Almost Deterministic Work Stealing

Shumpei Shiina, Kenjiro Taura

The University of Tokyo {shiina, tau}@eidos.ic.i.u-tokyo.ac.jp

Background – What is Task Parallelism?

Task parallelism is a form of parallelization done by specifying dependencies between tasks. In task parallelism, you can create many tasks at any point in your program, like Fig. 1. A computation of task parallelism is expressed in the form of directly acyclic graph (DAG) as shown in Fig. 2. For example, we can simulate particle interaction like smoothed particle hydrodynamics (SPH) (Fig. 4) in a straight-forward way by traversing the octree (Fig. 5), as shown in Fig. 6.

Work Stealing [1] is a popular scheduling strategy for task parallel programs. Workers have their own queue and they execute tasks in it, and when tasks are exhausted, they try to steal a task from other workers (Fig. 3). In random work stealing, workers choose victims randomly, which leads to **the data locality problem**; that is, for iterative applications, workers do not touch the same task at each iteration, and workers in the same NUMA node do not execute tasks close in the DAG.

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

Fig. 1 TBB-like fork-join notation

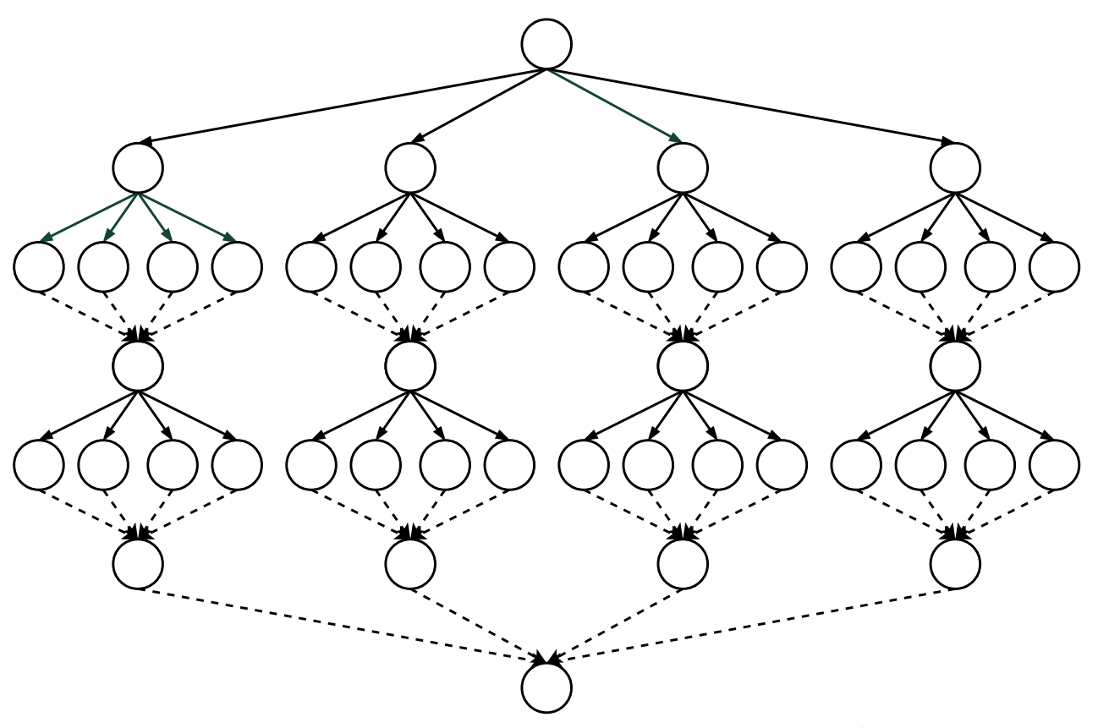


Fig. 2 Directly Acyclic Graph (DAG)

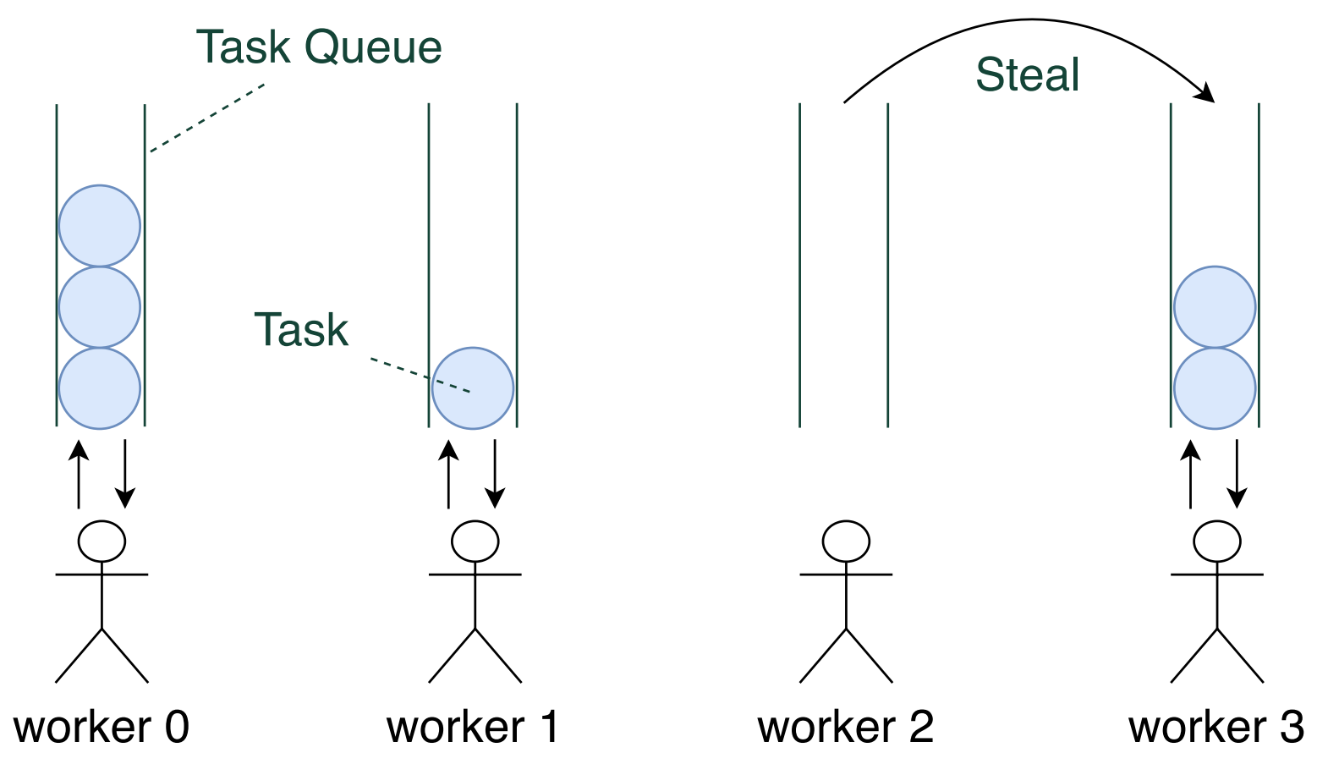


Fig. 3 Work Stealing

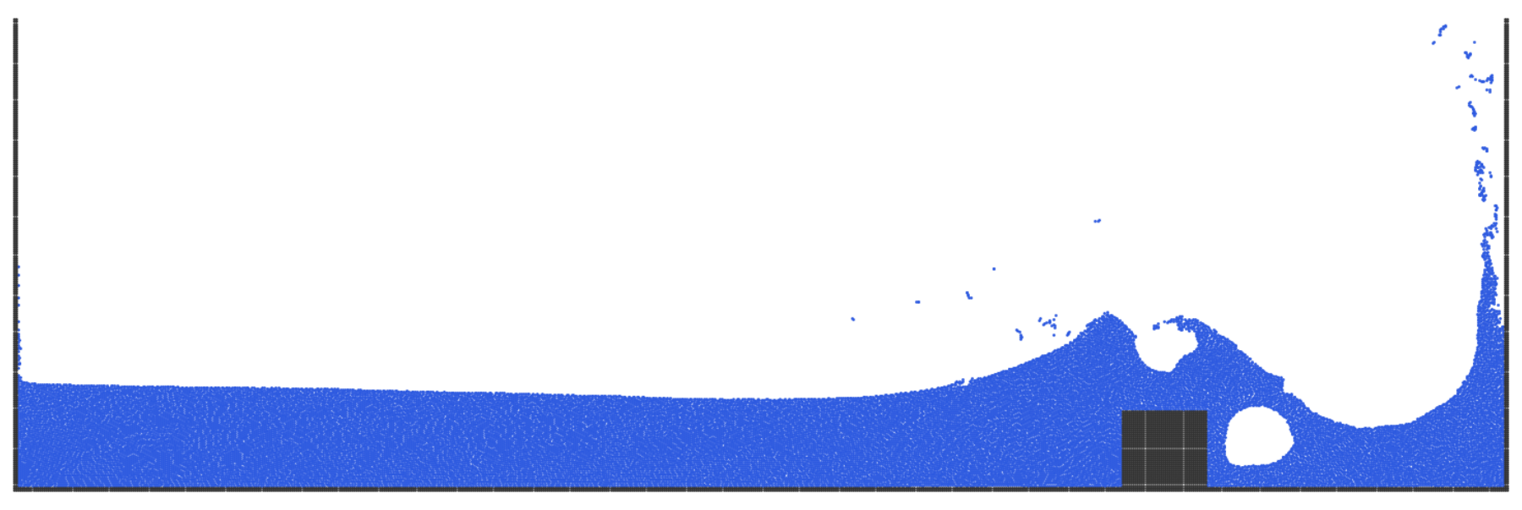


Fig. 4 2D dam breaking simulation with SPH

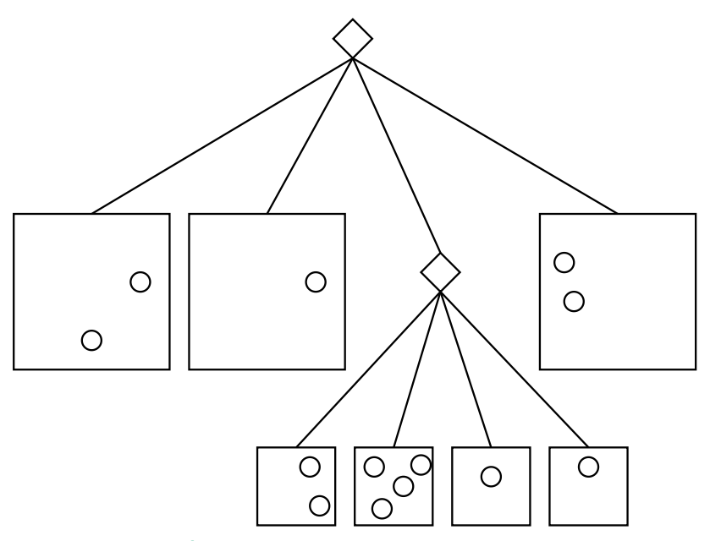


Fig. 5 Octree

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

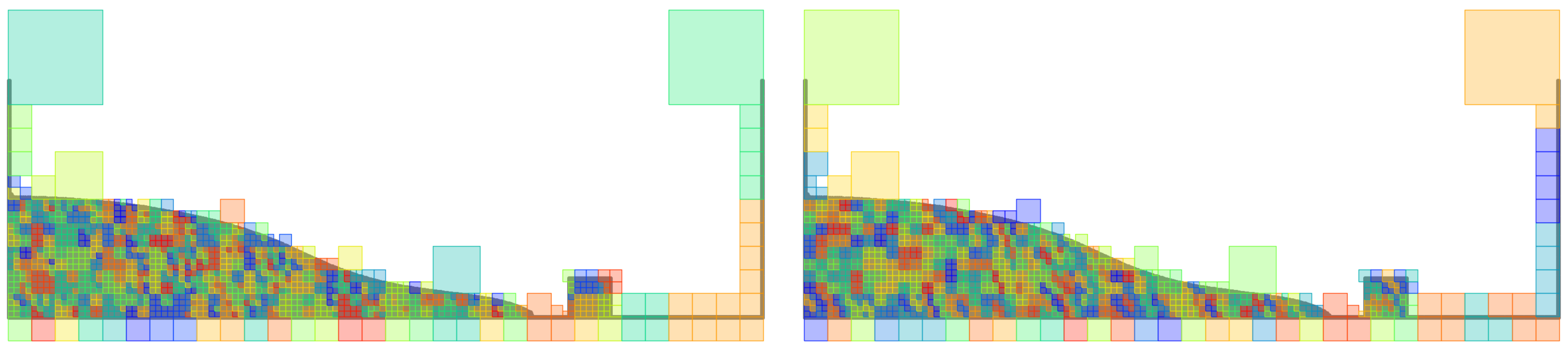
Fig. 6 Pseudocode of calculation of particle interaction parallelized by using fork-join model

Proposed Method – Almost Deterministic Work Stealing (ADWS)

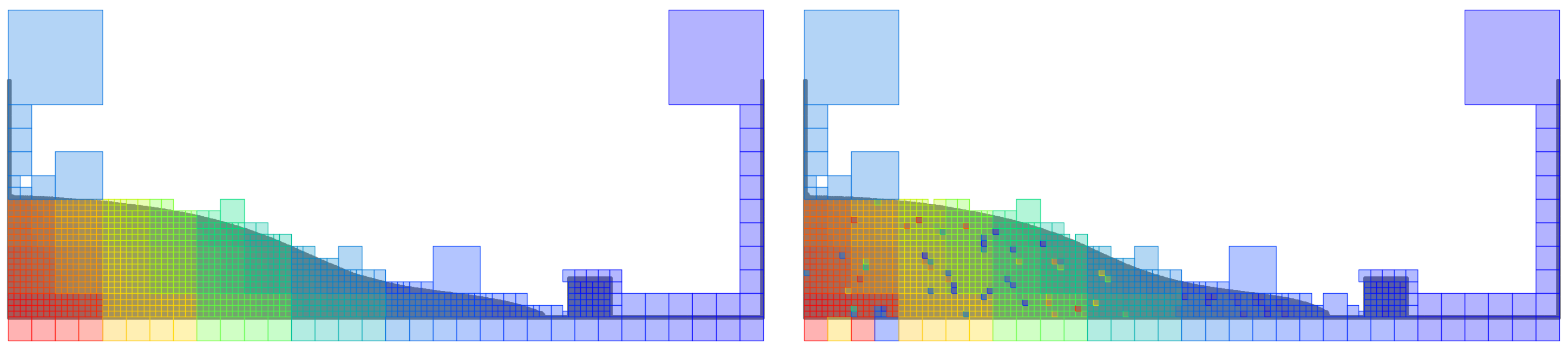
We propose **Almost Deterministic Work Stealing**, which consists of **Deterministic Task Allocation** and **Hierarchical Localized Work Stealing**. We believe what we need is,

1. **Easy fork-join programming interface**,
2. **Good data locality**, and
3. **Dynamic load balancing**.

Although programmers have to explicitly specify the amount of work of each task like Fig. 6, it is still an easy-to-understand fork-join program (requirement 1.). Fig. 7 visualizes the distribution of tasks among 64 workers on the particle simulation in Fig. 4. Random work stealing (a) and OpenMP dynamic (b) fullfill requirement 3., but not 2.. With deterministic scheduling policy like deterministic task allocation (c), requirement 2. is achieved, but not 3.. We can see that ADWS (d) achieves both of requirements 2. and 3..



(a) Random Work Stealing (b) OpenMP dynamic



(c) ADWS (no steal) (d) ADWS

Fig. 7 Visualization of task distribution among 64 workers in particle simulation.

Deterministic Task Allocation

Recursively allocate tasks to each worker based on the amount of work of each task spawned.

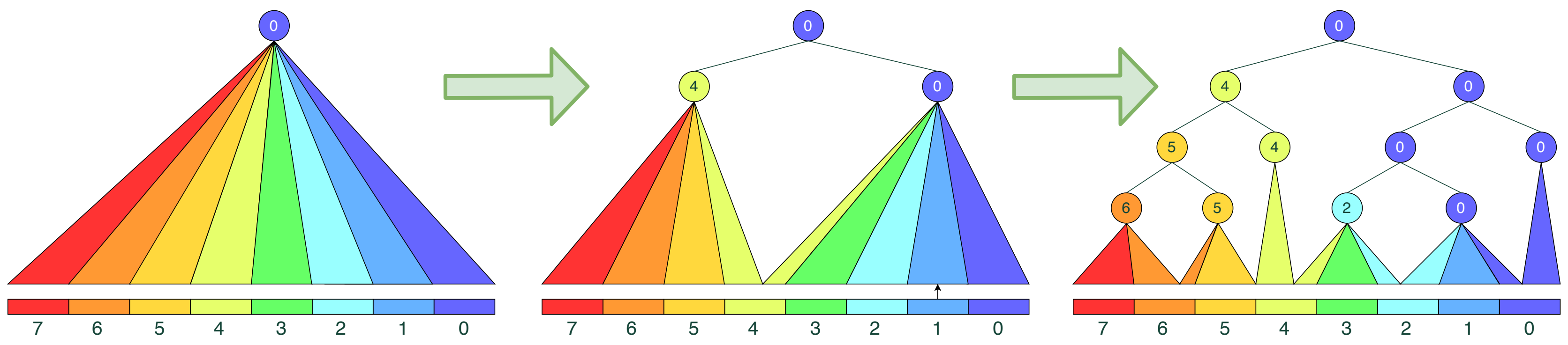


Fig. 8 An overview of the deterministic task allocation

Hierarchical Localized Work Stealing

Localize steals by managing **steal ranges**.

- Steal ranges are set during deterministic task allocation
- Activated from bottom up when tasks are exhausted
- Workers can steal tasks only from the current steal range

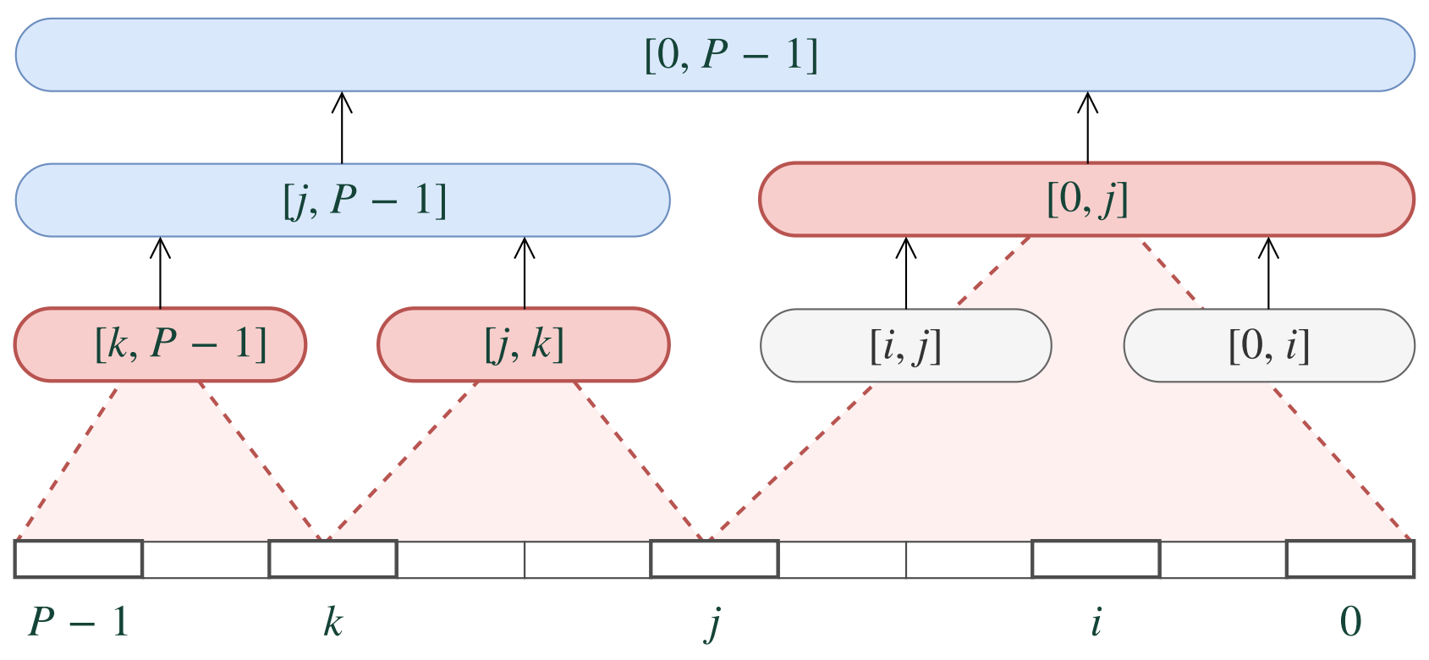


Fig. 9 A tree of steal ranges

Evaluation

We implemented ADWS on MassiveThreads[2] and conducted experiments in an environment of Tab. 1. The benchmarks are Heat2D and matrix-multiplication (matmul), the result of which is shown in Fig. 10 and Fig. 11, respectively. We also implemented task parallel computation of particle interaction in FDPS[3] and compared its performance to the original one (Fig. 12). In all of these benchmarks, ADWS outperforms others.

# of cores	64
# of sockets	4
microarchitecture	Skylake
model	Xeon Gold 6130
frequency	2.1 GHz
L1 data cache	32 KB/core
L2 cache	1 MB/core
L3 cache	22 MB/socket

Tab. 1 Environment

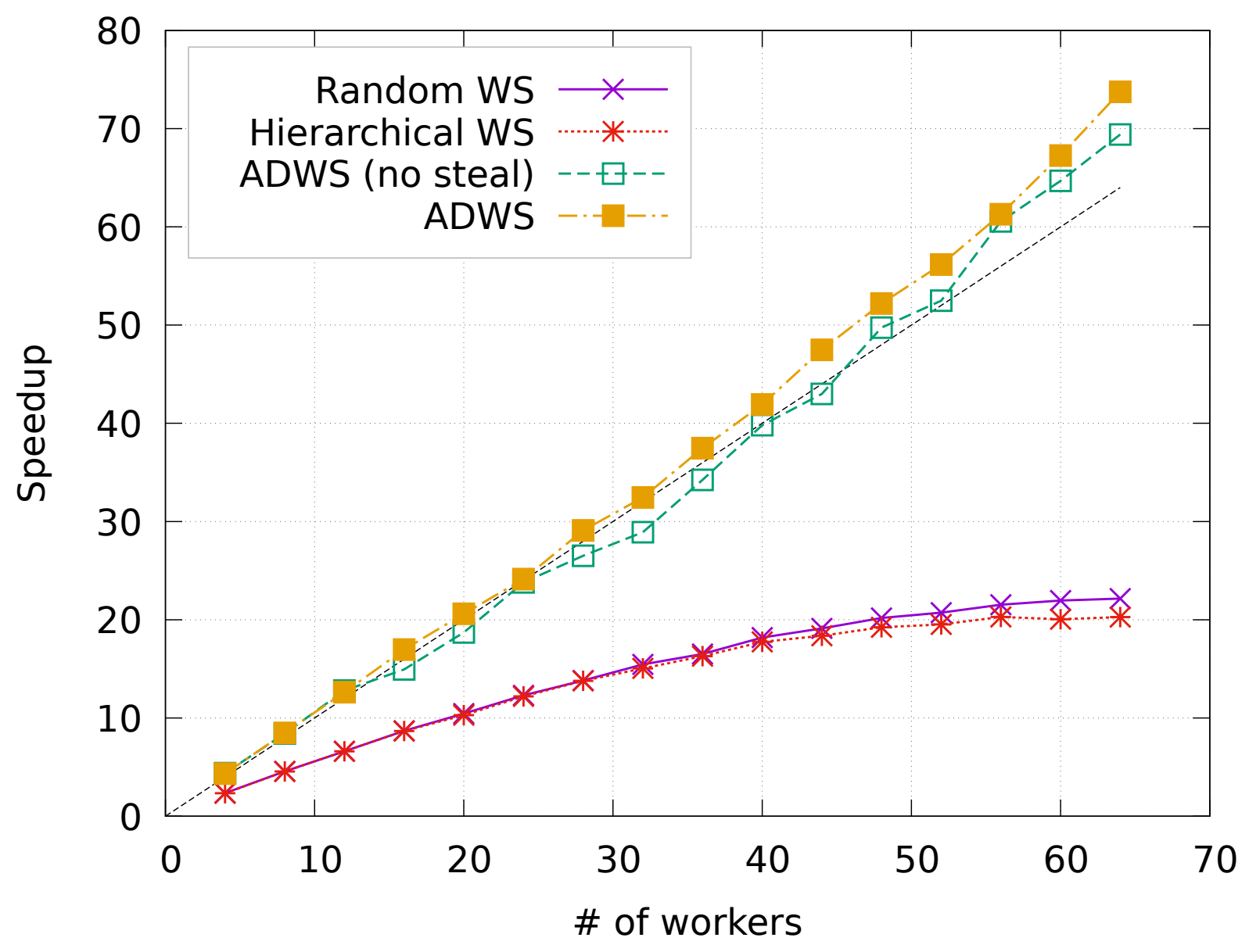


Fig. 10 Speedup of Heat2D (4096x4096)

Fig. 11 GFLOPS of Matmul (4096x4096)

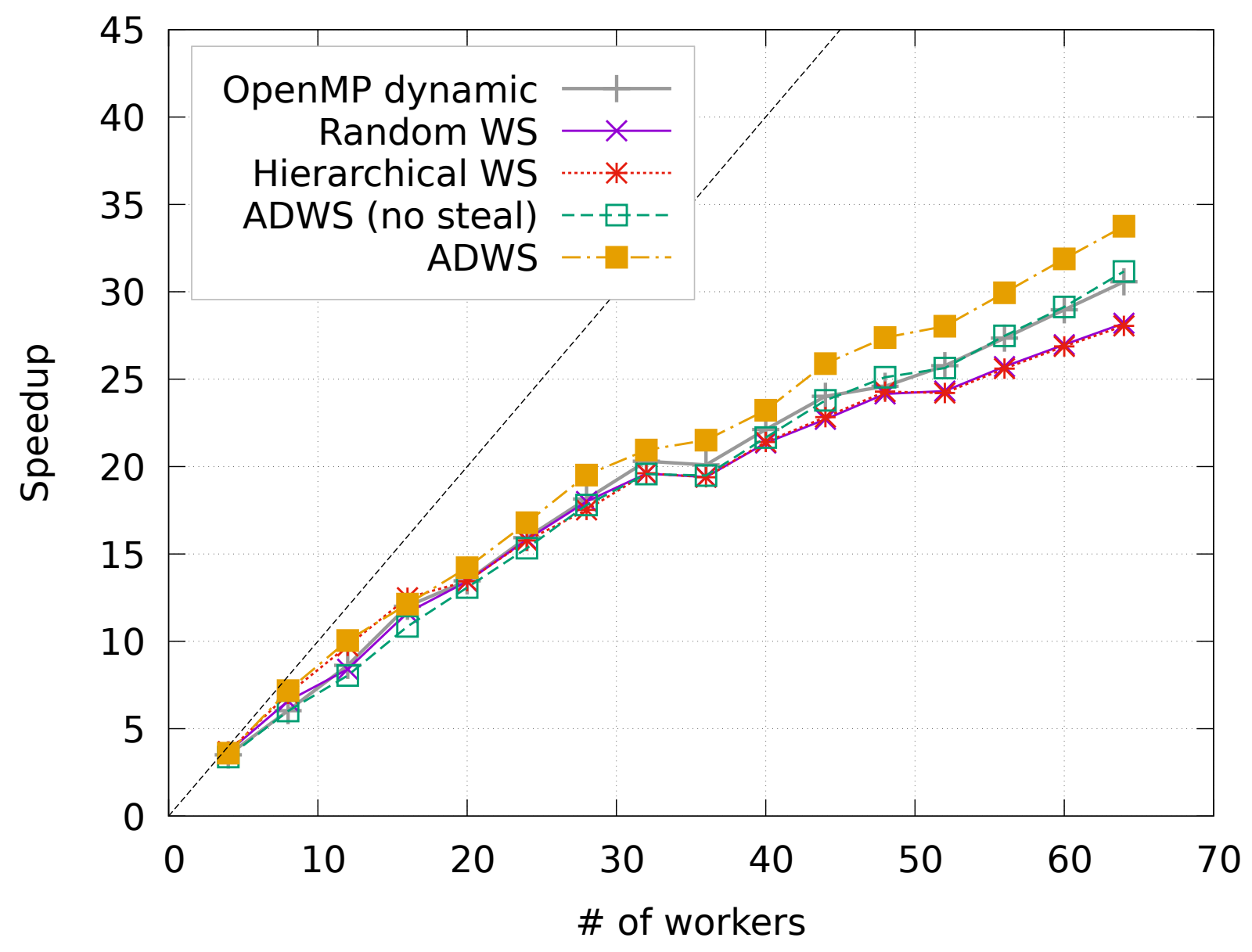


Fig. 12 Speedup of particle interaction in FDPS (N=138968)

---

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

[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.

[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.