Allocation Module 병렬화를 통한 Mesos 스케줄러의 확장성 및 성능 향상 기법

한호돌*, 오상윤**
*아주대학교 일반대학원 소프트웨어특성화학과
**아주대학교 소프트웨어융합학과
e-mail : {bab5ting, syoh}@ajou.ac.kr

Parallelization of Allocation Module for Scalability and Performance Improvement on Mesos Scheduler

Ho-Dol Han*, Sangyoon Oh**
*Dept. of Software Engineering, Graduate School of Ajou University
**Dept. of Software Convergence Technology, Ajou University

요 약

데이터 센터에서는 물리적인 규모 증가와는 달리 별도의 처리 없이는 분산처리 프레임워크가 동일한 클러스터 내에서 복수로 동작할 수 없어 전체 환경을 정적으로 분할하여 이들을 배치하는 것이 일반적이다. 그러나 최근 연구에서는 복수의 프레임워크를 한 클러스터 내에서 동작시킴으로써 클러스터의 활용률을 높이는 방향으로 이루어지고 있다. Mesos 는 복수의 분산처리 프레임워크를 한 클러스터에서 동작시키기 위한 시스템 중 하나로 각 프레임워크 스케줄러의 스케줄링을 지원하는 단일 Allocation Module 을 가진다. Allocation Module 은 모든 Slave 와 프레임워크 스케줄러들의 요청을 처리하는데, 시스템 규모가 커질수록 Allocation Module 으로 집중되는 부하가 증가하여 이에 따른 할당 속도 저하로 정상적인 동작이 불가능해진다. 이 문제를 해결하기 위해 본 논문에서는 Mesos 시스템의 Allocation Module 병렬화를 제안한다. 제안 방식을 통해 Allocation Module 의 부하를 분산함과 동시에 Head-of-line Blocking 으로 인한 스케줄링 지연 문제를 해결할 수 있을 것이다.

1. 서론

데이터 센터의 물리적인 규모는 매년 빠른 속도로 커지고 있다. 이러한 데이터센터에서 고성능 처리를 위해 Hadoop 과 같은 분산 프레임워크를 설치하여 사용하는 것이 보편적이다. 그러나 하나의 클러스터에서 복수의 분산처리 프레임워크가 동작할 수 없으며 또한 이들이 확장성에 제약을 가진다는 한계를 가지고 있기 때문에, 데이터 센터의 규모가 큰 경우 전체 자원을 분할하여 다수의 프레임워크를 운용하는 것이 일반적인 경향이다.

현재 분산 처리 프레임워크가 동작하는 환경에서는 전체 가용자원 환경을 고정된 크기로 나누어 분산 처리 프레임워크들을 배치하는 정적 파티션 접근법이 주로 사용되고 있다. 그러나 이 기법은 각 클러스터에 가해지는 부하가 균일하지 않은 경우 서로 다른 클러스터의 자원을 활용할 수 없기 때문에 클러스터 활용률이 낮아지는 단점을 가진다.

이에 대해 최근의 연구들에서는 전체 클러스터 활용도를 향상시키기 위해 다수의 분산 처리 프레임워크를 하나의 클러스터에서 함께 동작하도록 하는 방향으로 이루어졌다. 이러한 연구들에는 Apache Hadoop [1]의 Hadoop on Demand 스케줄러와 YARN [2], Mesos [3], 그리고 Google 의 Omega [4]가 대표적이다. 이와 같이 다수의 분산 처리 프레임워크가 동작하는 환경에서 현존하는 자원 할당에 사용하는 스케줄링 방식들은 크게 Monolithic, Two-level, Shared-state 세 가지 방식으로 분류될 수 있다 [4].

첫째, Monolithic 방식은 클러스터 내에 단일 스케줄러가 존재하며 복수개의 프레임워크에 대한 스케줄링을 지원한다. 이 경우 스케줄러 내에서 각 프레임워크에 특화된 스케줄링 로직 때문에 지원할 수 있는 프레임워크를 추가하기 어렵다. 또한 단일 스케줄러라는 특징 때문에 대기중인 다른 작업들이 한 작업의 스케줄링 시간 동안 지연되는 Head-of-Line Blocking 문제가 있다.

반면 Two-level 방식은 전체 클러스터 내에서 동작하는 다수의 스케줄러를 배치하고 이들을 조정하는 단일 중앙 조정자(Central Coordinator)를 둔다. 그러나 이 경우에는 단일한 중앙 조정자가 스케줄링의 병목지점이 되는 문제가 존재한다. Shared-state 방식은 각 클러스터의 스케줄러에서 클러스터 전체 상태의 동기화가 이루어진다. 이 특징으로 인해 클러스터의 크기와 스케줄러 수에 비례하여 각 스케줄러에 동기화로 가해지는 부하와 지연이 증가하는 문제가 있다.

Apache Hadoop, Spark[5]와 같이 클러스터를 구축하

는 대부분의 프레임워크가 사용하는 스케줄러는 Monolithic 스케줄러로 분류되며, Apache 의 Mesos 와 Hadoop on Demand 와 같이 다수의 프레임워크에 클러스터의 자원을 동적으로 할당해주는 자원 할당자가 존재하는 환경의 경우 Two-level 스케줄링, Google Omega 와 같이 각 스케줄러가 클러스터의 상태를 공유하는 경우 Shared-state 스케줄링 방식으로 분류된다.

본 논문에서는 Two-level 방식에 해당하는 Mesos 프레임워크의 단일 중앙 조정자에 해당하는 Resource Allocation Module 의 병렬화를 통해 기존 Two-level 방식에서 발생하는 병목 현상 문제를 제거하려 한다. 또한 이 접근 방법에 기반하여 기존 세 가지 분류에 비해 더 높은 확장성을 제공하며 큰 규모의 클러스터에서 좋은 스케줄링 성능을 제공할 수 있는 시스템 구조를 제안한다. 2 장에서는 기존 Two-level Scheduling 방식과 Mesos 에 대한 간략한 소개, 3 장에서는 Mesos 의 Allocation Module 병렬화로 발생하는 문제와 이에 대한 처리 방식을 설명하며 마지막으로 4 장에서는 본 논문의 결론을 맺는다.

2. 기존 Two-level Scheduling 방식

분산 처리 프레임워크에서의 작업은 큰 작업 단위인 Job 과 이를 구성하는 Task 로 나타낼 수 있다. 분산 처리 프레임워크는 일반적으로 단일 Master 와 다수의 Slave 로 클러스터를 구성한다. Master 는 작업 스케줄링, 로드밸런싱 등의 클러스터 관리를 담당하며 프레임워크에 요청되는 Job 의 Task 를 각 Slave 로 분배하며 Slave 는 할당된 Task 를 수행한다. 분산 처리 프레임워크들이 단일 클러스터에서 정상적으로 동작하기 위해서는 중간에 모든 Master 에 대해 각자의 할당을 조정하는 컴포넌트를 두거나 현재 클러스터의 자원 할당 현황에 대한 정보를 알게 함으로써 스케줄링 충돌 등의 상황을 피할 수 있어야 한다. Two-level Scheduling 방식은 전자에 해당한다.

Apache 의 Hadoop on Demand(HOD)는 RingMaster 와 다수의 HODRing 으로 구성된 HOD 레이어를 가진다. 사용자가 HOD 쉘을 통해 Hadoop 클러스터의 크기를 지정하여 요청하면 해당 클러스터와 사용자 간 연결을 지원하는 HODRing 이 생성되고 그 하부에서는 사용자의 요청에 부합하는 HOD 클러스터가 만들어져 해당 HODRing 과 연결된다. 이에 따라 Hadoop on Demand 또한 Two-level 스케줄링 방식으로 분류될 수 있으나 이 프레임워크는 Hadoop 클러스터만을 지원하며 이미 할당된 HOD 클러스터의 크기는 조절할 수 없는 문제가 존재한다.

Apache 의 Mesos 는 다수의 서로 다른 프레임워크의 스케줄링을 지원하는 단일 Allocation Module 이 Mesos Master 에서 동작한다. Mesos 는 각각의 개별적인 프레임워크를 로직 변화 없이 지원하기 위해 클러스터 내 Slave 들이 Allocation Module 에게 자신의 가용 자원을 수량화하여 Offer 라는 이름으로 제공하며 Allocation Module 은 주어진 가용 자원들을 종합해 각 프레임워크의 스케줄러에게 Offer 로 제공한다. 각 프레임워크 스케줄러는 주어진 Offer 에 기반하여 스케줄링 결정을 내리게 되고 실제 할당이 이루어진다. 이를 통해 각 프레임워크의 스케줄링 로직을 유지함과 동시에 Allocation Module 의 경량화를 달성할 수 있었다. 그러나 각 프레임워크에 대해 자원 할당의 공정성(Fairness)을 만족하기 위해 Offer 를 제공할 스케줄러를 순차적으로 선택하기 때문에 할당 과정이 직렬적이므로, 기존 Monolithic 스케줄러와 마찬가지로 Head-of-Line Blocking 문제가 존재한다. 이는 스케줄러에서 이루어지는 스케줄링 결정이 지연될 수록 이로써 대기중인 작업에서 나타나는 시작 지연이 더욱 커짐을 의미한다. 특히 [3] 에서는 기존 Mesos 의 즉각적인 Allocation 방식으로는 Slave 의 수가 15,000 개 이상인 경우 정상적인 동작이 불가능하였으며 이를 해결하기 위해선 즉각적인 Allocation 방식을 10 초 이상의 간격으로 배치 프로세싱으로 변경하는 것이 요구되었다.

정리하면, Mesos 는 이처럼 자원 관리 Module 을 추가함으로써 다양한 프레임워크를 단일 클러스터 내에 동작할 수 있게 하였으나 단일 Module 사용은 두 가지 문제를 가진다. 첫째, 직렬적인 할당 결정 방식으로 인해 한 스케줄러의 지연이 타 스케줄러들의 지연으로 이어진다. 둘째, Allocation Module 에 가해지는 과도한 네트워크 트래픽과 프로세싱 부하가 전체적인 할당 속도 저하를 불러와 큰 규모의 클러스터에서 정상적인 동작이 불가능하게 만든다. 따라서 대규모 클러스터에서 빠른 할당 속도를 유지하기 위해서는 Allocation Module 의 병렬화를 통해 이에 가해지는 부하를 분산시키는 방법이 유용할 것이다.

3. Resource Allocation Module 병렬화

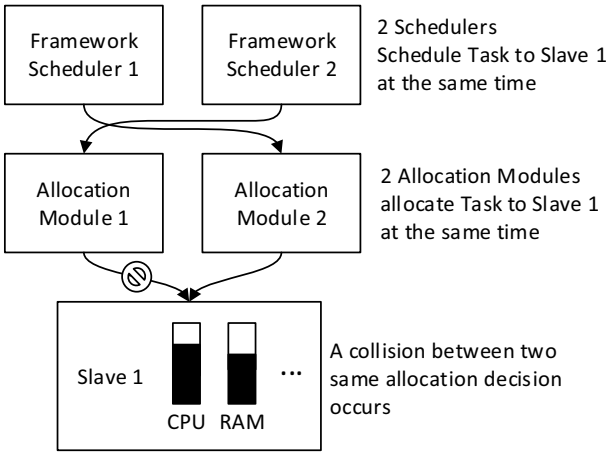
3.1. 기존 자원 할당 과정의 문제점

Apache Mesos 는 Offer 에 기반하여 동작하며 자원 할당은 다음과 같은 과정으로 이루어진다. 첫째, 각 Slave 노드가 Allocation Module 에 가용 자원을 제공(Offer)한다. 둘째, Allocation Module 은 이를 Dominant Resource Fairness(DRF)[6]에 따라 우선순위가 가장 높은 프레임워크 스케줄러를 선택하여 현재의 가용 자원을 알린다. 셋째, 프레임워크 스케줄러는 가용 자원 정보에 기반하여 스케줄링 결정을 내린다. 이 스케줄링 결정은 대상 Slave 들과 이들에 주어질 Task 들로 구성되며 각 Task 는 CPU, Ram 등 수행에 필요한 자원을 명시한다. 넷째, 스케줄러로부터 각 Slave 노드에 이를 할당한다. 이후 이 과정을 반복한다. 이러한 구조에서 Allocation Module 을 두 개 이상 배치하는 경우 다음과 같은 문제가 발생한다. 첫째, 병렬화된 스케줄링 결정으로 인해 나타나는 자원 경쟁 문제에 대한 해결이 필요하다. 둘째, Slave 에서 Offer 를 제공받을 Allocation Module 을 선택해야 한다. 셋째, 프레임워크 스케줄러로부터의 스케줄링 결정을 전달받을 Allocation Module 을 선택해야 한다.

3.2. 자원 경쟁

기존 할당 과정이 만약 다수의 Allocation Module 에

서 동일하게 일어나는 경우 이들이 서로 유한한 특정 자원에 대한 할당을 동시에 시도할 수 있다. 그림 1 은 이와 같은 할당 충돌을 도식화한 것으로 두 Allocation Module 중 하나는 할당에 실패할 수 있음을 의미한다. 두 Allocation Module 은 동시에 Slave 1 에 할당을 시도하나 Slave 1 의 자원은 한정되어 있다. 이 경우 두 할당 시도는 하나 이상 실패할 수 있는데, 이때 할당에 드는 비용이 크며 그 빈도가 잦은 경우 사전에 충돌을 피하는 방식이 이로울 것이다. 그러나 충돌을 피하기 위해 드는 비용이 더 크며 그 빈도가 잦지 않은 경우 별도의 충돌 회피 없이 단지 충돌이 발생했을 때의 경우에 한해 처리함으로써 전체적인 스케줄링 비용을 줄일 수 있게 된다.



(그림 1) 다중 Allocation Module 의 할당 충돌

3.4. 동시적(Concurrent) 자원 할당

본 절에서는 병렬화된 Allocation Module 환경에서의 할당 과정을 상세하게 설명한다. 다중 Allocation Module 을 사용하는 Mesos 환경에서 각 Allocation Module 은 프레임워크 스케줄러로부터 오는 스케줄링 결정의 충돌을 사전에 줄이기 위해, 스케줄링 결정을 처리할 때 결정에 포함된 각 Task 가 요구하는 Offer 에 대한 타 Allocation Module 이 가지고 있는 Offer 에 대한 동기화를 한다. 그 이후의 할당 과정에 대해서는 두 가지 방법을 고려할 수 있다.

첫째, 만약 Task 의 Offer 가 사용 가능한 경우 해당 Offer 를 잠금(Lock) 상태로 만든다. 이후 한 번의 추가적인 동기화를 통해 타 Allocation Module 에서 동일 Offer 에 대한 할당이 발생했는지 확인한다. 만약 해당 Offer 가 사용 가능하다면 자원 할당을 진행한다. 둘째, 현재 스케줄링 결정이 유효한 경우 우선 각 Slave 에 할당하며 할당 충돌이 일어나는 경우 Slave 에서 임의의 우선순위에 따라 해결하도록 하며 이를 Best Effort-Collision Resolve 방식으로 정의한다.

알고리즘 1 과 2 는 각각 잠금을 사용하는 경우와 Best Effort-Collision Resolve 방식을 사용할 때 Allocation Module 에서 스케줄링 결정을 처리하는 과정을 의사코드로 표현한 것이다. 잠금을 사용하는 경우 프레임워크 스케줄러로부터 스케줄링 결정이 전달되면 Allocation Module 은 현재 Offer 상태를 동기화한다. 그리고 스케줄링 결정에서 주어진 Task 들에 대해 할당 가능한 경우 해당 Task 가 요구한 Offer 에 대해 잠금을 하여 타 Allocation Module 에서의 동기화 요청이 오는 경우 해당 Offer 는 잠금 상태임을 알린다. 이후 한 번의 추가적인 동기화를 통해 잠근 Offer 들이 사용 가능한 지 확인하고 사용 가능한 경우 실제 할당을 시작한다.

Best Effort-Collision Resolve 방식의 경우 우선 현재의 전체 Offer 상태를 동기화한다. 그리고 프레임워크 스케줄러로부터 전달된 스케줄링 결정에서 주어진 Task 들에 대해 각 Task 에서 요구하는 Offer 와 현재 사용 가능 자원을 확인한다. 만약 할당이 가능하면 제출할 Task 목록에 추가하며 그렇지 못한 경우 거부된 Task 로 기록한다. 스케줄링 결정의 모든 Task 가 처리되면 이 과정으로 만들어진 제출 Task 목록을 가지고 각 Slave 에 실제 할당을 시작한다. 각 Slave 에서는 주어진 할당 요청에 대해 할당을 시도하며 할당에 성공한 Task 와 실패한 Task 를 보고한다. Allocation Module 은 이러한 결과를 종합하여 스케줄링 결정을 내린 프레임워크 스케줄러에 보고한다.

<알고리즘 1> 잠금을 사용하는 Allocation Module 의 스케줄링 결정 처리 의사코드

```
function processSchedule(decision)
  report := instant report for processing schedule
  lockedTasks := temporal empty set for locked tasks
  submittedTasks := temporal empty set for submitted tasks
  rejectedTasks := temporal empty set for rejected tasks

  this.offers := synchronize Offers

  foreach(task in decision.tasks)
    if this.offers.isCapable(task.offers) then
      this.offers.lock(task.offers)
      lockedTasks.add(task)
    else then
      rejectedTasks.add(task)
    end if
  end foreach

  offersAfterLocking := synchronize Offers
  foreach(task in lockedTasks)
    if offersAfterLocking.isCapable(task.offers) then
      this.offers.reduce(task.offers)
      submittedTasks.add(task)
    else then
      rejectedTasks.add(task)
    end if
  end foreach

  report.result := this.allocate(submittedTasks)
  report.submittedTasks := submittedTasks
  report.rejectedTasks := rejectedTasks

  this.sendSchedulingResult(report)
end function
```

<알고리즘 2> Best Effort-Collision Resolve 방식에서 Allocation Module 의 스케줄링 결정 처리 의사코드

```
function processSchedule(decision)
  report := instant report for processing schedule
  submittedTasks := temporal empty set
                    for submitted tasks
  rejectedTasks := temporal empty set for rejected tasks

  this.offers := synchronize Offers

  foreach(task in decision.tasks)
    if this.offers.isCapable(task.offers) then
      this.offers.reduce(offers)
      submittedTasks.add(task)
    else then
      rejectedTasks.add(task)
    end if
  end foreach
  report.result := this.allocate(submittedTasks)
  report.submittedTasks := submittedTasks
  report.rejectedTasks := rejectedTasks

  this.sendSchedulingResult(report)
end function
```

잠금을 통해 할당 충돌을 사전에 회피하는 경우 사전에 충돌을 줄일 수 있으나 충돌 회피에 드는 비용이 더 클 수 있다. 반면 잠금 없이 할당을 하는 경우 이와 반대로 충돌 회피 비용을 절감할 수 있으나 충돌이 잦은 경우 오히려 좋지 못하다. 이는 클러스터 내에서 시간 당 할당되는 작업의 수가 많아질 수록, 작업의 평균 규모가 클수록, 그리고 Slave 의 수가 적을수록 잦아지며 반대로 작업의 수가 적으며, 그 평균 규모가 작을수록, Slave 수가 많을수록 충돌의 횟수가 줄어들 것이다. 그러나 클러스터 전체 자원 사용량이 높은 경우 할당 충돌이 잦아지기 때문에 보다 효율적인 자원 할당을 위해서는 자원 사용량에 따라 두 가지 방식을 유동적으로 선택해야 할 것이다.

4. 결론

기존 데이터센터의 정적 분할을 이용한 클러스터 구성은 좋지 못한 자원 활용도를 보이는 문제가 있는데, 분할된 데이터센터 자원을 하나의 클러스터로 통일하여 사용함으로써 보다 높은 클러스터 활용도를 달성할 수 있을 것이다. 그러나 Monolithic, Two-level, Shared-state 총 세 가지 스케줄링 방식 모두 대규모 클러스터를 구성할 때 성능 혹은 확장성 문제가 발생한다. 본 논문에서는 이 중 Two-level Scheduling 방식인 Mesos 프레임워크의 Allocation Module 의 병렬화와 성능 향상의 가능성에 대해 Allocation Module 의 관점에서 설명하였다.

현재 본 논문에서 제시한 병렬화 방식을 적용하는 시도는 아직까지는 보이지 않으며, 특별히 시뮬레이션과 실험을 통한 성능 향상과 확장성 평가에 관한 연구는 찾을 수 없었다. 이와 같은 병렬성 연구는 주로 데이터베이스와 공유 및 트랜잭셔널 메모리 분야에서 찾을 수 있었으며 동시적 자원 할당에 대해 첫째는 Two-phase Commit, 둘째의 경우 Optimistic Concurrency Control 방식을 차용하였으며[7], 이 외에도 Distributed Shared Memory 분야의 연구를 분석함으로써 대규모 클러스터에서 병렬화된 자원 할당 방식에 유용하게 사용할 수 있을 것으로 생각된다. 본 논문에서 사용한 방식은 Shared-state 를 사용하는 Omega 와 유사하나 스케줄러와 할당 모듈 간 일 대 일의 높은 결합을 다 대 다의 낮은 결합도로 줄임으로써 보다 유연한 부하 분산이 가능하다. 향후 현재 제안 방식에 대한 수학적 모델링과 시뮬레이션, 그리고 실제 구현을 통한 성능 비교와 클러스터 규모에 따른 Allocation Module 조절 등의 이슈가 본 논문의 차후 연구과제로서 다루어질 것이다.

Acknowledgement

본 연구는 미래창조과학부 및 정보통신산업진흥원의 ICT/SW 창의연구과정사업의 연구결과로 수행되었음(NIPA-2014-H0510-14-1037)

참고문헌

[1] The Apache Software Foundation, "Apache Hadoop," (http://hadoop.apache.org/)

[2] V. K. Vavilapalli et al., "Apache Hadoop YARN: yet another resource negotiator," in *SOCC'13*, New York, 2013.

[3] B. Hindman et al., "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," EECS Department, University of California, Berkeley, May 2010.

[4] M. Schwarzkopf et al., "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys'13*, New York, 2013.

[5] M. Zaharia et al., "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[6] A. Ghodsi et al., "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *NSDI*, 2011.

[7] G. Coulouris et al., Distributed Systems: Concepts and Design (5th ed.), Addison-Wesley, 2012.