High-productivity Abstractions and Efficient Runtime for Dynamically Load-balanced Distributed Programs on Multi/Many-core Clusters

Finnerty, Patrick Martin

(Degree)
博士（工学）

(Date of Degree)
2022-09-25

(Date of Publication)
2024-09-25

(Resource Type)
doctoral thesis

(Report Number)
甲第8467号

(URL)
https://hdl.handle.net/20.500.14094/0100477893

（別紙様式 3）

論文内容の要旨

氏　　名　FINNERTY Patrick Martin

専　　攻　情報学専攻

論文題目（外国語の場合は，その和訳を併記すること。）

High-productivity Abstractions and Efficient Runtime for Dynamically Load-balanced Distributed Programs on Multi/Many-core Clusters
マルチコア／メニーコアクラスタにおける動的負荷分散プログラムのための高生産性抽象化と効率的ランタイム実装法

指導教員　太田　能　教授

Chapter 1: Introduction

Modern supercomputers rely on clusters of many-core processors, bringing large amount of parallelism both within a node and across nodes. These architectures revive interest in programming models and languages dedicated to supporting these new environments.

The Asynchronous Partitioned Global Address Space (APGAS) as implemented in the X10 language brings nice abstractions that allow programmers to handle the distributed nature of their program. However, this model is not perfect. Most notably, it does not support communication between processes. Neither does it inherently solve the issue of balancing the computational load across nodes.

High-level abstractions to dynamically balance the computation are therefore desired to allow non-expert programmers to

Chapter 2: Background

In the second chapter, we will recall useful background, covering existing parallel and distributed computing programming models and languages.

We will cover the existing state of the X10-style implementation of the Asynchronous Partitioned Global Address Space (APGAS) programming model. The semantics of the finish/async constructs will be demonstrated.

Chapter 3: Task Granularity Tuning for the lifeline-based multi-worker GLB

This scheme was first proposed in X10 but **suffered from implementation issues**. In the Java version discussed in this chapter, these issues were resolved by introducing a **yielding mechanism in the workers' main routine**. The mechanism goes against the "work first" principle generally observed in pool threads and work-stealing scheme. However, it is necessary in this case to ensure that worker threads do not monopolize the resources and that steal requests coming from other nodes can be scheduled successfully.

The concept of **granularity** appears in various contexts. Among the parameters of the hybrid load-balancing scheme discussed in this chapter, this specific setting is particularly **important to achieve good performance**. This parameter determines the

number of micro tasks that worker threads process before checking on the works-stealing runtime. A value too low causes overhead through excessive runtime checking, while a value too high will cause starvation to appear. Finding a suitable compromise is necessary to achieve good performance.

The problem with this setting is that it can be quite arbitrary. On the one hand, the dynamic load balancer cannot choose a value which will work in any situation against any computation. On the other hand, users of dynamic load balancers cannot be expected to know in advance what a good value should be. In practice, users may go through a wasteful trial-and-error process to find a value which gives appropriate performance for their particular workload.

In this chapter, we will introduce **a tuning mechanism capable of dynamically adjusting the task granularity in the context of the hybrid lifeline-based global load balancer**. This mechanism monitors the most recent situation on each host and uses two criteria to determine if either excessive overhead or starvation is occurring. It then automatically adjusts the granularity as necessary, without any user input.

While observing starvation is relatively straightforward, observing overhead is more difficult. We present a criterion capable of determining if excessive runtime checking is occurring by exploiting a data race in the load-balancing scheme.

To evaluate the performance of this mechanism, we challenge it against 4 backtrack-search algorithms implemented using this load balancer. The tuning mechanism built using these criteria is capable of handling the variety of exploration searches, starting from an unfavorable setting, the granularity is adjusted to a satisfactory level within a few seconds. The tuning mechanism we develop is also **robust against variations in the problem implementation and the hardware used** to perform the computation. Moreover, this tuning mechanism **does not generate any noticeable overhead** compared to executions without the mechanism.

Chapter 4: Relocatable Distributed Collections

Modern clusters and supercomputers based on many-core processors make available to computer scientist a high level of parallelism both between nodes and within nodes. However, the complexity of such distributed systems requires dedicated programming languages for programmers to successfully harness these architectures.

The elegant abstractions of the APGAS model as implemented in X10 and Java allows for rather simple management of termination detection. However, **the lack of support for collective computation/communication across processes** is a handicap for distributed

object-oriented programming.

To fill-in this gap, we introduce “relocatable distributed collections” which complements the APGAS for Java library. Under the model we propose, a **distributed collection is defined on a group of processes**, meaning it can contain records on the set of processes it is defined on. We mimic the usual collections from the standard Java library that programmers are already familiar with. The key innovation of our library lies in the **relocatable nature of the entries recorded into our distributed collections**, which can be relocated from a process to another using high-level abstractions provided by our library.

To handle the distributed nature of the computation**, we introduce of notion of teamed methods**. Unlike regular methods, the teamed methods need to be called on each process on which collections are defined. They create synchronization points between asynchronous activities running on different processes. Common computation patterns, such as reductions, and other features of our library are implemented using this concept.

Using these features, **non-experienced programmers are capable of writing sophisticated distributed programs** that would otherwise remain out of reach, or too complex to apprehend. The most prominent example of this is the PlhamJ financial market simulator which uses the features of our library extensively. **The features and abstractions supported by our library made it possible to introduce dynamic load balancing** into this simulator. In cases where the performance of the hosts differs, it is now possible **to dynamically adjust the distribution of data objects to match the actual performance available** in the cluster.

Chapter 5: Integrated Global Load Balancer

While the facilities presented in the previous chapter allow programmers to explicitly manage the distribution of entries across processes, **implementing a load-balancing strategy for each application still comes with non-negligible effort**. It would be preferable if such entry relocation for the purpose of **load-balancing could instead be left up to the library**. However, automatic load balancing facilities imply that the distribution of entries is surrendered to the library, when up until now control over the distribution of entries was entirely left up to the programmer.

To resolve this dichotomy, **we introduce a specific context within which the integrated load balancer operates**. Within this context, the computation to perform on the entries contained in the underlying distributed collection are expressed using a staging /

submission system. Outside of this specific context, the control over the distributed collections is left entirely up to the programmer. This design choice has the advantage of providing clear boundaries for the operation of the integrated load balancer and contribute to the greater programmability of our programming model.

Internally, **we re-visit the lifeline-based global load balancer** to implement these load-balancing facilities. There are however a number of key differences between the original scheme intent and the implementation used in our integrated load balancer.

First, contrary to the original scheme where computation tasks are self-contained, the source of the computation is external to the tasks in our scheme. As a consequence, **entries of the distributed collection are relocated along with the computation tasks** when inter-host load balancing procedures occur.

Secondly, the fact that **multiple computation operating on the same collection** is permissible under our scheme **challenges the typical "single englobing finish" used in the original scheme**. Under the canonical APGAS finish/async programming model, asynchronous activities are only allowed to spawn new activities which belong to the same finish. To guarantee the correctness of our scheme, we removed this constraint and allowed asynchronous activities to participate into multiple finish constructs.

Finally, **the notion of "lifelines" takes up a new meaning** in the context of our distributed collections.

We show that our current implementation is capable of dynamically balancing the load to some degree, but that further work is needed to reduce the redundant work relocation in cases where no load imbalance actually occurs.

Chapter 6: Conclusion

The last chapter will summarize the main findings of the previous chapter and open on new avenues for research. In particular, the concept of relocatable distributed collections have the **potential to provide appropriate abstractions for elastic programs** in which processes can be dynamically dropped or added to the computation to match the evolving parallelism needs of the program.