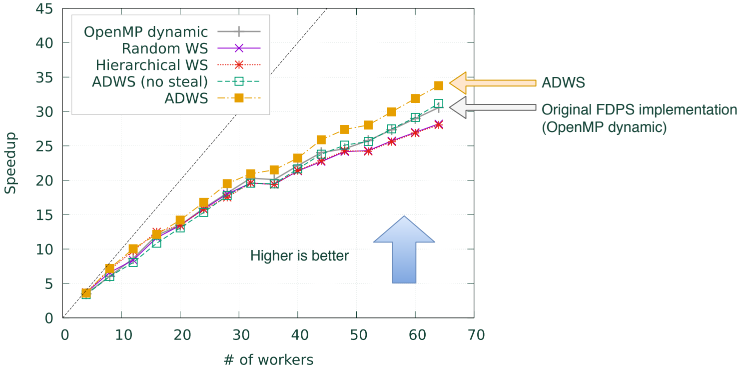
2 J. Nakashima and K. Taura, “MassiveThreads: A thread library for high productivity languages,” in Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday. Springer Berlin Heidelberg, 2014, pp. 222–238.

Performance Evaluation of Particle Interactions

We modified FDPS[3] to use nested fork-join parallelism

- Original implementation of FDPS uses GNU OpenMP parallel for (dynamic)

- # of particles: 138968
- 2D dam breaking



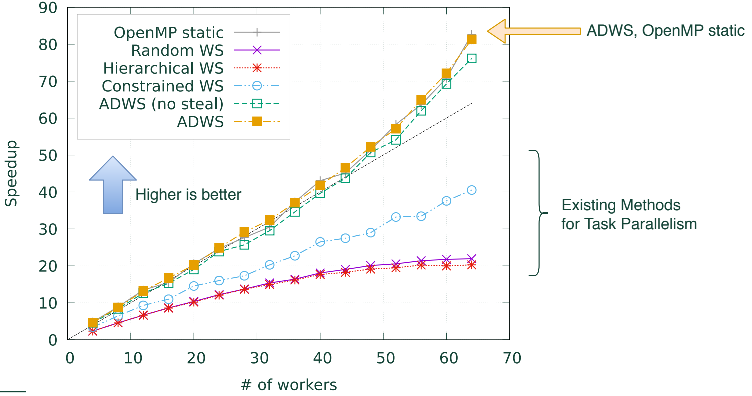
3 M. Iwasawa, A. Tanikawa, N. Hosono, et al., “Implementation and performance of FDPS: A framework for developing parallel particle simulation codes,” Publications of the Astronomical Society of Japan, vol. 68, no. 4, 2016.

Performance Evaluation of Heat2D

Highly **memory-bound** and **iterative** application (5-point stencil)

- It divides a 2D region into four parts recursively

- optimized (SIMD)
- 4096x4096 matrices
- cutoff = 64x64
- single precision
- Constrained WS: [4]



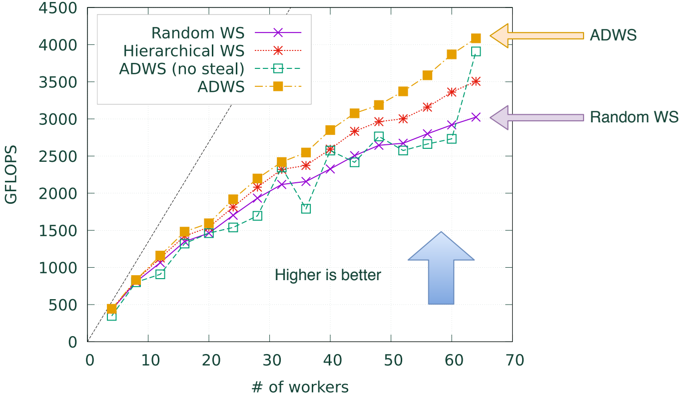
[4] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 857–868

Performance Evaluation of Matrix Multiplication

Not iterative application using a simple **divide-and-conquer** algorithm

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- optimized (SIMD)
- 4096x4096 matrices
- cutoff = 128x128
- single precision
- Hierarchical WS: [5]



5 S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11), vol. 625, 2011

Outline

Introduction
- Motivating Example: Calculation of Particle Interactions
- Task Parallelism and Work Stealing
- Data Locality in Work Stealing

Proposed Method: Almost Deterministic Work Stealing (ADWS)
- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

Evaluation

Related Work

Conclusion

Comparison to Related Work

- **Optimize scheduling by using metrics of previous iterations**[6][7][8]
  - ADWS is not specific to iterative programs
- **Optimize scheduling by using users' hardware-specific hints**[9][10]
  - ADWS requires users' hints, but they are not hardware-specific
- **Optimize a steal strategy based on memory hierarchy without hints**[11]
  - It does not optimize data locality of iterative programs

[6] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2000, pp. 1–12.

[7] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 857–868.

[8] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-aware work-stealing for multi-socket multi-core architectures," in Proceedings of the 28th ACM International Conference on Supercomputing, ACM, 2014, pp. 3–12.

[9] Y. Guo, J. Zhao, V. Cave, et al., "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12.

[10] J. Deters, J. Wu, Y. Xu, et al., "A NUMA-aware provably-efficient task-parallel platform based on the work-first principle," in 2018 IEEE International Symposium on Workload Characterization (IISWC), 2018, pp. 59–70.

[11] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11), vol. 625, 2011.

Outline

Conclusion

We have presented Almost Deterministic Work Stealing (ADWS), which:

- focuses on **nested fork-join parallelism**
- improves **data locality** in work stealing
    - memory hierarchy-aware deterministic scheduling

ADWS requires users' hints, but

- it is **hardware-independent** and application-specific
- it keeps **portability** of code

ADWS can speedup task parallel programs while keeping productivity

Future Work

- Automatic work estimation for iterative programs
  - Programmers do not have to specify hints
- More benchmarks
- Combine with cache-aware scheduling like CATS[12]

[12] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in Proceedings of the 26th ACM International Conference on Supercomputing, ACM, 2012, pp. 163–172.