Scalable and Highly Available Fault Resilient Programming Middleware for Exascale Computing

Atsuko Takefusa, Tsutomu Ikegami, Hidemoto Nakada,
Ryousei Takano, Takayuki Tozawa and Yoshio Tanaka
National Institute of Advanced Industrial Science and Technology (AIST)
Email: {atsuko.takefusa, t-ikegami, hide-nakada, takano-ryousei, t.tozawa, yoshio.tanaka}@aist.go.jp

I. Introduction

A hierarchical master-worker model is believed to be a promising programming paradigm that can achieve weak scaling on exascale-level high performance computers [1]. However, "fault resiliency" is one of the most important issues for exascale computing because the Mean Time Between Failure (MTBF) of such computers will be short [2]. We propose a fault resilient programming middleware called Falanx [3] for exascale computing that allows each application programmer to easily code an MPI-based fault resilient application with a hierarchical master-worker model. The Falanx middleware consists of a data store (DS) and a resource management system (RMS) in order to continue with an execution flow: The DS preserves data required for each application, and prevents data loss due to failures. The RMS allocates processes of each task, including data parallelism, to computing nodes avoiding nodes with failures. It is necessary that these components must be scalable and that they themselves have to be implemented in a fault resilient manner in exascale computing environments.

We design a scalable and highly available middleware, which consists of DS and RMS, and implement them by using Kyoto Cabinet [4] and Apache ZooKeeper [5]. Then, we investigate the basic performance from the preliminary experiments and confirm the feasibility from experiments using an actual chemical application, OpenFMO.

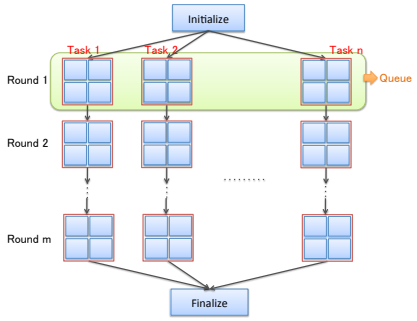

Fig. 1. A program logic image of the target applications.

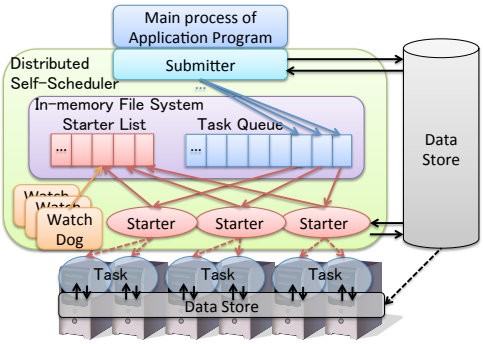

Fig. 2. Architecture overview of the proposed distributed self-scheduler.

II. The Falanx middleware

A. Overview of Falanx

Falanx employs a hierarchical master-worker programming model as shown in Fig. 1 for the scalability to exascale computing environments and allows users to develop exascale applications, by providing a fault resilient runtime environment as a middleware. Falanx provides an API, which enables to code application logic in the master process as workflow and specify whether a failed task is restarted or destroyed when failure has happened. Falanx has been developing by using ULFM-MPI [6] and a Falanx-based program is a single MPI job written in C. Fig. 1 shows an overview of a program logic image of the target applications of the proposed middleware.

B. Data Store (DS)

Falanx DS provides a high-performance key-value storage capability. It is an on-memory distributed storage, based on key hash indexing; i.e. the location of the data is determined the hash value of the key. It also replicates data for high availability. The replica will be placed right next node to the original one. We provide two types of write acks. One is called "full", which guarantees that the original and the replica are written to the Disk. The other is called "prim", which only guarantees that the original data was written. It allows programmers to choose one of them.

We implemented each DS node using Kyoto Cabinet, which is a open source KVS library that could be directly linked to the client process. While Kyoto Cabinet is not a on-memory DB originally, we disabled persistent write back so that it could be utilized as a on-memory storage.

C. Resource Management System (RMS)

We propose a scalable and highly available distributed self-scheduler, which provides the following functionalities in order to achieve scalability, fault resiliency, and persistency of both the target applications and the scheduler itself:

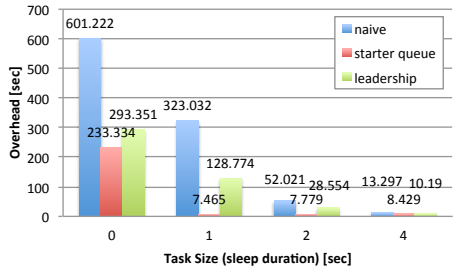

Fig. 3. The overheads of RMS using three ZooKeeper processes.

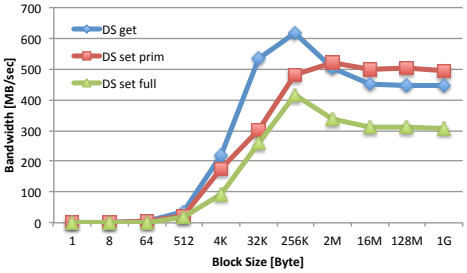

Fig. 4. Bandwidths of DS get and set prim and full processes.

1) Task management by multiple processes.
   a. Task submission to a task queue.
   b. Execution of a task in the task queue.
   c. Re-execution or deletion of a failed task.
2) Health monitoring of computing nodes and the networks between them.
3) Persistent and scalable management of resource management information.

In order to achieve these functionalities, we designed the distributed self-scheduler as shown in Fig. 2. The distributed self-scheduler consists of Submitter, Starter, WatchDog, and In-memory file system (In-memory FS) modules. Submitter has the functionalities of 1a., task submission and 1c., task re-execution or deletion. Starter is responsible for 1b., task execution. WatchDog is responsible for 2., health monitoring. In-memory FS achieves 3., persistent and scalable management of resource information such as a task queue, task statuses and Starter information.

We have implemented RMS by using Apache ZooKeeper, which provides an in-memory file system over multiple nodes and an "watch" mechanism in order to reduce the number of polling processes between nodes. We have also implemented three types of a task queue on ZooKeeper, Naive, Starter Queue and Leadership. Naive allows each starter to autonomically access the queue, and will easily cause contentions. Starter Queue and Leadership aim to reduce contentions. The former implements a simple FIFO queue and the latter provides a FIFO queue, which takes into account the priority.

III. Experiments

A. Preliminary experiments

We investigate the basic performance of RMS and DS. Fig. 3 shows the overheads of RMS using three ZooKeeper processes. We employed sixty Starters and ten thousand tasks, which process zero to four second sleep. The results show that Starter Queue and Leadership is better than Naive and the overheads are negligible when task size is 4 seconds.

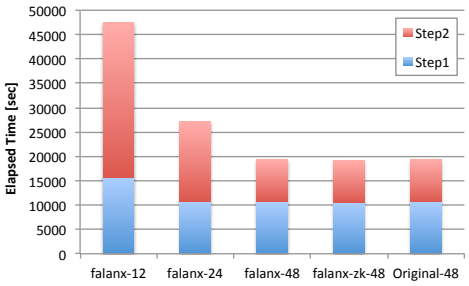

Fig. 5. Experimental results of OpenFMO.

Fig. 4 indicates bandwidths of DS get and set prim and full processes using 40G InfiniBand. The results show that prim is better than full in terms of bandwidth.

B. Performance of OpenFMO

We also investigate the performance of an OpenFMO application implemented by using Falanx. OpenFMO is an open-source software platform for Fragment Molecular Orbital (FMO) method. FMO is focused on ab initio electronic state calculations of macro molecules. In the application, a large molecule is divided into fragments, and electronic calculation for each fragment is processed on each worker. And then, the results are accumulated to check convergence. Fig. 5 shows the results of strong scaling measurement on a lysozyme molecule w/o water. The horizontal axis indicates implementations and the number of cores. The results show that the performance is suffered from a severe load-imbalance in Step 1, while that of Step 2 show a good scalability. We will implement a facility to rearrange the worker configuration in future work.

IV. Conclusion

We have designed and implemented Falanx, which makes it easier to develop exascale applications, by providing a fault resilient runtime environment as a middleware. The preliminary experiments showed fault resiliency and less overheads of Falanx. We will investigate the scalability and fault resiliency of various applications implemented by using Falanx.

Acknowledgment

This work was partly funded by KAKENHI 25871199.

References

[1] FOX Project (A Fault-oblivious Extreme-scale Execution Environment), http://fox.xstack.org/.
[2] Jack Dongarra et al., "International EXASCALE SOFTWARE PROJECT ROADMAP 1.1," http://www.exascale.org/mediawiki/images/a/a8/IESP-roadmap-1.1.pdf.
[3] The Falanx project, https://sites.google.com/site/spfalanx/.
[4] FAL Labs, "Kyoto Cabinet: a straightforward implementation of DBM," http://fallabs.com/kyotocabinet/.
[5] Apache ZooKeeper, http://zookeeper.apache.org/.
[6] Fault Tolerance Research in the Innovative Computing Lab at the University of Tennessee, http://fault-tolerance.org/.