Optimistic Threading Techniques for MPI+ULT

Shintaro Iwasaki
The University of Tokyo
Tokyo, Japan
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Abdelhalim Amer
Argonne National Laboratory
Lemont, Illinois
aamer@anl.gov

Kenjiro Taura
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

Pavan Balaji
Argonne National Laboratory
Lemont, Illinois
balaji@anl.gov

ABSTRACT

As the number of nodes and cores is increasing, MPI+Threads becomes a popular parallel programming model in distributed environments. OS-level threads have been used as "Threads", but it requires coarser-grained parallelism because of its large threading overheads. To exploit fine-grained parallelism of irregular applications, user-level threads (ULTs) are lightweight and thus regarded as promising threading techniques, which efficiently overlaps computation and communication. Since MPI functions might internally obtain locks to avoid conflicts between multiple threads, all threads calling MPI functions potentially suspend (i.e., yield) for overlap. Though the occurrence of conflicts depends on applications and is designed to be less frequent in general, most of the previous techniques impose the cost of the suspension feature on creation. Our work investigates dynamic threading techniques which adaptively employ the suspension feature. Because the cost of suspension is lazily paid, dynamic threading techniques achieve lower overheads when the suspension rate is low. Our preliminary evaluation of Graph500 shows the dynamic threading technique proposed by Eager and Jahorjan [1] successfully improved performance on a single node.

1 INTRODUCTION

Nowadays, exploitation of node-level and thread-level parallelism is essential to speed up programs on large-scale clusters and supercomputers. MPI is considered as a de facto standard programming model to parallelize applications in a distributed environment. Though inter-node parallelism has been exploited by MPI processes as well as inter-node parallelism in the past, a hybrid MPI+Threads programming model has become popular to utilize shared memory in each node. OS-level threads (e.g., pthreads) have been widely used as underlying threads, but they require coarse-grained parallelism to alleviate higher threading overheads. On the other hand, User-level threads (ULTs) significantly reduces the overheads since all operations are performed in user-space, and therefore have been adopted by a number of parallel systems and libraries. The combination of MPI and ULT is considered to be a promising programming model to exploit both inter-node and intra-node parallelism [2].
The key to efficiently overlapping communication and computation is suspension. When MPI functions internally get locks to avoid conflicts, threads calling those functions should suspend and switch to other threads immediately to utilize cores. Thus, all threads which might call MPI functions should be created as suspendable. However, the internal locks in MPI runtime systems are usually designed to be rarely taken, so few ULTs suspend in most cases. Most previous threading techniques fix threading features when threads are created, so the cost to employ the suspension feature, which might be required in MPI functions, must be paid even if threads are known in advance to infrequently block.
In this work, we evaluate a dynamic threading technique which is expected to reduce the thread creation and synchronization costs when threads rarely suspend. We focus on a dynamic threading technique proposed by Eager and Jahorjan [1]. Threads are adaptively set up if necessary by paying extra overheads and adding constraints. We characterize performance of threading methods by *the suspension rate*, which is the probability that ULTs suspend (i.e. fail to take a lock and switch to another thread) at runtime. The dynamic threading technique is placed in the midst of the two opposite threading techniques which have been intensively studied previously; one is a fully-fledged thread employing threading features on creation, and the other is a run-to-completion thread which has least overheads but cannot suspend. We note all of these techniques are for building general threading libraries, so neither compiler modifications, special compiler extensions, kernel modification nor source-to-source translations are required. We implemented them by extending Argobots [3] and evaluated the performance of these techniques on Intel Xeon Phi. Our preliminary experiment shows dynamic threading techniques successfully improved single-node performance of Graph500 parallalized by an MPI+ULT model.

2 THREADING TECHNIQUES

The essence of threads is how to save and restore the context of computation. Ultimately, the context is represented as state of registers and function stack; the context can be saved by taking a snapshot of registers and stack and be resumed by restoring them. We first explain the two popular methods which are adopted by a majority of previous threading libraries.

2.1 Fully-fledged thread (Full)

The first is a fully-fledged thread (**Full**), which has an independent stack and fully manages the context of functions. Registers are saved and restored by at least two user-level context switches; one is to invoke a thread, and the other is to finish that thread. Because

EuroMPI '17, Sept. 2017, Lemont, IL, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

it saves the context of a caller, a caller can be suspended and resume a caller if necessary.

2.2 Run-to-completion thread (RtC)

The latter is called a run-to-completion thread (**RtC**), which is invoked by a normal function call. Since the context is not saved, a caller and its callee are tightly coupled and cannot be suspended and resumed independently. **Full** can suspend but has a larger overhead, while **RtC** is lightweight but unsuspendable.

2.3 Scheduler activation (SA)

Both of these previous techniques determine a set of available features on creation. Scheduler activation, which was adopted by Chores [1], can adaptively obtain a suspension feature. When a caller is reentrant and its context is not necessary to be saved, we need to remove most operations to manage contexts. A caller can be restarted by running it from scratch on a newly allocated stack if it is reentrant. It is the case where a caller is a scheduling function since a scheduler (typically a random-work-stealing scheduler) does not have a state.

Table 1: Summary of threading techniques

	No Yield			Yield		
	Change stack?	Context switches?	Cost	Rerun Sched.?	Cost	Constraints
Full	Yes	Yes	High	No	Low	No
SA	No	No	∨	Yes	High	*
RtC	No	No	Low	-	-	**

* Scheduler must be reentrant. In addition, scheduler and ULT share size of stack.
** Suspension is not allowed.

Table 1 summarizes the overheads and constraints of these techniques.

3 PRELIMINARY RESULTS

We implemented all methods in Argobots [3], which adopts a parent-first scheduling policy. We used an Intel Xeon Phi 7210 (Knights Landing) processor which has 64 cores / 256 hardware threads running at 1.3GHz (fixed). We compiled all the programs with Intel Compiler 17.2.174.

3.1 Basic threading overheads

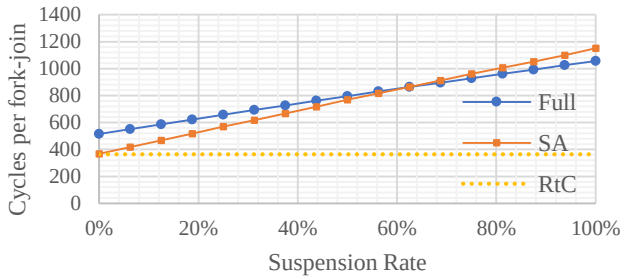

Figure 1: Cycles per fork-join

Figure 1 shows the overhead of a fork-join when repeating creating 256 threads and joining them on a single core. We change the number of threads which suspend to change the suspension rate. Because **RtC** cannot suspend, we show the $n = 0$ result as a baseline. It shows that the threading overhead of **SA** is smaller than that of **Full** when the suspension rate is low due to its dynamic behavior, while the additional cost is paid for lazy promotion if threads suspend in comparison with **Full**, while it can still suspend.

3.2 Graph500

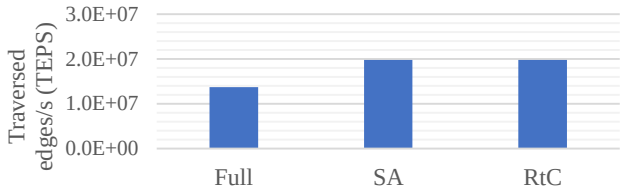

Figure 2: TEPS on a single node running 64 workers

Figure 2 presents a preliminary result of Graph500 using a single node which runs 64 workers. Since MPI functions are never called, the suspension rate is always zero. As shown in Figure 2, **SA** achieved better performance than **Full** due to its lower threading overhead. We note **RtC** cannot call MPI functions and thus is only applicable on a single node.

4 CONCLUSIONS

To exploit fine-grained parallelism, lightweight threading techniques which especially have lower costs when the suspension rate is low are promosing from the viewpoint of MPI+ULT. Scheduler activation we focused on enables threads to become suspendable dynamically, which reduces threading overheads in optimistic cases. However, as presented in Table 1, Scheduler Activation has several constraints, which narrows the applicability. For example, Scheduler activation is not straightforwardly applicable when the target applications have several types of tasks which require diffrent stack size. Developping other dynamic threading techniques which might add overheads but alleviates constraints is our future work.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

[1] D. L. Eager and J. Jahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, Feb. 1993.
[2] H. Lu, S. Seo, and P. Balaji. MPI+ULT: Overlapping communication and computation with user-level threads. In *Proceedings of the 17th International Conference on High Performance Computing and Communications*, HPCC '15, pages 444–454, Aug. 2015.
[3] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castello, D. Genet, T. Herault, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, and P. Beckman. Argobots: A lightweight, low-level threading and tasking framework. Argonne National Labolatory, 2016.