Scalable Load-Balancing Concurrent Queues in Modern Many-Core Architectures

Caleb Lehman
lehman.346@osu.edu
Ohio State University
Columbus, Ohio

Poornima Nookala (Advisor)
pnookala@hawk.iit.edu
Illinois Institute of Technology
Chicago, Illinois

Ioan Raicu (Advisor)
iraicu@cs.iit.edu
Illinois Institute of Technology
Chicago, Illinois

ABSTRACT

As the core counts of computing platforms continue to rise, parallel runtime systems with support for very fine-grained tasks become increasingly necessary to fully utilize the available resources. A critical feature of such task-based parallel runtime systems is the ability to balance work evenly and quickly across available cores. We highlight this by studying XTask, a custom parallel runtime system based on XQueue, which is a novel lock-less concurrent queuing system with relaxed ordering semantics that is geared to realizing scalability to hundreds of concurrent threads. We demonstrate the lack of adequate load balancing in the original XQueue design and present several solutions for improving load balancing. We also evaluate the corresponding improvements in performance on two sample workloads, computation of Fibonacci numbers and computation of Cholesky factorization. Finally, we compare the performance of several versions of XTask along with several implementations of the popular OpenMP runtime system.

KEYWORDS

concurrent data structures; fine-grained parallelism; lock-free; lock-less; queues; nonblocking; parallel runtime; tasks

1 INTRODUCTION

Modern computer systems are composed of many smaller computing devices. Efficient use of such systems is therefore contingent on a program's ability to use parallelism across many computing devices. Under task-based parallelism, the multiple cores in a shared-memory system are utilized by decomposing a computation into many shorter computations called tasks, which can be distributed to different cores, run in parallel, and recombined to produce the final result. Typically, a set of worker threads is spawned across the available cores. Workers retrieve and insert tasks into a concurrent queuing system, executing until the full computation is completed. Implementations of the task-based parallelism model also typically offer support for inter-task dependencies. Dependencies between tasks impose restrictions on the order in which tasks must be executed, yielding a partial order on the tasks, often encoded as a directed acyclic graph.

SC'19, November 2019, Denver, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 MOTIVATION

Fine-grained parallelism: This term refers to task-based decomposition of a job, in which the tasks are relatively short. Fine-grained tasks are desirable because they allow for a more balanced distribution of the workload across workers. However, workloads composed of many fine-grained tasks typically require more detailed task management and therefore place more emphasis on overheads present in the runtime system. This project is motivated by observations of poor performance of the GNU OpenMP implementation on fine-grained workloads. The overarching goal is to obtain improved performance on extremely fine-grained workloads by reducing overheads in the concurrent queuing system, compared to standard work stealing-based approaches. Prior work by the advisors explored lock-less, highly efficient, and scalable data structures (XQueue) that can increase the level of parallelism on modern architectures thereby improving performance.

Implicit Parallelism: Swift/T[3] is a parallel programming language designed for extreme scale, decentralized execution on a single, very large (exascale) system. Swift/T is implicitly parallel and the compiler can generate extreme parallelism based on the program and data flow. This idea is powerful, however since Swift/T uses MPI for intranode parallelism, the performance on single node is significantly degraded. A task-based runtime Xtask is under development by the Advisors and is aimed at enabling extreme fine-grained task execution on modern many core architectures while exploring the possibility of implicit parallelism using the techniques from Swift/T compiler. This work uses a preliminary version of XTask for evaluation purposes.

3 LOAD BALANCING IN XQUEUE

Original XQueue design implements an MPMC interface using one master and one auxiliary queue and communication between them is modelled after a dynamic task graph execution. However, this is not ideal for distributing tasks evenly across workers.

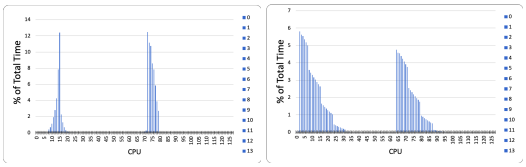


Figure 1: These figures represent the number of cycles each CPU spent executing tasks for the Fibonacci workload (left) and Cholesky workload (right).

This work focuses on studying and improving load balancing in XQueue. XQueue provides lower enqueue and dequeue latencies than standard approaches to concurrent queueing systems, which require locking or atomic operations, by utilizing multiple, lock-less, single-producer single-consumer queues. We extend the XQueue design by creating different topologies on the connections between cores in the XQueue system. We then test our versions of the XQueue system on several architectures for two sample workloads, a workload for the computation of Fibonacci numbers and a workload for the computation of Cholesky factorizations, both of which can be tuned to generate millions of fine-grained tasks.

3.1 Workstealing

Work stealing[2] is a common approach to achieve better load distribution between workers on various cores in which idle workers ask other workers for tasks. Original XQueue uses multiple lock-less SPSC queues and we modified it to use atomic primitives for supporting work stealing.

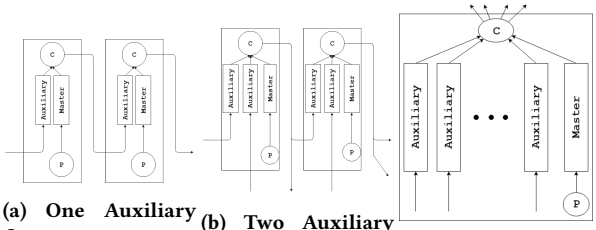
Let c_i denote the number of cycles the ith CPU spent executing tasks. Define *load balance error* by

$$\lambda := \frac{\max\{c_i\}}{\sum c_i / n} - 1$$

Our measurements yielded the following results:

	λ	
Design	**Fib.**	**Chol.**
XQueue	14.96	4.85
XQueue v2	3.99	7.94
XQueue vN	0.12	0.03
Work Stealing	0.10	0.01

3.2 XQueue Design Variations



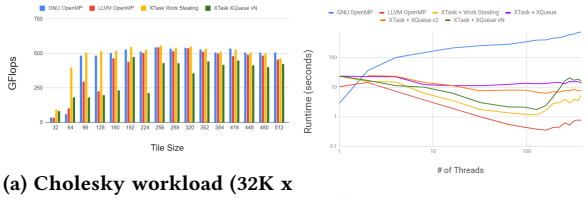
(a) One Auxiliary Queue **(b) Two Auxiliary Queues** **(c) N Auxiliary Queues**

Figure 2: XQueue Designs

The approach of sending tasks to a neighbor cannot be broadly applied to various applications since the task distribution entirely depends on the structure of the DAG. While work stealing is one of the solutions to achieve better load balancing, we also explored load balancing using multiple auxiliary queues. Figure 2 shows the designs we explored as part of this work.

4 EVALUATION

We found that our changes to the XQueue system led to significantly more optimal load balancing. Our final design achieved load balancing comparable to the load balancing achieved with a work stealing-based approach, a load balancing technique that is widely utilized in modern task-based runtime systems (see [2] or [3], for example). Additionally, our new XQueue design achieves speed ups of up to 6-10x compared to the original XQueue design when using 128 threads, while still remaining entirely lock-less. Our new XQueue design also demonstrates some scalability, showing increasingly good performance on the Fibonacci workload up to 128 threads. Finally, we show that our new XQueue design is still significantly outperformed by OpenMP[1] on the Fibonacci workload and a work stealing-based approach on the Fibonacci and Cholesky workloads, demonstrating a need for further investigation into aspects of the system other than load balancing in order to achieve good performance on a wide variety of applications.



(a) Cholesky workload (32K x 32K matrix) **(b) Runtime of `Fib(35)`**

Figure 3: These figures show the results obtained using various XQueue topologies and work stealing strategies

5 CONCLUSION AND FUTURE WORK

This paper presents several load balancing strategies for XQueue, which is an extremely scalable lock-less concurrent multiple producer multiple consumer out of order queue. Evaluation results show that XQueue is scalable up to hundreds of threads of execution. XTask is a work-in-progress runtime system which can achieve low latency and high throughput at extreme scale and achieve peak performance on modern multi-core architectures. Work stealing significantly improved the task distribution over the simplistic load balancing in XQueue. As part of the future work, we plan on integrating XQueue and/or XTask with the parallel programming systems Swift/T and Parsl [4] to exploit implicit parallelism capabilities, and OpenMP to accelerate many more applications with fine-grained parallelism transparently.

ACKNOWLEDGMENTS

This work is supported by the NSF CCF-1757964/1757970 REU award (BigDataX), and the NSF CNS-1730689 CRI award (Mystic).

REFERENCES

[1] L. Dagum and R. Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming, Vol. 5. IEEE Computational Science & Engineering.
[2] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12. 1094 – 1104. Issue 10.
[3] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing L. Lusk, and Ian T. Foster. 2013. Swift/T: scalable data flow programming for many-task applications. In *PPOPP*.
[4] B. Clifford Z. Li D. S. Katz R. Chard R. Kumar L. Lacinski J. Wozniak I. Foster M. Wilde K. Chard Y. Babuji, A. Woodard. 2019. Parsl: Pervasive Parallel Programming in Python. 28th International Symposium on High-Performance Parallel and Distributed Computing, New York, NY, USA.