Itoyori: Reconciling Global Address Space and Global Fork-Join Task Parallelism

SC23

Shumpei Shiina, Kenjiro Taura

The University of Tokyo

2023.11.14

What We Really Want to Reconcile: Productivity and Performance in HPC

Low-level programming models
that can achieve
the highest performance

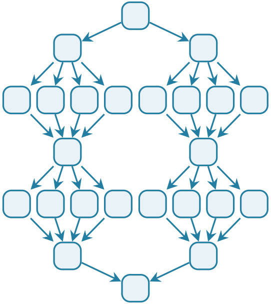
- Two different programming models for shared/distributed memory (**MPI+X** model)
  - X = Pthreads, OpenMP, TBB, ...
- Require much effort by HPC experts
  - Lower productivity

High-level programming models
that can **shortly** achieve
sufficiently good performance

- Desired properties:
  - A single, unified programming model for shared/distributed memory
  - General enough to easily express dynamic and irregular parallelism
- More is needed on this front

Fork-Join Task Parallelism on Distributed Memory?

- Parallel execution based on dynamically forked tasks
- Well suited for **dynamic** and **irregular** applications
- Programmers can focus on logical parallelism without considering hardware details (**processor-obliviousness**)
- Popular **shared-memory** programming models for fork-join task parallelism:



OpenCilk

oneAPI
(oneTBB)

OpenMP
(#pragma omp task)

... any systems for **distributed memory**?

Itoyori: A Distributed Task-Parallel Runtime System

- A C++17 library for fork-join task parallelism on distributed memory
  - It depends only on MPI (capable of **MPI-3 RMA**) → **Good Portability**
- "Itoyori" is the Japanese name of the fish "**Thread**fin Breams"
- Shared-memory-like simple global-view programming
- Yet highly scalable and efficient

GitHub:

https://github.com/itoyori/itoyori

What Itoyori Offers

- Work-stealing scheduler for fine-grained, **global fork-join task parallelism**
  - Tasks (user-level threads) can be scheduled **across different nodes**
  - Based on the **uni-address scheme** for inter-node dynamic thread migration
    - **[Akiyama & Taura, HPDC '15]**, scalability to > 100k cores **[Shiina & Taura, Cluster '22]**

What Itoyori Offers

- Work-stealing scheduler for fine-grained, **global fork-join task parallelism**
  - Tasks (user-level threads) can be scheduled **across different nodes**
  - Based on the **uni-address scheme** for inter-node dynamic thread migration
    - **[Akiyama & Taura, HPDC ’15]**, scalability to > 100k cores **[Shiina & Taura, Cluster ’22]**
- **Global address space**, a view of shared memory over distributed memory
  - More specifically, **Partitioned Global Address Space (PGAS)**

What Itoyori Offers

- Work-stealing scheduler for fine-grained, **global fork-join task parallelism**
  - Tasks (user-level threads) can be scheduled **across different nodes**
  - Based on the **uni-address scheme** for inter-node dynamic thread migration
    - **[Akiyama & Taura, HPDC ’15]**, scalability to > 100k cores **[Shiina & Taura, Cluster ’22]**
- **Global address space**, a view of shared memory over distributed memory
  - More specifically, **Partitioned Global Address Space (PGAS)**
- High-level C++ parallel STL-like interfaces
  - e.g., `transform()`, `reduce()`
  - They automatically call fork-join and global memory access APIs internally

What Itoyori Does NOT Offer

- Explicit point-to-point communication
  - Communication is implicitly issued when accessing the global address space

What Itoyori Does NOT Offer

- Explicit point-to-point communication
    - Communication is implicitly issued when accessing the global address space
- **Distributed shared memory (DSM)** that allows transparent global memory access
    - Explicit API calls are required for global memory access in Itoyori (PGAS)

What Itoyori Does NOT Offer

- Explicit point-to-point communication
  - Communication is implicitly issued when accessing the global address space
- **Distributed shared memory (DSM)** that allows transparent global memory access
  - Explicit API calls are required for global memory access in Itoyori (PGAS)
- APIs to distinguish between inter- and intra-node processes
  - No need for two-level parallelization (e.g., MPI+X)

What Itoyori Does NOT Offer

- Explicit point-to-point communication
  - Communication is implicitly issued when accessing the global address space
- **Distributed shared memory (DSM)** that allows transparent global memory access
  - Explicit API calls are required for global memory access in Itoyori (PGAS)
- APIs to distinguish between inter- and intra-node processes
  - No need for two-level parallelization (e.g., MPI+X)
- Complicated APIs for task-parallel execution

What Itoyori Does NOT Offer

- Explicit point-to-point communication
  - Communication is implicitly issued when accessing the global address space
- **Distributed shared memory (DSM)** that allows transparent global memory access
  - Explicit API calls are required for global memory access in Itoyori (PGAS)
- APIs to distinguish between inter- and intra-node processes
  - No need for two-level parallelization (e.g., MPI+X)
- Complicated APIs for task-parallel execution
- Special compilers other than ordinary C++17 compilers

Key Contributions of Our Research

- Proposing Itoyori, a distributed fork-join task-parallel runtime system
  - Itoyori **reconciles** PGAS and fine-grained fork-join task parallelism by introducing a **software cache** for global memory access
- Demonstrating high productivity and performance of Itoyori through a real-world application ExaFMM
  - 7.5× speedup when scaled from a single node to 12 nodes
  - comparable performance to a hand-optimized MPI implementation

Itoyori is expected to strike a good balance between productivity and performance!

Outline

Itoyori's Programming Model

Software Caching for Global Memory Access

Evaluation

Summary

Outline

Itoyori's Programming Model

Software Caching for Global Memory Access

Evaluation

Summary

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Divide the input array into two sub-arrays and sort them recursively (**divide-and-conquer**)

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Divide the input array into two sub-arrays and sort them recursively (**divide-and-conquer**)

Merge the two sorted arrays

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Switch to a fast sequential algorithm for small arrays

Divide the input array into two sub-arrays and sort them recursively (**divide-and-conquer**)

Merge the two sorted arrays

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a       , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Distributed parallel code in Itoyori:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

Sequential C++ code:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Distributed parallel code in Itoyori:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
```

Parallel tasks can be dynamically forked and joined, even recursively (**Nested fork-join parallelism**)

Itoyori's Programming Model ▷ An Example of Parallel Merge Sort

In order to access global memory, programmers need to call **checkout/checkin API**

Sequent… …lel code in Itoyori:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {
    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
```

Parallel tasks can be dynamically forked and joined, even recursively (**Nested fork-join parallelism**)

Checkout/Checkin APIs

Raw virtual addresses can be used for global memory access

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

Checkout/Checkin APIs

Raw virtual addresses can be used for global memory access

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

- Requests local access to global memory region $[a, a + n)$
- Specifies the access mode (`read`, `read_write`, or `write`)
  - If `read` or `read_write`, the latest data may be fetched from owners

Checkout/Checkin APIs

Raw virtual addresses can be used for global memory access

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

- Requests local access to global memory region $[a, a + n)$
- Specifies the access mode (read, read_write, or write)
  - If read or read_write, the latest data may be fetched from owners

- Claims the completion of memory access
- Passes the same arguments as the corresponding checkout call
  - If read_write or write, this region is considered modified

Outline

Global Address Space + Global Task Parallelism = ?

Partitioned Global Address Space (PGAS) model:

- Programmers optimize data movement by explicitly distinguishing between global and local data
- We want to aggregate communication for different tasks working on the same data

Inter-node dynamic load balancing (global task parallelism):

- The runtime system can dynamically move tasks across nodes for load balancing
- Requiring each task independently issue communication for its own data

If we naively combine these two...

⇨ **Redundant, fine-grained communication**

Redundant, Fine-Grained Communication in Parallel Merge Sort

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

- At merge, we want to reuse remote data fetched in the previous sort functions
- However, it is difficult for programmers to do so because these tasks **may** run on different nodes

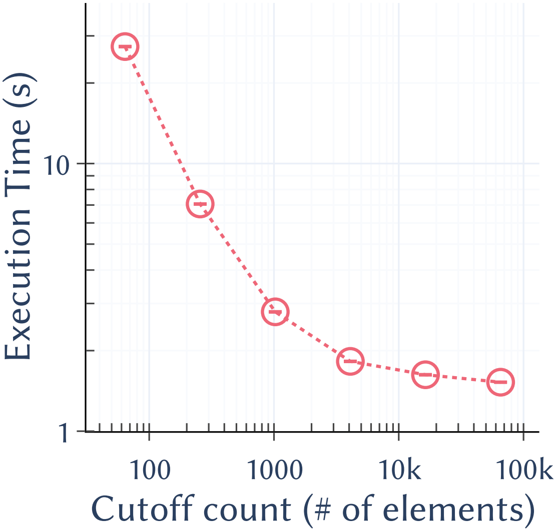
Redundant, Fine-Grained Communication in Parallel Merge Sort

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    thread th = fork([=]{ msort(a, n/2); });
    msort(a + n/2, n/2);
    th.join();
    merge(a, n, n/2);
  }
}
```

- As a result, global memory accesses are issued for each task
- More fine-grained tasks
  ⇨ More fine-grained communication

- At merge, we want to reuse remote data fetched in the previous sort functions
- However, it is difficult for programmers to do so because these tasks **may** run on different nodes

Performance of Naive Combination of PGAS and Dynamic Load Balancing

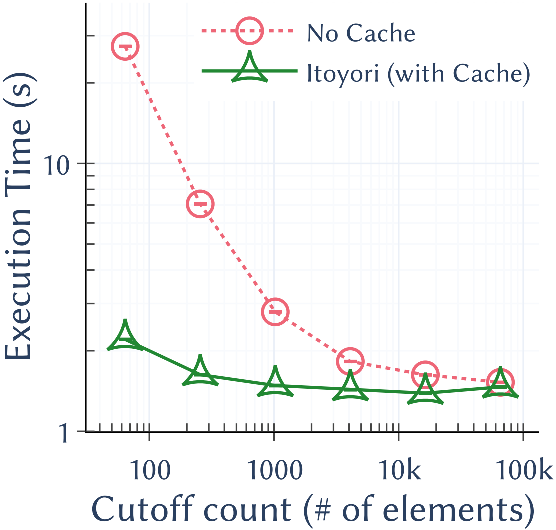


- Recursive parallel merge sort
  - called **Cilksort**
- Ran on 12 nodes (576 cores)
- More fine-grained tasks
  - ⇨ More fine-grained communication
  - ⇨ Worse performance

Reconciling Them by Software Caching!

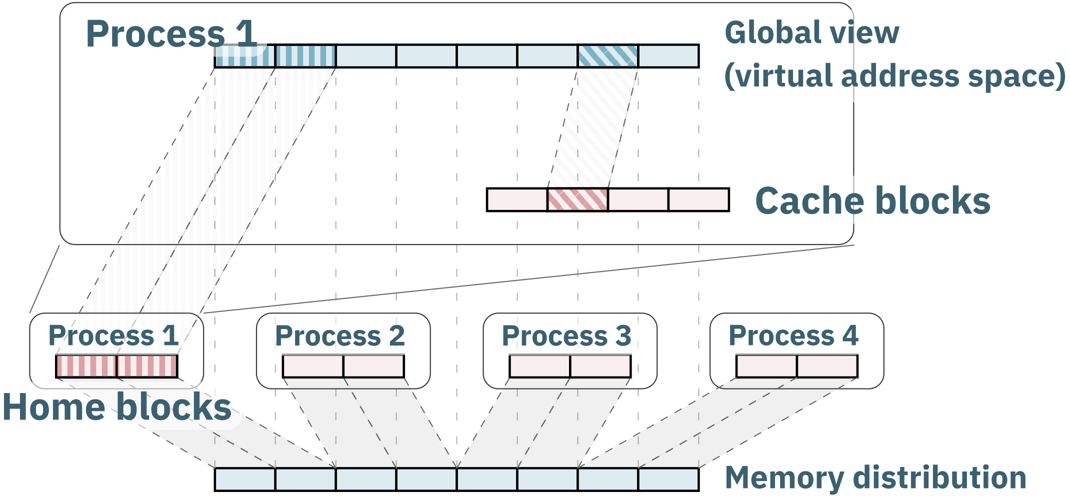
- By introducing a software cache, the runtime system, rather than programmers, can **aggregate communication for tasks that are scheduled on the same node**
- Exploit **spatial locality** by fetching larger data than requested
- Exploit **temporal locality** by reusing fetched data across different tasks
- We designed **checkout/checkin APIs** for efficient software caching
  - Avoid unnecessary copy overhead that would occur in traditional PGAS APIs (GET/PUT)
  - See our paper for more details!

Performance Improvement by Software Caching

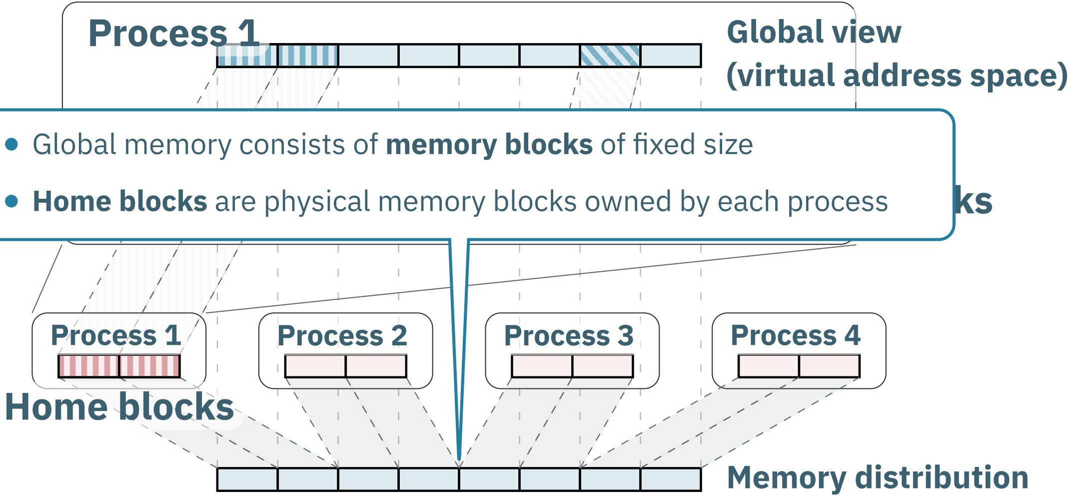


- By software caching, Itoyori becomes more robust to fine-grained parallelism

Virtual Memory Mappings for Software Cache

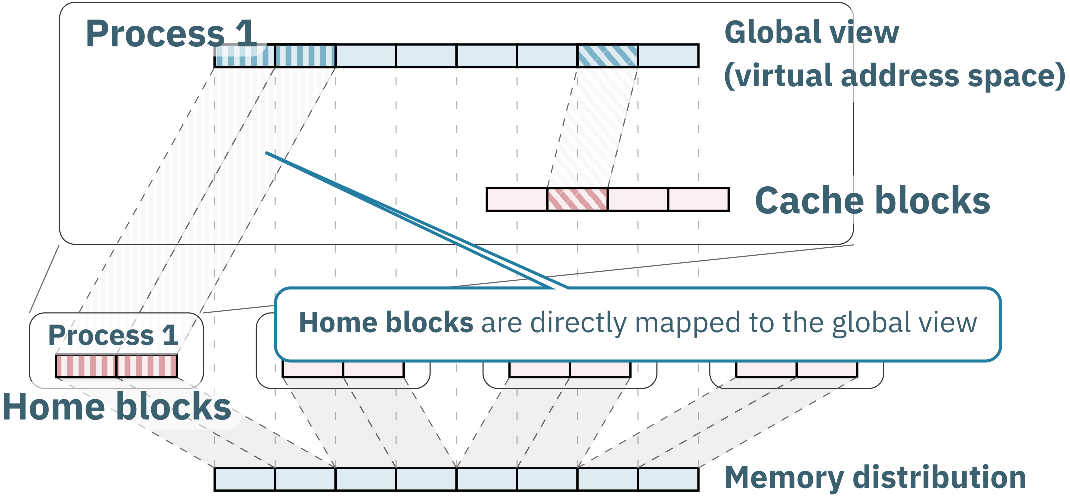


Virtual Memory Mappings for Software Cache



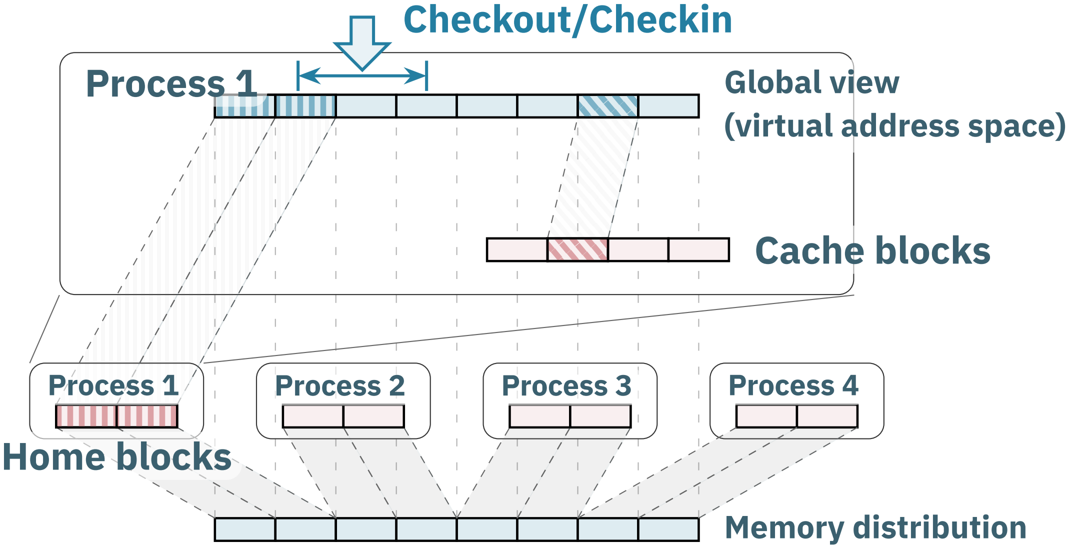
- Global memory consists of **memory blocks** of fixed size
- **Home blocks** are physical memory blocks owned by each process

Virtual Memory Mappings for Software Cache

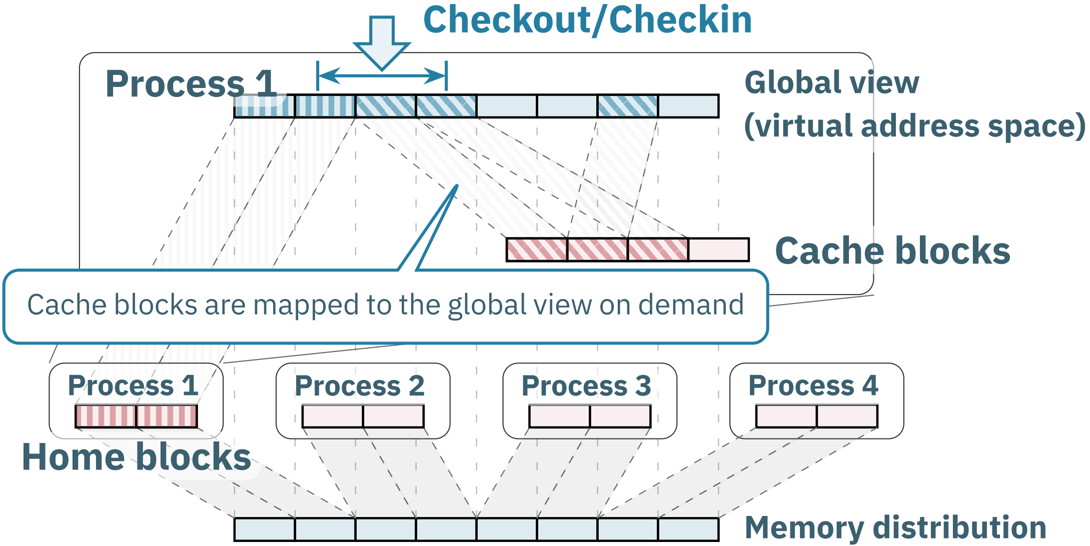


Home blocks are directly mapped to the global view

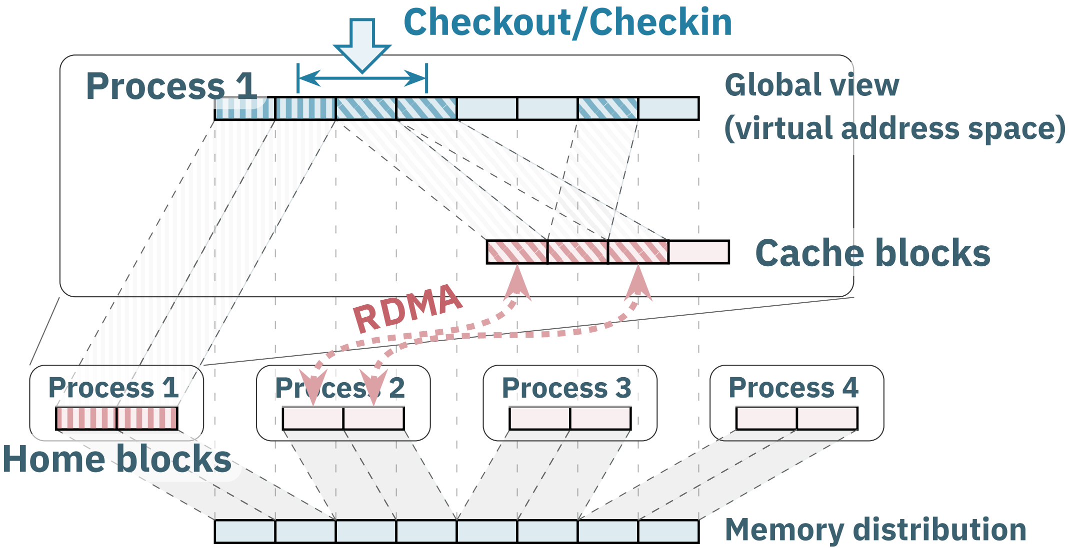
Virtual Memory Mappings for Software Cache



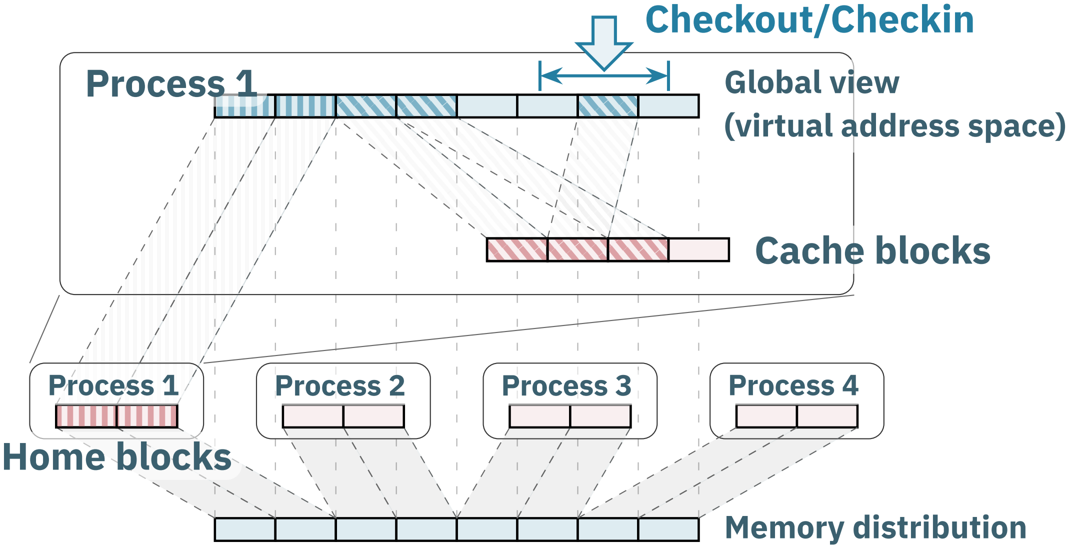
Virtual Memory Mappings for Software Cache



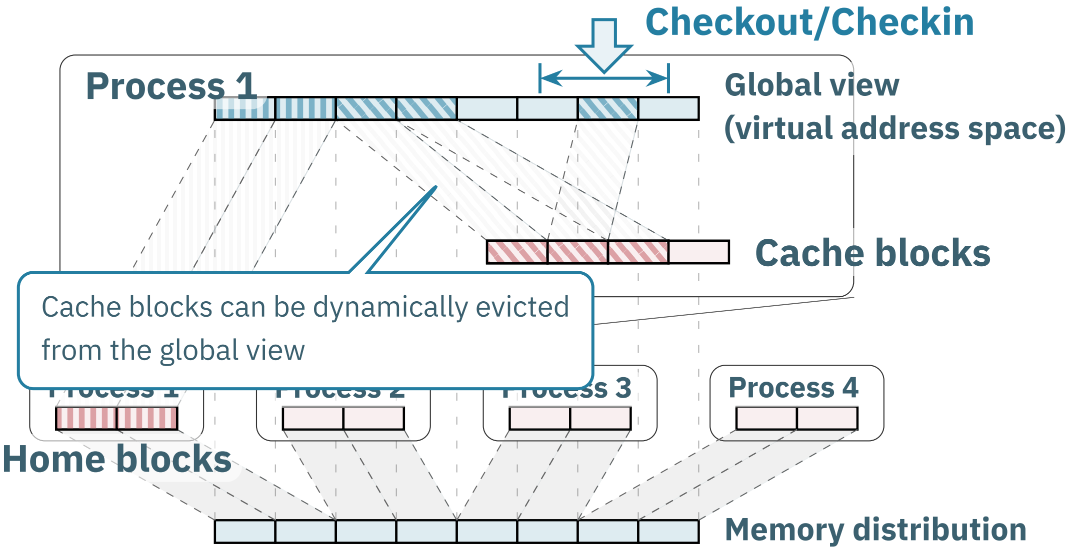
Virtual Memory Mappings for Software Cache



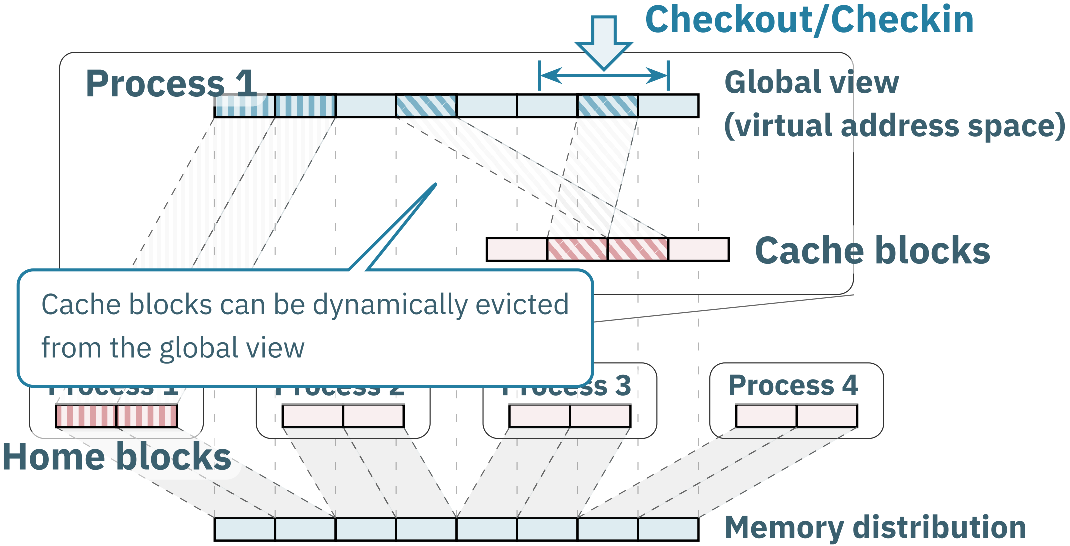
Virtual Memory Mappings for Software Cache



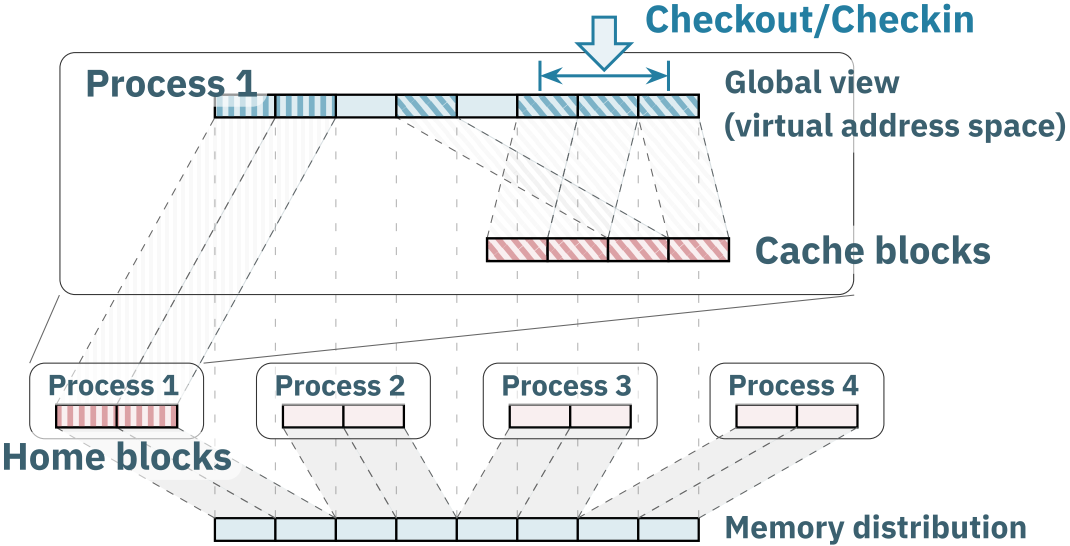
Virtual Memory Mappings for Software Cache



Virtual Memory Mappings for Software Cache



Virtual Memory Mappings for Software Cache



Memory Consistency and Cache Coherence

- Itoyori employs a relaxed memory consistency model that assumes that the program is **data-race-free**
  - No data race is allowed in Itoyori programs
- Caches can be invalidated and written back to their home at fork-join points
  - but only when work-stealing events happen
- RDMA-based efficient cache management for work stealing
- Please check out the paper for more details!

Outline

Itoyori's Programming Model

Software Caching for Global Memory Access

Evaluation

Summary

Performance Evaluation of Itoyori

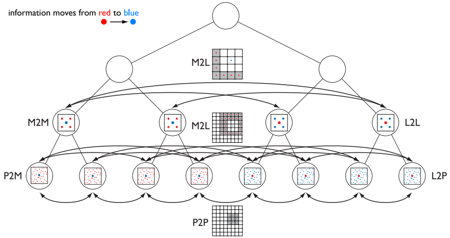
- We evaluated Itoyori with three fork-join applications
  - Cilksort, UTS-Mem, and **ExaFMM**
- In this talk, we show the result for **ExaFMM** only

Experimental environment:

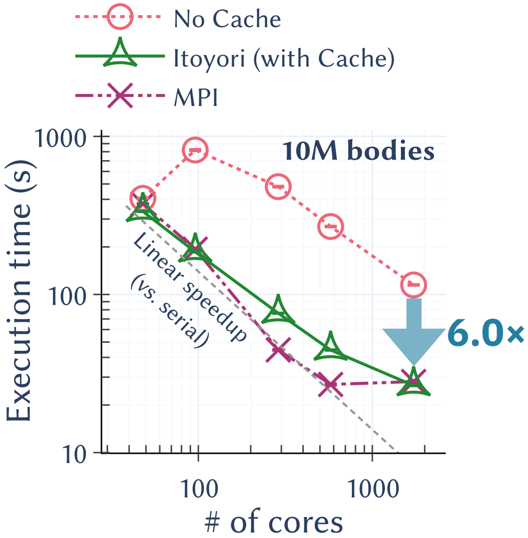
- Wisteria/BDEC-01 Odyssey supercomputer at The University of Tokyo
- Similar configuration to Fugaku Supercomputer
  - **CPU:** Fujitsu A64FX (48 cores/node)
  - **Memory:** HBM2 (32 GiB/node)
  - **Network:** Fujitsu MPI over Tofu Interconnect D

ExaFMM (High-Performance FMM Library) [Yokota+, CPC ’13]

- ExaFMM approximates interactions between far-enough particles by using a global tree
  - Highly dynamic and irregular parallelism
- We ported a shared-memory fork-join task-parallel implementation of ExaFMM [Taura+, ScalA ’12] to Itoyori
- **The overall parallel algorithm was not changed** from the original shared-memory code, except for microscopic changes
  - If we were to use MPI, we would have to redesign the parallel algorithm itself

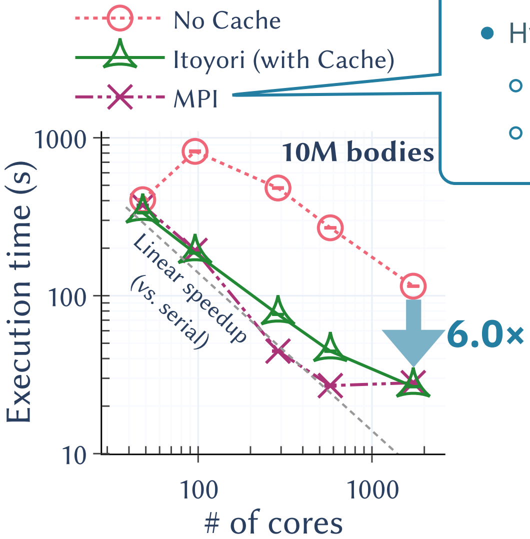


Tree-based computation in ExaFMM from [Yokota+, CPC ’13]



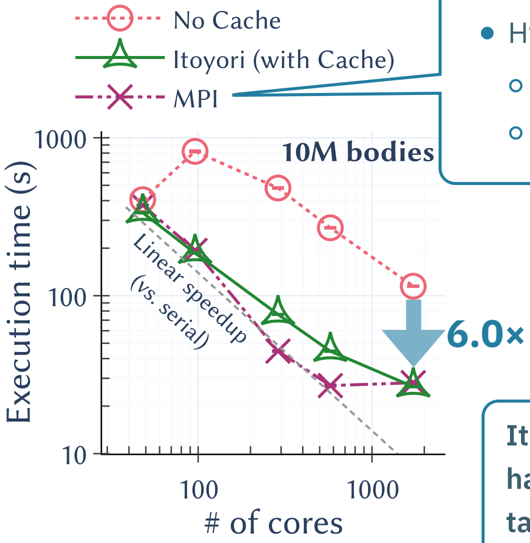
- Software caching improved performance by up to 6.0×
- **7.5× speedup on 12 nodes** (vs. 1 node)

ExaFMM ▷ Strong Scaling



- An existing MPI implementation of ExaFMM
- Hybrid of MPI and fork-join task parallelism
  - **Inter-node:** static load balancing using MPI
  - **Intra-node:** task parallelism (the same)

- Software caching improved performance by up to 6.0×
- **7.5× speedup on 12 nodes** (vs. 1 node)



- An existing MPI implementation of ExaFMM
- Hybrid of MPI and fork-join task parallelism
  - **Inter-node:** static load balancing using MPI
  - **Intra-node:** task parallelism (the same)

- Software caching improved performance by up to 6.0×
- **7.5× speedup on 12 nodes** (vs. 1 node)

Itoyori performs competitively to the hand-optimized MPI version, while maintaining high productivity

Outline

Itoyori's Programming Model

Software Caching for Global Memory Access

Evaluation

Summary

Summary

- Itoyori is a C++ global-view programming framework for fork-join task parallelism
- Software caching is a key to scale fork-join parallelism to distributed memory
- We designed efficient software cache with **checkout/checkin APIs**
- Our experiments suggested that Itoyori could achieve **a good balance between productivity and performance**

GitHub:

https://github.com/itoyori/itoyori