Preliminary Evaluation towards Task Priority Control in HPX

Suhang Jiang
GSIS
Tohoku University
Sendai, Miyagi, Japan
jiang.suhang.s7@dc.tohoku.ac.jp

Mulya Agung
GSIS
Tohoku University
Sendai, Miyagi, Japan
agung@dc.tohoku.ac.jp

Ryusuke Egawa
Cyberscience Center
Tohoku University
Sendai, Miyagi, Japan
egawa@tohoku.ac.jp

Hiroyuki Takizawa
Cyberscience Center
Tohoku University
Sendai, Miyagi, Japan
takizawa@tohoku.ac.jp

1 Introduction

Recently, task-based execution has been attracting attention because it does not perform expensive synchronization as long as the dependencies among tasks are not violated. High Performance ParalleX (HPX)[1] is one 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.

HPX employs hybrid threading implementations[2], in which *M* logical threads (HPX threads) are assigned to *N* worker threads. By using a task queue, tasks are first assigned to HPX threads without any help of the operating system. Whenever a worker thread becomes idle, an HPX thread is retrieved from the thread pool, and assigned to the worker thread.

In the original runtime implementation, all tasks are managed using a single default task queue, and assigned to HPX threads in a thread pool in a first-in-first-out fashion since every task basically has the same priority of being executed. However, this could be inefficient, even though some important tasks should start execution earlier than others.

2 Use of multiple task queues

In the original HPX runtime system, tasks are assigned to HPX threads in a thread pool, which are initially not bound to any cores yet. With a task queue, task assignment is done in a first-in-first-out fashion. Then, the HPX threads are retrieved from the thread pool, and executed by worker threads running on cores. One difficulty in this task assignment is that, if the execution of a task on the critical path is delayed by executing other threads, the total execution time increases.

However, if the result of a preceding task is required by other subsequent tasks, the subsequent tasks are blocked until the result of the preceding task becomes available. Suppose that a task on the critical path, called a *critical task*, is blocked. Then, if some other tasks are also ready for execution, they might be executed earlier than the critical task, resulting in increasing the critical path. Therefore, we need to give a higher priority to critical tasks so as to prevent prolonging the critical path.

To this end, we propose to decouple the default task queue by using two task queues, one is for critical tasks, and the other is for the other tasks. Since only the task at the head of a task queue is assigned to the HPX thread pool, we can give a higher priority for the critical tasks, and they can be assigned earlier if they are pushed into a dedicated queue.

In addition, a thread mapping method, called NUMA-balanced, has been applied to map the worker threads to processor cores on the NUMA system. We apply thread mapping when the application is launched. Our results show that the thread mapping method can further improve the performance results of our task queue decoupling method because it reduces the load imbalance among the NUMA nodes.

3 Evaluation and Discussions

The experimental evaluation has been conducted on a NUMA system based on Intel Xeon Phi Knights Landing (KNL) processors. The benchmark used for the comparison is blocked Cholesky factorization[3][4], and its Cholesky decomposition task is considered as a critical task for "look-ahead" execution.

Compared with the implementation of using the default task queue in Figure 1, the proposed implementation can increase the performance by 31.76% in terms of execution time. In addition, the thread mapping can further increase the performance by 4.8%.

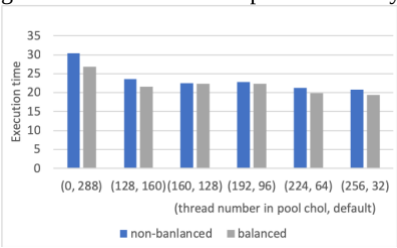

Figure 1: Execution time of using different number of threads in decoupled thread pools, comparing with using NUMA-balanced to map the threads based on the former.

4 Conclusions and future work

In this work, we have proposed a method to use decoupled task queues, and we have also evaluated the impacts of thread mapping on the performance results of our method. Our preliminary results show that assigning tasks to different task queues can improve the performance by giving a higher execution priority to critical tasks, and thread mapping can reduce the execution time of the tasks.

In the future, we will automatically identify critical tasks and assign them to a dedicated queue.

REFERENCES

[1] Kaiser, Hartmut, et al. "HPX: A task based programming model in a global address space." Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. ACM, 2014.

[2] Grubel, Patricia, et al. "The performance implication of task size for applications on the HPX runtime system." 2015 IEEE International Conference on Cluster Computing. IEEE, 2015.

[3] Cayrols, Sébastien, Iain Duff, and Florent Lopez. "Parallelization of the solve phase in a task-based Cholesky solver using a sequential task flow model." NLAFET Working Note (2018).

[4] Dorris, Joseph, et al. "Task-based Cholesky decomposition on knights corner using OpenMP." International Conference on High Performance Computing. Springer, Cham, 2016.