Task Priority Control for the HPX Runtime System[1][2]

Suhang Jiang
Tohoku University

1. INTRODUCTION

In recent years, high-performance computing (HPC) is experiencing a phase change with the challenges of programming, management of heterogeneous multicore system architectures, and large scale system configurations. Synchronization [1] is the process of coordinating the behaviors of two or more processes, which may be running on different processors. Synchronization is needed for various collective communications such as parallel reduction. As the HPC system size increases, the overhead of a global synchronization involving a large number of processes is likely to become larger. Therefore, there is a demand for avoiding such expensive global synchronizations as much as possible to achieve high performance on an unprecedentedly large HPC system in the future.

Task-based execution is one promising approach to minimizing global synchronizations, because it does not perform expensive synchronization as long as task dependencies are not violated. High Performance ParalleX (HPX) [2][3] is one promising candidate of task-based programming and execution models, which provides a C++ class library to describe tasks and their dependencies, and also a runtime system for parallel computing based on the partitioned global address space (PGAS) model [4]. Therefore, this work focuses on HPX for task-based parallel execution.

In HPX, an OS-level thread called a worker thread executes tasks in a queue in a first-in first-out (FIFO) policy; tasks in the queue cannot overtake their preceding tasks. As a result, the execution of a task on the critical path can be delayed by executing other non-critical tasks. The OpenMP specification version 4.5 [5] and later support task priority, resulting in higher performance. However, as far as we know, such a task priority control mechanism has hardly been discussed for the HPX runtime system. Therefore, this paper discusses task priority control to improve the performance of a task-based HPX application by decreasing the waiting time of critical tasks in the queue.

The main contributions of this work are as follows:

1) Task priority control in HPX is achieved by using decoupled thread pools,

2) An appropriate thread mapping strategy to prioritize critical tasks is explored, and

3) Performance improvement by task priority control is quantitatively evaluated.

---

[1] 課題採用時の情報は、2019 年度若手・女性後期採択課題の「並列プログラミングモデル HPX と XMP の比較と改善」となります。
[2] この論文は iWAPT2020 にて発表しました。

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 presents the proposed mechanism to achieve task priority control in the HPX runtime system. Section 4 shows some evaluation results, and discusses the impact of task priority control on performance. Finally, Section 5 gives some concluding remarks, describes our future work.

2. RELATED WORK

Task-based execution is promising to prevent global synchronizations and thus to achieve high performance on a largescale future HPC system, on which a global synchronization is even more expensive [6]. HPX [2][3] is a parallel runtime system designed to overcome the limitations of conventional runtime systems such as starvation, communication latencies, remote resource access overheads, and the waiting time until contention resolution. It also provides a C++ class library for task-based programming.

HPX uses standard C++ classes for asynchronous operations, such as future and promise. A future class object is used to retrieve a value that is set by a different thread using a promise class object. In C++11 and later, thus, two different threads can share data using future and promise class objects. In HPX, the thread using a promise object could run on a different node. First, when a future class object is created by a thread running on one node, the thread is not blocked and continues to execute. In the background, another thread is launched on the same node to implicitly manage the inter-node communication. Then, the latter thread is waiting until the value of a promise class object is set by a different thread potentially running on another node. Finally, the value is shared via inter-node communication.

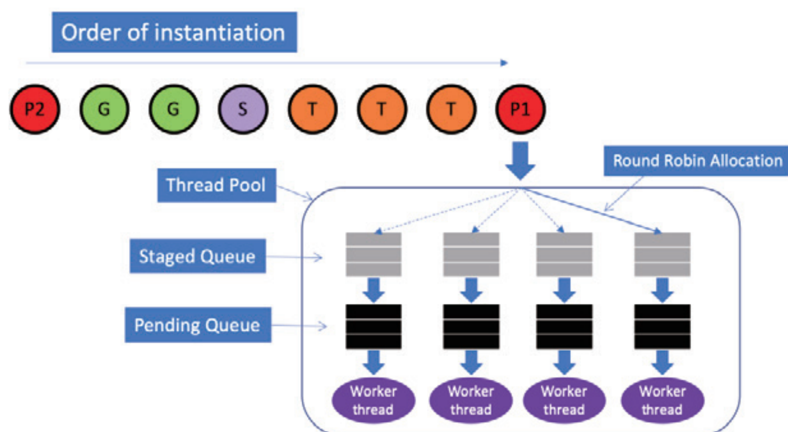

Fig. 1. Task execution with the original mechanism. Every task is executed by one of worker threads in the default thread pool. A red task (P1 and P2) is critical and can start running right after its preceding purple task (S).

Figure 1 illustrates how tasks are executed in the original HPX runtime system. 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. Each task can be in one of five states. The first one is running, which means that the context is active, and the code is running as if it runs with a native thread. The second state is suspended, which means that the task is waiting for a synchronization event. The third state is staged, which means that the task has been created, but cannot start execution yet, because its dependency is not satisfied. The fourth state is pending, which means that the task is ready to run, but still needs to wait in the task queue until a worker thread becomes available. The fifth state is terminated, which usually means that the task execution is finished. To manage the stages of each task, a worker thread has two kinds of task queues, the staged and pending queues. When a task is assigned to a worker thread, the task is first pushed to the staged queue until its dependency is satisfied. When the task dependency is satisfied, the task is moved to the pending queue and waits until the corresponding worker thread becomes available for the execution. Tasks in the pending queue are executed in a FIFO policy, and cannot overtake their preceding tasks. Thus, the execution of a task on the critical path could be delayed by executing other threads, if the critical task is moved to the pending queue after the noncritical ones. Suppose that red tasks in Figure 1, P1 and P2, are critical, and P2 needs only the result of task S. Then, task P2 can start running right after task S. However, task P2 needs to wait for task G if tasks G and P2 are assigned to the same worker thread. As a result, the critical task needs to wait longer than necessary, and hence the total execution time increases. Therefore, this paper improves the HPX performance by giving a higher execution priority to critical tasks.

The OpenMP specification version 4.5 [5] and later support a mechanism to specify the priority of a task in the task queue. This priority is a hint to the OpenMP runtime system for the order of task execution. During the execution, the task scheduler will execute higher priority tasks before lower priority ones as long as the task dependencies are not violated. In HPX, a task can still wait in the pending queue even if its dependency is satisfied. The task will wait until a worker thread becomes available in the default thread pool. In contrast to the mechanism of OpenMP, our proposed method uses decoupled thread pools to control the priority of tasks. By using different thread pools for critical and non-critical tasks, the critical tasks can be executed by worker threads without interference from non-critical tasks.

This paper also discusses a thread mapping method to map each worker thread to a different processor core. A better thread mapping method can mitigate the load imbalance and improve the performance of a task-based application. HPX provides four different built-in thread mapping methods. One of the four built-in methods is selected when an HPX application is launched. The first method is the most standard way of thread mapping, called compact. Using this method, all the threads will concentrate on as fewer sockets or NUMA domains [2] as possible. The second method, called scatter,

evenly assigns threads to physical cores and NUMA domains. The third and fourth methods, called balanced and NUMA-balanced, are similar to scatter. They assign threads to physical cores and NUMA domains in a round-robin fashion.

However, balanced assigns consecutive threads to different physical cores, while NUMA-balanced assigns consecutive threads to different NUMA domains. One problem of the built-in mapping methods is that all the methods assume to use the default task management of HPX. If multiple thread pools are used for the execution as proposed later in this paper, the built-in mapping methods may not effectively improve performance. In this work, therefore, it is necessary to design and implement a thread mapping method for exploiting the potential of multiple thread pools.

3. TASK PRIORITY CONTROL FOR HPX

As shown in Figure 1, in the original HPX runtime system, a task is created when a future class object is instantiated. Then, tasks are assigned to worker threads in a round-robin fashion. Each worker thread has two kinds of task queues. A task waits in the staged queue until the task dependency is satisfied. After that, the task moves to the pending queue and waits until the worker thread is available for the task execution. As a result, tasks assigned to one worker thread are executed in an FIFO fashion; a subsequent task cannot overtake its preceding tasks. Therefore, the execution of critical tasks can be delayed by executing its preceding tasks no matter if the preceding ones are critical.

In this work, we propose a task priority control mechanism for the HPX runtime system so that critical tasks can be executed right after their task dependencies are satisfied. In the proposed mechanism, 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. Therefore, since critical and non-critical tasks are respectively assigned to different thread pools, the execution of critical tasks is never blocked by the execution of non-critical tasks.

In the proposed mechanism, critical tasks are prioritized as follows. Suppose that the task dependencies of an application are represented as a directed acyclic graph (DAG). Then, critical tasks can be selected with considering the DAG. The proposed mechanism assumes that programmers are responsible for identifying critical tasks of an application. As a typical example, in this paper, critical tasks are defined as the tasks that appear most frequently on the critical path of an application. The critical tasks are automatically detected by using the following three steps. First, we find the critical path of the DAG by finding the longest path in the DAG. Second, we calculate the number of occurrences of each task in the critical path. Finally, the tasks that have the highest number of occurrences in the critical path are detected as the critical tasks. After that, 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. In addition, since worker threads in each of the two thread pools can be mapped to processor cores in various ways, this paper also proposes a thread mapping method for decoupled thread pools.

As shown in Figure 1, in the original HPX runtime system, all tasks are assigned to worker threads in the default thread pool. If the result of one task is required by other subsequent tasks, the subsequent tasks are blocked in the staged queues until the result of the preceding task becomes available. Suppose that a critical task is blocked. Then, if some other tasks are in the pending queue, they are executed earlier than the critical task, potentially increasing the waiting time of the critical task and hence prolonging the critical path. Therefore, the original HPX mechanism needs an additional mechanism for giving a higher priority to critical tasks so as to prevent prolonging the critical path.

Figure 2 illustrates how tasks are executed with the proposed mechanism. Unlike the original mechanism, the proposed mechanism can prioritize to execute critical tasks by using a dedicated thread pool for the critical tasks, called a critical thread pool. In the proposed mechanism, critical task P2 can be executed right after task S. Since the critical tasks and non-critical tasks are assigned to different thread pools, the critical tasks in the pending queues do not need to wait for noncritical tasks. Therefore, the proposed mechanism can prevent delaying the execution of critical tasks. Because for each task queue, it is divided into staged queue and pending queue, between the decoupled thread pools, task dependencies can be achieved better.

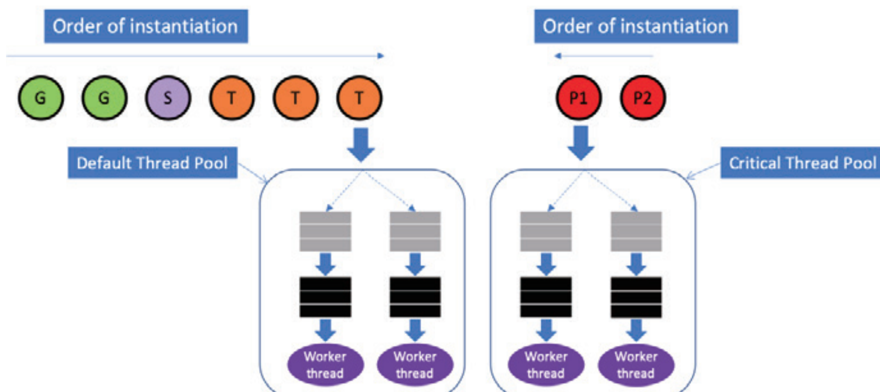

Fig. 2. Task execution with the proposed mechanism. Only critical tasks (P1 and P2) are assigned to worker threads in the critical thread pool.

In the original HPX runtime system, all worker threads are in a single thread pool, and selected for individual tasks while considering only their localities, i.e., NUMA domains. As shown in Figure 2, however, the proposed mechanism uses two types of worker threads for critical and non-critical tasks, respectively. A newly arisen problem is how to map worker threads to processor cores with considering not only their localities but also their types. Therefore, in Section IV, we discuss the performance gain by the

proposed mechanism, and also the effect of using a different thread mapping method on performance.

In HPX, a task can be executed by any threads of a thread pool by using a task executor called **pool_executor**. We use **hpx::async** function call to execute all the tasks asynchronously. Which thread pool is used for executing a task can be determined by specifying the parameter of thread pool in the executor. Thus, to assign the critical tasks to the critical pool, we set the critical pool as a parameter of the executor that runs the critical tasks. On the other hand, we use the default thread pool for the executor running the non-critical tasks. Listings 1 and 2 show the code snippets of the merge sort benchmark using the default thread pool and the decoupled thread pools, respectively. In this benchmark, the tasks are divided in a recursive way, and they are run in parallel by using the hpx::async function. The differences between the default and the proposed mechanisms are shown in the code, starting from Line 22. As shown in Listing 2, the proposed mechanism first sets the critical pool as the thread pool of the critical_executor (Lines 23-24). Then, the critical pool is used for executing the merge task by specifying the critical_executor as the task executor for the task.

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 }
```

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 }
```

A. Thread Mapping for Multiple Thread Pools

Since the built-in mapping methods of HPX are only applicable for the default thread pool, this work proposes a thread mapping algorithm for multiple thread pools. We adopt the built-in NUMA-balanced method of HPX to implement the proposed algorithm, NUMA-balanced-mtp. The key difference between these two algorithms is that NUMA-balanced-mtp distributes the threads from multiple thread pools among the NUMA domains. We choose the NUMA-balanced mapping because it can effectively improve performance on a NUMA system by reducing the load imbalance among the NUMA domains [7][8].

Algorithm 1 depicts the NUMA-balanced-mtp algorithm. First, the algorithm retrieves the available NUMA domains using the resource::partitioner object of HPX (Line 1). Then, it iterates the threads of all the thread pools. The next(pools) function starts from the critical thread pools to give a higher priority to the critical thread pools to be mapped than the other thread pools (Lines 2 and 9). For each thread, the algorithm selects a target NUMA domain in a round-robin fashion (Line 6). The threads are distributed among the NUMA domains to reduce the load imbalance. Then, it maps the thread to a processor core of the target NUMA domain (Line 7). To map a thread to a processor core, the map function uses the add_resource routine of the resource::partitioner object.

Algorithm 1 The NUMA-balanced-mtp Algorithm

Input: *rp* {Resource partitioner object of HPX}
Input: *pools* {Thread pools}
1: *domains* ← *rp.numa_domains*()
2: *pool* ← *next*(*pools*)
3: **while** *current_pool* **do**
4: **for** *thread in pool* **do**
5: {Iterate the NUMA domains in a round-robin fashion}
6: *target_domain* ← *next*(*domains*)
7: *map*(*thread*, *target_domain*)
8: **end for**
9: *pool* ← *next*(*pools*)
10: **end while**

TABLE I THE CONFIGURATIONS OF THE OAKBRIDGE-CX AND KNL4 SYSTEMS.

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

4. EVALUATION AND DISCUSSIONS

As shown in Figure 1, in the original HPX runtime system, a task is created when a future class object is instantiated. Then, tasks are assigned to worker threads in a round-robin fashion. Each worker thread has two kinds of task queues. A task waits in the staged queue until the task dependency is satisfied. After that, the task moves to the pending queue and waits until the worker thread is available for the task execution.

A. Evaluation Setup

In this section, we evaluate the performance gain by the proposed mechanism, using two benchmark programs on the following two different systems. The first system is a large cluster system of Intel Xeon Platinum 8280 processors, called Oakbridge-CX (OCX) [9], which is installed at the Information Technology Center, the University of Tokyo. The second is an Intel Xeon Phi Knights Landing (KNL) system [10], called KNL4. The system is configured as a four-node NUMA system by setting Sub-NUMA clustering (SNC) mode as the clustering mode of the KNL. In this mode, the system is partitioned into four NUMA nodes, with 72 logical cores per NUMA node. The hardware configurations of OCX and KNL4 are shown in Table I. HPX version 1.4.0 is used as the runtime system for all the experiments.

One benchmark used to evaluate the performance is Cholesky factorization [11] to compute,

$$A = LL^*; \quad (1)$$

where A is a Hermitian positive-definite matrix, L is a lower triangular matrix with real and positive diagonal entries, and L* denotes the conjugate transpose of L. The benchmark program consists of four kinds of tasks; Cholesky decomposition (potrf), solving the triangular matrix equation (trsm), matrix multiplication (gemm), and

symmetric rank-k update (syrk). A DAG of the task dependency for tiled Cholesky factorization of 4×4 tiles is shown in Figure 3. Once the k-th panel is factorized and then broadcast, the next most urgent task to complete is factorization of panel k + 1, which is a so-called look-ahead [12]. Therefore, it is already known that the potrf tasks in the DAG are critical.

The other benchmark is an implementation of merge sort. In the case of merge sort, the DAG consists of split, sort and merge tasks, and there is no clear critical path, as shown in Figure 4. However, as shown in the graph, the split and marge tasks appear most frequently on every path. Compared with the split task, the merge task is more computationally expensive because it needs to merge the previous results of the sort tasks in a sorted order. Therefore, in this work, the merge task is considered as a critical task. Only merge tasks are assigned to worker threads in the critical thread pool.

B. Evaluation Results

First, the performance gain by the proposed mechanism is evaluated with the tiled Cholesky factorization whose DAG is shown in Figure 3. The tile size is set to 4×4 elements, and thus each task operates on a submatrix of 4×4 elements. Two thread mapping methods, the default and NUMA-balanced-mtp thread mapping methods are compared in the following evaluation. In the HPX runtime system, as shown in the Figure 5, balanced mapping is used as the default mapping. Thus, the default mapping distributes worker threads to all the physical cores without considering the NUMA domains. On the other hand, the NUMA-balanced-mtp distributes all worker threads of the decoupled thread pools among the NUMA domains as follows. First, it assigns consecutive threads of the critical thread pool to different NUMA domains. Then, it assigns consecutive threads of the default pool to different NUMA domains.

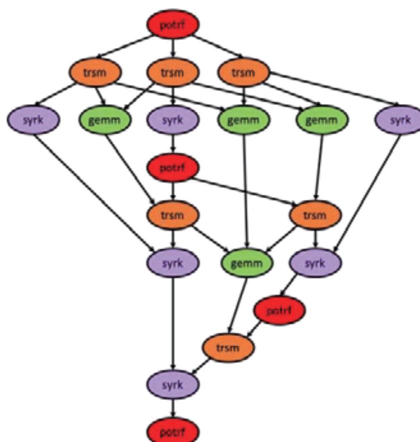

Fig. 3. Task dependency for tiled Cholesky factorization of 4×4 tiles. The potrf tasks are critical.

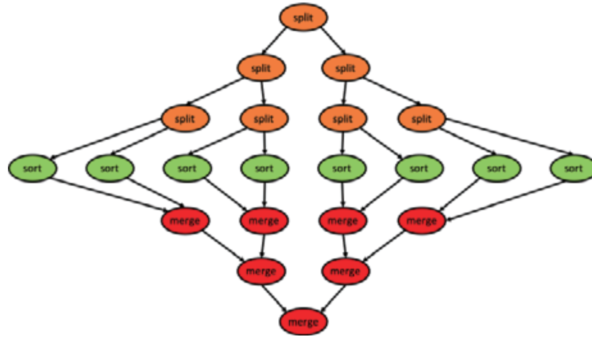

Fig. 4. Task dependency for merge sort (8-way parallel execution). The merge tasks are handled as critical tasks in the evaluation.

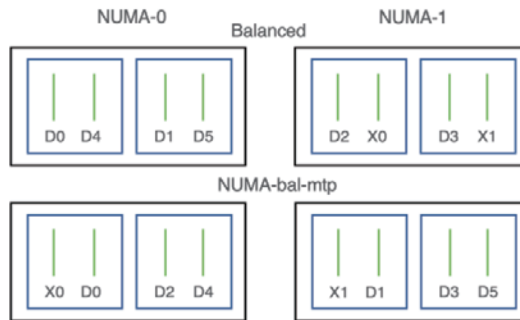

Fig. 5. Example of mapping results with two NUMA domains, six threads in the default pool, two threads in the critical pool. Here Xi is the worker thread i of the critical thread pool, and Di is the worker thread of the default thread pool.

For the evaluation on OCX, only one node is used and thus 56 threads are executed on two NUMA domains. The size of the matrix is set to 2048×2048, and the numbers of tiles are thus 512. The evaluation results are shown in Figure 6. In the figure, the horizontal axis indicates the configuration of thread pools used for executing the benchmark. The tuple shown in the horizontal axis represents the numbers of threads in the critical thread pool and the default thread pool. A tuple (c, d) means that c threads are in the critical thread pool, and d threads in the default thread pool. The first column, called default, uses the single default thread pool. As shown in the figure, the use of decoupled thread pools with the default thread mapping method could decrease 33.5% of the execution time at most on OCX by properly adjusting the configuration of thread pools. The results from (28, 28) configuration show that balancing the number of threads in default and critical thread pools cannot achieve the best performance. On the other hand, the (50, 6) configuration shows the highest performance, indicating that allocating more threads to the critical thread pool can significantly improve the performance even if the number of threads of critical pool is much higher than that of the default pool. These results also suggest that the time spent for waiting the

critical tasks has a significant impact on the performance of the Cholesky benchmark.

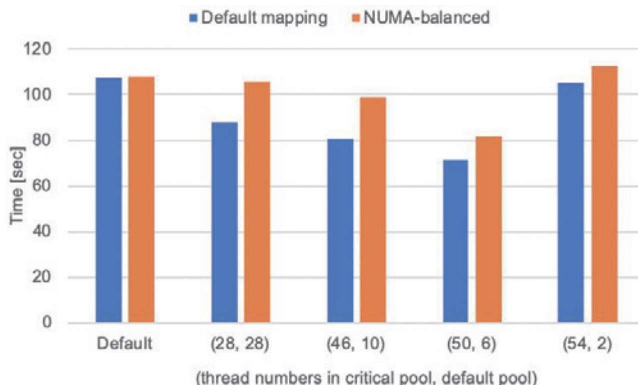

Fig. 6. Performance evaluation results of Oakbridge-CX for the Cholesky benchmark.

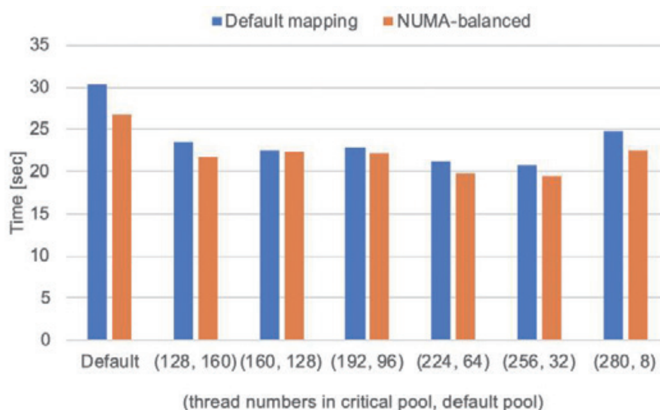

Fig. 7. Performance evaluation results of Knights Landing for the Cholesky benchmark.

On the KNL4 system, all of 288 threads are running on different logical cores. The size of the matrix is set to 512×512, and the numbers of tiles are thus 128. The evaluation results on KNL4 system are shown in Figure 7. The results indicate that the performance gain by the proposed mechanism increases as the number of worker threads in the critical pool, and the execution time is reduced by 31.8% at the thread pool configuration of (256, 32) with the default thread mapping method. These results show that the use of decoupled thread pools can significantly improve performance even if the number of threads in the critical pool is sufficiently large. The performance results on OCX and KNL4 clearly show that the proposed mechanism can achieve a higher performance than the default mechanism of HPX by giving a higher priority to critical tasks. Figure 8 shows the waiting time in the staged and pending queues with different numbers of threads allocated for critical and non-critical tasks. These results are obtained using two HPX performance counters, called the staged time and the pending

time. As shown in the figure, the configuration with 256 threads in critical thread pool can achieve the shortest pending time and also the shortest staged time. This explains why this configuration can achieve the shortest execution time in Figure 7. Compared with the default, all the configurations of the proposed mechanism can reduce the waiting time. If the number of threads in the default thread pool is lower than 32, the number of threads for the non-critical tasks is too small, the waiting time of the tasks will increase significantly. Therefore, the proposed mechanism can reduce the waiting time of tasks even if the optimal number of worker threads in the critical thread pool is not known in advance.

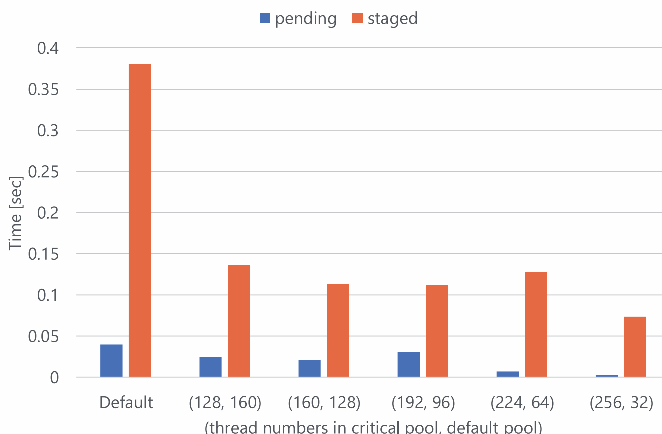

Fig. 8. The waiting time in the staged and pending queues.

Next, the effects of thread mapping policies on performance are investigated using the Cholesky benchmark program. Figures 6 and 7 also show the performance evaluation results with the two thread mapping methods. Figure 7 shows that, for the same problem, using the NUMA-balanced-mtp thread mapping can further increase the performance by 4.8% on KNL4. It is because, by distributing worker threads in different thread pools over different NUMA domains, the NUMA-balanced-mtp thread mapping method reduces the load imbalance among NUMA domains. The results also show that the proposed mechanism with the NUMA-balanced-mtp thread mapping method can further improve the performance, suggesting that the load balance among NUMA domains is important in improving performance of the proposed mechanism. However, Figure 6 shows that the NUMA-balanced-mtp thread mapping method decreases the performance on OCX, because the communication cost among NUMA domains on OCX is larger and thus cancels the performance gain by reducing the load imbalance. It is because the number of processor cores in the OCX is much lower than that in the KNL4. The impacts of reducing the load imbalance to the execution time of the Cholesky benchmark is lower than those of the communication cost. Therefore, the best thread mapping method depends on the hardware configuration, and hence the thread mapping method should be carefully selected considering the target system.

Finally, the performance gain by the proposed mechanism is also done with the merge sort benchmark on the KNL4 system. In this evaluation, the thread mapping method of NUMA-balanced-mtp is used because it can improve the performance of the proposed mechanism as discussed above. The evaluation results with the merge sort benchmark on the KNL4 system is shown in Figure 9. The merge sort benchmark uses an array of 10^5 numbers as input. As shown in the figure, all the configurations of the proposed mechanism can achieve a higher performance than the default mechanism. In this particular benchmark program, the best configuration is the (256, 32) configuration, which decreases the execution time by 47% in comparison with the default mechanism. Accordingly, the results clearly show that the proposed mechanism can improve the performance even if there is no clear critical path in the DAG. As previously analyzed, the merge tasks are more computationally-expensive than the others, and hence they should be executed earlier so that the execution of the merge tasks can be overlapped with that of other tasks. These results also show the importance of identifying the critical tasks in improving the performance of the task-based application.

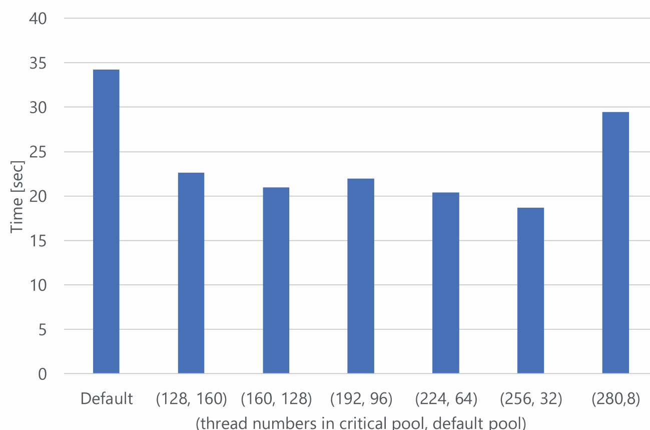

Fig. 9. Performance evaluation results of Knights Landing for the merge sort benchmark.

TABLE II RATIOS OF THE NUMBER OF EXECUTED THREADS AND THREAD TIME OF THE BENCHMARKS.

Ratios	Number of executed threads	Thread time
Cholesky decomposition	0.1	13.3
Merge sort	0.99	0.6

As an analysis of the results on the KNL4 system, Table II shows the ratios of the number of executed threads and thread times of the critical thread pool and the default thread pool. As shown in the table, the ratio of the number of executed threads of the two benchmarks are significantly different. For the Cholesky decomposition benchmark, the number of executed threads of the critical thread pool is much lower than that of the default thread pool. The ratio of the number of executed threads of Cholesky is

0.1, which means that the number of threads executed for critical tasks is only 10% of that is executed for non-critical tasks. In contrast, for the merge sort benchmark, the number of executed threads of the critical thread pool is almost equal to that of the default thread pool. As previously analyzed from the DAG of the merge sort benchmark, the numbers of split and merge tasks in each path are equal. The results of the number of executed threads show that these two benchmarks have different workloads, although the best number of threads configurations of the benchmarks are the same. Moreover, the ratio of thread times of the merge sort benchmark shows that the decoupled thread pools mechanism can significantly decrease the average execution time of critical tasks. The ratio of thread times of the merge sort is 0.6, which means that the average execution time of critical tasks is 60% shorter than that of the non-critical tasks. The results shown in Table II suggest that the impacts of the decoupled thread pools mechanism depend on the application workload and the hardware configuration of the system.

5. CONCLUSIONS

This paper 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. The performance evaluation results clearly demonstrate that the proposed mechanism can reduce the staged time and the pending time, in which tasks are waiting for other tasks. As a result, the proposed mechanism can reduce the total execution time of the Cholesky benchmark program by approximately 33.5% and 36.1% at maximum on OCX and KNL4, respectively. In addition, the NUMA-balanced-mtp thread mapping method can further improve the KNL4 performance, even though it degrades the OCX performance. Accordingly, the proposed mechanism can improve the performance of task-based HPX applications by prioritizing critical tasks with a low runtime overhead especially if its parameters, such as the number of threads in each thread pool and the thread mapping method, are appropriately adjusted for the target system.

As mentioned above, several parameters of the proposed method are empirically adjusted by hand for the application and target system in advance. Since the performance is sensitive to the parameter values, auto-tuning of them will be an interesting research topic. Automatic detection of critical tasks in a DAG would also be important in practical use. In addition, we can expect that the proposed mechanism can improve the runtime systems for other task-based PGAS languages, such as XcalableMP [13]. These topics will be discussed in our future work.

ACKNOWLEDGMENTS

This work was propelled by Suhang Jiang, Mulya Agung, Ryusuke Egawa, Hiroyuki Takizawa.

The authors would like to thank Prof. Hiroaki Kobayashi of Tohoku University and Dr. Kentaro Sano of RIKEN R-CCS for fruitful discussions on this work.

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.

REFERENCES

[1] T. E. Hart, P. Mckenney, A. K. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," Journal of Parallel and Distributed Computing, vol. 67, no. 12, pp. 1270-1285, 2007.

[2] 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.

[3] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: an advanced parallel execution model for scaling-impaired applications," in International Conference on Parallel Processing Workshops, 2009, pp. 394-401.

[4] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham et al., "Productivity and performance using partitioned global address space languages," in The 2007 International Workshop on Parallel Symbolic Computation, 2007, pp. 24-32.

[5] OpenMP Architecture Review Board, "OpenMP application programming interface, version 4.5," 2015. [Online]. Available: https://www.openmp.org/specifications/

[6] J. Loaiza, S. Chandrasekaran, and N. MacNaughton, "Using local locks for global synchronization in multi-node systems," 2008, uS Patent 7,376,744.

[7] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Qu ́ema, R. Lachaize, and M. Roth, "Challenges of memory management on modern numa systems," Communications of the ACM, vol. 58, no. 12, pp. 59-66, 2015.

[8] M. Agung, M. A. Amrizal, R. Egawa, and H. Takizawa, "Deloc: A locality and memory-congestion-aware task mapping method for modern numa systems," IEEE Access, vol. 8, pp. 6937-6953, 2020.

[9] K. Nakajima, "Parallel multigrid method on multicore/manycore clusters," in Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops, ser. HPCAsia2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 5-9. [Online]. Available: https://doi.org/10.1145/ 3373271.3373273

[10] J. Jeffers, J. Reinders, and A. Sodani, Intel Xeon Phi Processor High Performance

Programming: Knights Landing Edition. Morgan Kaufmann, 2016.
[11] J. Dorris, J. Kurzak, P. Luszczek, A. YarKhan, and J. Dongarra, “Taskbased Cholesky decomposition on knights corner using OpenMP,” in International Conference on High Performance Computing, 2016, pp.
544-562.
[12] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: past, present and future,” Concurrency and Computation: practice and experience, vol. 15, no. 9, pp. 803-820, 2003.
[13] K. Tsugane, J. Lee, H. Murai, and M. Sato, “Multi-tasking execution in PGAS language XcalableMP and communication optimization on many-core clusters,” in International Conference on High Performance Computing in Asia-Pacific Region, 2018, pp. 75-85.