2022. 12. 02

강원대학교

통합 네트워크 매니저 설계서

문서 정보

|  |  |
| --- | --- |
| **버 전** | 1.0 |
| **작성일** | 2022-12-02 |
| **상 태** | 🗹 완료 🞎 진행 중 🞎 초안 |
| **작성자** | 정래원, 채신영, 길명선, |
| **검토자** | 길명선, 문양세 |
| **승인자** | 문양세 |

**목 차**

[1. RDMA 다대다 통신 라이브러리 1](#_Toc121411964)

[1.1. 개요 1](#_Toc121411965)

[1.2. RDMA 다대다 통신 라이브러리 설계 15](#_Toc121411966)

[1.2.1. 전체 동작 구조 15](#_Toc121411967)

[1.2.2. 주요 함수 36](#_Toc121411968)

[2. 다양한 RDMA 오퍼레이션 모드에 대한 벤치마크 테스트 92](#_Toc121411969)

[2.1. 개요 92](#_Toc121411970)

[2.2. 일반화된 RDMA 통신 모델 설계 103](#_Toc121411971)

[2.3. 일반화된 모델에서 RDMA 오퍼레이션 벤치마크 테스트 138](#_Toc121411972)

[3. 참고 문헌 161](#_Toc121411973)

# RDMA 다대다 통신 라이브러리

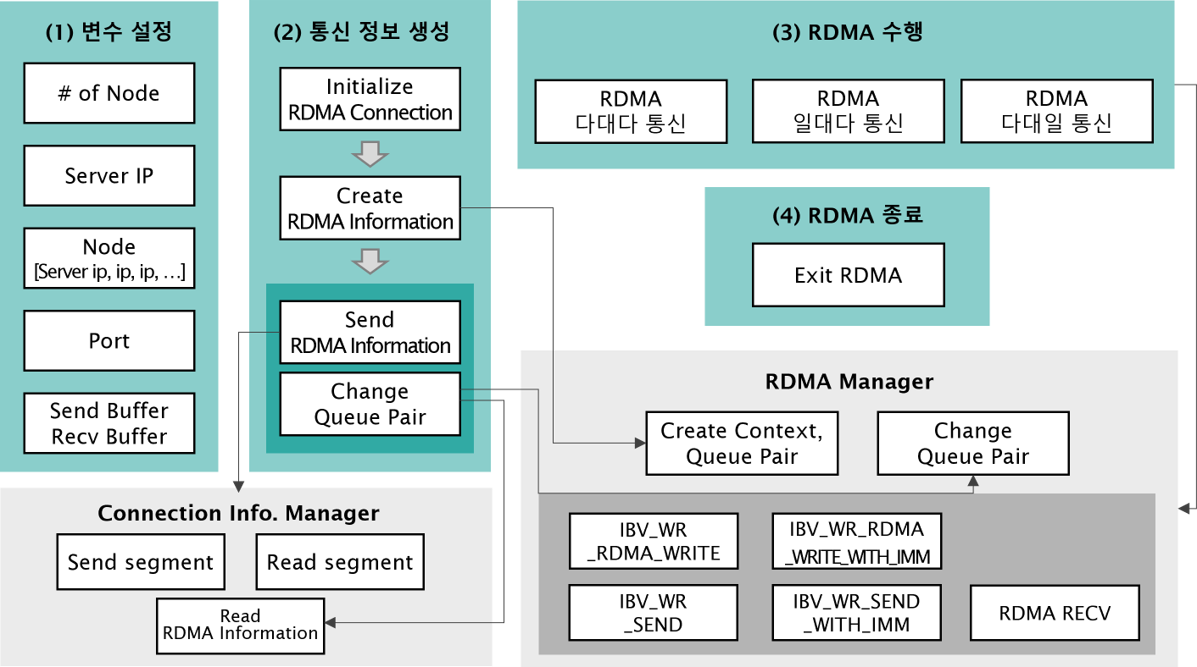
## 개요

빅데이터 처리나 딥러닝과 같은 복잡한 알고리즘 수행에는 분산 클러스터가 필수이다. 특히, HPC(High Performance Computer) 급 사양에서는 각 노드를 연결하는 네트워크 통신이 전체 성능에 큰 영향을 끼친다. 이 같은 분산 처리 환경에 사용되는 대표적인 기술로 RDMA(Remote Direct Memory Access)가 있다. RDMA는 CPU를 거치지 않고 노드 간 메모리에 직접 데이터를 송수신하는 기술로, 이더넷과 인피니밴드 환경에서 사용 가능하다. 일반적으로 이러한 RDMA는 Verbs라는 프로그래밍 모델로 개발한다. 그러나, 이는 구현 복잡도가 매우 높고, 실제 응용 개발 시 매우 많은 단계 정의가 필요하다. 또한, 일대일 통신 기반으로 매뉴얼이 구성되어 있어 분산 환경에 적용하기 위해서는 여러 시행 착오를 거칠 수밖에 없다. 5.2절에서 이러한 RDMA의 기존 프로그래밍모델을 일반화하여 구현 복잡도를 낮추고, 다수 노드 환경에도 쉽게 적용할 수 있는 새로운 라이브러리를 설계한다.

## RDMA 다대다 통신 라이브러리 설계

### 전체 동작 구조

그림 40은 제안 라이브러리를 활용한 RDMA 통신 모듈의 동작 구조를 나타낸다. 통신 모듈은 1) 변수 설정, 2) 통신 정보 생성, 3) RDMA 수행, 4) RDMA 종료의 네 단계로 구성된다. 첫째, 변수 설정 단계에서는 다대다 통신에 필요한 노드 수, 서버 IP, 포트 번호, 송수신 버퍼 주소 등의 변수들을 설정한다. 둘째, 통신 정보 생성 단계에서 RDMA 통신에 필요한 정보와 Queue Pair, Completion Queue등 데이터 송수신 관련 버퍼들의 생성, 등록 과정을 수행한다. 이를 위해, 변수 설정 단계에서 지정한 변수들을 등록하고 각 노드들을 소켓 통신으로 연결시킨다. 연결이 완료되면, RDMA 통신에 사용되는 큐들을 생성하고, 이들을 메모리 영역(memory region)에 등록한다. 각 노드들은 소켓 통신으로 데이터 송수신에 사용되는 큐들의 정보를 교환하며, 이를 통해 큐의 상태를 변경한다. 셋째, RDMA 수행 단계에서는 앞 단계의 정보와 Verbs API를 기반으로 일대다, 다대다 통신을 수행한다. 이 때, 제안하는 RDMA 라이브러리는 SEND, SENDw/Imm, WRITE, WRITE w/Imm 오퍼레이션들을 지원한다. 모든 데이터 통신이 완료되면, RDMA 종료 단계로 넘어간다. 넷째, RDMA 종료 단계에서는 통신을 하기위해 생성했던 각종 큐들을 모두 삭제하고, 큐의 상태 및 등록을 관리하는 영역들 역시 초기화한다. 해당 과정이 끝나면 모든 통신 프로세스를 종료한다.



<그림 40> RDMA 다대다 통신 라이브러리 구조도

### 주요 함수

본 절에서는 제안 라이브러리의 각 단계에서 사용되는 주요 함수에 대해 설명한다. 첫째, 변수 선언 단계의 initialize\_rdma\_connection()는 사용자가 선언한 변수를 등록하고 소켓 통신으로 노드들을 연결시키는 함수이다. 해당 함수는 현재 노드의 IP, 사용할 노드의 IP들, 사용할 노드의 개수, 포트 번호, send buffer, recv buffer 파라미터를 입력 받아 스레드를 이용해 각 노드의 서버와 다른 노드의 클라이언트를 각각 연결시킨다.

둘째, 통신 정보 생성 단계의 create\_rdma\_info()는 RDMA 통신 정보를 생성하는 함수이다. 해당 함수에서는 Context, Queue Pair, Completion Queue 등 RDMA통신에 필요한 정보들을 생성하고, 사용할 버퍼들의 메모리 영역을 등록한다. 이 함수를 사용하면 기존 RDMA 구현 시 단계별로 정의했던 복잡한 정보들을 한번에 생성∙등록할 수 있다. 이때, 생성된 각 노드 정보를 이용해 큐의 상태를 변경하기 때문에 노드들끼리 서로 해당 정보들을 공유할 수 있도록 벡터에 저장되도록 설계하였다.

셋째, create\_rdma\_info() 함수와 마찬가지로 통신 정보 생성 단계인 send\_info\_change\_qp() 함수는 create\_rdma\_info() 함수를 통해 생성된 RDMA 통신 정보를 각 노드들끼리 교환하고 Queue Pair의 상태를 변경하는 함수이다. 해당 함수는 생성한 정보들(버퍼 주소, lkey, rkey, queue pair number, lid 등)을 교환하고, 수신한 정보를 이용하여 Queue Pair 상태를 변경하는 기능을 제공한다. 정보 교환은 소켓 프로그래밍을 이용하여 서로가 서로에게 보내고 받는 형식으로 설계하였다. Queue Pair의 상태는 Init 상태, 수신 받을 수 있는 상태인 RTR(Ready to Receive) 상태, 송신할 수 있는 상태인 RTS(Ready to Send) 상태가 있다. 각 노드들이 보낸 정보를 수신한 후 그 정보들을 이용해 send\_buffer에 대해서는 데이터를 송신할 수 있는 상태인 RTS까지 변경하고, recv\_buffer는 데이터를 수신 받을 수 있는 상태인 RTR까지만 변경하게 설계하였다.

넷째, rdma\_comm() 함수는 RDMA 수행 단계에서 실제 RDMA를 이용한 다대다 데이터 송수신을 처리하며, 이를 위해 오퍼레이션별 동작 과정을 상세히 정의한다. 먼저, SEND, SEND w/Imm, WRITE w/Imm은 로컬이 원격에 데이터를 보내는 오퍼레이션인데, 이 때 전송할 데이터와 수신 확인을 위한 32비트의 정수 값을 함께 보내도록 설계하였다. 원격에서는 Completion Queue에 쌓여 있는 정수 값을 폴링(polling)하여 수신이 완료되었음을 알 수 있다. 반면에, WRITE는 다른 오퍼레이션들과는 다르게 로컬이 원격에 데이터를 보낸 뒤 수신이 성공해도 판단할 수 있는 방법이없다. 따라서 소켓 통신을 이용해 ‘send’ 시그널을 보내고 해당 시그널을 받으면 수신 성공으로 판단하도록 설계하였다.

실제 RDMA 수행에는 총 세 가지 함수가 사용된다. 먼저, rdma\_one\_to\_many\_send\_msg()는 RDMA 기반 일대다 통신을 위해 사용하는 함수이다. 일대다 통신은 변수 설정시 정의한 서버 IP를 가진 노드가 메시지를 송신하고, 서버 IP가 아닌 노드들이 메시지를 수신하는 구조이다. 이와 반대로, rdma\_many\_to\_one\_send\_msg()와 rdma\_many\_to\_one\_recv\_msg() 함수는 RDMA 기반 다대일 통신을 위한 기능을 담당한다. 다대일 통신은 서버가 아닌 노드들이 서버 노드에 메시지를 송신하고, 서버 노드는 각 노드에서 오는 메시지들을 수신하는 과정으로 구성된다. 이때, 서버에 해당하는 노드는 rdma\_many\_to\_one\_recv()를 통해 사용할 오퍼레이션을 인수로 넘겨주고, 나머지 노드는 rdma\_many\_to\_one\_send\_msg()를 통해 오퍼레이션과 송신한 메시지를 인수로 전달한다.

마지막으로 exit\_rdma() 함수는 RDMA 종료 단계에서 통신을 끝내기 위해 생성한 정보를 정리하는 기능을 제공한다. 즉, 통신에 사용한 큐를 모두 삭제하고, 큐 관리를 위해 생성한 메모리 영역들 역시 할당 해제하는 과정을 진행하게 된다. 따라서, 해당 함수는 RDMA 통신이 모두 끝난 뒤 프로그램 종료 전에 호출된다.

# 다양한 RDMA 오퍼레이션 모드에 대한 벤치마크 테스트

## 개요

최근 SNS, IoT 등 실시간으로 대량의 데이터가 생성되면서 이를 빠르게 처리하기 위한 고성능 네트워크 처리 기술들이 주목을 받고 있다. 네트워크 성능을 향상시키기 위해 많은 기업에서 InfiniBand를 사용하여 고성능 처리를 지원한다. InfiniBand는 높은 대역폭과 낮은 지연시간을 보장하는 고성능 장비로 다양한 통신 프로토콜을 제공한다. 이 중, 불필요한 버퍼 복사 없이 직접 메모리에 데이터를 전송하는 RDMA에 대한 관심이 높다. 6.2 절에서는 다수 노드로 확장한 일반화 된 RDMA 통신 모델을 구현하고, 6.3 절에서 이를 여러 RDMA 오퍼레이션에 적용해 벤치마크 테스트를 진행한다. 즉, 다수의 노드로 일반화된 RDMA 통신 모델을 설계 및 구현하고, 노드 수와 데이터 크기에 따른 RDMA 오퍼레이션의 성능을 비교한다.

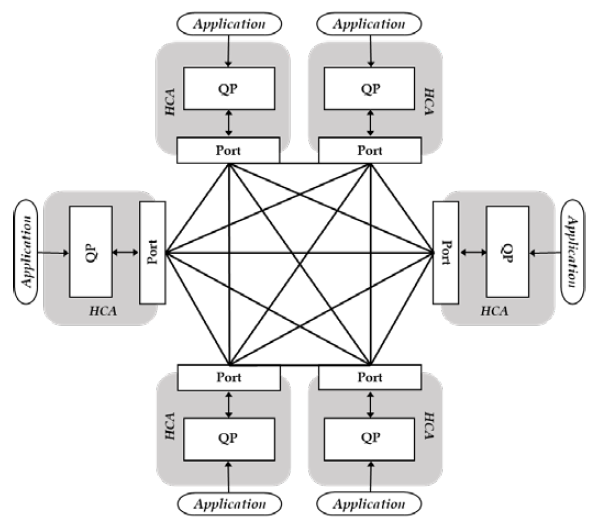
## 일반화된 RDMA 통신 모델 설계

본 절에서는 RDMA 오퍼레이션에 대한 벤치마크를 위해 구현한 통신 모델과 벤치마크 방법을 소개한다. 또한, 구현한 모델의 유효성 검증을 위해, OFED(Open Fabrics Enterprise Distribution) 벤치마크 툴과 비교 실험을 진행한다. 실험 환경은 표 12과 같다.

|  |  |
| --- | --- |
| **장비** | **사양** |
| **Node** | **- Intel Xeon 2.4GHz 6 Core**  **- 64GB RAM**  **- HCA : Mellanox MT4099** |
| **Network** | **- InfiniBand: Mellanox SwitchXⓡ-2 MSX6012F-1BFS Managed FDR 56Gbps** |
| **Software** | **- OS: Ubuntu 18.04**  **- InfiniBand Driver: MLNX\_OFED\_LINUX-4.9-3.1.5** |

<표 12> 실험 환경

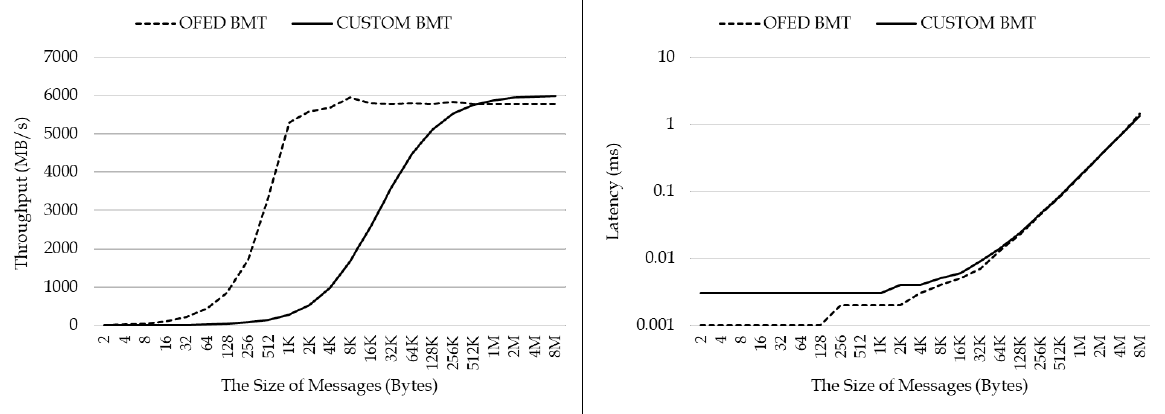
설계한 통신 모델에서 모든 노드는 송신과 수신을 함께 수행하기 때문에 각 노드는 서버와 클라이언트 역할을 동시에 담당한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 이는 복잡한 네트워크 환경을 가정하여 RDMA 통신 모델을 일반화하기 위함이다. 그림 41은 설계한 일반화된 RDMA 통신 모델 중 참여 노드 수가 6대 일 때의 토폴로지 상태를 나타낸다. 실험에서는 노드를 2대 부터 하나씩 토폴로지에 추가하며 6대까지 진행한다.



<그림 41> 일반화된 RDMA 통신 모델 구조도(6-nodes)

구현한 모델을 벤치마크에 활용하기 위해, 각 노드의 서버와 클라이언트는 RDMA 오퍼레이션을 지정된 횟수만큼 반복한다. 또한, 서버는 RDMA 오퍼레이션을 수행할 때마다 폴링을 통해 작업 완료 여부를 즉시 확인한다. 이는 원격 노드에서 데이터 수신을 알 수 없는 write와 같은 오퍼레이션에 대해서도 동일하게 실험하기 위함이다. 각 오퍼레이션에 대한 성능은 데이터를 보내고, 처리하는 시간을 종합하여 처리량과 지연시간을 측정한다.

일반화된 모델의 유효성 검증을 위해 노드 2대로 구성된 일대일 모델에서 OFED 벤치마크 툴과 비교한다. 오퍼레이션은 write를 사용하며, 수행 횟수를 10만회로 고정한다. 데이터 크기는 2B부터 8MB까지 2배씩 증가시킨다. 그림 42는 비교 결과를 나타낸다. 그림을 보면, 메시지 크기가 커질수록 처리량과 지연시간이 높아지며, 실험에 사용한 장비의 최대 성능에 가깝게 치리됨을 알 수 있다. 하지만, 메시지 크기가 512KB 미만일 때, 제안하는 모델이 OFED 툴보다 성능이 낮다. 그 이유는 일반화된 모델의 경우 오퍼레이션 수행마다 매번 폴링하고, OFED 툴은 처음과 마지막에만 폴링을 수행하기 때문이다. 이로 인해, 메시지 크기가 작을 때는 폴링으로 인한 대기가 발생하여 처리량이 낮고 지연시간이 높게 나타난다. 하지만, 메시지 크기가 커질수록 전송에 더 많은 시간이 소요되기 때문에 그 차이가 줄어든다. 실제로, OFED툴과 동일한 결과를 보이는 Infinity 벤치마크를 대상으로 매번 폴링하도록 실험한 결과, 구현 모델과 동일하게 나타났다. 이를 통해, 일반화 모델의 유효성을 검증하고, 폴링 오퍼레이션 성능에 미치는 영향을 확인할 수 있다.

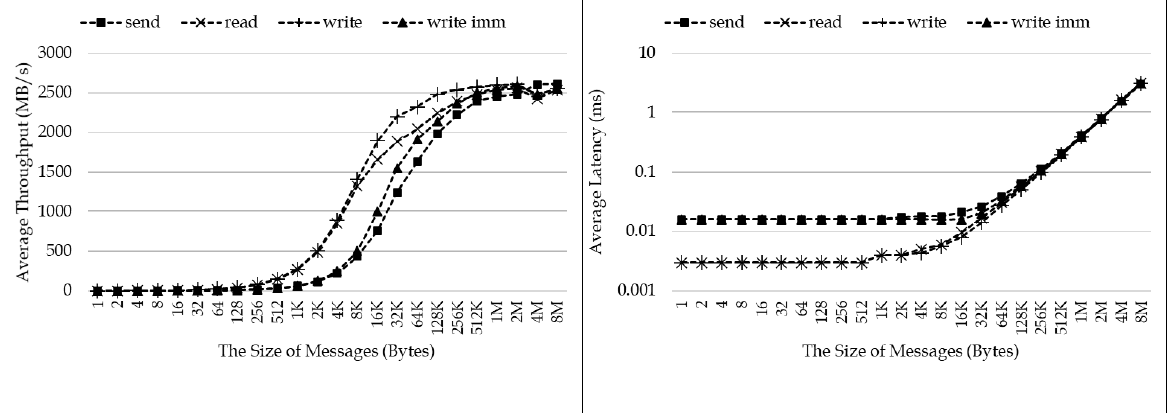


<그림 42> OFED 벤치마크 툴과 제안하는 모델의 비교 결과

## 일반화된 모델에서 RDMA 오퍼레이션 벤치마크 테스트

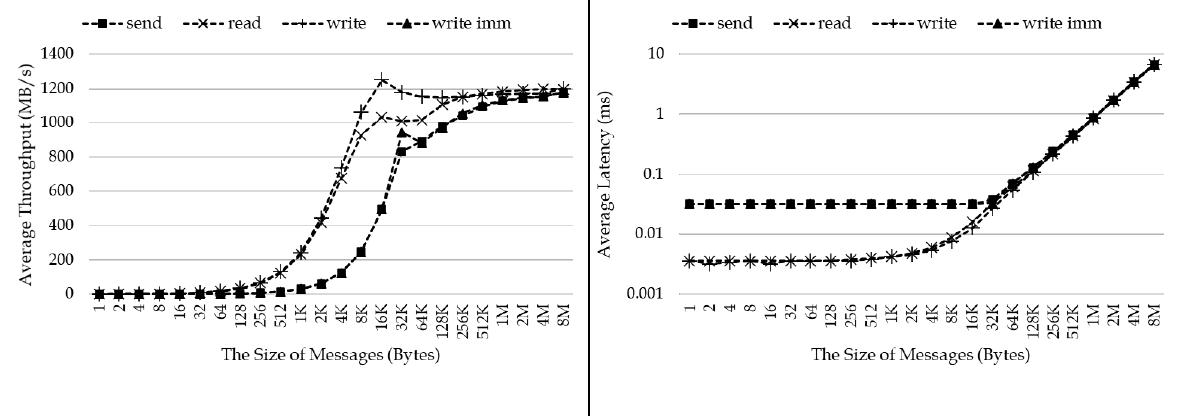
본 절에서는 다수 노드로 일반화된 RDMA 통신 모델에서 RDMA 오퍼레이션에 대한 벤치마크 테스트를 진행한다. 메시지 크기를 1B부터 8MB까지 2배씩 증가시키고, 노드를 하나씩 증가시키며 실험한다. 오퍼레이션은 send (receive), write, read, write imm 네 가지를 대상으로 비교한다.

그림 42은 노드가 3 대일 때, 각 오퍼레이션에 대한 평균 처리량과 지연시간을 나타낸다. 그림을 보면, 네 가지 오퍼레이션 모두 메시지 크기가 증가할수록 처리량과 지연시간이 증가한다. 자세히 살펴보면, write와 read가 send와 writre imm에 비해 높은 처리량과 낮은 지연시간을 보인다. 이러한 결과가 나타나는 이유는 write와 read는 폴링이 필요하지 않은 one-sided 오퍼레이션이기 때문이다. 반면, send와 write imm은 원격 노드에 대한 폴링이 필요한 two-sided 오퍼레이션으로, 완료 시그널을 전달받을 때까지 대기하기 때문에 상대적으로 평균 처리량이 낮게 나타난다.



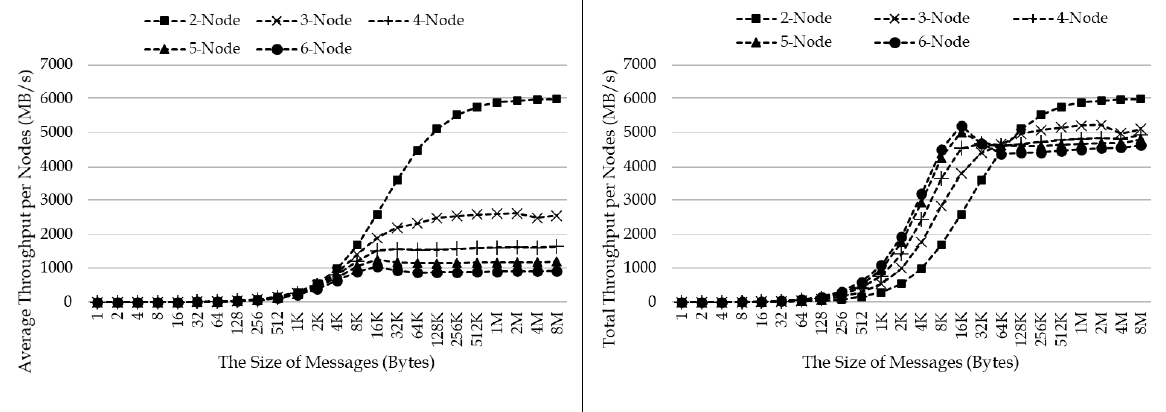
<그림 42> 3-nodes 토폴로지에서 오퍼레이션 성능 비교

그림 43은 5 대의 노드로 구성된 환경에서 오퍼레이션에 대한 실험 결과를 나타낸다. 그림 42와 마찬가지로 각 오퍼레이션의 성능이 비슷한 추세를 보이며, 8KB 결과에서는 오히려 그 차이가 더욱 두드러지는 것을 볼 수 있다. 이를 통해, 노드 수가 증가해도 one-sided와 two-sided 오퍼레이션 간의 차이는 변하지 않음을 알 수 있다. 하지만, 노드 수가 증가할수록 평균 처리량이 감소한다. 원인을 분석하기 위해, 모든 실험에서 평균 및 전체 처리량을 추가적으로 분석한다.



<그림 43> 5-nodes 토폴로지에서 오퍼레이션 성능 비교

그림 44는 2 대부터 6 대까지 모든 실험에서 write 오퍼레이션에 대한 평균 및 전체 처리량을 나타낸다. 그림의 왼쪽을 보면, 노드 수를 n이라 할 때 3-6-nodes 결과는 2-nodes 결과 대비 1/(n-1) 만큼의 평균 처리량을 보인다. 이는 같은 네트워크 내에서 처리할 수 있는 메시지의 평균 처리량이 통신에 참여하는 노드(QP) 수에 반비례하기 때문이다. 이러한 현상은 통신 노드 수가 증가하면서 네트워크 부하가 가중되기 때문에 나타나며, 그림 42와 그림 43에서 평균 처리량이 줄어든 것도 같은 이유 때문이다. 그림 44의 오른쪽을 보면, 같은 네트워크 내에서 전체 메시지 처리량은 64KB 이상부터 모두 4500MB/s 이상으로 일정한 것을 확인할 수 있다.



<그림 44> 모든 실험에서 write에 대한 처리량 비교

# 참고 문헌

IBTA. (2007년 11월). Infiniband Architecture Specificaiton Volume 1. Release 1.2.1.

IBTA. (2014년 10월). Supplement to InfiniBand Architecture Specification Volume 1 Relaese 1.2.1 Annex A17: RoCEv2.

IBTA. (2016년 11월 18일). “About Infiniband”. infinibandta.org.에서 검색됨

“infiniband”. (2016). wikipedia: https://en.wikipedia.org/wiki/InfiniBand에서 검색됨

Mellanox. (2016). Mellanox OFED for Linux User Manual Rev 3.40.

OFA. (2016). “OFA Overview”. openfabrics: https://www.openfabrics.org/에서 검색됨

Oracle. (2016). Oracle: http://www.oracle.com/에서 검색됨

PfisterF.G. (2001). An introduction to the infiniband architecture. , High Performance Mass Storage and Parallel I/O (페이지: 617-632).

TOP500.org. (2016년 11월). “top500 list”. TOP500.org: https://www.top500.org/에서 검색됨

MPI-Forum. "MPI docs", www.mpi-forum.org에서 검색됨

MPI. "MPI를 이용한 병렬프로그래밍", http://k-atoms.ksc.re.kr/에서 검색됨

OpenMP. (2021). wikipedia: https://en.wikipedia.org/wiki/OpenMP에서 검색됨

Jost G. (2003). Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster., EWOMP, Vol 3.

RDMA. (2015). RDMA Aware Networks Programming User Manual Rev 1.7

Grzegorz M. (2010). Pregel: a system for large-scale graph processing. SIGMOD (페이지: 135-146).

Joseph E. (2012). PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, USENIX (페이지 17-30)

Giraph. (2020). https://giraph.apache.org/에서 검색됨