Task Priority Control for the HPX Runtime System

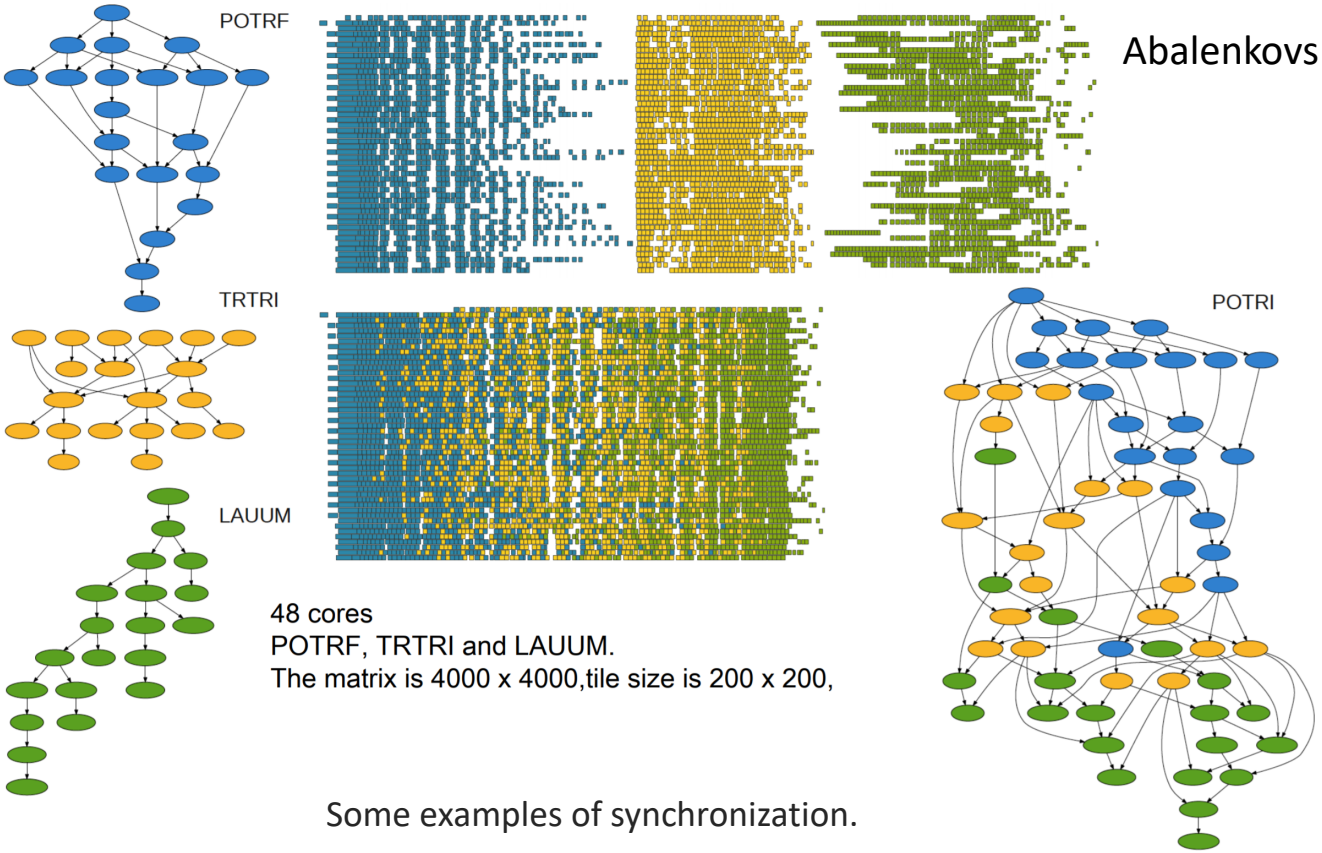
Suhang Jiang[1], Mulya Agung[2], Ryusuke Egawa[3,2], Hiroyuki Takizawa[2]
jiang.suhang@fujitsu.com, agung@tohoku.ac.jp, egawa@tohoku.ac.jp, takizawa@tohoku.ac.jp
1. Graduate school of Information Sciences, Tohoku University
2. Cyberscience Center, Tohoku University
3. Graduate School of Engineering, Tokyo Denki University
Suhang Jiang is presently with Fujitsu Corporation.

Outline

- Background
  - HPX - High Performance ParalleX
  - Task dependency
- Motivation and Contributions
- Proposed Approach
  - Decoupled thread pools
  - Thread mapping for multiple thread pools
- Evaluations
  - Evaluation setup
  - Decoupled thread pools evaluation
  - Thread mapping evaluation
- Conclusions & Future work

Background

- Synchronization is expensive!



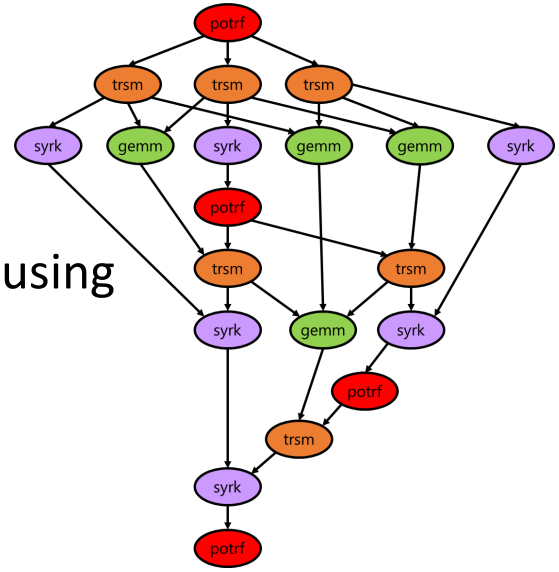
Abalenkovs et al.@2015

Some examples of synchronization.

The impact of synchronization will increase drastically on large-scale systems.

HPX – High Performance ParalleX [1]

- **Runtime system and C++ classes for large-scale task-based execution**
  - C++11 standard classes for multi-threading, such as **future** and **promise**, are used for task-based execution on a distributed-memory parallel computing system
    - Threads of an application can be executed on different NUMA domains.
    - Tasks and their dependencies can be represented as a directed acyclic graph (**DAG**).
    - Tasks can be executed asynchronously by using hpx::async.



The DAG of right view of Blocked Cholesky Factorization

[1] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task-based programming model in a global address space," in 2015 IEEE International Conference on Cluster Computing, 2015, pp. 682–689.

Motivation

- **OpenMP version 4.5 or later supports task priorities to improve performance [2].**
- **Task priority control in HPX is not perfect.**
  - Execution of critical tasks can be delayed by executing non-critical tasks.
    → Prioritizing critical tasks will improve performance.

This Work

- **A lightweight task priority control mechanism for the HPX runtime system**
  - A higher priority is given to execution of some **critical tasks**
    - Decoupled thread pools
  - A method of mapping threads to cores is also discussed
    - The built-in thread mapping methods of HPX are not applicable for the decoupled thread pools

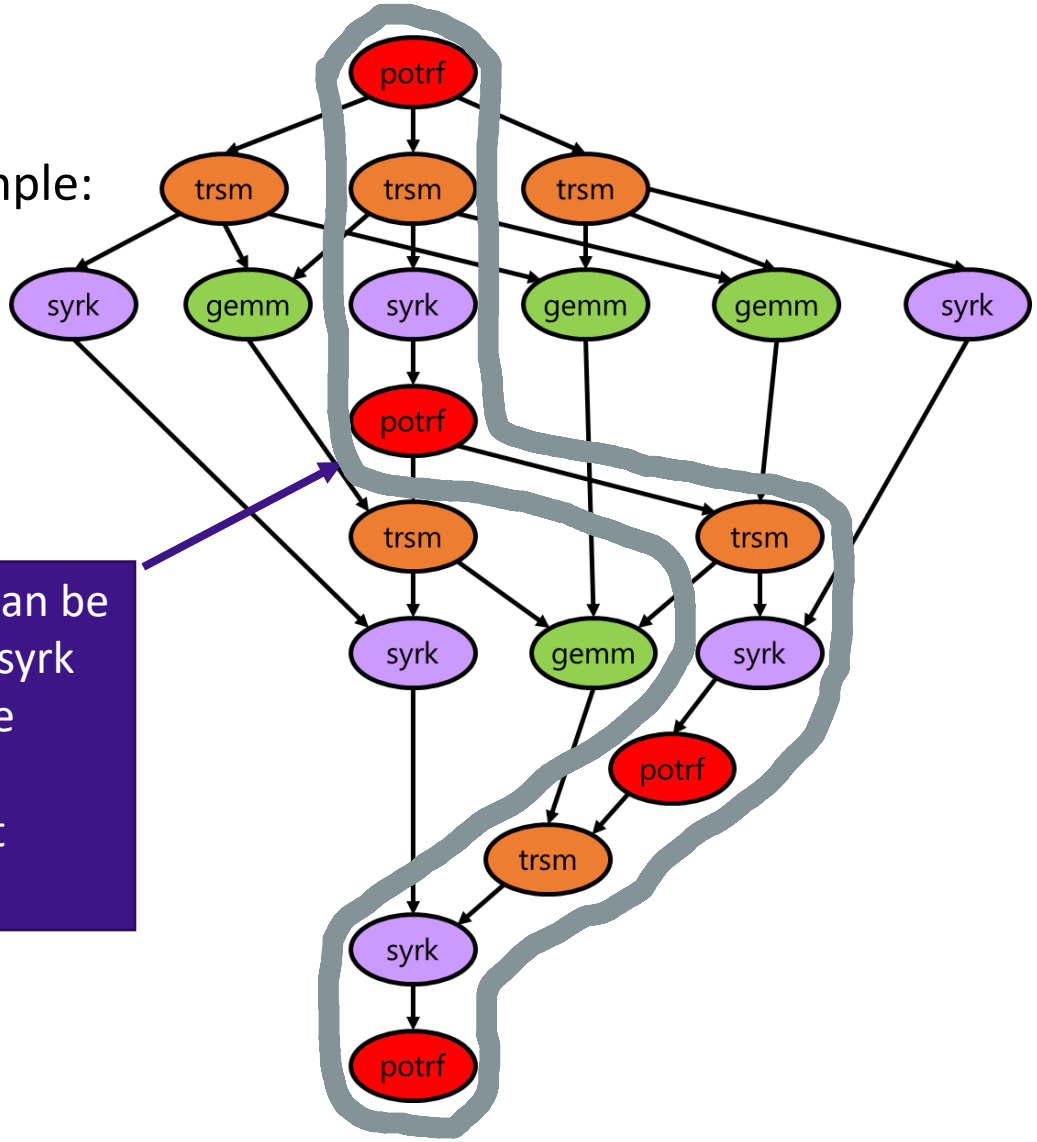
[2] Sinnen, Oliver, Jsun Pe, and Alexei Vladimirovich Kozlov. "Support for fine grained dependent tasks in OpenMP." International Workshop on OpenMP. Springer, Berlin, Heidelberg, 2007.

Contributions

1. **Task priority controls in HPX by using decoupled thread pools**
2. **A thread mapping method for decoupled thread pools**
3. **The impacts of decoupled thread pools and thread mapping mechanisms on the performance of task-based execution**
   - The sources of performance improvements are also clarified

Task Dependency

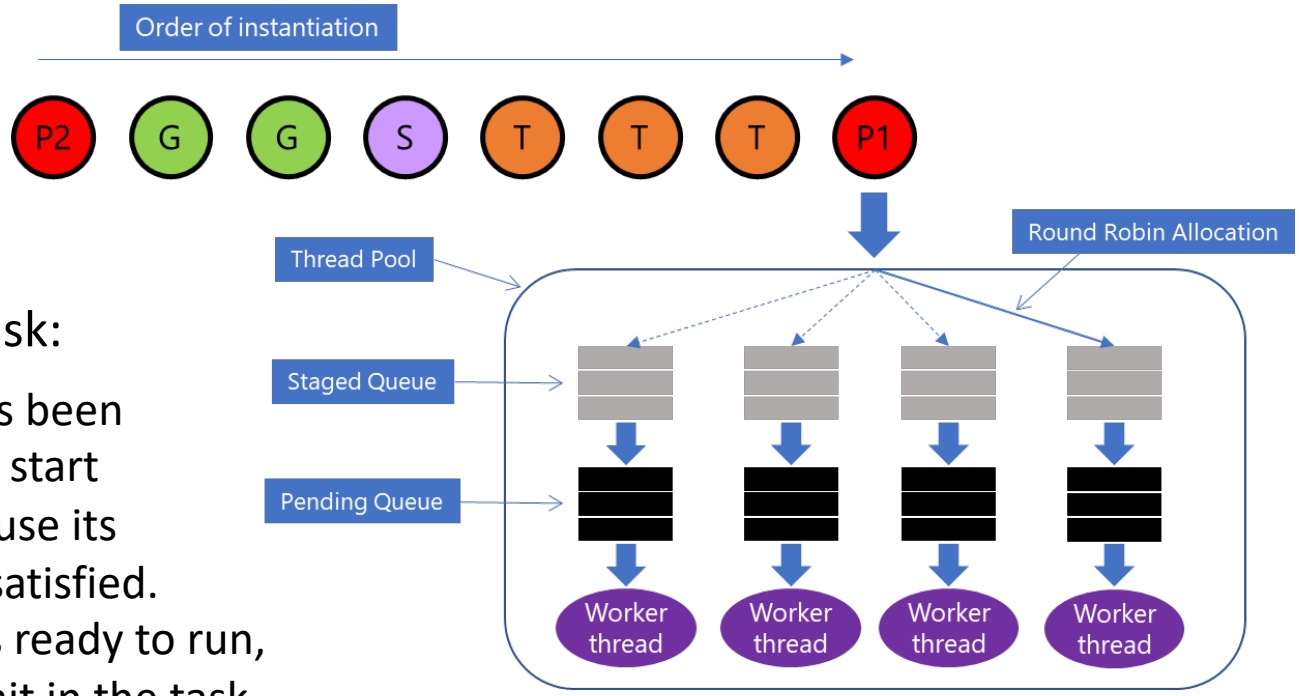
- Take the right DAG of Blocked Cholesky Factorization as an example:



The potrf task on the critical path can be executed right after the preceding syrk task. However, other tasks might be executed earlier.
So tasks on the critical path are not necessarily executed earlier.

HPX Thread Management

- In the HPX runtime system, a task is created when a future class object is instantiated, and then assigned to one of the worker threads in a round-robin fashion.



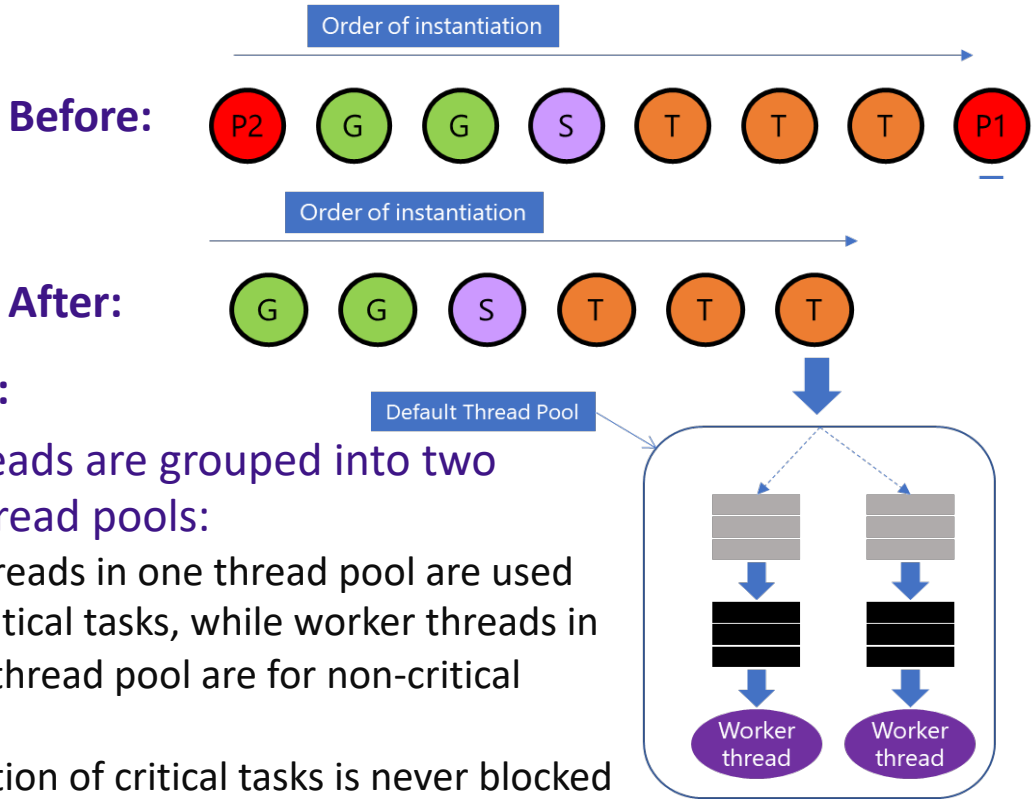
Task execution with the default mechanism of HPX

- The states of each task:
  - **Staged**: The task has been created, but cannot start execution yet, because its dependency is not satisfied.
  - **Pending**: The task is ready to run, but still needs to wait in the task queue until a worker thread becomes available.

Proposed Thread Management

Suppose programmers are responsible for finding critical tasks.

- Only the critical tasks are assigned to a dedicated thread pool instead of the default thread pool, while the other non-critical tasks are assigned to the default thread pool.



Task execution with the proposed mechanism

Differences:

Worker threads are grouped into two different thread pools:

- Worker threads in one thread pool are used only for critical tasks, while worker threads in the other thread pool are for non-critical tasks.
- The execution of critical tasks is never blocked by the execution of non-critical tasks.

Implementation on HPX

The critical tasks are assigned to the critical pool using `hpx::executors`

Listing 1. The merge sort with the default thread pool.

```
1  std::vector<int> mergesort_par(std::vector<int> v)
2  {
3    if (v.size() <= 2)
4    {
5      // Sort task
6      return serial_sort(v);
7    }
8    else
9    {
10     //Split task
11     std::tuple<std::vector<int>,
12       std::vector<int>> splits = split(v);
13     std::vector<int> left = std::get<0>(splits);
14     std::vector<int> right = std::get<1>(splits);
15
16     //Divide tasks recursively
17     hpx::future<std::vector<int>> f_left = hpx::async(
18       mergesort_par, hpx::find_here(), left);
19     hpx::future<std::vector<int>> f_right = hpx::async(
20       mergesort_par, hpx::find_here(), right);
21
22     //Merge task using the default thread pool
23     hpx::future<std::vector<int>> fm =
24       hpx::async(merge, hpx::find_here(), f_left.get(),
25         f_right.get());
26   }
27 }
```

Use the default executor

Listing 2. The merge sort with the decoupled thread pools.

```
1  std::vector<int> mergesort_par(std::vector<int> v)
2  {
3    if (v.size() <= 2)
4    {
5      // Sort task
6      return serial_sort(v);
7    }
8    else
9    {
10     //Split task
11     std::tuple<std::vector<int>,
12       std::vector<int>> splits =  split(v);
13     std::vector<int> left = std::get<0>(splits);
14     std::vector<int> right = std::get<1>(splits);
15
16     //Divide tasks recursively
17     hpx::future<std::vector<int>> f_left = hpx::async(
18         mergesort_par, hpx::find_here(), left);
19     hpx::future<std::vector<int>> f_right = hpx::async(
20       mergesort_par, hpx::find_here(), right);
21
22     //Merge task using the critical thread pool
23     hpx::threads::executors::pool_executor
24       crit_executor(CRITICAL_POOL_NAME);
25     hpx::future<std::vector<int>> fm =
26       hpx::async(critical_executor, merge,
27         hpx::find_here(), f_left.get(), f_right.get());
28   }
29 }
```

Set the executor to use the critical pool

The code snippets of the default and the decoupled thread pool mechanisms

Thread Mapping

How to map critical threads to NUMA domains?

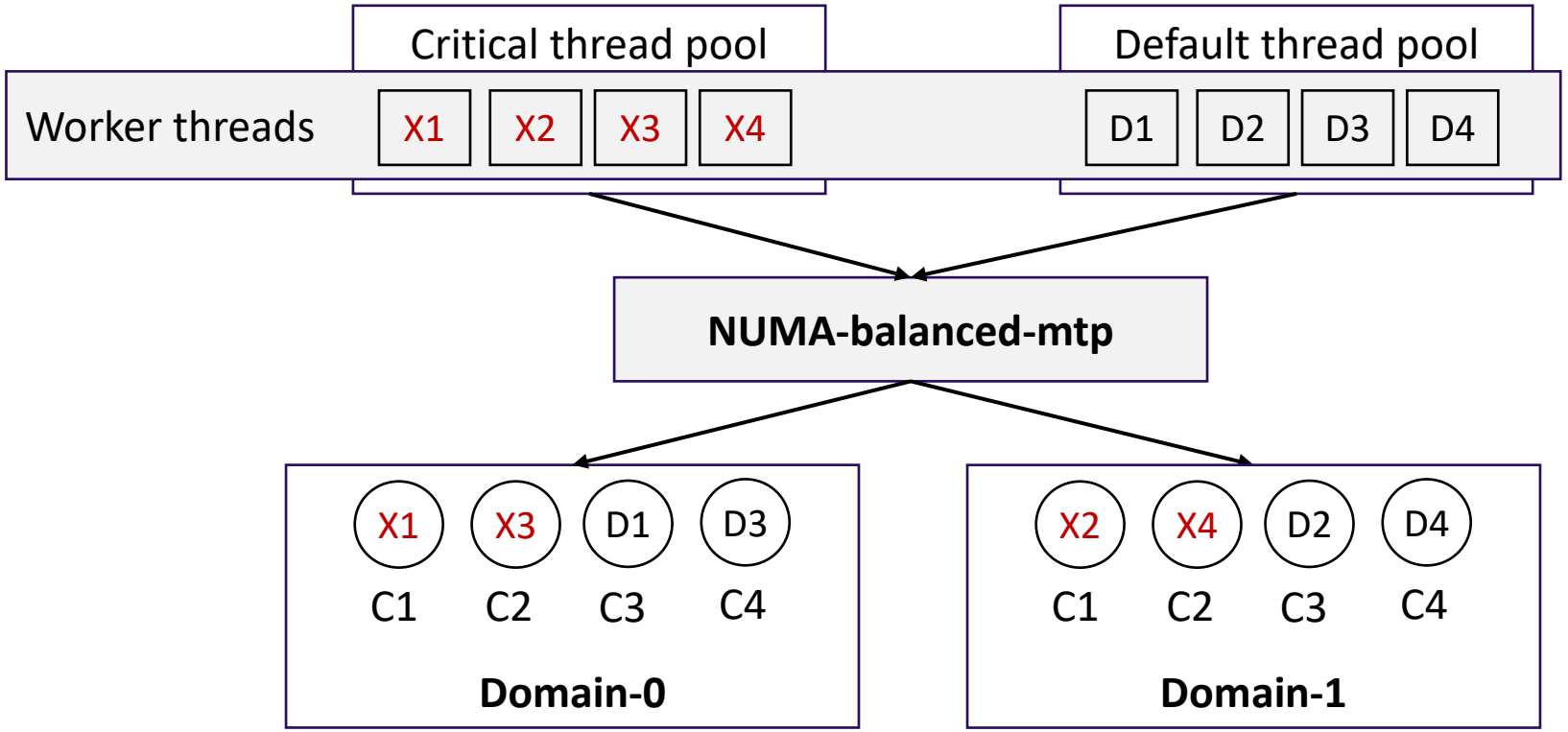
- Since worker threads in each of the two thread pools can be mapped to processor cores in various ways, thread mapping can improve performance.
- We propose a thread mapping method, called **NUMA-balanced-mtp**, to reduce the load imbalance among the NUMA domains.

  - Default : critical tasks are placed closer.

  - NUMA-balanced-mtp: loads of critical tasks are balanced.
    Distribute the threads from multiple thread pools among the NUMA domains

Thread mapping in two NUMA domains

NUMA-balanced-mtp Algorithm



The result of thread mapping on two NUMA domains

Evaluation Setup

- Evaluate the performance gain by the proposed mechanism using two benchmarks on two different systems.

Machine	NUMA domains	Cores/domain	Threads/core
Intel Xeon Phi KNL	4	18	4
Oakbridge-CX	2	28	1

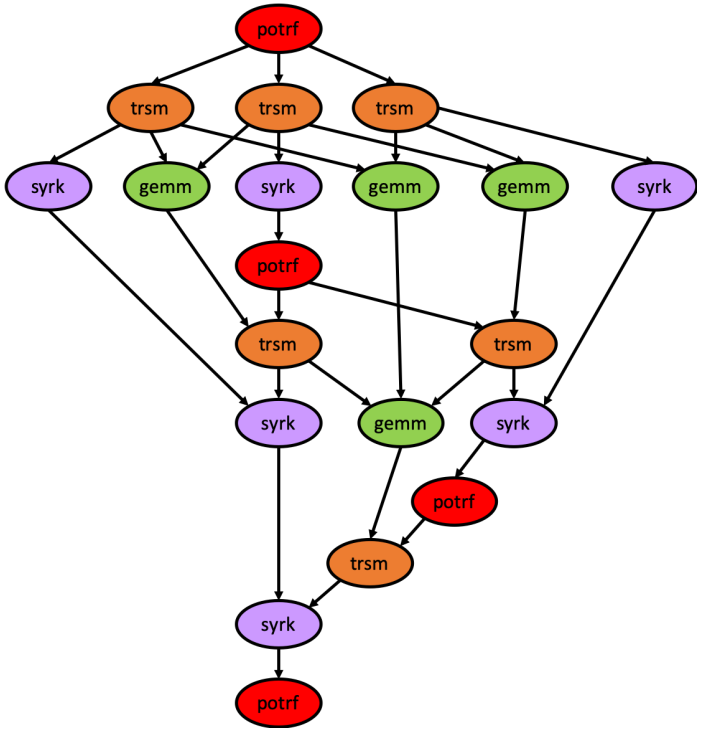
The configurations of the Intel Xeon Phi KNL(**KNL4**) and Oakbridge-CX(**OCX**) systems.

- Two different benchmarks:

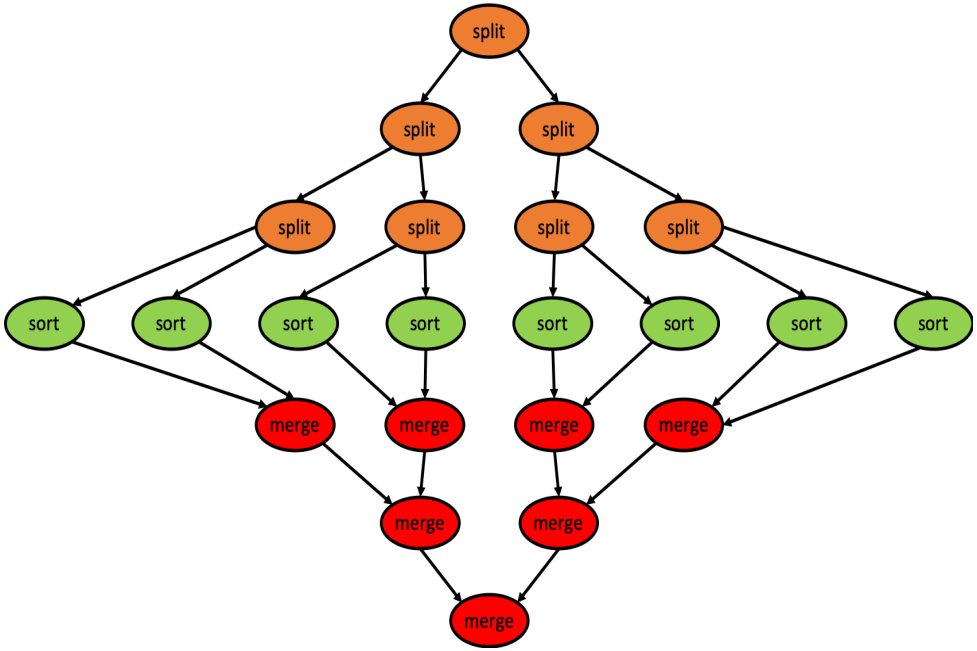
	Data format	Parallelization
Cholesky factorization	matrix-based	tasks with loops
Merge sort	array-based	Recursive tasks

- Compare the execution times of the proposed mechanism to those of the default mechanism of HPX
  - Evaluate different configurations of the number of threads in each thread pool
  - Evaluate the impacts of the proposed thread mapping mechanism on the performance
- Investigate the sources of performance improvements by measuring the waiting time of tasks in the staged and pending queues

Critical Tasks of the Benchmarks



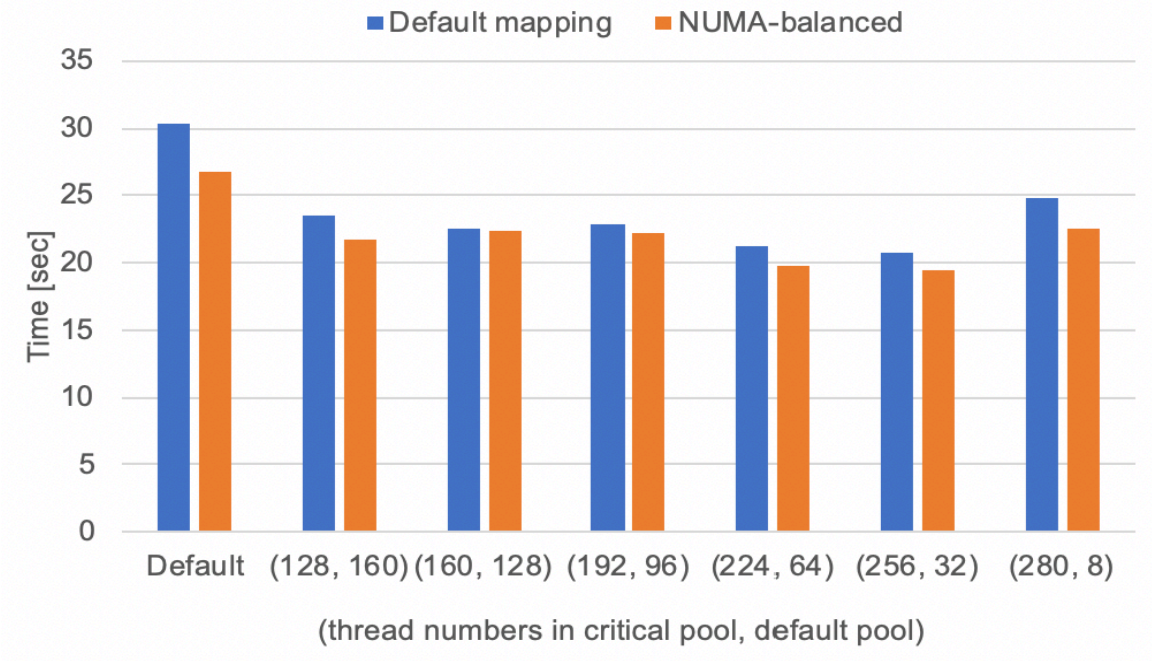
The DAG of the Cholesky factorization benchmark. The potrf task is detected as the critical task



The DAG of the merge sort benchmark. The merge task is detected as the critical task

Performance Results on the KNL4

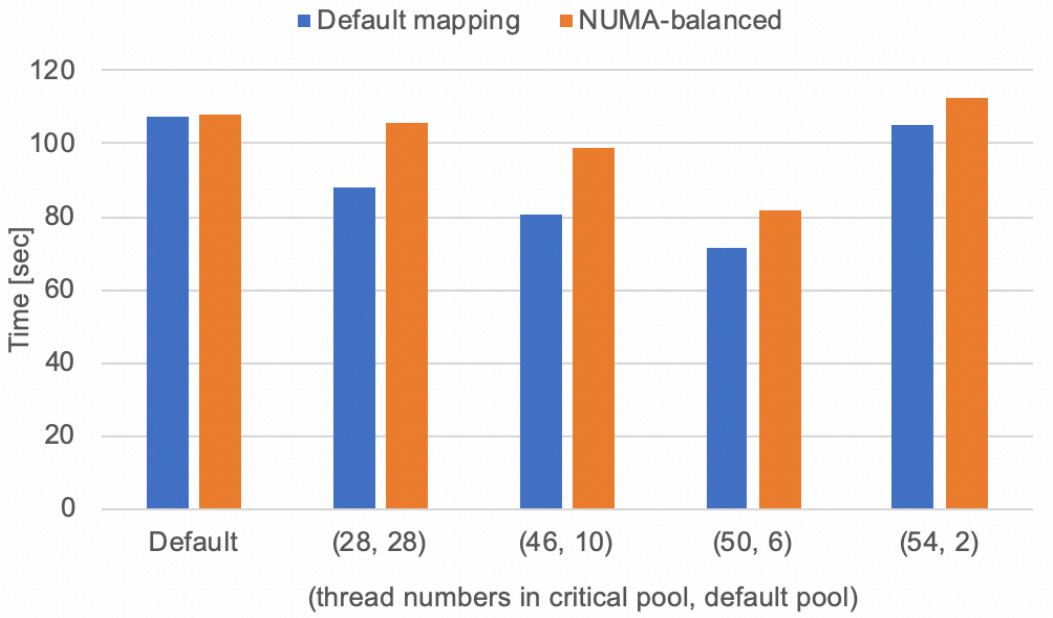
- The execution time is reduced by 31.8% at the thread pool configuration of (256, 32) with the default thread mapping method.
- For the same problem, using the NUMA-balanced-mtp thread mapping can further increase the performance by 4.8%.



Performance results of the Cholesky benchmark on the KNL4
(288 threads, 512x512 matrix size)

Performance Results on the OCX

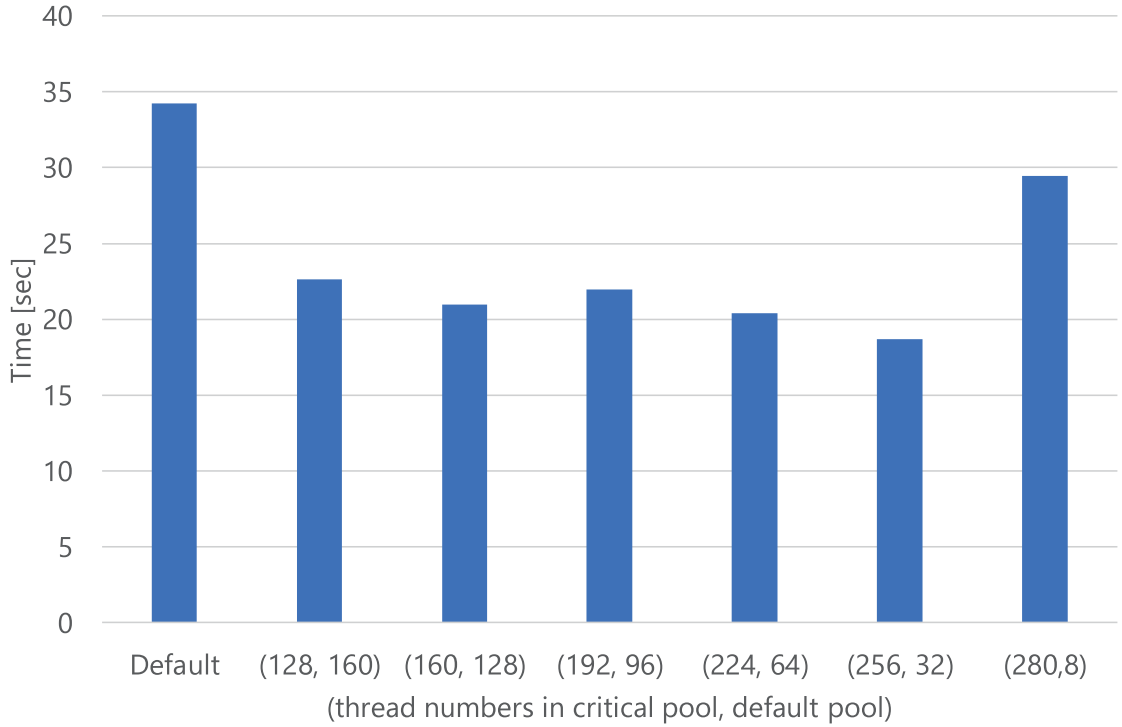
- The proposed implementation can increase the performance by 33.5% in terms of execution time.
- The mapping method decreases the performance because the communication cost among NUMA domains on OCX is larger and thus cancels the performance gain by reducing the load imbalance.



Performance results of the Cholesky benchmark on the OCX
(56 threads, 2048x2048 matrix size)

Performance Results of Merge Sort

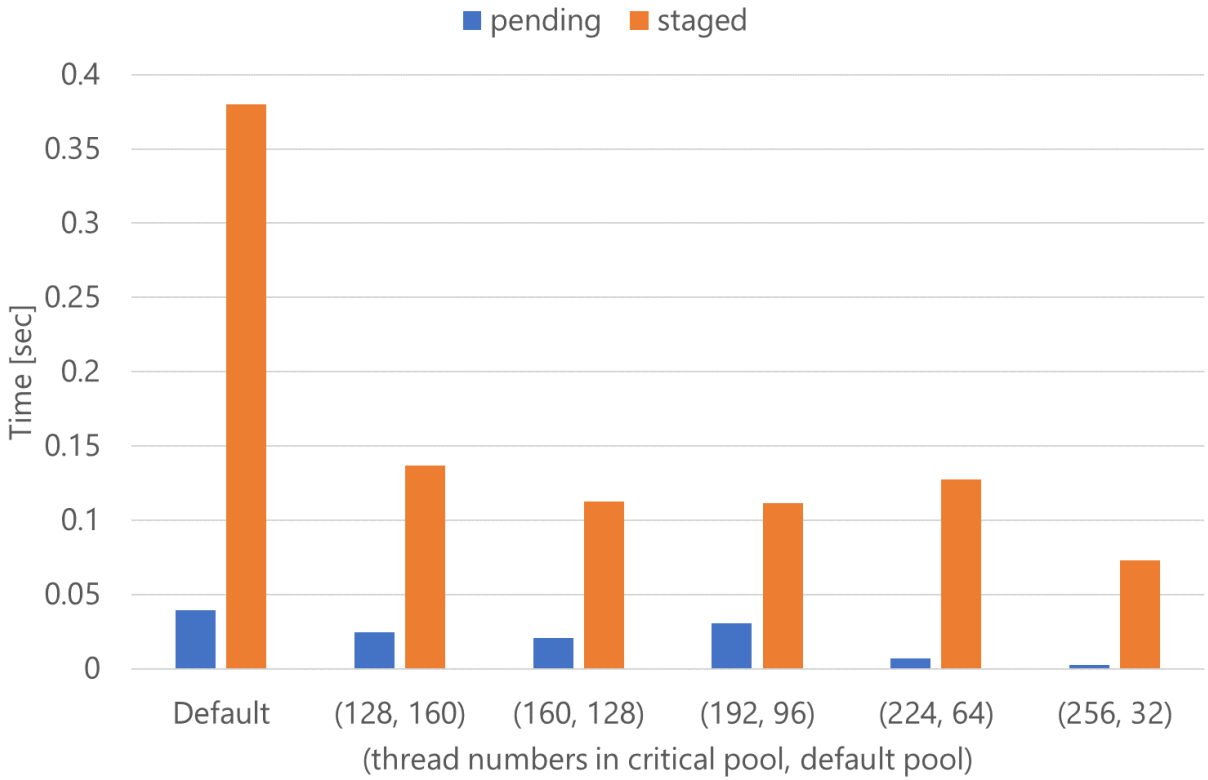
- All the configurations of the proposed decoupled thread pools mechanism can achieve a higher performance than the default mechanism.
- The best configuration is shown by the (256, 32) configuration, which decrease the execution time by 47% compared with the default mechanism.



Performance results of the merge sort benchmark on the KNL4 (288 threads, 10^5 array size)

Sources of Performance Improvements

- The configuration with 256 threads in critical thread pool can achieve not only the shortest pending time but also the shortest staged time.
- This explains why this configuration can achieve the shortest execution time.



The waiting times of tasks in the staged and pending queues. The results are obtained using the HPX counters on the KNL4

Workloads of Critical Tasks of the Benchmarks

- The workloads of the critical tasks are analyzed from the **ratio of the number of executed threads and thread times of the critical pool and default pool**
- In the Cholesky benchmark, the execution times of critical tasks are much longer than those of the non-critical tasks.
- In merge sort, the number of executed critical tasks is almost equal to that of the non-critical tasks.

The impacts of the decoupled thread pools mechanism depend on the application workload

Conclusions

This work has proposed a task priority control mechanism that uses decoupled thread pools in order to prioritize the execution of critical tasks.

- ✓ By using a pool of worker threads dedicated to critical tasks, the proposed mechanism can prevent critical tasks from waiting for non-critical tasks.
- ✓ As a result, the proposed mechanism can significantly reduce the waiting time of critical tasks, and hence the total execution time.
- ✓ In addition, the effects of using different thread mapping methods are also investigated empirically.

Future Work

- **The parameters of the numbers of pools and threads in each pool are empirically adjusted by hand in advance.**
  - Since the performance is sensitive to the parameter values, auto-tuning of the parameters will be an interesting research topic.

Thank you!

Acknowledgments

The authors would like to thank Prof. Hiroaki Kobayashi of Tohoku University and Dr. Kentaro Sano of RIKEN R-CCS. This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program “R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications,” Grant-in-Aid for Scientific Research(B) #16H02822 and #17H01706, and Initiative on Promotion of Supercomputing for Young or Women Researchers, Information Technology Center, The University of Tokyo.