Distributed Continuation Stealing is More Scalable than You Might Think

IEEE Cluster ’22

Shumpei Shiina, Kenjiro Taura

The University of Tokyo

2022.09.06

Dynamic Load Balancing on Distributed Memory

- **Dynamic load balancing** is particularly important for irregular algorithms
  - e.g., sorting, sparse matrix arithmetic, tree-based algorithms, AMR, ...
- Hardware is becoming massively parallel → **Manual load balancing is getting harder**
  - intra-/inter-node, many-core, deep cache hierarchy, NUMA, ...
- **Work stealing** is a popular algorithm for automatic load balancing by the runtime
  - Only when a processor becomes idle, it attempts to steal work from another processor

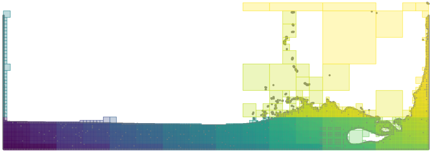


Fig. Fluid simulation (SPH)

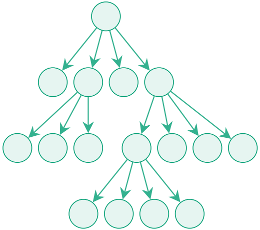


Fig. Unbalanced Tree

Task Parallelism (Fork-Join Parallelism)

- Spawn (fork) a concurrent thread and join its completion
  - threads can be created recursively → well suited to divide-and-conquer algorithms

```
thread th = spawn([=]{ A(); });
B(); // A() and B() are executed concurrently
th.join(); // ensures completion of A()
```

- General parallel execution model for many algorithms
  - e.g., matrix arithmetic, FFT, sorting, dynamic programming, game tree search, space-partitioning tree, N-body, ...
  - Adopted by many runtimes: Cilk, Intel TBB, Java fork/join, OpenMP, ...
- A bunch of threads ($\gg$ # of cores) can be created
  - The underlying runtime performs dynamic load balancing

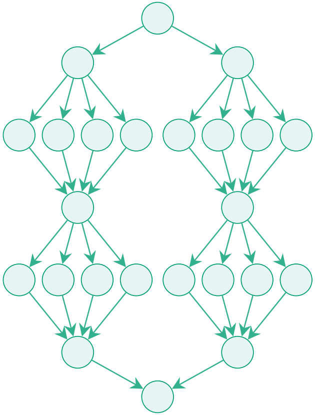
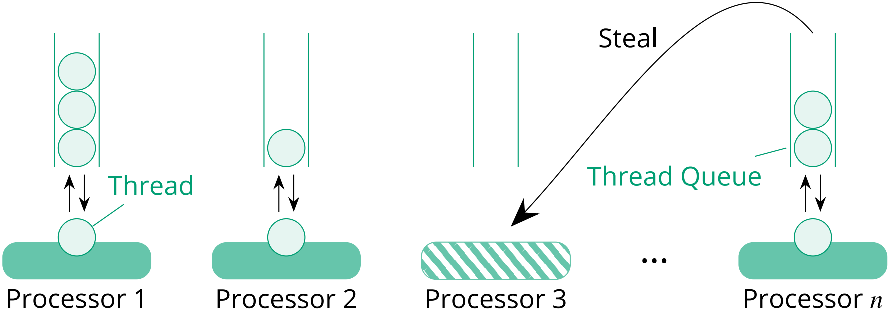


Fig. Task dependency graph

Work Stealing [Blumofe and Leiserson, JACM '99]

- Widely used scheduling strategy for task-parallel programs
- Each processor has its own **thread queue**
- A processor pushes stealable threads to its thread queue
- A processor pops a thread from its local queue when the current thread is completed
- When the local queue is empty, it steals a thread from a randomly selected processor

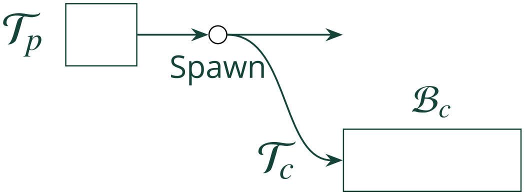


What is Continuation Stealing?

```
Thread 𝒯p:
    ...
```

What is Continuation Stealing?

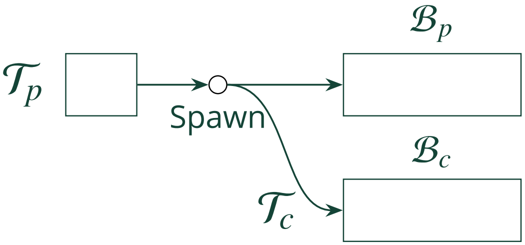
```
Thread 𝒯p:
    ...
    thread 𝒯c = spawn([=]{ ℬc; });
```



What is Continuation Stealing?

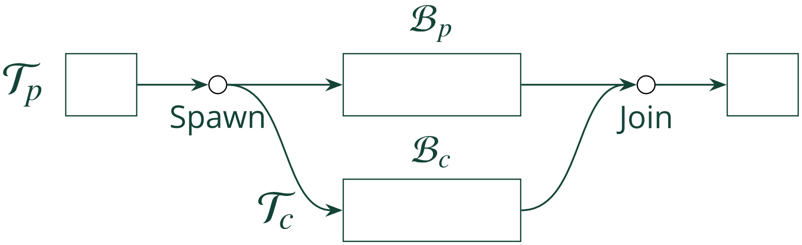
Thread $\mathcal{T}_p$:

```
    ...
    thread 𝒯c = spawn([=]{ ℬc; });
    ℬp;
```



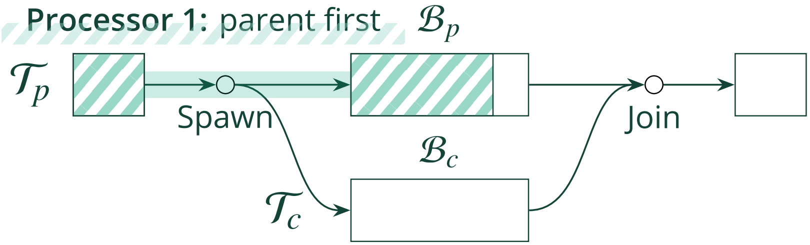
What is Continuation Stealing?

```
Thread 𝒯_p:
    ...
    thread 𝒯_c = spawn([=]{ ℬ_c; });
    ℬ_p;
    𝒯_c.join();
```



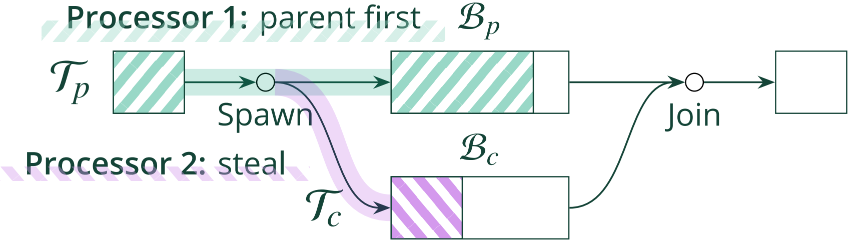
What is Continuation Stealing?

```
Thread 𝒯_p:
    ...
    thread 𝒯_c = spawn([=]{ ℬ_c; });
    ℬ_p;
    𝒯_c.join();
```



What is Continuation Stealing?

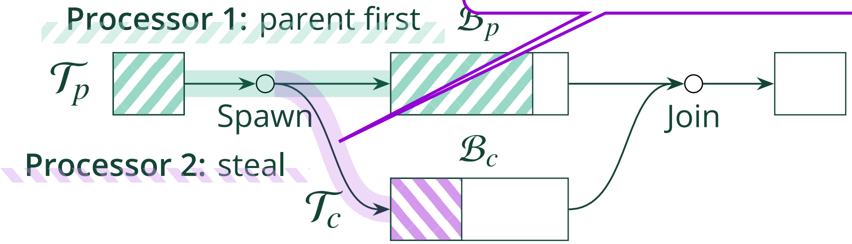
```
Thread 𝒯_p:
    ...
    thread 𝒯_c = spawn([=]{ ℬ_c; });
    ℬ_p;
    𝒯_c.join();
```



What is Continuation Stealing?

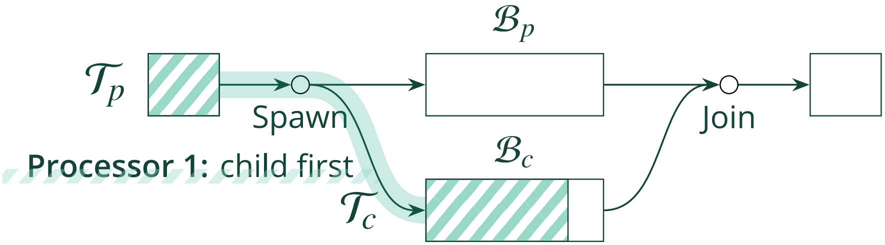
```
Thread 𝒯_p:
    ...
    thread 𝒯_c = spawn([=]{ ℬ_c; });
    ℬ_p;
    𝒯_c.join();
```

The newly created child is stolen
→ **Child stealing**



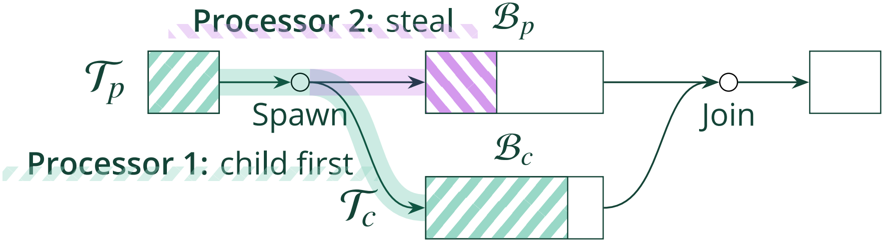
What is Continuation Stealing?

```
Thread 𝒯p:
    ...
    thread 𝒯c = spawn([=]{ ℬc; });
    ℬp;
    𝒯c.join();
```



What is Continuation Stealing?

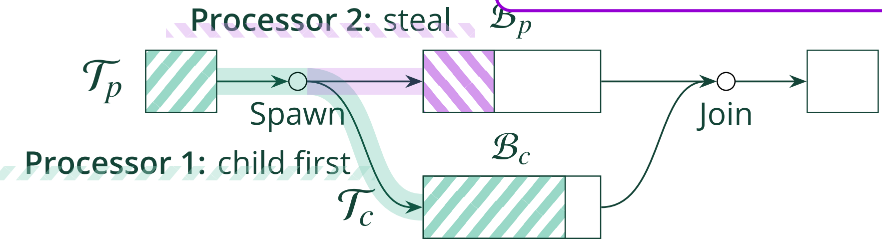
```
Thread 𝒯p:
    ...
    thread 𝒯c = spawn([=]{ ℬc; });
    ℬp;
    𝒯c.join();
```



What is Continuation Stealing?

```
Thread 𝒯_p:
    ...
    thread 𝒯_c = spawn([=]{ ℬ_c; });
    ℬ_p;
    𝒯_c.join();
```

The parent's continuation is stolen
→ **Continuation stealing**



Why We Consider Continuation Stealing is Better

- Many shared-memory runtimes (e.g., Cilk) use continuation stealing because of its efficiency
- Good characteristic: **Continuation stealing preserves the serial execution order**
  - i.e., the execution order of programs with spawn/join keywords removed
  - **Ordinary function call:** execute the called function $\longrightarrow$ its continuation
  - **Continuation stealing:** execute the spawned thread $\longrightarrow$ its continuation
- Because of this execution order, continuation stealing is unlikely to be blocked at join
  - Later, I'll explain why

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...
  1. It is **difficult to implement continuation stealing as a library** with unmodified C/C++ compilers

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...

1. It is **difficult to implement continuation stealing as a library** with unmodified C/C++ compilers
2. **Continuation stealing is less efficient than child stealing** because a whole call stack needs to be copied across nodes

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...
  1. It is **difficult to implement continuation stealing as a library** with unmodified C/C++ compilers
  2. **Continuation stealing is less efficient than child stealing** because a whole call stack needs to be copied across nodes
- The first concern was addressed by [Akiyama and Taura, HPDC '15]
  - Efficient RDMA-based continuation stealing by copying call stacks across nodes

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...

  1. It is **difficult to implement continuation stealing as a library** with unmodified C/C++ compilers
  2. **Continuation stealing is less efficient than child stealing** because a whole call stack needs to be copied across nodes

- The first concern was addressed by [Akiyama and Taura, HPDC '15]
  - Efficient RDMA-based continuation stealing by copying call stacks across nodes
- The second concern has not previously been addressed
  - No performance comparison against child stealing or any other existing runtimes

Continuation Stealing on Distributed Memory

- However, most distributed work-stealing runtimes use child stealing
- This is presumably because people might think...

  1. It is **difficult to implement continuation stealing as a library** with unmodified C/C++ compilers
  2. **Continuation stealing is less efficient than child stealing** because a whole call stack needs to be copied across nodes

- The first concern was addressed by [Akiyama and Taura, HPDC '15]
  - Efficient RDMA-based continuation stealing by copying call stacks across nodes
- The second concern has not previously been addressed
  - No performance comparison against child stealing or any other existing runtimes
- **This paper suggests that the second assumption is not true**

Contributions

Technical Contribution:

- Efficient **join** implementations over **RDMA**, which were not covered by previous work
  - in order to reveal the full potential of continuation stealing

Experimental Results:

- Despite a small increase in steal latency, **continuation stealing often outperforms child stealing** overall
- **Even compared with existing runtimes, continuation stealing is reasonably fast** (showing great scalability to **more than 100k cores**)
- As well as steal policies, **different join policies largely affect performance** particularly for programs with a complicated dependency pattern

Continuation stealing is beneficial even on distributed memory!

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Child Stealing vs. Continuation Stealing on distributed memory

	Child Stealing	Continuation Stealing
Easy to implement?	Yes	Not so easy
Representation of a stealable task	Function pointer + arguments	Call stack
Preserves serial execution order?	No	Yes
Likely to efficiently resolve join?	No	Yes

Child Stealing vs. Continuation Stealing on distributed memory

Continuation stealing requires **thread migration** (without compiler support), which is nontrivial to implement (but possible [Akiyama and Taura, HPDC '15])

	Child Stealing	Continuation Stealing
Easy to implement?	Yes	Not so easy
Representation of a stealable task	Function pointer + arguments	Call stack
Preserves serial execution order?	No	Yes
Likely to efficiently resolve join?	No	Yes

Child Stealing vs. Continuation Stealing on distributed memory

Continuation stealing needs to copy the call stack, which is typically larger than a function pointer and its arguments → **Larger steal cost?**

	Child Stealing	Continuation Stealing
Easy to implement?	Yes	Not so easy
Representation of a stealable task	Function pointer + arguments	Call stack
Preserves serial execution order?	No	Yes
Likely to efficiently resolve join?	No	Yes

Child Stealing vs. Continuation Stealing on distributed memory

Continuation stealing preserves the serial execution order (the order without spawn/join primitives) → good theoretical bounds are known

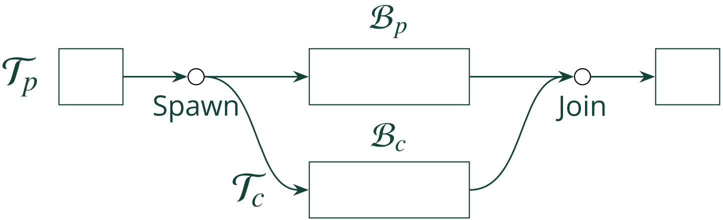
	Child Stealing	Continuation Stealing
Easy to implement?	Yes	Not so easy
Representation of a stealable task	Function pointer + arguments	Call stack
Preserves serial execution order?	No	Yes
Likely to efficiently resolve join?	No	Yes

Child Stealing vs. Continuation Stealing on distributed memory

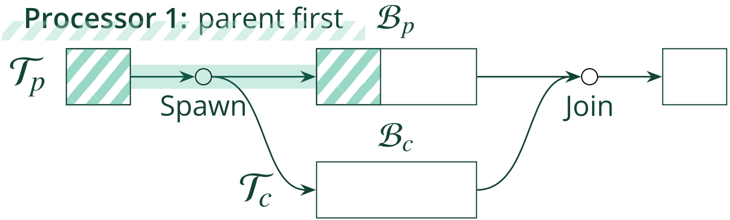
Continuation stealing is considered more efficient for resolving join → **Next slide**

	Child Stealing	Continuation Stealing
Easy to implement?	Yes	Not so easy
Representation of a stealable task	Function pointer + arguments	Call stack
Preserves serial execution order?	No	Yes
Likely to efficiently resolve join?	No	Yes

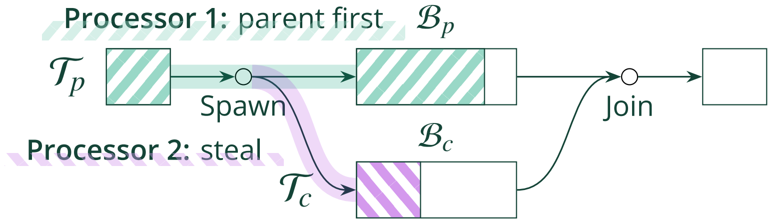
Child Stealing → Joins are Likely to be Blocked



Child Stealing → Joins are Likely to be Blocked

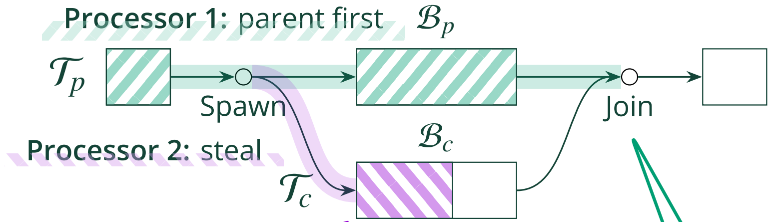


Child Stealing → Joins are Likely to be Blocked



Execution of the child $\mathcal{B}_c$ is delayed compared to the parent $\mathcal{B}_p$

Child Stealing → Joins are Likely to be Blocked

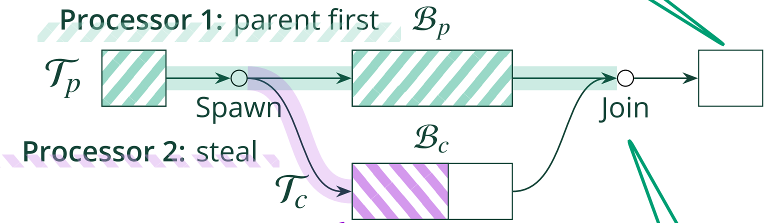


Execution of the child $\mathcal{B}_c$ is delayed compared to the parent $\mathcal{B}_p$

The parent is likely to reach the join earlier
→ The parent is **likely to be blocked at join**

Child Stealing → Joins are Likely to be Blocked

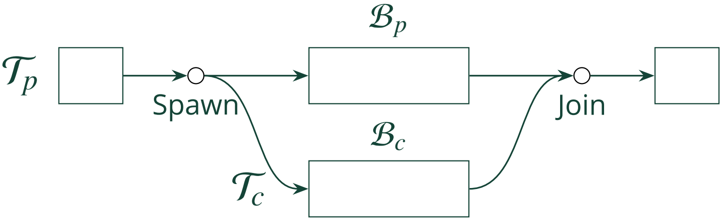
The parent can be resumed when the child is completed,
but processor 1 may be busy when the join is resolved
→ **execution of the continuation of the join can be delayed**



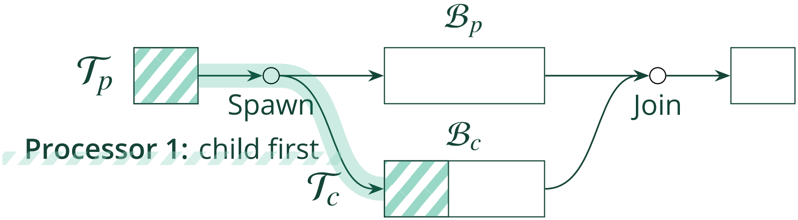
Execution of the child $\mathcal{B}_c$ is delayed compared to the parent $\mathcal{B}_p$

The parent is likely to reach the join earlier
→ The parent is **likely to be blocked at join**

Continuation Stealing → Joins are Unlikely to be Blocked

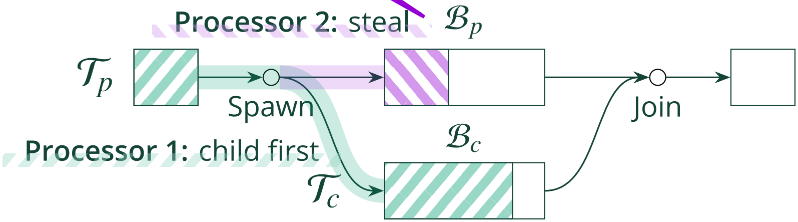


Continuation Stealing → Joins are Unlikely to be Blocked

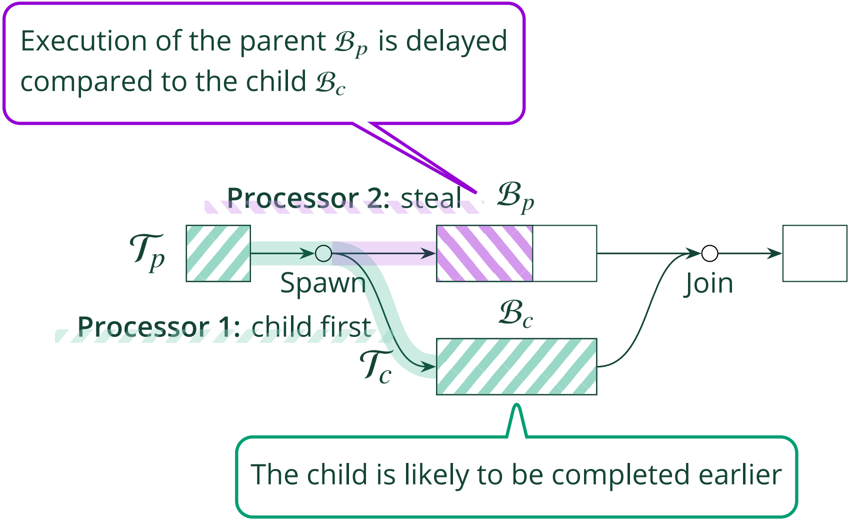


Continuation Stealing → Joins are Unlikely to be Blocked

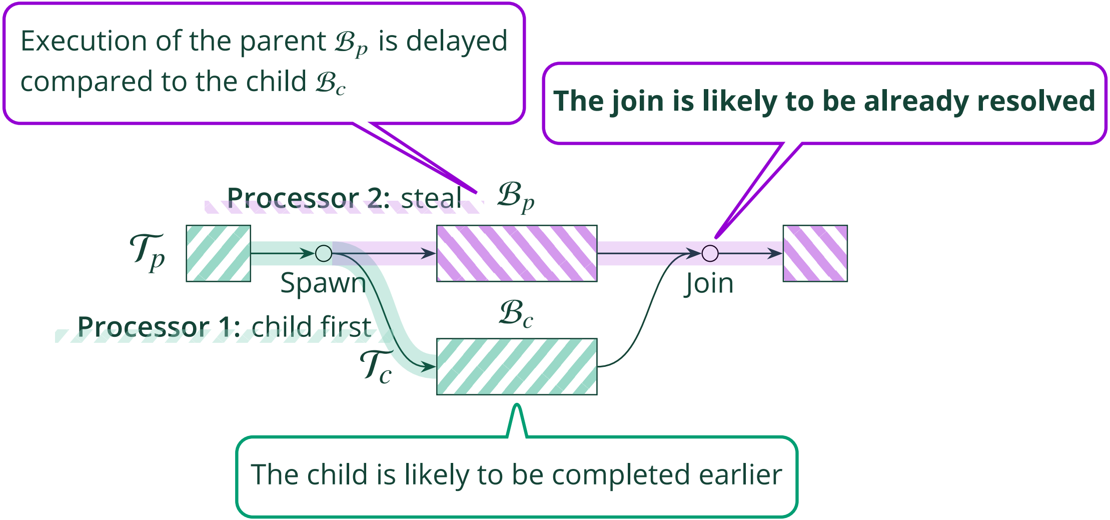
Execution of the parent $\mathcal{B}_p$ is delayed compared to the child $\mathcal{B}_c$



Continuation Stealing → Joins are Unlikely to be Blocked



Continuation Stealing → Joins are Unlikely to be Blocked



Previous Work – Uni-Address Threads [Akiyama and Taura, HPDC '15]

- Efficient RDMA-based continuation stealing without compiler modification
- Basic idea: copy thread stacks to the same virtual address before threads are executed
- Good scalability to 4096 cores was reported
- **No comparison with child stealing or existing distributed task-parallel runtimes**

We still don't know whether distributed continuation stealing is worth implementing

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

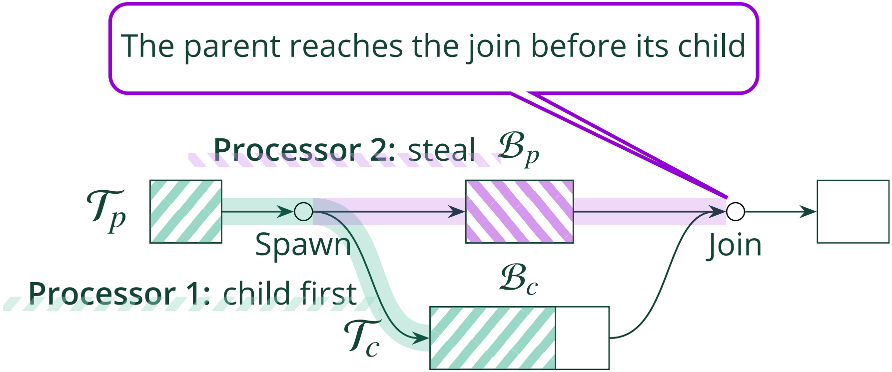
Performance Improvements for Join

- To evaluate the full potential of continuation stealing, we need to carefully design **join** implementations, which were not well considered in previous work
- We introduced two improvements for join
  1. **How to efficiently resume the continuation of the join when blocked**
  2. How to efficiently free memory needed for join remotely (see the paper)
- In this talk, we will explain the first improvement

Joining Strategy – Overview

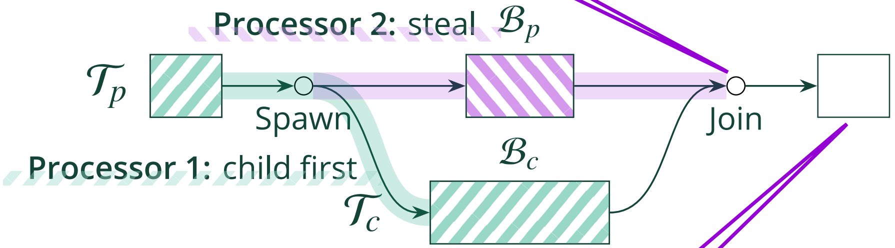
- In general, there are two strategies to resolve a join in fork-join programs
  1. **Stalling join:** across a join, the executing processor is the same (e.g., Intel TBB)
  2. **Greedy join:** the processor who runs the parent or the child, whichever reaches a join point later, executes the continuation of the join (e.g., Cilk)
- Theoretically, **greedy join** is considered better than **stalling join**
- In practice, **greedy join** is more difficult to implement because it involves thread migration
- The previous work [Akiyama and Taura, HPDC '15] uses **stalling join** strategy
- We implemented **greedy join** strategy by using RDMA atomic operations

Stalling Join (Previous Implementation)



Stalling Join (Previous Implementation)

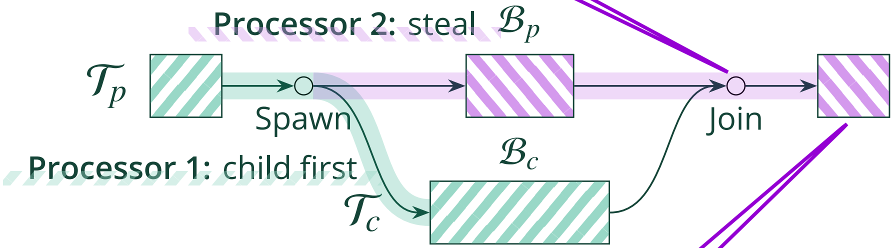
The parent reaches the join before its child



Even if the child is completed, the continuation of the join must be resumed by **processor 2**, who suspended the parent thread

Stalling Join (Previous Implementation)

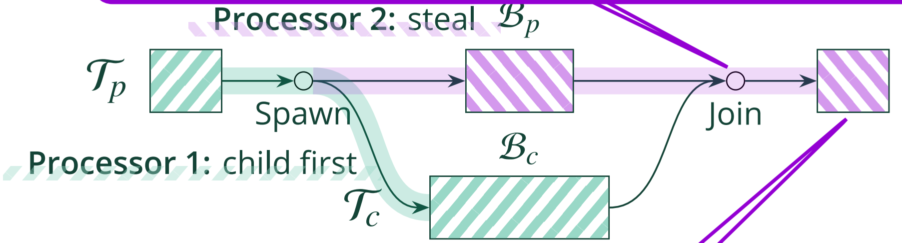
The parent reaches the join before its child



Even if the child is completed, the continuation of the join must be resumed by **processor 2**, who suspended the parent thread

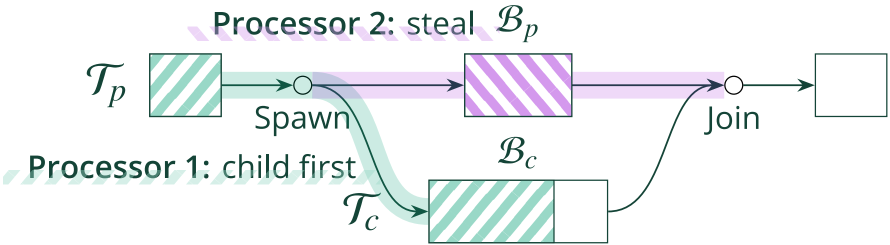
Stalling Join (Previous Implementation)

However, **processor 2** may be busy when the join is resolved
→ **execution of the continuation of the join can be delayed**

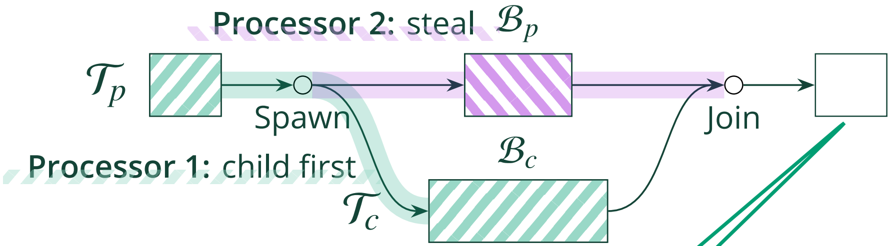


Even if the child is completed, the continuation of the join must be resumed by **processor 2**, who suspended the parent thread

Greedy Join (Our Improvement)

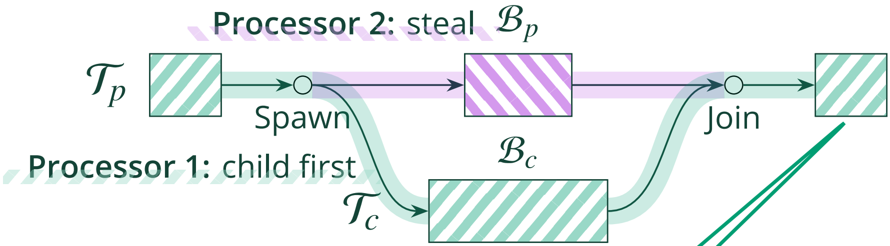


Greedy Join (Our Improvement)



If the child thread is completed later, the continuation of the join is immediately resumed by **Processor 1**

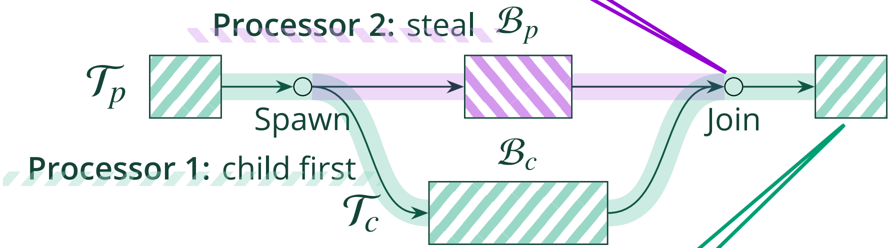
Greedy Join (Our Improvement)



If the child thread is completed later, the continuation of the join is immediately resumed by **Processor 1**

Greedy Join (Our Improvement)

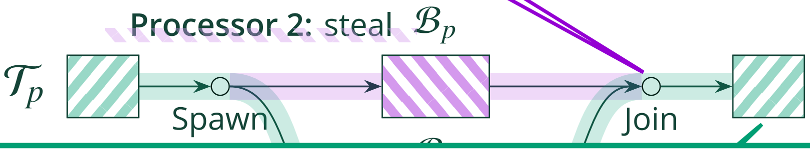
The parent thread needs to be migrated from **Processor 2** to **Processor 1** at join



If the child thread is completed later, the continuation of the join is immediately resumed by **Processor 1**

Greedy Join (Our Improvement)

The parent thread needs to be migrated from **Processor 2** to **Processor 1** at join



We devised an efficient RDMA-based implementation of **greedy join** in a lock-free manner by utilizing **RDMA atomic operations**

If the child thread is completed later, the continuation of the join is immediately resumed by **Processor 1**

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Evaluation – Overview

Research questions:

1. How does continuation stealing perform compared with child stealing?
2. Is it practical to use continuation stealing in distributed task-parallel runtimes?
3. How important is thread migration (like greedy join) on distributed memory?

Evaluation – Overview

Research questions:

1. How does continuation stealing perform compared with child stealing?
2. Is it practical to use continuation stealing in distributed task-parallel runtimes?
3. How important is thread migration (like greedy join) on distributed memory?

Summary of our findings:

1. Despite a small increase in steal latency (only less than 20%), **continuation stealing often outperforms child stealing by efficiently resolving joins**
2. **Even compared with existing bag-of-tasks runtimes (without joins), our system is reasonably fast** (96.4% parallel efficiency on **110,592 cores** in UTS benchmark)
3. **Lack of thread migration capability (at either fork or join) leads to bad performance** when we intensively use threads as futures (in LCS benchmark)

Experimental Settings

- We implemented various strategies over MPI-3 RMA in a C++ library developed in the previous work (MassiveThreads/DM [Akiyama and Taura, HPDC '15])
- Two variants of continuation stealing (**stalling** and **greedy** join)
- Two variants of child stealing (RtC and Full), which mimic prevalent implementations
  - Only the one with better performance (Full) is shown in this presentation

Experimental environment:

- ITO-A: **ITO supercomputer (subsystem A)** at Kyushu University (up to 256 nodes)
  - Intel Xeon Gold 6154 (36 CPU cores/node), InfiniBand EDR 4x (100 Gbps), Open MPI v5.0.x
- Wisteria-O: **Wisteria/BDEC-01 Odyssey** at the University of Tokyo (up to 2304 nodes)
  - Fujitsu A64FX (48 CPU cores/node), Tofu Interconnect-D, Fujitsu MPI

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Synthetic Benchmark

- Recursive parallel-for benchmark (RecPFor)
- Two-way divide-and-conquer + parallel for loop at each recursion
- Common pattern for many divide-and-conquer algorithms
  - e.g., quicksort, tree construction (kdtree, decision tree)
- Each leaf task spins for 10 μs ($M = 10\mu s$)

```
RecPFor(int n) {
  if (n == 1) {
    compute(M); // run for duration of M
  } else {
    for (int k = 0; k < 5; k++) {
      parallel_for (int i = 0; i < n; i++)
        compute(M); // run for duration of M
    }
    thread th = spawn([=] { RecPFor(n/2); });
    RecPFor(n/2);
    th.join();
  }
}
```

Synthetic Benchmark – Performance Analysis

- We profiled the execution of the RecPFor benchmark ($N = 2^{22}$)
- **ITO-A:** 576 cores (16 nodes), **Wisteria-O:** 1728 cores (36 nodes)

System	Steal Strategy	Time	# of Outstanding Joins	# of Steals (Successful)	Avg. Steal Latency (Successful)	Avg. Stolen Task Size
ITO-A	Cont. Steal (greedy)	**8.30 s**	**56876**	**474991**	31.6 μs	1845 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3** μs	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9** μs	**55 bytes**

Synthetic Benchmark – Performance Analysis

Continuation stealing outperforms child stealing
(We will see a larger performance difference between stalling and greedy join in LCS)

System	Steal Strategy	Time	# of Outstanding Joins	# of Steals (Successful)	Avg. Steal Latency (Successful)	Avg. Stolen Task Size
ITO-A	Cont. Steal (greedy)	**8.30 s**	**56876**	**474991**	31.6 μs	1845 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3** μs	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9** μs	**55 bytes**

Synthetic Benchmark – Performance Analysis

Child stealing incurs a much larger number of **outstanding joins**
→ **less parallelism available**

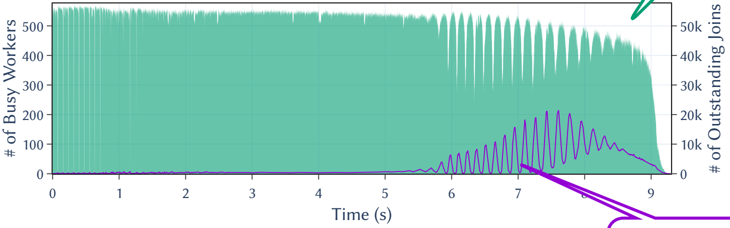
An **outstanding join** is a join that is blocked due to a steal event

System	Steal Strategy	Time	# of Outstanding Joins	# of Steals (Successful)	Avg. Steal Latency (Successful)	Avg. Stolen Task Size
ITO-A	Cont. Steal (greedy)	**8.30 s**	**56876**	**474991**	31.6 μs	1845 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3** μs	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9** μs	**55 bytes**

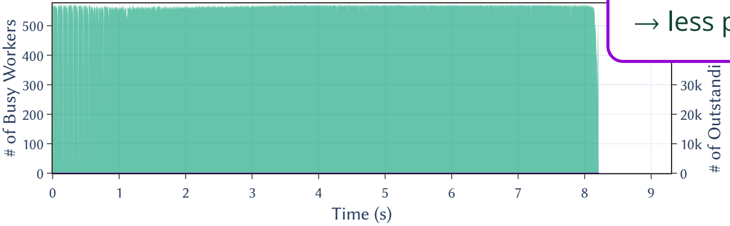
Synthetic Benchmark – Performance Analysis

Many idle workers

Child Steal:



Cont. Steal (greedy):



Many outstanding joins → less parallelism

that
nt

System	...g. Stolen Task Size
ITO-A	55 bytes
Wisteria-O	139 bytes
	156 bytes
	55 bytes

Synthetic Benchmark – Performance Analysis

- less parallelism $\rightarrow$ larger number of steals
- Greedy join is the most efficient, as it can immediately resolve outstanding joins

System	Steal Strategy	Time	# of Outstanding Joins	# of Steals (Successful)	Avg. Steal Latency (Successful)	Avg. Stolen Task Size
ITO-A	Cont. Steal (greedy)	**8.30 s**	**56876**	**474991**	31.6 μs	1845 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3** μs	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9** μs	**55 bytes**

Synthetic Benchmark – Performance Analysis

- Cont. steal needs to copy the call stack → larger stolen task size
- Nevertheless, the steal latency is only less than 20% longer than child stealing

System	Steal Strategy	Time	# of Outstanding Joins	# of Steals (Successful)	Avg. Steal Latency (Successful)	Avg. Stolen Task Size
ITO-A	Cont. Steal (greedy)	**8.30 s**	**56876**	**474991**	31.6 μs	1845 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3** μs	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9** μs	**55 bytes**

Synthetic Benchmark – Performance Analysis

- Cont. steal needs to copy the call stack → larger stolen task size
- Nevertheless, the steal latency is only less than 20% longer than child stealing

Despite a small increase in steal latency, continuation stealing has an overall performance benefit

System			# of		Avg. Steal	Stolen sk Size
ITO-A						5 bytes
	Cont. Steal (stalling)	8.33 s	72546	807097	30.4 μs	1790 bytes
	Child Steal	9.31 s	3208417	6038858	**29.3 μs**	**55 bytes**
Wisteria-O	Cont. Steal (greedy)	**5.94 s**	**229154**	**1790018**	20.4 μs	1139 bytes
	Cont. Steal (stalling)	5.97 s	279035	3086034	20.6 μs	1156 bytes
	Child Steal	7.69 s	8602558	15704498	**19.9 μs**	**55 bytes**

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Unbalanced Tree Search (UTS) Benchmark

- A widely used benchmark to measure the load balancing capability of runtime systems
- **X-axis:** the number of processes, up to 110,592 cores (2304 nodes)
- **Y-axis:** throughput of the benchmark (higher is better)
- **Our system could achieve 96.4% parallel efficiency on 110,592 cores**

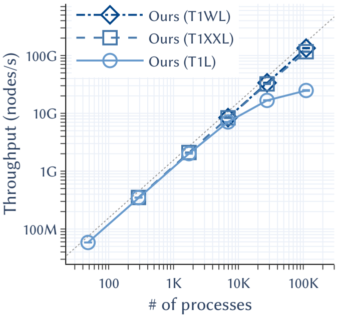


Fig. Result on Wisteria-O

Settings:

- UTS counts the number of all nodes in an unbalanced tree in parallel
- Throughput: the number of counted nodes per second
- Ideal throughput (the straight line) is calculated by serial performance
- Tree sizes: T1L < T1XXL < T1WL
- Cont. steal (greedy join)

Comparison with Existing Bag-of-Tasks Runtimes in UTS

- Also compared our runtime with existing systems on ITO-A (with InfiniBand)
- **Competitors:** three state-of-the-art work-stealing systems based on bag-of-tasks
  - Child stealing, **no join** primitive → global termination detection is needed
- Our system and SAWS [Cartier+, ICPP '21] → RDMA-based, good scalability
- Charm++ and X10/GLB → no RDMA, worse scalability

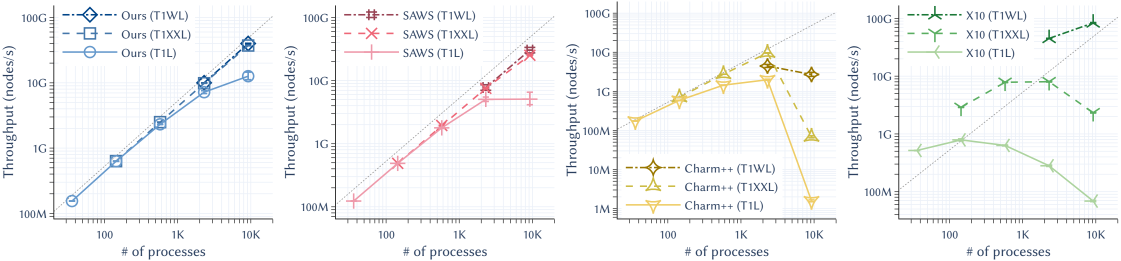


Fig. **Our Runtime System** Fig. SAWS Fig. Charm++ Fig. X10/GLB

Comparison with Existing Bag-of-Tasks Runtimes in UTS

- Also compared our runtime with existing systems on ITO-A (with InfiniBand)

Summarizing, our system

- Has more general synchronization (join) and thread migration capability
- Performs as well as or even better than bag-of-tasks counterparts

- Charm++ and X10/GLB → no RDMA, worse scalability

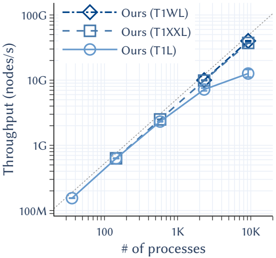


Fig. **Our Runtime System**

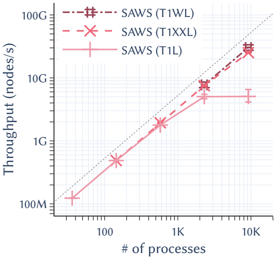


Fig. SAWS

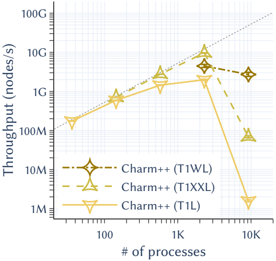


Fig. Charm++

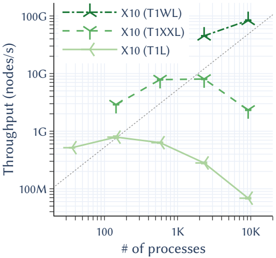


Fig. X10/GLB

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Preliminaries: Fork-Join and Futures

- Our thread implementation is not only for **fork-join** but also for **futures**
- **Fork-join:** a thread must be joined by its parent (parallelism is nested)
- **Future:** a thread (called future) can be joined at any point (not strictly nested)
- Our **longest common subsequence (LCS)** benchmark intensively uses futures
  - It combines recursive space decomposition and futures to represent true task dependencies
  - **Lack of thread migration capability easily falls into bad load balancing** in this benchmark

Results of the LCS Benchmark

- We compared three scheduling policies with different thread migration capabilities
- Child stealing: No thread migration
- Cont. Steal (stalling): Migration at **steal**
- Cont. Steal (greedy): Migration at **steal** + Migration at **join**
- **Lack of either migration capability led to much worse performance** (due to bad load balancing)

Tab. Execution times with 576 cores (16 nodes).

Size	**Cont. Steal (greedy)**	Cont. Steal (stalling)	Child Stealing
2^{18}	**0.569 s**	3.44 s	93.1 s
2^{22}	**45.9 s**	433 s	2.11×10^4 s

Settings:

- Run on 16 nodes of ITO-A
- Find an LCS length of N 1-byte characters (randomly generated)
- Serial cutoff size: 2^9

Outline

Background

Joining Threads over RDMA

Evaluation

- Performance Analysis of Various Scheduling Policies (Synthetic Benchmark)
- Scalability Study with State-of-the-Art Systems (UTS Benchmark)
- Thread Migration Capability and Futures (LCS Benchmark)

Conclusion and Future Work

Summary

Our artifact is available at:

`https://github.com/s417-lama/cluster22-contsteal-artifact`

Conclusion:

- We introduced an efficient RDMA-based greedy join implementation
- Despite a small increase in steal latency, continuation stealing has an overall performance benefit in nested fork-join programs
- Thread migration (both continuation stealing and greedy join) is particularly important for programs with a complicated dependency pattern

Future Work:

- Integrate with PGAS to handle global data
  - Current system only allows function arguments and return values for data exchange
- Apply memory hierarchy-aware scheduling to improve data locality
  - e.g., Almost Deterministic Work Stealing (ADWS) **[Shiina and Taura, SC '19 and TPDS '22]**