2021. 11.

강원대학교

IPoIB/RDMA 기반 네트워크 매니저 설계서

문서 정보

|  |  |
| --- | --- |
| **버 전** | 1.0 |
| **작성일** | 2021-11-18 |
| **상 태** | 🗹 완료 🞎 진행 중 🞎 초안 |
| **작성자** | 길명선, 김현종, 남궁주홍, 서지희, 정래원 |
| **검토자** | 문양세 |
| **승인자** | 문양세 |

**목 차**

[IPoIB/RDMA 기반 네트워크 매니저 설계서 1](file:///C:\\Users\\wnghd\\Desktop\\DKE\\그래프\\설계서\\IPoIB_RDMA_기반 네트워크 매니저 설계서_v.0.8.docx" \l "_Toc88150718)

[문서 정보 2](#_Toc88150719)

[1. 인피니밴드 기술 분석 1](#_Toc88150720)

[1.1. 인피니밴드 개요 1](#_Toc88150721)

[1.1.1. 인피니밴드 역사 2](#_Toc88150722)

[1.1.2. OFED 개요 3](#_Toc88150723)

[1.1.3. OFA 개요 4](#_Toc88150724)

[1.2. 관련 기술 및 경쟁 기술 6](#_Toc88150725)

[1.2.1. iWARP 6](#_Toc88150726)

[1.2.2. RDMA over Converged Ethernet (RoCE) 6](#_Toc88150727)

[1.3. 인피니밴드 구조 분석 7](#_Toc88150728)

[1.3.1. 인피니밴드 구성 7](#_Toc88150729)

[1.3.2. OFED 구조 14](#_Toc88150730)

[1.4. 인피니밴드 버전별 성능 15](#_Toc88150731)

[1.5. 인피니밴드 적용 사례 16](#_Toc88150732)

[1.5.1. 데이터 센터 16](#_Toc88150733)

[1.6. 구축 및 동작 확인 18](#_Toc88150734)

[1.6.1. 서버 환경 구성 18](#_Toc88150735)

[1.6.2. 성능 19](#_Toc88150736)

[2. MPI 및 OpenMP 22](#_Toc88150737)

[2.1. MPI 22](#_Toc88150738)

[2.1.1. 개요 22](#_Toc88150739)

[2.1.2. 목표 22](#_Toc88150740)

[2.1.3. 역사 23](#_Toc88150741)

[2.2. MPI 기본 개념 24](#_Toc88150742)

[2.2.1. 프로세스와 프로세서 24](#_Toc88150743)

[2.2.2. 메시지 24](#_Toc88150744)

[2.2.3. Tag 26](#_Toc88150745)

[2.2.4. 커뮤니케이터(communicator) 26](#_Toc88150746)

[2.2.5. 프로세스 랭크(rank) 27](#_Toc88150747)

[2.2.6. 점대점(Point to Point) 통신 27](#_Toc88150748)

[2.2.7. 집합 통신(Collective Communication) 31](#_Toc88150749)

[2.3. OpenMP 33](#_Toc88150750)

[2.3.1. 개요 33](#_Toc88150751)

[2.3.2. 역사 33](#_Toc88150752)

[2.3.3. 설계 34](#_Toc88150753)

[2.3.4. 핵심 요소 34](#_Toc88150754)

[2.4. Hybrid Model 36](#_Toc88150755)

[3. TurboGraph++ 네트워크/메시지 전송 구조 분석 36](#_Toc88150756)

[4. IPoIB/RDMA 기반 네트워크 매니저 36](#_Toc88150757)

[4.1. 개요 36](#_Toc88150758)

[4.2. Vertex-centric 분산 그래프 처리 시스템 38](#_Toc88150759)

[4.3. IPoIB 네트워크 매니저 프로토타입 40](#_Toc88150760)

[4.4. RDMA 네트워크 매니저 프로토타입 42](#_Toc88150761)

5. RDMA 다대다 통신 라이브러리 44

5.1.개요 44

5.2.RDMA 다대다 통신 라이브러리 설계 45

5.2.1. 전체 동작 구조 45

5.2.2. 주요 함수 46

6. 다양한 RDMA 오퍼레이션 모드에 대한 벤치마크 테스트 48

6.1. 개요 48

6.2. 일반화된 RDMA 통신 모델 설계 48

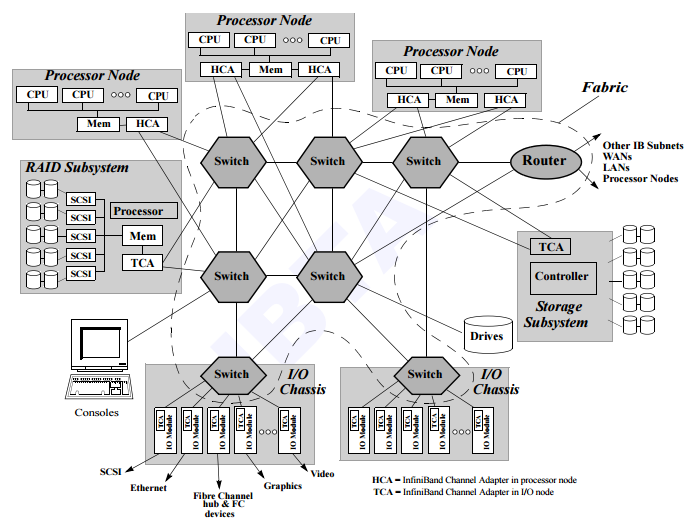
6.3. 일반화된 모델에서 RDMA 오퍼레이션 벤치마크 테스트 50

[7. 참고 문헌 52](#_Toc88150762)

# 인피니밴드 기술 분석

## 인피니밴드 개요

인피니밴드는 서버, 통신 장비, 스토리지, 임베디드 시스템들의 상호 연결에 사용하는 입출력 아키텍쳐를 정의하는 산업 표준 사양서이다. 높은 대역폭과 낮은 지연시간, 높은 안정성, 확장성을 특징으로 가지고 있다. 고성능 컴퓨팅과 기업용 데이터 센터에서 많이 사용하는 스위치 패브릭 방식의 통신 연결방식이다 (IBTA, 2016). 스위치 패브릭 방식을 사용하여 패킷을 수평적으로 전달할 수 있다. 노드와 스위치는 Channel Adapter (CA)라는 네트워크인터페이스 카드와 연결된다. Host Channel Adapter (HCA)는 계산노드에서 사용하며 Target Channel Adapter (TCA)는 입출력노드에서 사용한다. 그림 1은 인피니밴드 컨셉을 보여준다 (IBTA, 2007).



<그림 1> IBA System Area Network

### 인피니밴드 역사

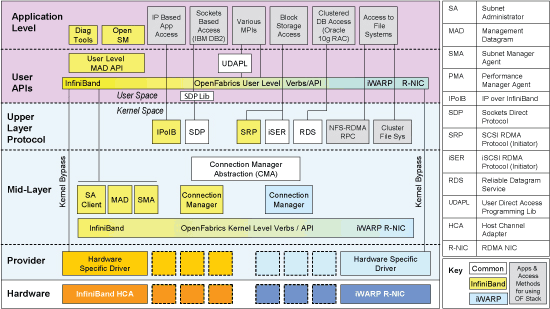
1999년에 Future I/O와 Next Generation I/O의 두 가지 기술이 병합되어 인피니밴드가 탄생했다. Compaq, Dell, Hewlett-Packard, IBM, Intel, Microsoft, Sun 등이 멤버로 가입한 인피니밴드 Trade Association (IBTA)에서 표준화를 이끈다. PCI bus를 이용한 상호 연결에는 병목현상이 발생하여 PCI-X를 이용한 인피니밴드 사양서 Version 1.0이 2000년에 처음으로 공개되었다. 2001년에 멜라녹스가 10Gbit/s 장비를 출시하였고, 2002년에 Microsoft는 인피니밴드 개발을 멈추었다. 2003년에 인피니밴드 기반의 클러스터가 그 해에 TOP500 프로젝트에서 3위를 기록하였다. 이 클러스터는 Virginia Tech에서 구축하였다. TOP500은 세계에서 비분산형 컴퓨터시스템을 1위부터 500위까지로 성능의 순위를 매기는 프로젝트이다. 2004년에 OpenFabrics Alliance에서 표준화된 리눅스 기반의 인피니밴드 스택을 개발함으로 리눅스가 인피니밴드를 지원하였다. 2009년에 TOP500에서 기가비트 이더넷은 259개에 설치되었고 인피니밴드는 181개에서 사용되었다. 2010년에 인피니밴드 시장을 주도하는 멜라녹스와 볼테어가 합병했고, 오라클은 멜라녹스에 대규모로 투자하였다. 2011년 International Supercomputing Conference에서 FDR 스위치가 발표되었다 (infiniband, 2016). 현재 IBTA에 가입한 멤버에는 "Advanced Photonics, Inc.", "Amphenol Interconnect Products", "Anritsu", "Applied Optoelectronics, Inc.", "Bay Microsystems", "Broadcom", "Bull SAS", "ConnPro Industries Inc.", "Cray, Inc.", "Finisar Corporation", "Foxconn Interconnect Technology, Ltd.", "Fujitsu Limited", "Genesis Technology USA, Inc.", "Hewlett-Packard", "IBM", "Intel Corporation", "Jess-Link Products Co., Ltd.", "Keysight Technologies, Inc.", "Lawrence Livermore National Laboratory (LLNL)", "LUXSHARE-ICT Co., Ltd.", "Mellanox Technologies, Inc.", "Microsoft", "Molex Inc.", "NetApp", "Oracle", "QLogic", "Samtec", "Semtech", "Shanghai Net Miles Fiber Optics Technology Co. Ltd.", "Software Forge, Inc.", "TE Connectivity", "Tektronix", "UNH InterOperability Lab", "Unisys Corporation", “Volex inc.", "Woodward McCoach, Inc.", "Yamaichi Electronics USA" 등이 있다. 그림 2는 IBTA를 주도하는 스티어링 위원회이다.



<그림 2> IBTA 스티어링 위원회

### OFED 개요

OFED(OpenFabrics Enterprise Distribution) 혹은 OFS(OpenFabrics Software)는 RDMA(Remote Direct Memory Access)와 커널 바이패스 애플리케이션을 위한 오픈소스 소프트웨어이다. OFED는 커널 레벨 드라이버, 채널 지향 RDMA send/receice 오퍼레이션, 커널 바이패스를 위한 유저 레벌 API, 병렬 처리 시스템인 MPI, 소켓 데이터 교환(RDS, SDP), file system/database system(iSER, NFS-RDMA, SRP)를 포함한다. 높은 효율의 네트워크, 스토리지 연결, 병렬 컴퓨팅 등을 필요로 하는 데이터 센터, HPC에서 사용된다. 이러한 OFED는 10 기가비트 이더넷, iWARP 기술, RDMA over Converged Ethernet (ROCE), 인피니밴드 등의 네트워크 기술에서 사용할 수 있다. OFED는 Red Hat Enterprise Linux (RHEL), Novell SUSE Linux Enterprise Distribution (SLES), Oracle Enterprise Linux (OEL), 마이크로소프트 Windows Server 등의 다양한 운영체제에서 사용이 가능하다 (OFA, 2016). 그림 3은 OFED 스택 구조를 보여준다. 2.1.2절에서 OFED에 대해 자세히 설명한다.



<그림 3> OFED 스택

### OFA 개요

OpenFabrics Alliance (OFA)는 오픈소스 기반의 기관으로 OFED의 개발, 테스트, 라이선스 관리, 배포 등을 수행한다. 이 기관의 미션은 최소한의 CPU 부하로 최대의 대역폭, 낮은 지연시간 빠른 유선 메시징으로 고효율의 애플리케이션 개발을 가능하게 하는 것이다. 2004년 6월에 OpenIB Alliance로 벤더와 독립적으로 리눅스 기반의 인피니밴드 스택 개발을 목표로 시작되었다. 2005년에 마이크로소프트 Windows를 지원하였다. 2006년에 iWARP를 지원하여 기관을 확장하였고 2010년에 RoCE를 지원하기 시작했다. 멤버쉽의 종류는 Promoter, Adopter, Supporter, Academic, Individual, Consulting Participant로 나누어진다. Promoter는 가장 강한 영향력을 가진 것으로 가입비가 3000달러이고 매년 회비는 10000달러이다. Adopter는 기관에 개발과정에 공헌하고 참여하지만 강한 영향력을 가지고 있지 않다. 가입비는 3000달러이고 연회비는 5000달러이다. Supporter는 OFED를 사용하는 멤버들로 실제적으로 공헌하지는 않는다. 가입비는 3000달러이고 연회비는 1500달러이다. Academic은 교육기관들로 OFED에 공헌하고 개발 프로세스, 테스팅, 문제 해결의 역할을 수행하는 멤버다. 연회비는 2000달러이다. Individual은 OFED 사용자, 개발자들로 이루어진 멤버로 연회비는 200달러이다. Consulting Participant는 OFED 기반의 기술에 공헌한 기관, 회사들로 명예멤버로 연회비는 없다. 다음 그림 4, 그림 5, 그림 6, 그림 7은 OFA 멤버쉽 명단을 보여준다 (OFA, 2016).



<그림 4> OFA Promoters 멤버



<그림 5> OFA Adopters 멤버



<그림 6> OFA Supporters 멤버



<그림 7> OFA Consulting Participants 멤버

## 관련 기술 및 경쟁 기술

### iWARP

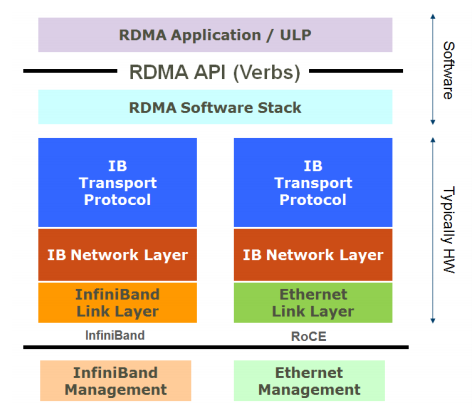
iWARP는 Internet 프로토콜 네트워크에서 효율적인 데이터 전송을 위한 RDMA 네트워크 프로토콜이다. 인피니밴드와 경쟁기술로 자리잡고 있다. IETF에 RFC 문서가 등록되어있다. iWARP를 정의하는 RFC 문서는 표 1에서 보여준다.

<표 1> iWARP에 관한 RFC 문서

|  |  |
| --- | --- |
| RFC 번호 | RFC 내용 |
| RFC 5040 | A Remote Direct Memory Access Protocol Specification |
| RFC 5041 | Direct Data Placement over Reliable Transports |
| RFC 0543 | Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation |
| RFC 5044 | Marker PDU Aligned Framing for TCP Specification |
| RFC 5042 | Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security |
| RFC 6580 | IANA Registries for the Remote Direct Data Placement (RDDP) Protocols |
| RFC 6581 | Enhanced Remote Direct Memory Access (RDMA) Connection Establishment |
| RFC 7306 | Remote Direct Memory Access (RDMA) Protocol Extensions |

### RDMA over Converged Ethernet (RoCE)

이더넷 네트워크에서 RDMA를 사용가능하게 해주는 네트워크 프로토콜이다. 인피니밴드기반의 애플리케이션을 이더넷에서 사용가능하게 해준다. IBTA에서 표준화 작업 및 관리하고 있다. 그림 8에서 RoCE 프로토콜 스택을 보여준다 (IBTA, 2014).

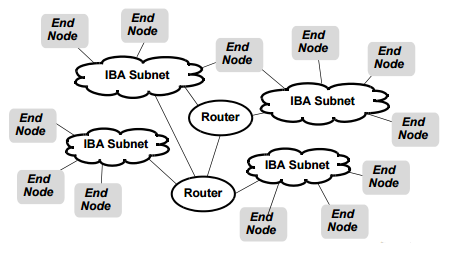


<그림 8> 인피니밴드와 RoCE 프로토콜 스택

## 인피니밴드 구조 분석

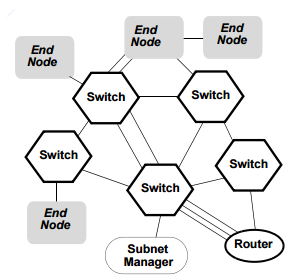
### 인피니밴드 구성

인피니밴드는 크게 스위치 패브릭 구조와 RDMA 오퍼레이션을 통해 커널을 거치지 않는 특징이 있다. 이를 통해 인피니밴드는 낮은 지연시간과 높은 대역폭을 지원한다. 또한 물리적 한 개의 링크를 Virtual Lane (VL) 0~ VL 15로 논리적으로 나누어 우선순위가 높은 것을 VL15, 가장 낮은 것을 VL0으로 패킷을 보내어 QoS를 보장한다 (IBTA, 2007). 다음 내용부터 각각을 세부적으로 살펴본다. 인피니밴드 서브넷은 라우터를 통해서 연결을 한다. 그림 9는 인피니밴드의 네트워크 토폴로지를 보여준다.



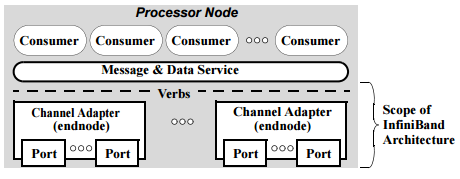
<그림 9>IBA 네트워크 컴포넌트

한 개의 서브넷 안에는 여러 개의 스위치가 존재할 수 있으며 반드시 한 개의 서브넷매니저가 필요하다. 스위치, 라우터, 노드 간의 링크가 여러 개가 존재할 수 있다. 또한 토폴로지 구조가 이더넷의 계층적 구조가 아닌 패브릭 방식으로 이루어진 것을 알수있다. 기존의 계층적 구조의 이더넷은 서버간의 통신에서 상위 스위치 장비를 거쳐 수직적으로 패킷이 전달되었다. 패브릭 방식은 패킷이 수평적으로 전달되어 안정적인 메시지 전달을 할 수 있다. 그림 10은 인피니밴드 서브넷의 토폴로지를 보여준다.



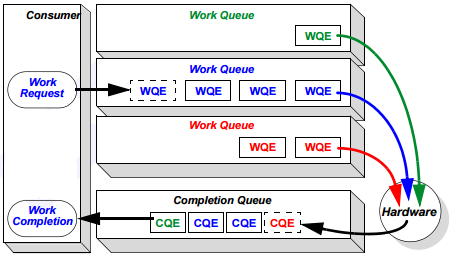
<그림 10>IBA 네트워크 서브넷 컴포넌트

각 노드에는 Channel Adapter를 통해 인피니밴드 네트워크와 연결한다. 노드의 애플리케이션에서 IBA Verbs를 통해서 설정, 관리, 오퍼레이션을 할 수 있는 API 이다. 그림 11은 노드의 구조도이다.



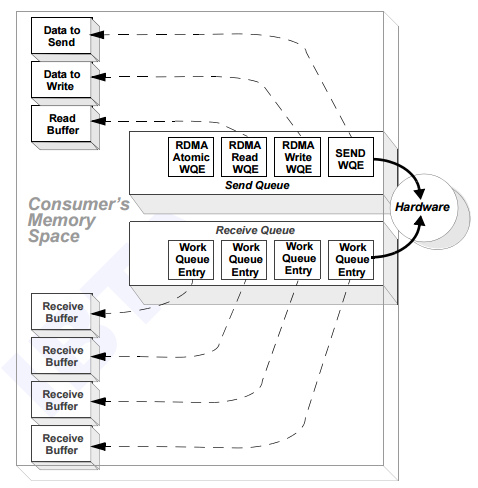
<그림 11> 각 노드의 구조도

통신에서 오퍼레이션들은 큐잉 모델을 통해서 이루어진다. 오퍼레이션을 담는 큐를 work queue라고 하며 만들어 질 때 기본적으로 쌍으로 만들어진다. 이것을 Queue Pair (QP)라고 한다. 한 개는 다른 소비자에게 보내는 오퍼레이션을 위한 것이고 한 개는 다른 소비자에게서 받는 오퍼레이션을 위한 것이다. 소비자는 Work Request를 통해서 지시하면 Work queue에 Work Queue Element (WQE) 를 넣는다. 그리고 Channel Adapter에서 처리하면 Completion Queue에 Completion Queue Element (CQE) 를 넣는다. 그림 12에서 큐잉 모델을 보여준다.

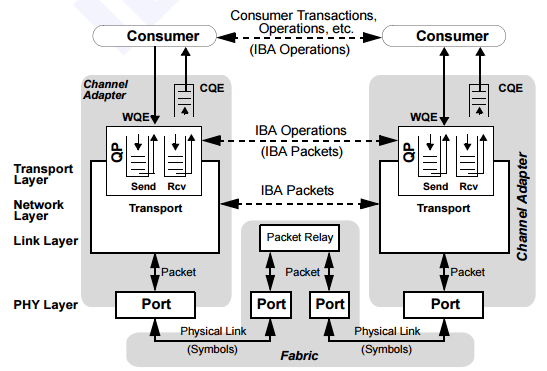


<그림 12> 소비자 큐잉 모델

인피니밴드에서 오퍼레이션은 SEND, RDMA, MEMORY BINDING 등이 있다. SEND 오퍼레이션은 소비자의 메모리에서 데이터 블록을 네트워크를 통해 목적지 소비자에게 데이터를 전송하는 WQE를 만든다. RDMA 오퍼레이션은 RDMA-WRITE, RDMA-READ, ATOMIC의 3가지 타입으로 WQE를 만든다. RDMA-WRITE는 소비자의 메모리에서 원격 소비자의 메모리로 데이터를 전송한다. RDMA-READ는 원격 메모리에서 소비자의 메모리로 데이터를 전송한다. ATOMIC는 원격 메모리의 특정 위치의 값을 읽어오고 다시 그곳에 값을 수정 또는 대체하는 오퍼레이션이다. MEMORY BINDING 오퍼레이션은 소비자가 전송할 때 사용할 수 있는 Memory Region을 설정할 수 있다. 이 메모리 영역에는 R\_KEY가 생성되는데 이것은 RDMA 오퍼레이션을 수행할 때 권한이 있는지 판단하기위해 사용한다. 그림 13은 Work Queue 오퍼레이션을 보여주고 그림 14는 인피니밴드 통신 구조를 보여준다.



<그림 13> Work Queue 오퍼레이션



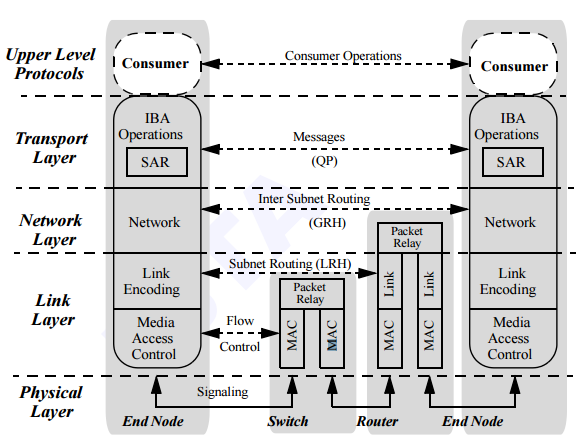
<그림 14> 인피니밴드 통신 구조

종단 간의 노드에서 각각의 QP는 같은 서비스 타입으로 만들어진다. Connection Oriented와 Datagram의 차이는 한 노드의 QP가 다른 노드의 QP와 연결이 맺어지는 것이 Connection Oriented 이고 Datagram은 한 노드의 QP에서 어떤 노드의 QP던지 송신 혹은 수신이 가능하다. Acknowledged는 신뢰성이 있는 데이터 통신을 위해 ACK 메시지를 응답으로 보내는 것을 말한다. 표 2는 인피니밴드 서비스 타입을 보여준다.

<표 2> 인피니밴드 서비스 타입

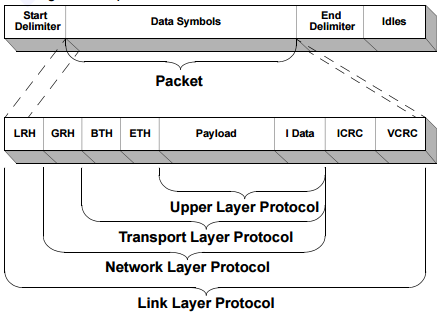
|  |  |  |  |
| --- | --- | --- | --- |
| Service Type | Connection Oriented | Acknowledged | Transport |
| Reliable Connection | Yes | Yes | IBA |
| Unreliable Connection | Yes | No | IBA |
| Reliable Datagram | No | Yes | IBA |
| Unreliable Datagram | No | No | IBA |
| RAW Datagram | No | No | Raw |

인피니밴드 프로토콜 스택은 물리 계층, 링크 계층, 네트워크 계층, 전송 계층, 상위 계층으로 나누어 진다. 그림 15는 인피니밴드 프로토콜 스택을 보여준다.



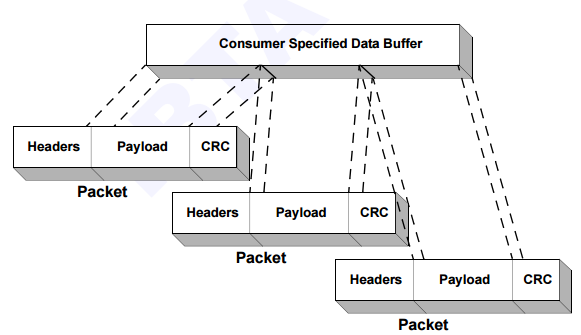
<그림 15> 인피니밴드 프로토콜 스택

물리 계층은 데이터의 전송을 비트 단위로 전송한다. 링크 계층은 링크 관리 패킷과 데이터 패킷으로 나누어 진다. 링크 관리 패킷은 링크를 만들고 유지하는 오퍼레이션을 위해 사용된다. 데이터 패킷은 인피니밴드 오퍼레이션, 데이터 등을 전달할 때 사용한다. 그림 16은 데이터 패킷의 포맷을 보여준다.



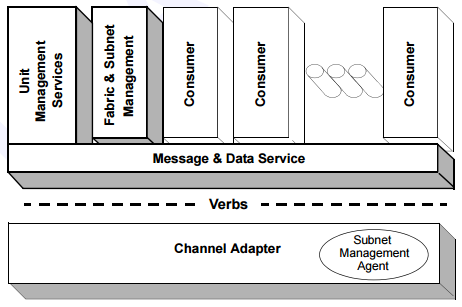
<그림 16> 인피니밴드 데이터 패킷 포맷

Local Route Header (LRH)는 로컬 출발지 포트에서 로컬 목적지 포트의 전송에 관한 부분을 담당하며 스위치가 전달하여준다. LRH 안에는 LID 정보가 들어있는데 LID는 서브넷 매니저가 포트별로 할당하여 준다. 또한 ICRC와 VCRC를 통해 데이터의 무결성을 검사한다. 네트워크 레이어는 서브넷 간의 데이터 전송을 위해 존재한다. Global Route Header (GRH)가 패킷에 존재한다. GID라는 IPv6 형식의 주소가 헤더 안에 존재하여 출발지에서 목적지로 전송한다. 라우터는 GRH를 기반으로 패킷은 전달한다. 전송 계층에서는 Maximum Transfer Unit (MTU)에 따라 데이터를 나누어서 전송하고 다시 합치는 역할을 수행한다. Base Transport Header에는 오퍼레이션 코드, 패킷 넘버 등의 정보가 들어 있다. 그림 17은 전송 계층에서 데이터를 나누고 합치는 것을 보여준다.



<그림 17> 전송계층에서 데이터 나누기/합치기

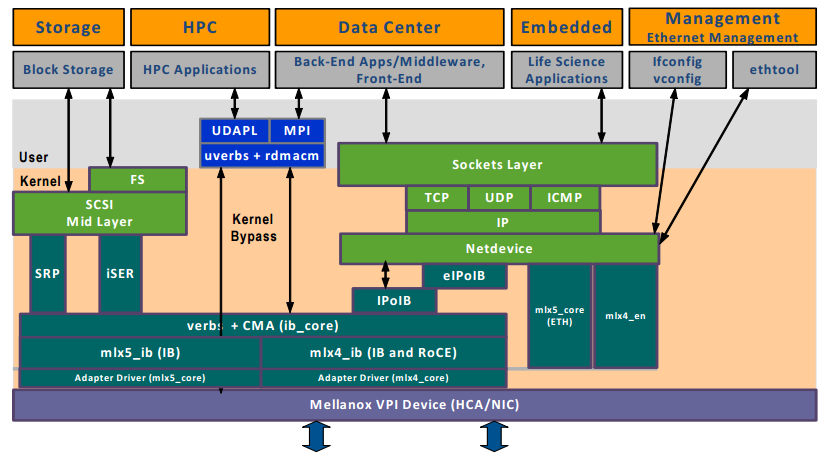
상위 계층에서는 다양한 소비자들이 존재한다. 소비자들은 Verbs를 통해서 데이터를 송수신할 수 있다. 그림 18은 상위 계층의 구조를 보여준다 (IBTA, 2007).



<그림 18> 상위 계층

### OFED 구조

인피니밴드를 사용하기 위해서는 OFED를 설치해야한다. OFED에는 드라이버까지 포함되어서 배포된다. 그림 19는 멜라녹스 OFED 스택으로 상위 계층 프로토콜이 어떻게 자원을 사용하는지 보여준다.



<그림 19> 멜라녹스 OFED 스택

mlx4\_ib, mlx4\_ib는 장비의 드라이버 역할을 수행한다. 상위 계층으로는 IPoIB가 있다. IPoIB는 인피니밴드 위에서 IP 데이터그램을 송수신 가능하게 해주는 것으로 쉽게 인피니밴드를 이용할 수 있다는 장점이 있다. IPoIB는 두종류의 MTU가 존재한다. Datagram mode의 경우는 MTU 사이즈가 4KByte이고 Connected Mode는 MTU가 64KByte이다. iSCSI Extensions for RDMA (iSER) 은 iSCSI를 RDMA로 확장한 것으로 데이터를 SCSI 버퍼에서 데이터 카피가 없이 바로 전송이 가능하게 해준다. User Direct Access Programming Library (uDAPL)은 표준 API로 인피니밴드, RoCE에서 높은 안정성, 확장성, 고성능 메시지 전송을 위해 사용한다. Message Passing Interface (MPI)는 병렬 소프트웨어 개발을 가능하게 해주는 라이브러리로 병렬 컴퓨터, 클러스터, 헤테로지니어스 네트워크 등에서 사용한다. 멜라녹스 OFED는 MPI를 포함하고 있다. Open MPI로 오픈소스 MPI-2 버전을 구현한 것이다 (Mellanox, 2016). OFED 소프트웨어 배포는 크게 Channel Adaapter 드라이버, 중간 계층의 Verbs, CM, uVerbs 등, ULPs의 IPoIB, SRP 등, MPI, OpenSM(인피니밴드 서브넷 매니저), 진단 프로그램, 성능 테스트 프로그램, 펌웨어 툴 등이 포함되어 있다.

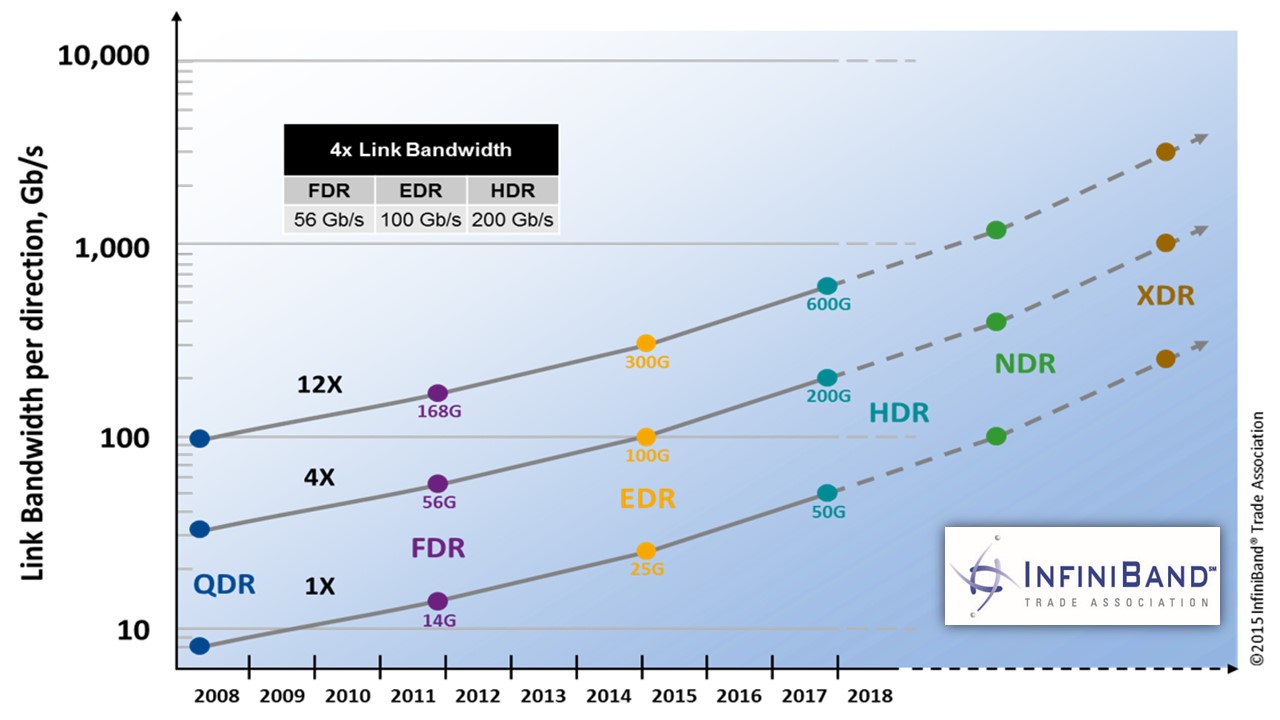
## 인피니밴드 버전별 성능

인피니밴드는 시그널 속도에 따라 SDR, DDR, QDR, FDR-10, FDR, EDR이라고 버전을 부른다. 표 3에서 자세한 속도를 확인할 수 있다.

<표 3> 인피니밴드 속도 버전

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | SDR | DDR | QDR | FDR-10 | FDR | EDR |
| 1X | 2 Gbit/s | 4 Gbit/s | 8 Gbit/s | 10 Gbit/s | 13.64 Gbit/s | 25 Gbit/s |
| 4X | 8 Gbit/s | 16 Gbit/s | 32 Gbit/s | 40 Gbit/s | 54.54 Gbit/s | 100 Gbit/s |
| 12X | 24 Gbit/s | 48 Gbit/s | 96 Gbit/s | 120 Gbit/s | 163.64 Gbit/s | 300 Gbit/s |

인피니밴드는 시그널링 속도와, 링크 수로 전송 속도가 결정된다. 링크 수는 케이블 자체에 와이어 수를 의미한다. 와이어 수는 기본 1x는 4개이고 4x는 16개, 12x는 48개이다. 지금 현재 최신 인피니밴드 시그널링 속도는 EDR(Enhanced Date Rate)로 4x 기준으로 100Gbps이고 12x라면 300Gbps이다. 일반적으로 4x가 많이 쓰인다 (PfisterG., 2001). 속도 버전에 대한 개발은 IBTA 워킹그룹이 주관하고 있다. 그림 20은 IBTA에서 제공하는 인피니밴드 개발 로드맵이다 (IBTA, 2016).



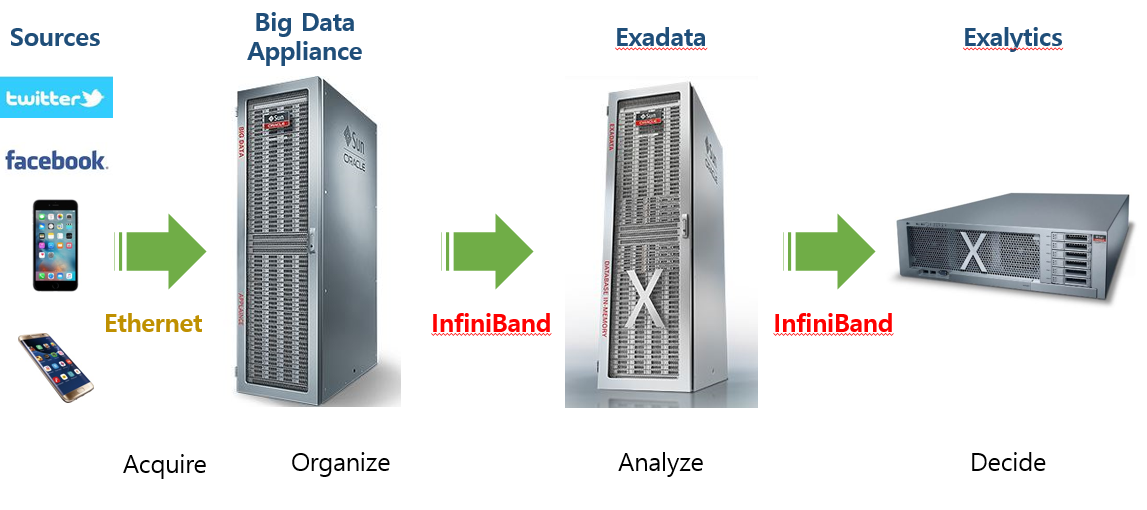
<그림 20> 인피니밴드 개발 로드맵

## 인피니밴드 적용 사례

### 데이터 센터

* **오라클 빅데이터 솔루션**

오라클은 인피니밴드 관련 네트워크 제품들을 판매하고 있다. 또한, 인피니밴드 기반의 솔루션, 서비스들을 판매한다. 오라클 빅데이터 솔루션은 데이터를 수집하여 정제하고 분석하여 저장하는 것이다. 다음 그림 21은 오라클 빅데이터 솔루션을 나타낸다.



<그림 21> 오라클 빅데이터 솔루션

빅데이터 어플라이언스는 Hadoop을 탑재하여 빅데이터 수집, 저장 처리를 담당하는 제품이다.데이터 연결폭 한계를 극복하기 위해 인피니밴드 네트워크를 사용한다. 엑사데이터 베이스 머신은 오라클 데이터베이스 소프트웨어를 위해 컴퓨트시스템과 스토리지 시스템을 결합한 것이다. 스케일 아웃 컴퓨트 서버와 스케일 아웃 스토리지 서버, 인피니밴드, 특화된 소프트웨어가 결합된 것을 엑사데이터라고 한다. 이 제품을 사용하는 기업들은 다음과 같다 (Oracle, 2016).

* 신세계 푸드 – 수불데이터 집계, 분석을 위해 사용한다.
* SK Telecom – 실시간 과금 신뢰도 및 업무효율성 향상에 사용한다.
* 친애저축은행 – 기업의 정보계 시스템과 개인 계정계 시스템에 사용한다.
* IBK – 온라인 금융 검래 지원 시스템 속도 향상과 전산시스템 통합에 사용한다.
* **슈퍼컴퓨터 (www.top500.org)**

1993년에 시작된 프로젝트로 세계에서 가장 강력한 비분산 컴퓨터 시스템을 1위부터 500위까지 순위를 매긴다. 일년에 두번 업데이트를 한다. 첫번째는 International Supercomputing Conference에서 6월에 업데이트하고 두번째는 ACM/IEEE Supercomputing Conference에서 11월에 업데이트를 한다. 2016년 11월 기준, 전세계 슈퍼컴퓨터 500위에서 187개 슈퍼컴퓨터가 인피니밴드를 사용하여 인피니밴드가 가장 큰 비중을 차지한다. 표 4는 슈퍼컴퓨터 500위에서 사용한 네트워크를 종류별로 나타낸다 (TOP500.org, 2016).

<표 4> TOP500 네트워크 종류별 사용 비율

|  |  |
| --- | --- |
| 이름 | 비율 |
| Infiniband | 37.4 % |
| 10G Ethernet | 35.6 % |
| Custom Interconnect | 14.2 % |
| Gigabit Ethernet | 5.6 % |
| omnipath | 5.6 % |
| 기타 | 1.6 % |

* **The Ohio State University의 논문 연구 사례**
* Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., ... & Panda, D. K. (2012, November). High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (p. 35). IEEE Computer Society Press.
* Wasi-ur-Rahman, Md, et al. "High-performance rdma-based design of hadoop mapreduce over infiniband." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013.
* Lu, Xiaoyi, et al. "High-performance design of Hadoop RPC with RDMA over InfiniBand." *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013.
* 주요 연구 내용

인피니밴드 네트워크에서 높은 성능의 MPI 구현하여 이를 바탕으로 Hadoop HDFS, Hadoop MapReduce 등에 적용하여 성능을 향상시키는 연구를 주로 한다. 또한, 클러스터, High Performance Computing에 관한 연구를 많이 한다.

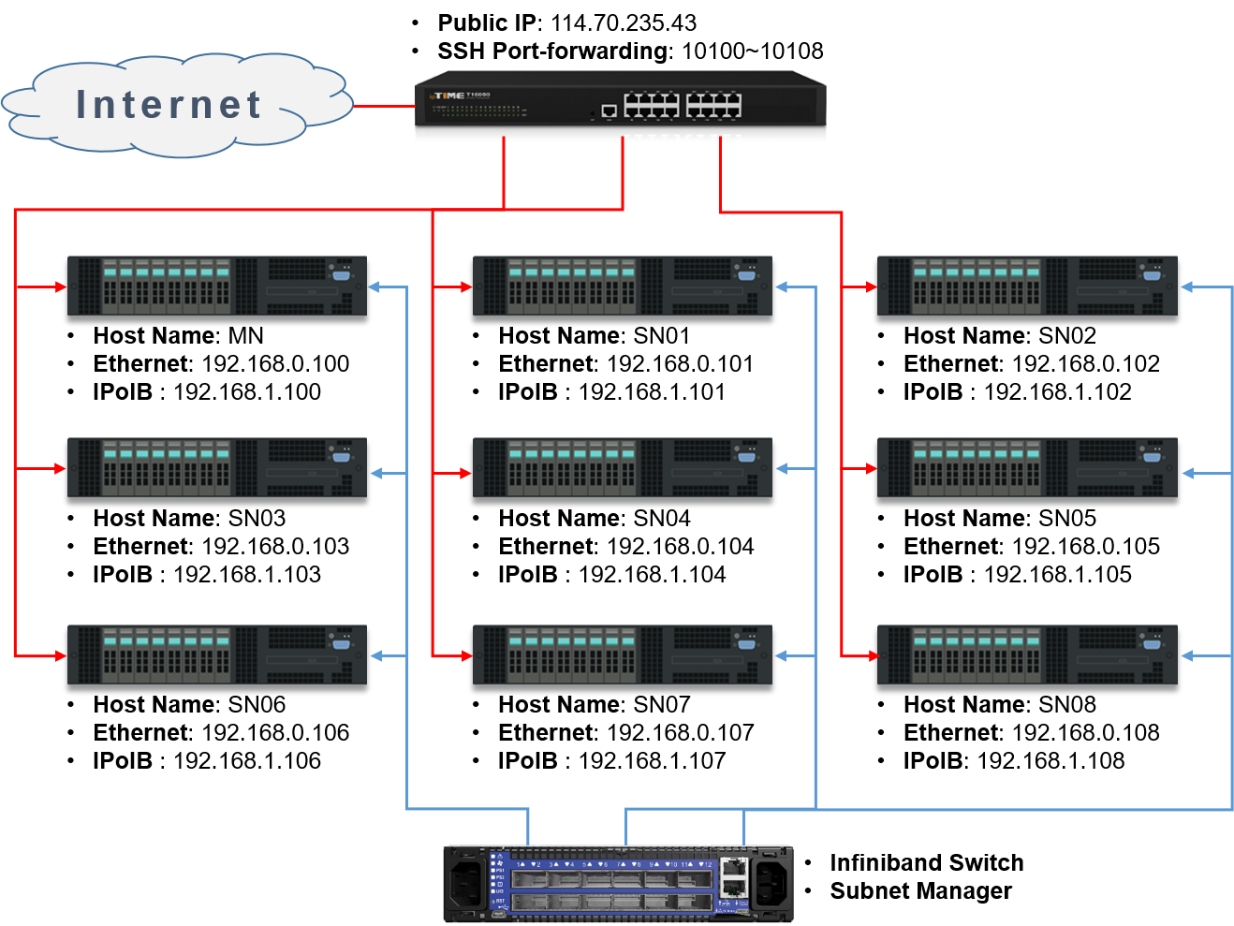
* **인하대 논문 연구 사례**
* 허희성, 이광수, 메히디, & 김덕환. (2015). 인피니밴드 스토리지 네트워크를 적용한 오픈스택 클라우드 스토리지 시스템의 설계 및 스토리지 가상화 성능평가. *정보과학회 컴퓨팅의 실제 논문지*, *21*(7), 470-475.
* 주요 연구 내용

입출력 연산 시 서버와 스토리지 노드 간에 발생할 수 있는 병목현상을 해결하기 위해 고속 통신을 지원하는 인피니배늗 스토리지 네트워크를 적용한 오픈스택 클라우드 스토리지 시스템의 구조를 제안하고, 오픈스택에 적용한 인피니밴드 스토리지 네트워크가 고성능 스토리지 가상화에 적합함을 연구한다.

## 구축 및 동작 확인

### 서버 환경 구성

본 과제에서는 마스터노드 한 개와 데이터 노드 8개로 클러스터를 구성하였다. 네트워크는 1Gb/s 이더넷 공유기와 56Gb/s 인피니밴드 스위치를 사용하여 구축하였다. 인피니밴드의 IP는 IPoIB의 사용을 위한 IP주소이다. 인피니밴드 스위치는 서브넷 매니저 기능을 같이 수행한다. 이더넷 공유기를 통해 Internet과 연결되고 외부와 통신이 가능하다. 그림 22는 네트워크 구성도를 보여준다.



<그림 22> 네트워크 구성도

CPU가 Master Node는 8 Core이고 Slave Node는 6 Core인 점을 제외하고 사양은 모두 같다. 각 노드의 사양은 표 5에서 보여준다.

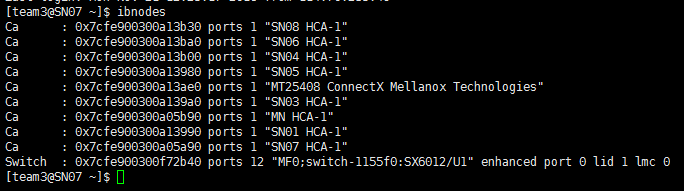
<표 5> 노드 사양

|  |  |
| --- | --- |
| CPU | Xeon E5-2630V3 2.5GHz, 8 Core(Master Node) / 6 Core(Slave Node) |
| Memory | PC4-1700 REG ECC 32GB |
| SSD | 256G SSD 850 PRO |
| HDD | 기업용 1TB ST1000NM0033 |
| NIC | Gbit LAN on Board \* 2 / ConnectX-3 adapter card, FDR IB (56Gb/s) |

### 성능

* **인피니밴드 동작 확인**

인피니밴드 네트워크가 올바르게 동작하고 있는지 확인이 가능한 명령어는 OFED에 포함되어 있는 ibnodes이다. 현재 인피니밴드 네트워크에 연결되어 있는 노드를 알 수 있다. 스위치와 노드 9개가 확인 가능함으로 인피니밴드 네트워크에 모두 접속했음을 알 수 있다. 이 명령어의 결과화면은 그림 23과 같다.



<그림 23> ibnodes 명령어 결과

* **인피니밴드 성능 측정**
* qperf 명령어

한쪽 노드에서는 qperf 서버를 동작하고 다른 노드에서는 qperf 클라이언트 역할을 수행하여 대역폭, 지연시간을 측정하는 명령어이다. tcp\_bw, tcp\_lat는 이더넷 성능과 IPoIB의 성능을 측정하는 것으로 IP주소를 다르게 사용하면 된다. 이 명령어는 OFED에 포함되어 있다. 표 6에서 측정할 테스트 목록을 보여준다.

<표 6> qperf 테스트 목록

|  |  |  |
| --- | --- | --- |
| 통신 종류 | 테스트 이름 | 설명 |
| 소켓 기반 | tcp\_bw | TCP 대역폭 측정 |
| tcp\_lat | TCP 지연시간 측정 |
| RDMA Send/Receive | rc\_bw | RC 전송 대역폭 측정 |
| rc\_lat | RC 전송 지연시간 측정 |
| RDMA | rc\_rdma\_read\_bw | RC RDMA read 오퍼레이션 대역폭 측정 |
| rc\_rdma\_read\_lat | RC RDMA read 오퍼레이션 지연시간 측정 |
| rc\_rdma\_write\_bw | RC RDMA write 오퍼레이션 대역폭 측정 |
| rc\_rdma\_write\_lat | RC RDMA write 오퍼레이션 지연시간 측정 |

이더넷 tcp\_bw는 1 GbE 이더넷을 사용한 것이고 IPoIB tcp\_bw는 IPoIB tcp\_bw는 OFED에서 제공하는 IPoIB를 사용한 것이다. RDMA를 사용했을 때 대역폭이 약 6.28 GB/s로 가장 크고, RDMA READ에서 지연시간이 4.94 μs로 가장 낮다. 표 7은 qperf를 이용한 성능 측정 결과이다.

<표 7> qperf 성능 측정 결과

|  |  |  |  |
| --- | --- | --- | --- |
| 이더넷 tcp\_bw | 118 MB/s | 이더넷 tcp\_lat | 30.6 μs |
| IPoIB tcp\_bw | 2.52 GB/s | IPoIB tcp\_lat | 10.4 μs |
| rc\_bw | 6.28 GB/s | rc\_lat | 7.44 μs |
| rc\_rdma\_read\_bw | 6.29 GB/s | rc\_rdma\_read\_lat | 4.94 μs |
| rc\_rdma\_write\_bw | 6.29 GB/s | rc\_rdma\_write\_lat | 6.98 μs |

* 파일 전송 프로그램

파일 전송 프로그램을 작성하여서 파일 전송 속도를 측정할 때 사용하였다. 클라이언트에서 서버에게 파일을 전송하고 서버에서는 파일을 받으면서 속도를 측정하는 프로그램이다. 프로그램의 간략한 코드는 표 8에서 보여준다.

<표 8> 파일 전송 프로그램

|  |  |
| --- | --- |
| 서버 | 클라이언트 |
| int main()  {  clnt\_sock=accept();  fd = open();  clock\_gettime();  while( (str\_len=read(clnt\_sock)) != 0)  {  write(fd);  total += str\_len;  }  close(fd);  clock\_gettime();  } | main()  {  sock=socket();  connect();  fd = open(argv[3], O\_RDONLY);  while( (str\_len=fileRead(fd)) > 0)  {  networkWrite(sock);  }  close();  } |

파일 전송은 1GB 더미 파일을 전송하였다. 네트워크가 이더넷과 IPoIB에서 파일시스템을 각각HDD, SSD, RamDisk에서 전송 속도를 측정하였다. IPoIB, RamDisk를 사용했을 때 1376 MB/s 속도로 가능 높다. 성능 결과는 표 9에서 보여준다.

<표 9> 파일 전송 속도 결과

|  |  |  |  |
| --- | --- | --- | --- |
|  | HDD | SSD | RamDisk |
| Ethernet | 112 MB/s | 112 MB/s | 112 MB/s |
| IPoIB | 153 MB/s | 357 MB/s | 1376 MB/s |

# MPI 및 OpenMP

## MPI

### 개요

MPI(Message Passing Interface)는 병렬 컴퓨팅 아키텍처에서 작동하도록 설계된 표준화되고 이식 가능한 메시지 전달 표준이다. 프로세스들 사이의 통신을 위해 코드에서 호출해 사용하는 서브루틴(Fortran) 또는 함수(C)들의 라이브러리이다. Fortran 또는 C로 작성된 메시지 패싱 프로그램들에게 순차 프로그램들처럼 다양한 아키텍처들에 대한 풍부한 소스코드 이식성(source-code portability)을 제공하고자 하는 표준화 작업의 결과이다.

### 목표

어떤 플랫폼상에서도 컴파일과 실행이 가능한 소스코드 이식성, 다양한 아키텍처에 걸쳐 효율적인 수행을 가능하게 하는 것, 메시지 패싱 프로그래밍을 위한 풍부한 기능 제공을 목표로 한다. 현재 MPI는 집합 연산, 사용자 정의 데이터 타입과 토폴로지, 다양한 방식의 통신 등의 다양한 기능들과 이기종 병렬 아키텍처(heterogeneous parallel architecture)에 대한 지원 등을 제공하며, MPI-2에서는 코드가 실행되는 동안 프로세스의 수를 변화시키는 동적 프로세스 관리, 원격 메모리 접근, 병렬 I/O 등을 지원하고 있다.

이 외에도 다음과 같은 MPI 목표가 있다.

* 응용 프로그래밍 인터페이스를 설계한다(컴파일러나 시스템 구현 라이브러리에 반드시 필요한 것은 아님).
* 효율적인 통신을 허용한다. 메모리에서 메모리로의 복사를 피하고 계산과 통신의 중첩을 허용한다. 가능한 경우 통신 보조 프로세서로 오프로드한다.
* 이기종 환경에서 사용할 수 있는 구현을 허용한다.
* 인터페이스에 대해 편리한 C와 Fortran 77 바인딩을 허용한다.
* 신뢰할 수 있는 통신 인터페이스를 가정한다. 통신에 대한 오류는 기본 통신 하위 시스템에서 처리되어 통신 실패에 대처할 필요가 없다.
* PVM, NX, Express, p4 등 현재 관행과 크게 다르지 않은 인터페이스를 정의하고 더 큰 유연성을 허용하는 확장을 제공한다.
* 기본 통신 및 시스템 소프트웨어를 크게 변경하지 않고 많은 공급업체의 플랫폼에서 구현할 수 있는 인터페이스를 정의한다.
* 인터페이스의 의미는 언어 독립적이어야 한다.
* 인터페이스는 스레드 안전성을 허용하도록 설계되어야 한다.

### 역사

1994년, 40개의 서로 다른 기구들을 대표하는 약 60명의 메시지 패싱 시스템 전문가들로 구성된 MPIF(MPI Forum)는 MPI-1 표준을 내놓았고, 1997년에 기존의 MPI-1에 병렬 I/O, C++와 Fortran90 지원, 동적 프로세스 관리 등의 도구를 추가한 MPI-2를 발표하였다. MPI-2에서 동적 프로세스 관리, 확장된 커뮤니케이션, 병렬 I/O, C++ and FORTRAN 90 bindings, 실시간 확장 기능을 추가했다. 현재 MPI를 구현하는 제품들은 다음과 같이 있다.

* MPI/Pro: MPI Software Technology implementation
* IBM MPI: IBM product implementation for the SP and RS/6000 workstation clusters
* MPICH: Argonne National Lab and Mississippi State University implementation
* UNIFY: Mississippi State University implementation
* CHIMP: Edinburgh Parallel Computing Centre implementation
* LAM: Ohio Supercomputer Center implementation

## MPI 기본 개념

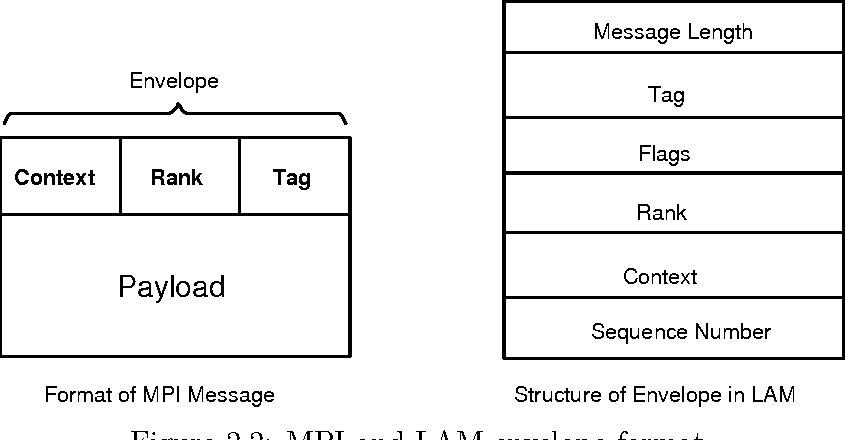
### 프로세스와 프로세서

실제 MPI 애플리케이션은 서로 데이터를 주고받는 여러 개의 프로세스들로 이루어져있다. 각각의 프로세스들은 MPU 애플리케이션이 구동될 때 병렬적으로 시작된다. 각각의 프로세스들은 하나의 문제 해결을 위해 다른 프로세스와 데이터를 주고 받는다. 이는 하드웨어적으로 분산되어 있는 컴퓨터 간에도 데이터를 주고 받을 수 있다는 장점이 있다. 이를 통해 인피니밴드와 같이 병렬처리를 위해 만들어진 병렬컴퓨터에서도 MPI 프로그램은 잘 동작한다.

스레드 기준으로 작업을 할당하는 OpenMP와 달리 MPI는 프로세스를 기준으로 작업을 할당한다. 일반적으로 한 개의 프로세서에서 여러 개의 프로세스가 실행(일대다)될 수 있으나, 프로그램의 최적화된 성능을 위해 하나의 프로세서에서 하나의 프로세스가 실행(일대일)되는 것이 바람직하다.

### 메시지

메시지는 데이터와 데이터의 송신자와 수신지 주소를 나타내는 봉투(Envelope)로 구성된다. 메시지 전송은 한 서브프로그램의 변수에서 다른 서브프로그램의 변수로 데이터가 이동하는 것이며, 메시지 패싱 시스템은 데이터의 값 자체보다 그 이동에만 관심이 있다. 일반적으로 메시지 전송 시스템이 메시지 전송을 확인하기 위해 어떤 프로세스가 메시지를 보내는가, 메시지를 보내는 프로세스의 어디에 데이터가 있는가, 보내는 데이터는 어떤 종류인가, 데이터의 양은 얼마나 되는가, 어떤 프로세스가 메시지를 받는가, 메시지를 받는 프로세스의 어느 위치에 데이터가 들어가는가, 얼마나 많은 양의 데이터를 받을 준비를 해야 하는가 같은 정보가 필요하다. 메시지를 받는 프로세스는 전달되는 데이터가 모두 도착해야 그 데이터를 사용할 수 있다. 보내는 프로세스도 보낸 메시지가 모두 도착했는지 알아야 할 경우가 있을 것이다. 따라서, 메시지 패싱 시스템은 메시지 전달뿐 아니라 통신에 대한 진행상황에 대한 정보도 제공해야 한다. 메시지 전송은 메시지 동기화에 대한 정보를 제공한다. 그림 24는 MPI 메시지의 포맷을 도식화한 것이다.



<그림 24> MPI 메시지 포맷

다음 표 10, 11은 MPI에서 메시지 송신을 위한 데이터 타입과 Fortran, C 유형에 대한 인수 값을 나타낸다.

<표 10> Fortran에 대한 데이터 타입

|  |  |
| --- | --- |
| **MPI datatype** | **Fortran datatype** |
| MPI\_INTEGER | INTEGER |
| MPI\_REAL | REAL |
| MPI\_DOUBLE\_PRECISION | DOUBLE PRICISION |
| MPI\_COMPLEX | COMPLEX |
| MPI\_LOGICAL | LOGICAL |
| MPI\_CHARACTER | CHARACTER(1) |
| MPI\_BYTE |  |
| MPI\_PACKED |  |

<표 11> C에 대한 데이터 타입

|  |  |
| --- | --- |
| **MPI datatype** | **C datatype** |
| MPI\_CHAR | signed char |
| MPI\_SHORT | signed short int |
| MPI\_INT | signed int |
| MPI\_LONG | signed long int |
| MPI\_UNSIGNED\_CHAR | unsigned char |
| MPI\_UNSIGNED\_SHORT | unsigned short int |
| MPI\_UNSIGNED | unsigned int |
| MPI\_UNSIGNED\_LONG | unsigned long int |
| MPI\_FLOAT | float |
| MPI\_DOUBLE | double |
| MPI\_LONG\_DOUBLE | long double |
| MPI\_BYTE |  |
| MPI\_PACKED |  |

### Tag

프로세스 간 주고받는 메시지를 구분하고 확인할 목적으로 각 메시지에는 태그를 붙인다. 예를 들어 1번 프로세스(송신)에서 2번 프로세스(수신)로 10개의 메시지를 보낸다면 태그를 이용해 각 메시지를 구분할 수 있으며, 수신 프로세스에서는 태그를 이용해 받는 메시지를 순서대로 처리할 수 있다. MPI는 와일드카드 태그 사용도 지원하고 있다. 태그 값의 범위는 0에서 MPI\_TAG\_UB를 포함한 반환된 값까지이다. 이 값은 MPI를 실행하는 동안 변경되지 않는다. 태그 상한 값은 적어도 32767이다.

### 커뮤니케이터(communicator)

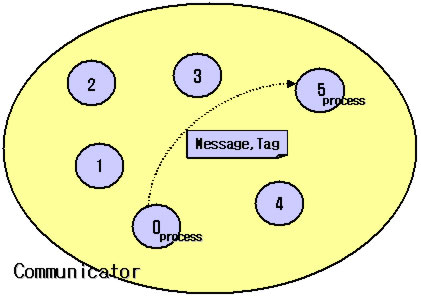
커뮤니케이터는 서로 통신할 수 있는 프로세스들의 집합을 나타낸다. MPI에서 통신은 같은 커뮤니케이터를 공유하는 즉, 같은 프로세스 그룹에 속한 프로세스들끼리 통신이 가능하다. 커뮤니케이터 이름은 모든 점대점 통신과 집합 통신에 인수로 필요하며, 송신과 그에 대응하는 수신은 커뮤니케이터가 일치해야 한다. 한 프로그램 내에 여러 커뮤니케이터가 있을 수 있고, 한 프로세스는 서로 다른 여러 커뮤니케이터에 속할 수 있다. MPI\_COMM\_WORLD는 MPI에서 기본적으로 제공하는 커뮤니케이터로 MPI 헤더파일에 정의되어 있으며, 병렬 작업에 참여하는 모든 프로세스들로 구성된다.

### 프로세스 랭크(rank)

커뮤니케이터 내에서 프로세스들은 0부터 시작되는 연속적인 정수들을 할당 받게 되는데, 이 정수들은 커뮤니케이터 내에서 프로세스를 구분 짓는 고유의 번호로서 프로세스 식별자 역할을 한다.

### 점대점(Point to Point) 통신

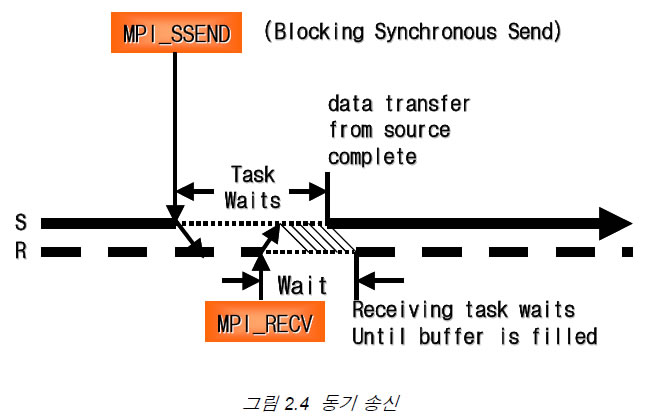
점대점 통신은 가장 간단한 메시지 전달 방식으로 MPI 통신의 기본이 된다. 그림 25와 같이 하나의 송신 프로세스로부터 다른 하나의 수신 프로세스로 메시지가 전송되며, 오직 두 개의 프로세스만이 그 메시지에 대해 알 필요가 있을 때 사용된다. 메시지 송신 특성, 송신과 프로그램 실행이 어떻게 상호작용하는가에 따라 몇 가지 통신 모드가 존재한다.



<그림 25> MPI의 메시지 패싱

점대점 통신의 송신 루틴은 4가지 통신모드로 구분할 수 있으며 각각 블록킹 루틴과 논블록킹 루틴으로 구분해 아래와 같이 정리할 수 있다.

* 블록킹 동기 송신 (Synchronous Send)

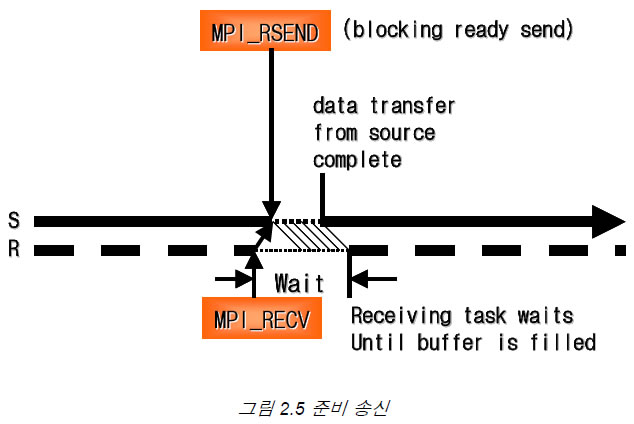


<그림 26> 동기 송신

그림 26에서 시간은 왼쪽에서 오른쪽으로 흐르며 S는 송신 프로세스가 송신 작업을 수행하는 시간을 나타내고, R는 수신 프로세스가 수신 작업을 수행하는 시간을 나타내고 있다. 송/수신 수행 시간을 나타내는 선이 끊어지며 나타나는 점선은 메시지 패싱 이벤트 때문에 발생한 인터럽트로 인해 각 프로세스가 대기 상태에 있음을 나타낸다.

MPI\_Ssend를 호출하면 송신 프로세스는 우선 송신 준비가 되었음을 수신 측에 알린다. 수신 프로세스는 수신 루틴을 호출하고 수신 준비가 되었음을 송신 측에 알리고 송신 프로세스가 수신 준비 완료 메시지를 받으면 데이터 전송이 시작된다. 데이터 전송에 소비되는 시스템 부하와 동기화 과정으로 인한 동기화 부하로 인해 성능은 느리지만 메시지가 전달되지 않거나 사라지는 상황을 방지할 수 있어 안전한 통신을 수행할 수 있다.

* 블록킹 준비 송신 (Ready Send)

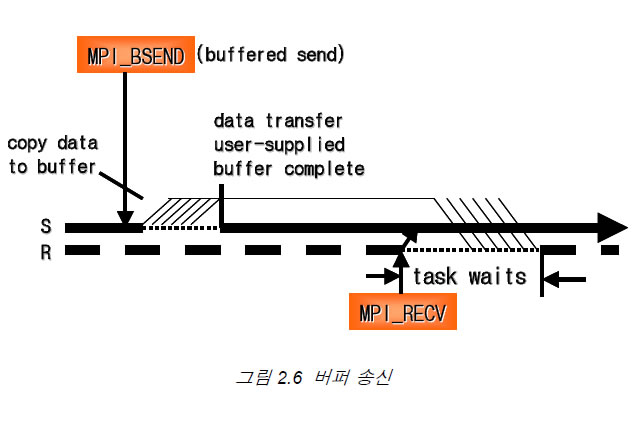


<그림 27> 준비 송신

준비 송신은 목적지 프로세스에 수신 루틴이 준비되어 있어야 한다. 송신 프로세스는 메시지를 통신 네트워크로 보내고 즉시 완료되며, 수신 프로세스가 메시지를 받기 위해 대기(ready) 하고 있음을 가정한다. 만약 수신 프로세스가 메시지를 받을 준비를 하고 있으면 무사히 수신될 것이고 그렇지 않으면 메시지는 사라지고 오류가 발생한다.

준비 송신은 송신 작업에 의한 시스템 부하와 동기화 부하를 최소화하기 때문에 성능 향상에 유리하다. 그러나 수신 측에서 대응되는 송신보다 얼마나 일찍 준비를 하고 있느냐에 의존하는 동기 부하는 여전히 남아있으며, 디버깅에 어려움이 있다.

* 블록킹 버퍼 송신 (Buffered Send)



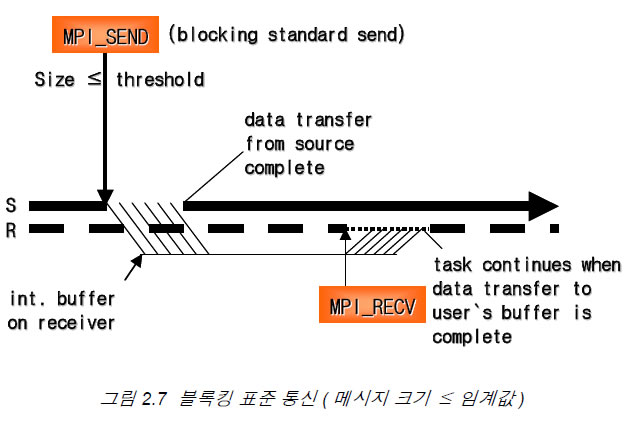
<그림 28> 버퍼 송신

MPI는 송신될 때까지 데이터를 보관할 수 있는 버퍼를 사용자가 지정해 쓸 수 있도록 하는 기능을 제공한다. 버퍼 송신은 버퍼 송신 루틴(MPI\_BSend)이 호출되면 즉시 유저가 정의한 버퍼로 데이터를 복사해 두고 수신 준비가 완료되면 이 버퍼로부터 수신 프로세스로 송신을 하게 된다. 송신 프로세스는 준비된 버퍼로 데이터 복사가 완료되면 송신 버퍼를 다른 목적으로 사용할 수 있다. 버퍼 송신은 송신 버퍼에서 사용자 정의 버퍼로의 작업으로 인해 추가적인 시스템 부하가 생기지만 송신 측의 동기화 부하는 없어진다. 수신 작업에 의한 수신 측의 동기화 부하는 여전하며 사용자는 버퍼 공간 관리를 직접 해야 한다. 사용자는 MPI 루틴 MPI\_BUFFER\_ATTACH와 MPI\_BUFFER\_DETACH를 이용해 통신에 이용할 버퍼를 확보하고 해제 시킬 수 있다.

* 블록킹 표준 송신 (Standard Send)

블록킹 표준 모드 송신은 메시지 크기에 따라 두 가지 모드로 작동한다. 메시지 크기가 임계값보다 작으면 시스템에 준비된 시스템 버퍼를 이용하여 버퍼 송신과 같이 작동하며, 임계값보다 메시지 크기가 크면 동기 송신처럼 작동하게 된다. 임계값은 시스템 상황에 따라 다르게 설정할 수 있으며 참고로, IBM의 MPI 라이브러리 구현 제품인 PE에서는 환경 변수 MP\_EAGER\_LIMIT을 이용해 임계값 설정을 할 수 있다.

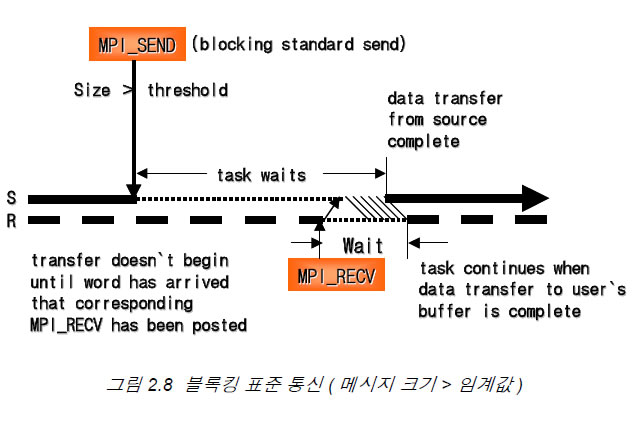
표준 모드 송신이 두 가지 모드로 작동되는 이유는 시스템 상황에 따라 통신의 안전성과 성능을 고려하여 최적화된 통신을 하기 위한 것이다. 표준 송신 모드는 메시지 크기에 따라 작동 방식이 다르기 때문에 사용자는 송신 완료가 메시지 도착을 의미하지 않는다는 사실을 유념해야 한다.



<그림 29> 블록킹 표준 통신(메시지 크기 임계값)

보내고자 하는 메시지 크기가 시스템 임계값보다 작을 때 루틴 MPI\_Send를 호출한 수신 노드는 네트워크를 거쳐 수신 노드의 시스템 버퍼로 메시지를 우선 복사한다. 그리고 곧바로 리턴되어 다른 작업을 수행할 수 있다. 수신 루틴 MPI\_RECV가 호출되면 시스템 버퍼의 메시지는 수신 프로세스로 복사되고 복사가 완료되면 수신 프로세스는 다른 작업을 수행할 수 있다.

시스템 버퍼를 사용하여 송신 측에서 동기화 부하를 없애는 것은 버퍼 송신과 동일하나 표준모드에서 사용되는 시스템 버퍼는 프로그램 시작 시에 프로세스마다 하나씩 부가되어 사용자의 관리가 따로 필요 없다.

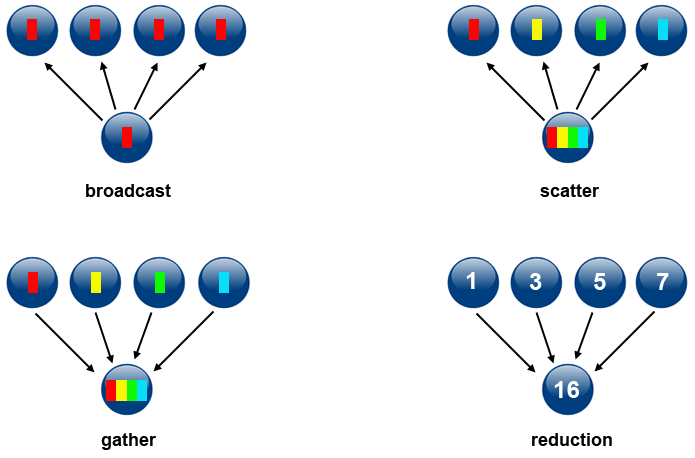


<그림 30> 블록킹 표준 통신(메시지 크기 임계값)

메시지 크기가 임계값보다 클 때 표준 모드 송신은 동기 모드 송신과 동일하게 행동한다. 사용자는 임계값 설정을 0으로 둠으로써 모든 표준 모드 송신을 가장 안전한 동기 송신과 똑같이 행동하도록 할 수 있다.

### 집합 통신(Collective Communication)

메시지 패싱 시스템들은 한 쌍의 프로세스만이 참여하는 점대점(Point to Point) 통신뿐 아니라 한 무리의 프로세스들이 동시에 통신에 참여하는 다양한 집합 통신을 제공한다. 일대다(one-to-many), 다대일(many-to-one) 형식과 같은 여러 가지 통신 방법이 있는데, 집합 통신 루틴을 호출하는 한 줄의 코드가 여러 번의 점대점 호출을 대신하여 오류의 가능성이 줄어들고, 코드를 좀 더 쉽게 읽을 수 있어 디버깅 작업 등이 간단해지는 장점이 있다. 최적화된 집합 통신 루틴의 호출은 같은 동작을 하도록 하는 점대점 루틴들의 호출보다 일반적으로 빠르다.



<그림 31> 집합 통신 루틴

그림 31의 Broadcast는 MPI\_Bcast로 사용하며 루트 프로세스의 메모리에 있는 데이터를 동일 커뮤니케이터 내의 다른 모든 프로세스들의 메모리의 같은 위치로 복사하는 일대다(one-to-all) 통신이다. Scatter는 MPI\_Scatter로 사용하는 일대다 통신이다. MPI\_Scatter가 호출되면 루트 프로세스는 데이터를 같은 크기로 나누어 동일 커뮤니케이터 내의 각 프로세스에 랭크 순서대로 하나씩 전달한다. 처음 sendcount 개의 데이터가 프로세스 0으로 다음 sendcount 개 데이터가 프로세스 1로 전달되는 식이다. Gather는 MPI\_Gather로 사용하며 커뮤니케이터에 있는 모든 프로세스들로부터 하나의 수신 프로세스(루트 프로세스)로 데이터를 전송하는 다대일 통신이다. MPI\_Gather 루틴이 호출되면, 루트 프로세스를 포함한 모든 프로세스는 각자의 송신 버퍼에 있는 데이터를 루트 프로세스로 보내고, 루트 프로세스는 데이터를 받아 순서대로 메모리에 저장한다. 이때, 각 gather된 데이터의 크기는 모두 같아야 한다. Reduction은 MPI\_Reduce로 사용되며 각 프로세스들로부터 데이터를 모아 하나의 값으로 환산(reduce) 하여 그 결과를 루트 프로세스에 저장한다.

## OpenMP

### 개요

멀티 스레드 기반 공유 메모리 병렬 프로그램을 위한 표준 API로, C, C++, Fortran 언어와 유닉스 및 마이크로소프트 윈도우 플랫폼을 비롯한 여러 플랫폼을 지원한다. 공유 메모리 아키텍처 환경에서 사용한다. Open MP는 프로그래머에게 간단하고 유용한 인터페이스를 만들기 위해 portable하고 확장성이 있는 모델을 사용한다. 병렬 프로그래밍의 하이브리드 모델로 작성된 응용 프로그램은 OpenMP와 MPI(Message Passing Interface)를 둘 다 사용하거나, 더 투명성 있는 방식으로 비공유 메모리 시스템을 위한 OpenMP 확장을 사용하여 컴퓨터 클러스터 상에서 구동할 수 있다.

### 역사

OpenMP Architecture Review Board(ARB)는 1997년 10월에 첫 API인 OpenMP fortran 1.0을 발표했다. 이듬해 10월에 C/C++ 표준을 발표했다. 2000년에는 Fortran 버전 2.0이 출시되었고 2002년에는 C/C++ 버전 2.0이 출시되었다. 버전 2.5는 2005년 릴리스된 C/C++/Fortran specification이다.

버전 2.0까지 OpenMP는 기본적으로 매우 규칙적인 루프를 병렬화하는 방법을 지정했다. 이는 진입 시 루프의 반복 횟수를 알고 있는 행렬 지향 수치 프로그래밍에서 발생하기 때문이다. 이는 한계로 인식되어 다양한 병렬 확장 작업이 구현하는 데에 추가되었다. 2005년에는 작업 병렬 처리를 표준화하기 위한 노력이 이루어졌으며 Cilk, X10, Chapel의 작업 병렬 처리 기능에서 영감을 받아 2007년에 제안이 발표되었다.

버전 3.0은 2008년 5월에 릴리스되었다. 3.0에서는 작업 개념과 작업 구성이 포함되어 OpenMP 2.0의 대부분을 구성하는 병렬 루프 구성을 넘어 OpenMP의 범위를 확장했다.

버전 4.0은 2013년 7월에 릴리스되었다. 가속기 지원, 오류 처리, 스레드 친화성, 작업 확장, 사용자 정의 감소, SIMd지원, Fortran 2003을 지원한다.

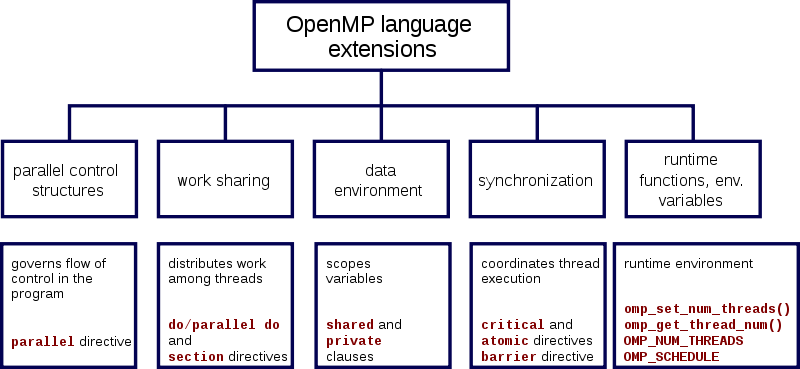
현재 버전은 2020년 11월에 릴리스된 5.1이다. 모든 컴파일러(및 OS)가 최신 버전의 전체 기능을 모두 지원하는 것은 아니다.

### 설계

기본 스레드(연속적으로 실행되는 명령어)가 지정된 수의 하위 스레드를 분기하고 시스템이 이들 사이에서 작업을 분할하는 멀티스레딩 구현이다. 그런 다음 스레드는 다른 프로세서에서 스레드를 할당하는 런타임 환경과 함께 동시에 실행된다. 병렬로 실행되는 코드는 실행되기 전에 스레드가 만들어지도록 하는 지시문과 함께 표시된다. 각 스레드에는 함수(omp\_get\_thread\_num())를 사용하여 얻을 수 있는 연결된 id가 있다. 스레드 id는 정수이고 기본 스레드는 0인 id를 갖는다. 병렬화된 코드를 실행한 후 스레드는 기본 스레드에 다시 결합되어 프로그램이 끝날 때까지 계속된다. 기본적으로 각 스레드는 병렬화된 코드 섹션을 독립적으로 실행한다. 작업 공유 구조는 각 스레드가 할당된 코드 부분을 실행하도록 스레드 간에 작업을 나누는데 사용할 수 있다. 이러한 방식으로 OpenMP를 사용하여 작업 병렬 처리와 데이터 병렬 처리를 모두 달성할 수 있다. 런타임 환경은 사용량, 머신 로드 및 기타 요인에 따라 프로세서에 프로세스를 할당한다. 런타임 환경은 환경 변수를 기반으로 스레드 수를 할당하거나 코드에 함수를 사용하여 할당할 수 있다. OpenMP 함수는 C/C++에서 omp.h 헤더 파일에 포함되어있다.

### 핵심 요소

OpenMP의 핵심 요소는 스레드 생성, 워크로드 분배(작업 공유), 데이터 환경 관리, 스레드 동기화, 사용자 레벨 런타임 루틴 및 환경 변수에 대한 구성이다. 그림 32는 OpenMP의 구성도를 나타낸 것이다.



<그림 32> OpenMP 구성 차트

Parallel region은 “omp parallel” pragma를 사용해 스레드를 만든다. 각 스레드는 구성된 블록에서 코드의 복사본을 실행한다. 런타임 함수는 특정 스레드 수를 요청하고 스레드 id를 리턴한다. 각 스레드는 동일한 코드를 중복 실행한다. 각 스레드는 모든 스레드 자원을 공유한다 스레드는 barrier로 모든 스레드가 완료되기 전까지 대기한다.

Work sharing은 스레드 간에 작업을 분할한다. 예를 들어 “for” loop를 work-sharing 한다면 스레드 간에 루프 반복을 분할한다. Sections work-sharing은 각 스레드에 다른 구조화된 블록을 제공한다.

Data environment는 기본 저장 속성을 나타낸다. 공유 메모리 프로그래밍 모델의 변수는 기본적으로 공유된다. 전역 변수는 스레드 간에 공유된다. Fortran은 COMMON 블록, SAVE 변수, MODULE 변수가 있고 C는 File scope 변수, static 변수가 해당된다. 하지만 모든 것이 공유되는 것은 아니다. 병렬 영역에서 호출되는 하위 프로그램의 스택 변수는 private이고 명령문 블록 내의 자동 변수는 private이다. SHARED, PRIVATE, FIRSTPRIVATE, THREADPRIVATE 절을 사용하여 스토리지 속성을 선택적으로 변경할 수 있다. 병렬 루프 내부의 private 값은 LASTPRIVATE 절을 사용하여 루프 외부의 값으로 전송할 수 있다. 기본 상태는 DEFAULT(PRIVATE | SHARED | NONE)으로 수정할 수 있다.

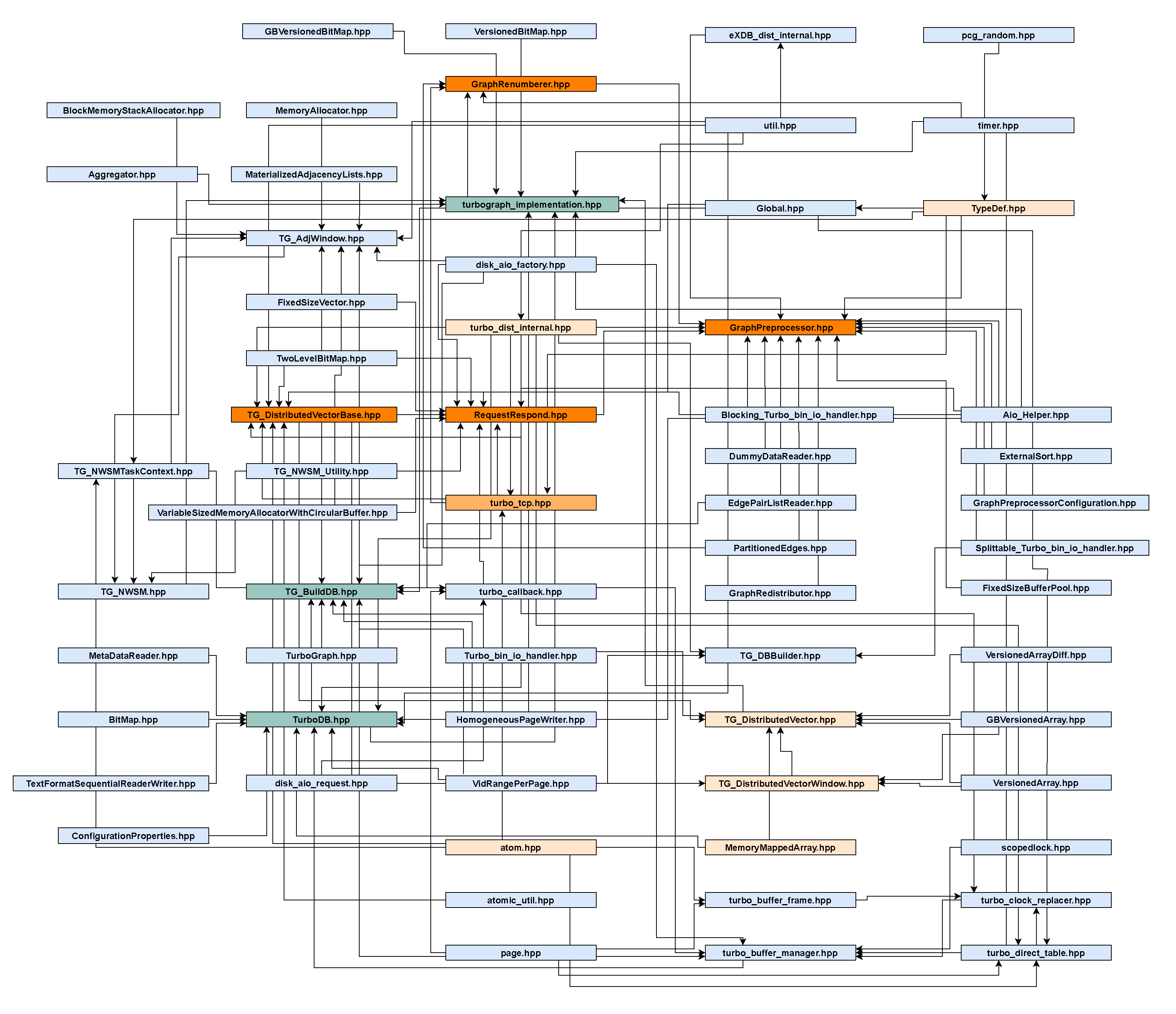
Synchronization은 동기화를 지원하기 위한 여러 구성을 나타낸다. Critical section, atomic, barrier, flush, ordered, single, master가 있다. Runtime functions/env. Variables는 실행 환경에서 라이브러리 루틴을 나타낸다. 스레드 수를 수정하거나 확인, parallel region에 있는지 확인, 시스템의 몇 개의 프로세스가 있는지 확인하는 루틴이 있다.

## Hybrid Model

MPI와 OpenMP 프로그래밍 모델을 하이브리드 패러다임으로 결합하여 병렬성을 활용할 수 있다. 하이브리드 모델의 주요 목표는 컴파일러 지시문에 의한 loop level에서 domain decomposition과 fine grain 병렬 처리에서 얻어지는 process level coarse-grain 병렬 처리를 결합하는 것이다. 하이브리드 접근 방식은 노드가 병렬 처리에 MPI가 필요하고 노드 내 loop level 병렬 처리를 활용하는 데 OpenMP를 사용할 수 있는 SMP 노드 클러스터에 적합하다.

# TurboGraph++ 네트워크/메시지 전송 구조 분석

TurboGraph++는 지금까지 설명한 MPI와 OpenMP 프로그래밍 모델을 하이브리드로 사용하는 병렬 구조로, 그래프 처리 고속화가 가능한 고성능 그래프 데이터베이스이다. TurboGraph++는 총 178개의 헤더 파일, 27개의 소스 파일로 구성된 C++ 프로젝트로, 이 중 코어 헤더 파일의 참조 관계는 다음 그림과 같다. 그림에서, 주황색과 녹색으로 표시된 헤더들이 TurboGraph++의 외부 통신, 즉 분산 노드간 통신과 관련된 헤더로, 본 설계서에서는 이를 중심으로 그래프 처리 구조를 분석하고, 새로운 IPoIB/RDMA 기반 네트워크 매니저를 설계한다.



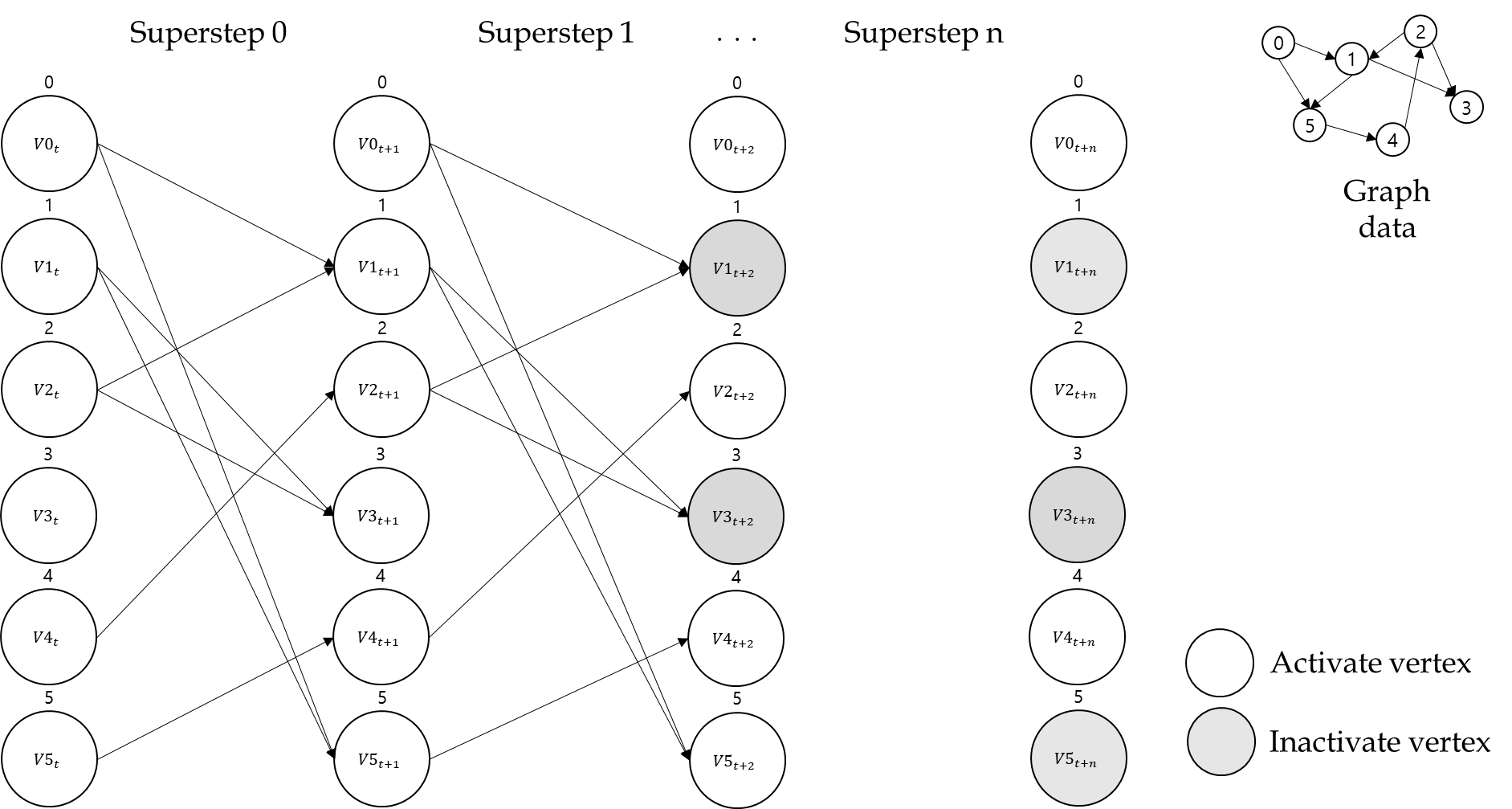
# IPoIB/RDMA 기반 네트워크 매니저

## 개요

그래프 데이터는 정점(vertex)과 간선(edge)으로 이루어져 있으며, 트위터, 페이스북과 같은 소셜 네트워크 데이터, 통신망과 같은 네트워크 데이터를 표현하는 데 주로 사용된다. 이 같은 대용량 그래프 데이터를 단일 노드 기반 시스템으로 처리하는데 한계가 있다. 이를 해결하기 위해, 분산 그래프 처리 시스템이 연구되고 있으며 대표적으로 Pregel, Apache Giraph, PowerGraph 등이 있다.

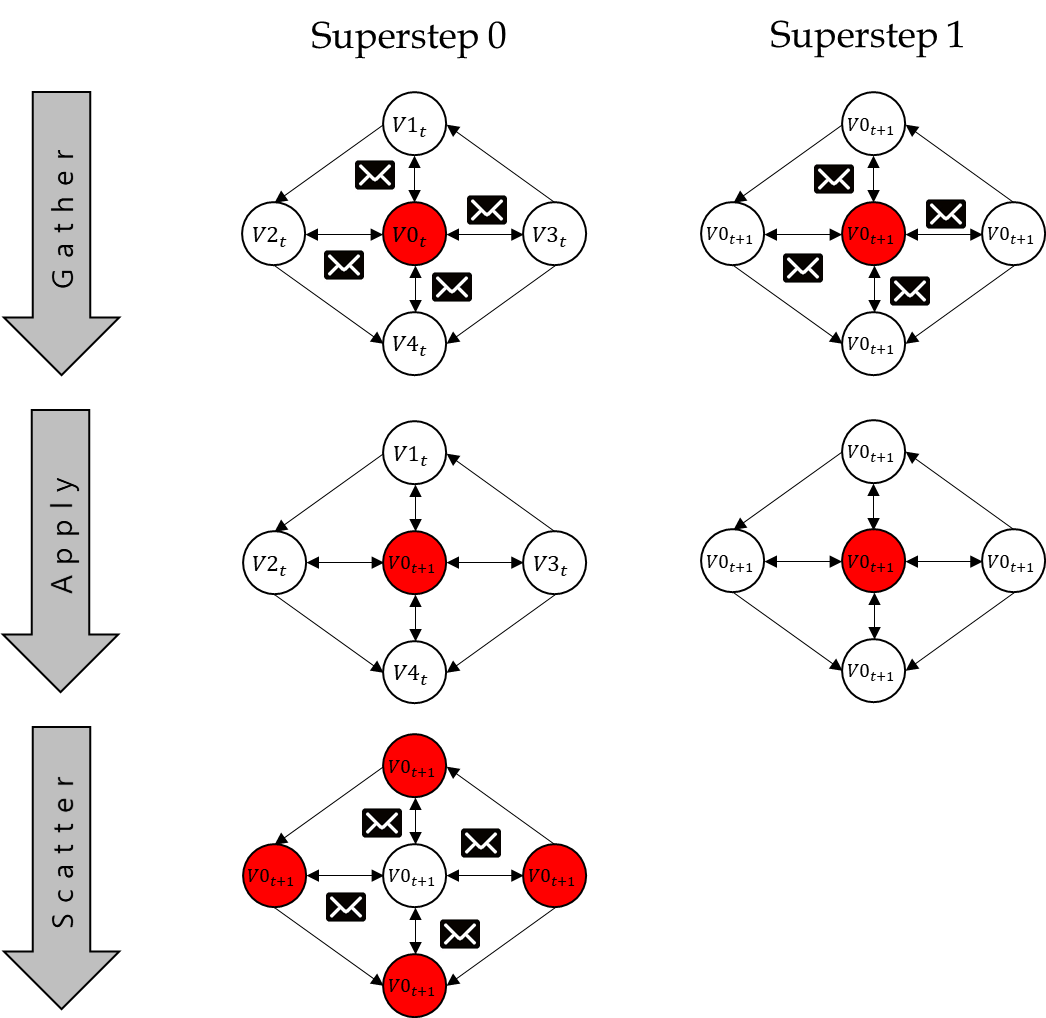
분산 그래프 처리 시스템은 노드 간 네트워크 통신을 이용하여 사용자가 요구한 질의를 수행한다. 이 때, 네트워크 통신 과정에서 네트워크 유량이 증가함에 따라 오버헤드도 증가한다. 또한, 정확한 질의 결과를 얻기 위해 노드 간의 송수신되는 메시지의 신뢰성도 보장되어야 한다. 따라서, 분산 그래프 처리 시스템에서 네트워크 매니저는 질의 대상이 되는 데이터와 네트워크 환경에 따라 통신 오버헤드를 줄이기 위해 적절한 메시지 블록 크기를 지원할 수 있어야 하며, 송수신 되는 메시지의 신뢰성 또한 보장되어야 한다. 결과적으로 앞선 요구사항을 만족하면서 네트워크 통신을 수행하고 관리하는 네트워크 매니저를 설계할 필요성이 있다.

대부분의 분산 그래프 처리 시스템은 vertex-centric 방식과 GAS(Gather-Apply-Scatter) 방식으로 그래프 알고리즘을 처리한다. Vertex-centric은 그래프 데이터의 분산 처리를 위한 프로그래밍 모델로, 그림 33과 같이 각 정점마다 독립적으로 연산을 수행한다. 그래프 처리 연산을 독립적으로 수행함으로써 병렬 처리가 가능하며, 대용량 그래프를 빠른 속도로 처리할 수 있다. 하지만 이웃 정점들 간에 통신을 통해 값을 교환하면서 반복적으로 알고리즘을 수행하기 때문에, 네트워크 처리 성능이 중요하다.



<그림 33> Vertex-centric 모델의 동작 구조

현실 세계의 그래프는 상대적으로 소수의 정점에 많은 정점이 이웃하는 power-law 그래프이다. 이러한 power-law 그래프에 vertex-centric 모델을 적용하여 처리한다면, 정점간 불필요한 통신이 발생할 수 있다. GAS 방식은 power-law 그래프에 vertex-centric 모델을 적용했을 때 발생할 수 있는 문제점을 해결하기 위해 제시된 프로그래밍 모델로, 그림 34와 같이 한 정점에 이웃하는 정점의 값을 모아 연산을 수행하여, vertex-centric에 비해 상대적으로 네트워크 유량이 적다. 하지만, 중앙 정점의 선정과 특정 파티셔닝을 사용하기 때문에 추가적인 연산이 필요하다.

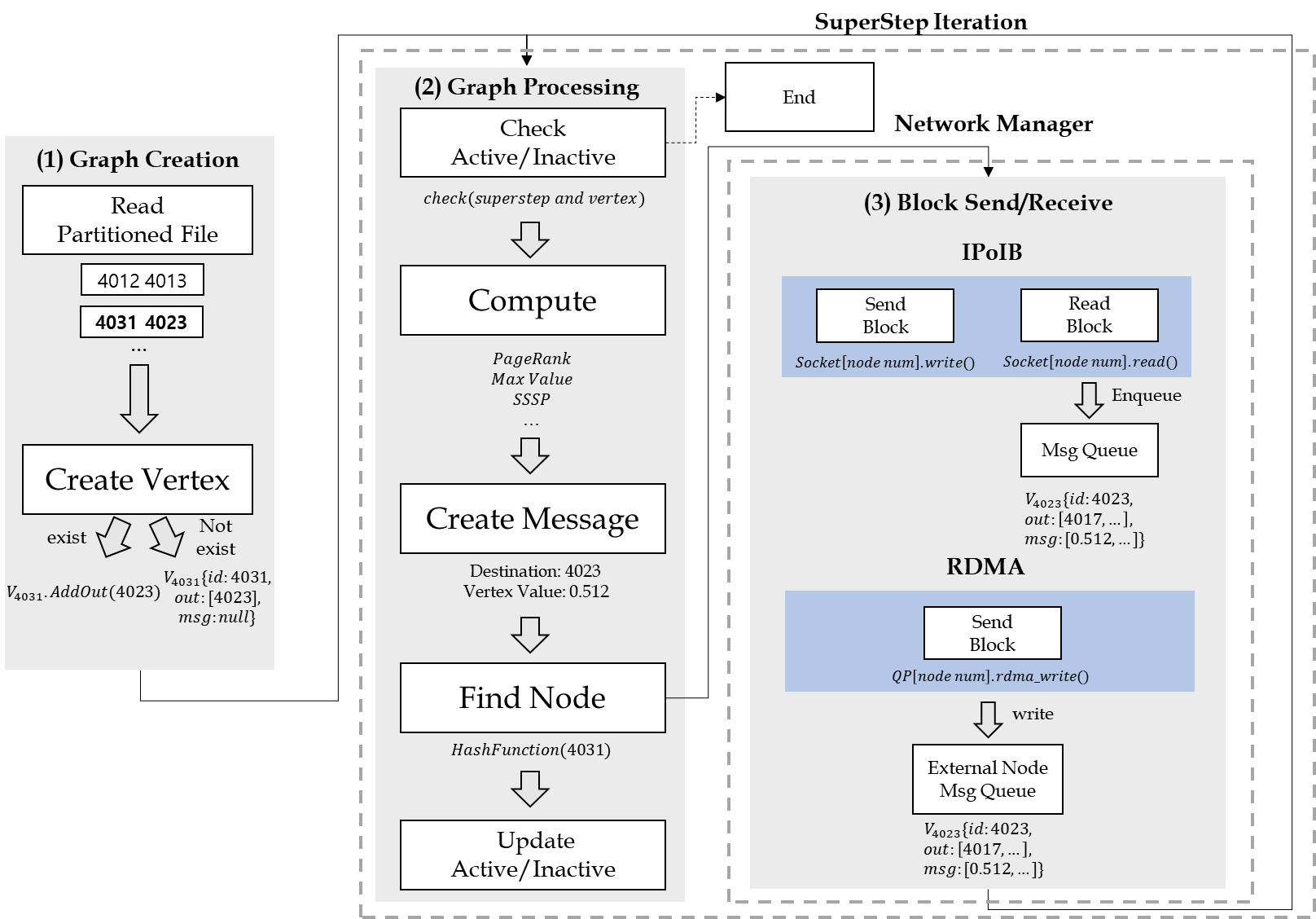


<그림 34> GAS 모델의 동작 구조

본 장에서는 TurboGraph++의 IPoIB/RDMA 기반 네트워크 매니저를 설계하기 위해 상대적으로 네트워크 유량이 많아 네트워크 환경에 따른 성능 비교에 적합한 vertex-centric 분산 그래프 처리 시스템에서의 네트워크 매니저 프로토타입을 설계한다. 4.2절에서는 IPoIB와 RDMA를 사용하는 vertex-centric 방식의 분산 그래프 처리 시스템을 설계한다. 4.3절과 4.4절에서는 각각 IpoIB와 RDMA를 위한 네트워크 매니저 프로토타입을 설명한다.

## Vertex-centric 분산 그래프 처리 시스템

그림 35는 vertex-centric 분산 그래프 처리 시스템의 구조를 나타낸다. 시스템은 (1) 그래프 생성, (2) 그래프 처리, (3) 블록 송수신 단계로 구성된다. 먼저, 그래프 생성 단계에서 각 노드는 간선 정보가 담긴 파일을 읽는다. 이 때, 파일의 그래프 데이터는 해시 함수를 통해 파티셔닝 되어 있는 상태이다. 이후 출발 정점을 id로 하는 Vertex 클래스가 생성된다. Vertex 클래스는 식별자인 *id*, 목적지 정점인 *out*, 정보를 담을 *msg*로 구성된다. 모든 데이터에 대한 Vertex 클래스 생성이 끝나면 그래프 생성이 완료된다.



<그림 35> Vertex-centric 그래프 처리 시스템 구조도

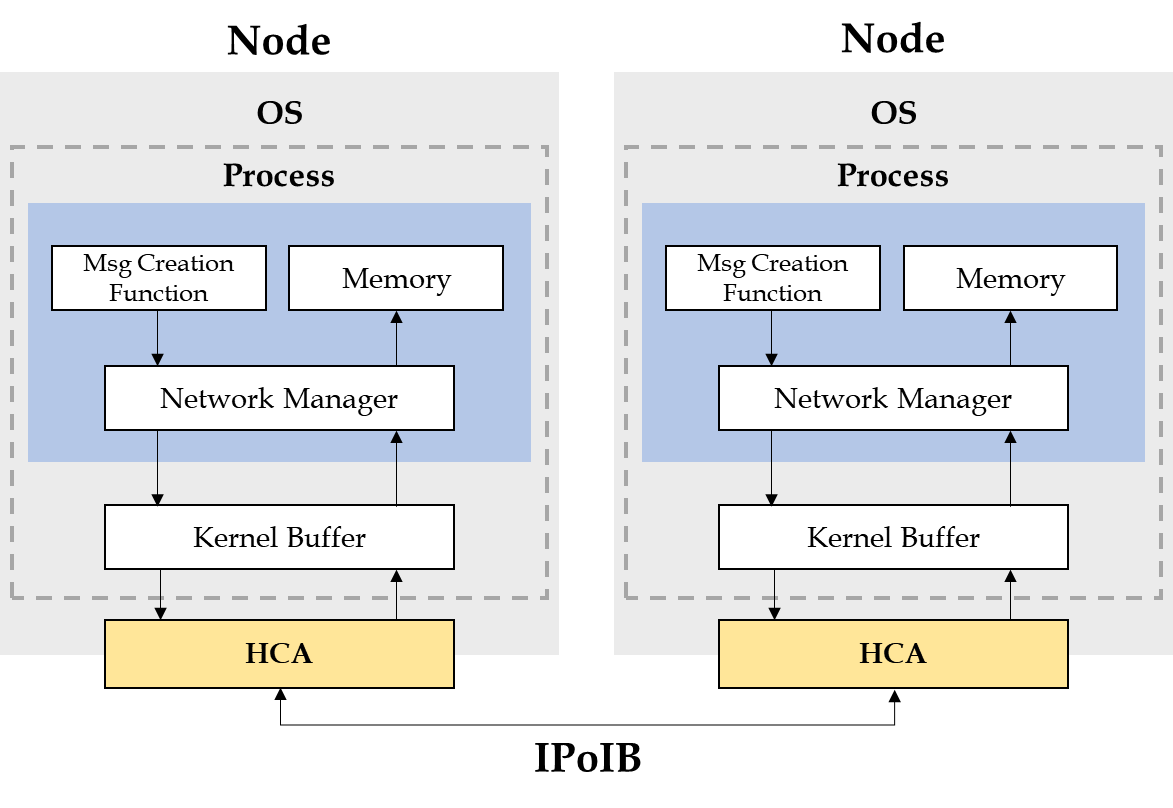
그래프 처리 단계는 그래프 쿼리 수행, 메시지 생성, 목적지 노드 검색, 활성/비활성화 갱신의 세부 단계로 구성된다. 그래프 쿼리 수행 단계에서 각 정점은 superstep의 진행여부를 결정하고, 병렬적으로 그래프 쿼리를 수행한다. 만약, superstep이 처음 수행되는 경우라면 각 정점은 초기값만 계산한다. 그렇지 않은 경우, 수신된 메시지와 vertex 값을 종합하여 쿼리를 수행한다. 메시지 생성 단계는 목적지 정점에게 전달할 메시지를 생성한다. 메시지는 목적지 *id*와 *value* 값을 가진다. 메시지 생성이 완료되면 목적지 노드 검색 단계에서 목적지 정점이 있는 노드를 탐색하는데, 여기서는 해시 함수를 사용한다. 이후, 생성된 메시지는 네트워크 매니저에 의해 실행되는 블록 송수신 단계로 전달한다. 메시지 전달 이후, 각 정점은 활성/비활성화 갱신 단계를 통해 다음 superstep에서 자신의 상태를 결정한다.

마지막으로 블록 송수신 단계이다. 본 시스템에서의 메시지 송수신은 블록 단위로 이루어진다. 송신 단계에서는 메시지가 블록 사이즈만큼 쌓일 때까지 대기한 후 목적지로 전송한다. 수신 단계 역시 블록 사이즈만큼 수신하고, 마지막 메시지 블록이 도착하면 종료한다. 이때, 수신하는 블록 안의 메시지는 각 vertex 클래스의 메시지 큐(*msg*)에 들어간다. 블록 송수신 단계는 통신 속도 향상을 위해 동시에 수행된다. IPoIB 네트워크를 사용할 경우 블록 송신 이후 수신을 시작하면 송수신을 위한 버퍼에 대기가 발생하고, 처리 시간이 증가한다. 이러한 문제를 해결하고자 블록 송수신을 동시에 수행하여 버퍼 대기 시간을 최소화한다.

## IPoIB 네트워크 매니저 프로토타입

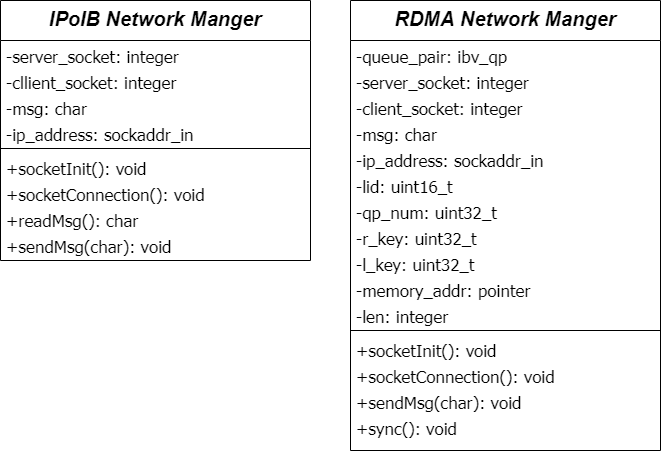
IPoIB 네트워크에서 다른 노드에 메시지를 송신할 때 프로세스는 송신 메시지를 커널 버퍼에 복사해야 한다. 이와 반대로, 메시지를 수신할 때는 커널 버퍼에 있는 수신 메시지를 프로세스에 할당된 메모리로 복사해야 한다. 따라서, IPoIB 네트워크 매니저는 생성된 메시지를 커널 버퍼에 복사하는 기능과 커널 버퍼에 있는 메시지를 프로세스에 할당된 메모리에 복사하는 기능이 필요하다.

그림 36은 IPoIB 네트워크 매니저를 사용하는 그래프 분산 시스템의 네트워크 송수신 순서이다. 그림을 보면, 송신 단계에서 네트워크 매니저는 메시지 생성 함수에서 받은 메시지를 커널 버퍼에 보내고, OS는 커널 버퍼의 메시지를 InfiniBand의 HCA(Host Channel Adapter)에게 외부 노드로 송신을 요청하는 것을 알 수 있다. 또한, 수신 단계에서 네트워크 매니저는 HCA에 의해 커널 버퍼에 올라간 외부 메시지를 읽어와 프로세스에 할당된 메모리에 넣는 것을 알 수 있다.



<그림 36> IPoIB 네트워크 송수신 순서

그림 37은 IPoIB 네트워크 매니저 클래스 다이어그램이다. IPoIB 네트워크 매니저 클래스를 보면, 메시지 송수신을 담당하는 sendMsg와 readMsg 함수 외에 소켓을 초기화 하는 socketInit과 socket을 연결하는 socketConnection 함수가 있는 것을 확인할 수 있다. IPoIB 네트워크를 이용하기 위해서는 소켓 사용이 필수적이다. 따라서, IPoIB 네트워크 매니저는 메시지 송수신을 담당하는 함수 외에 추가적으로 소켓 초기화 함수와 클러스터에 참여하는 노드의 소켓 간의 연결을 담당하는 함수가 필요하다.

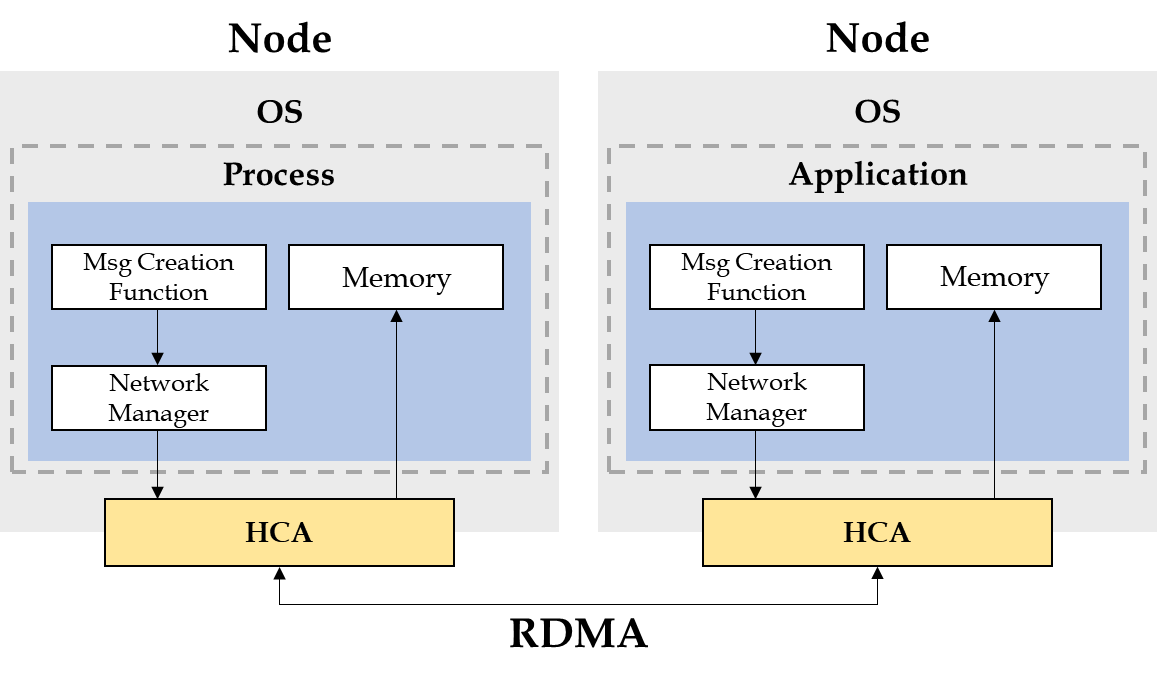


<그림 37> IPoIB 네트워크 매니저 클래스 다이어그램

## RDMA 네트워크 매니저 프로토타입

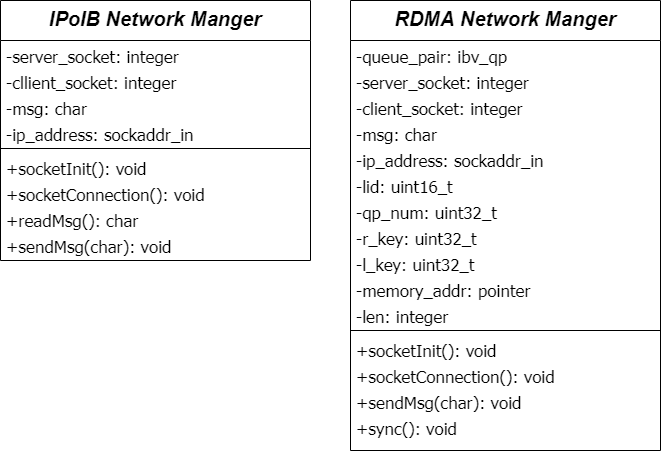
RDMA 네트워크를 이용한 통신은 IPoIB와 다르게 커널 버퍼를 사용하지 않는다. RDMA 네트워크 통신은 다른 노드로 메시지를 송신할 때 노드의 프로세스에 할당된 메모리에 직접 전달한다. 메시지를 수신할 때는 IPoIB와 다르게 특별한 동작이 필요하지 않고 다른 노드로부터 메시지가 전달된 메모리에서 읽으면 완료된다. 따라서 RDMA 네트워크 매니저는 기능은 외부 노드의 프로세스에 할당된 메모리에 메시지를 직접 전달할 수 있는 기능만 있으면 된다.

그림 38은 RDMA 네트워크 매니저를 사용하는 그래프 분산 시스템의 네트워크 송수신 순서이다. 그림을 보면, 송신 단계에서 네트워크 매니저는 메시지 생성 함수에서 받은 메시지를 HCA에게 외부 노드의 프로세스에 할당된 메모리에 송신을 요청한다는 것을 알 수 있다. 그러나, IPoIB의 네트워크 매니저와 다르게 메시지 수신에 대해 특별한 동작을 하지 않은 것도 알 수 있다. 이와 같이 RDMA를 위한 네트워크 매니저는 메시지 생성 함수에서 받은 메시지를 HCA에게 외부 노드의 프로세스에 할당된 특정 메모리에 메시지를 송신하는 한가지 기능만 필요하다



<그림 38> RDMA 네트워크 송수신 순서

그림 39는 RDMA 네트워크 매니저 클래스 다이어그램이다. RDMA 네트워크 매니저 클래스를 보면, 메시지 송신을 담당하는 sendMsg 함수 외에 소켓의 초기화를 담당하는 socketInit 함수, socket 연결을 담당하는 socketConnection 함수, 클러스터 노드간 동기화 과정을 담당하는 sync 함수가 있는 것을 확인할 수 있다. RDMA 통신을 시작하기 전 클러스터에 참여하는 각 노드는 자신 이외 노드들의 LID(Local Identifier), QPN(Queue Pair Number), 메모리 주소, rKey(Remote Key), lkey(Local Key)가 필요하다. 이를 위해 각 노드들의 RDMA 네트워크 매니저는 sync 함수를 이용하여 TCP 통신을 통해 필요한 정보를 교환한다. 따라서, IPoIB와 같이 socket 초기화 과정과 연결 과정을 담당하는 함수가 필요하다.



<그림 39> RDMA 네트워크 매니저 클래스 다이어그램

# RDMA 다대다 통신 라이브러리

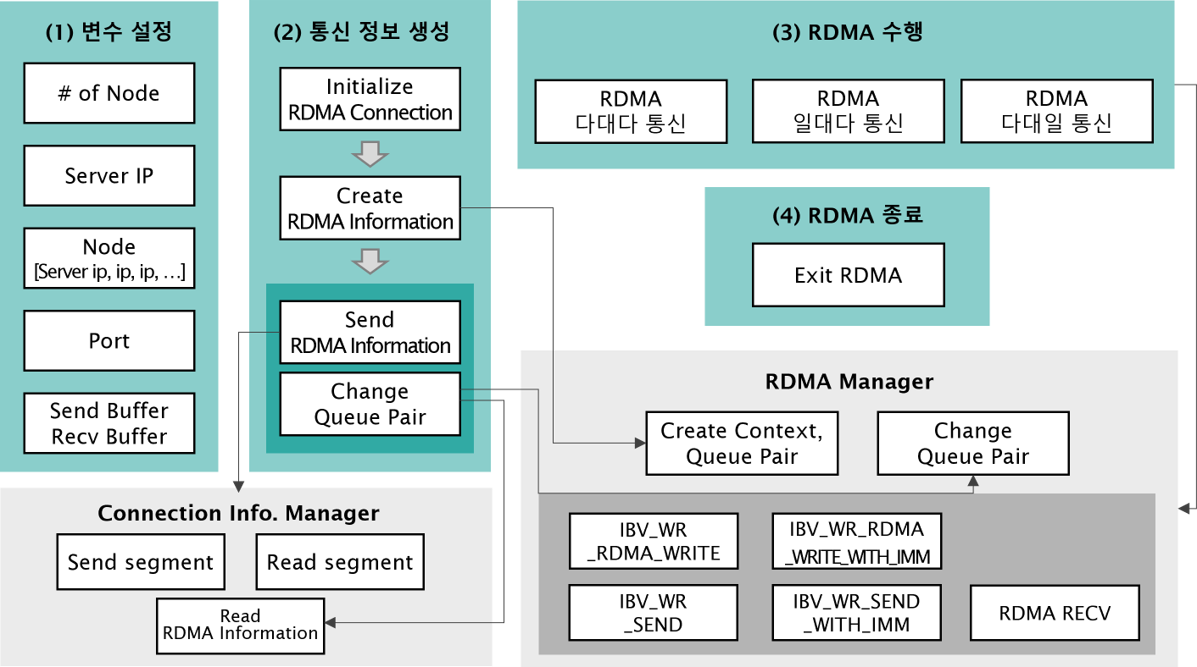
## 개요

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.

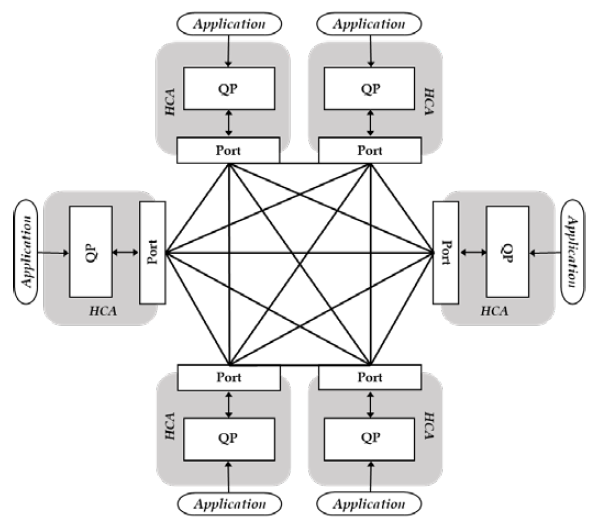
## 일반화된 RDMA 통신 모델 설계

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

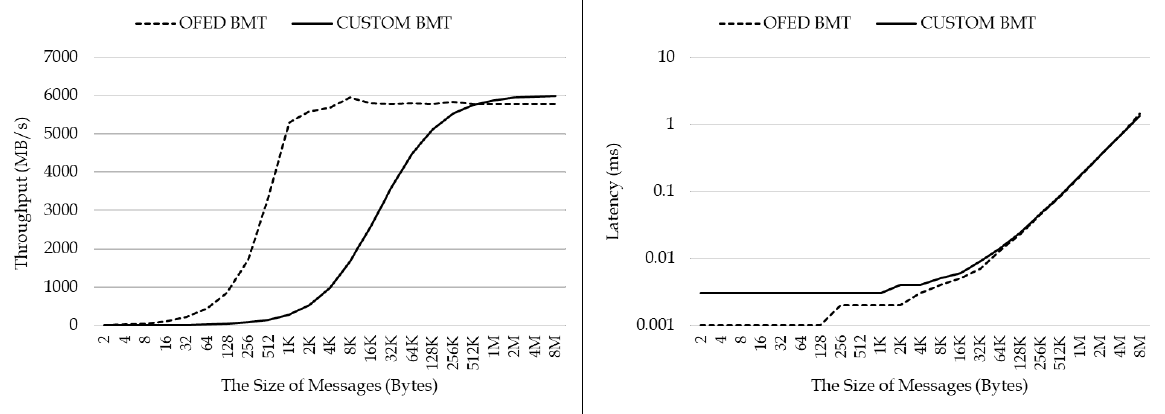
설계한 통신 모델에서 모든 노드는 송신과 수신을 함께 수행하기 때문에 각 노드는 서버와 클라이언트 역할을 동시에 담당한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 또한, 모든 노드가 서로 완전히 연결된 매시 토폴로지 형태로 동작한다. 이는 복잡한 네트워크 환경을 가정하여 RDMA 통신 모델을 일반화하기 위함이다. 그림 41은 설계한 일반화된 RDMA 통신 모델 중 참여 노드 수가 6대 일 때의 토폴로지 상태를 나타낸다. 실험에서는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

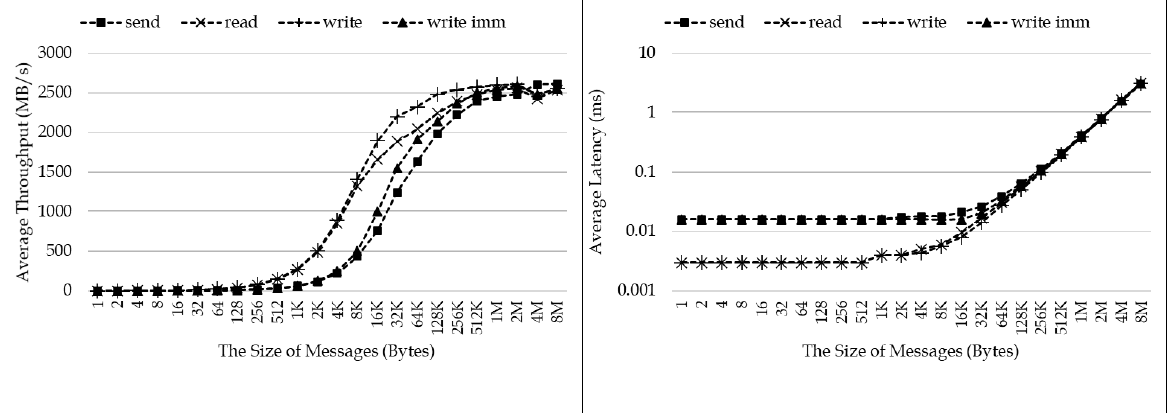


<그림 42> OFED 벤치마크 툴과 제안하는 모델의 비교 결과

## 일반화된 모델에서 RDMA 오퍼레이션 벤치마크 테스트

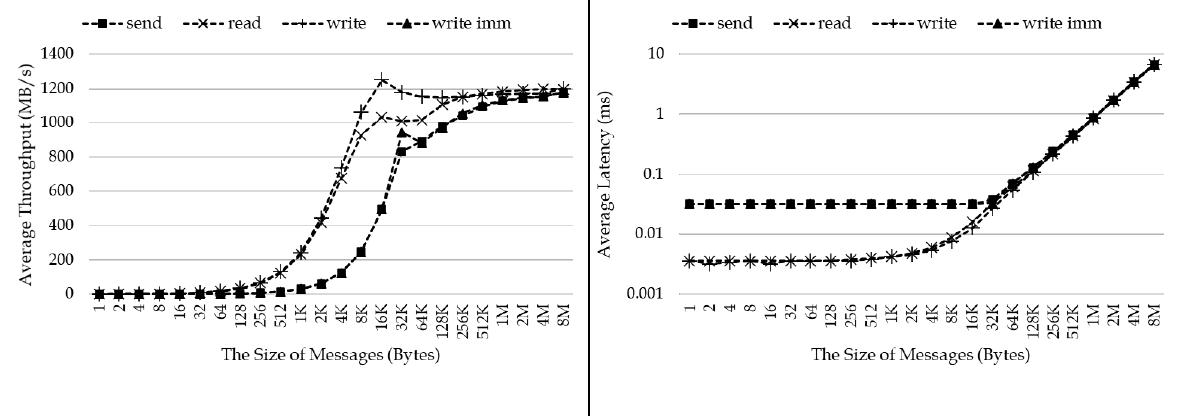
본 절에서는 다수 노드로 일반화된 RDMA 통신 모델에서 RDMA 오퍼레이션에 대한 벤치마크 테스트를 진행한다. 메시지 크기를 1B부터 8MB까지 2배씩 증가시키고, 노드를 하나씩 증가시키며 실험한다. 오퍼레이션은 send (receive), write, read, write imm 네 가지를 대상으로 비교한다.

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.



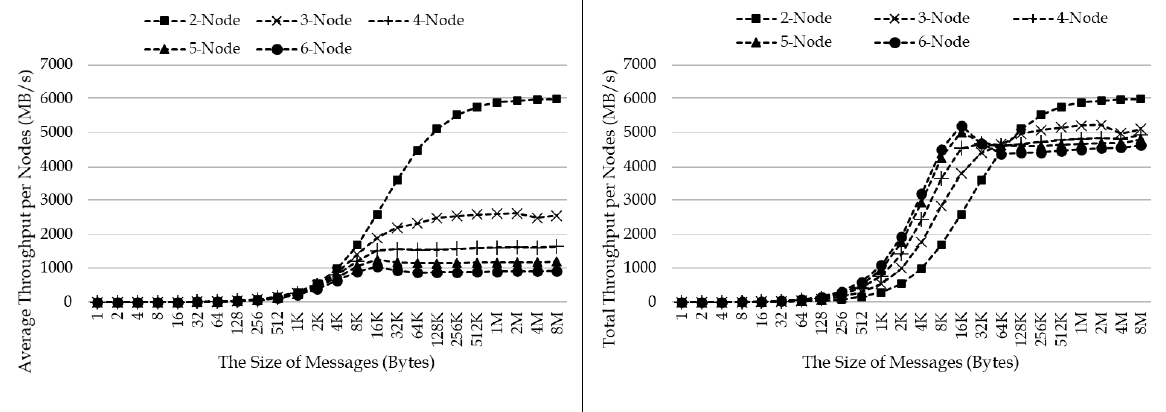
<그림 42> 3-nodes 토폴로지에서 오퍼레이션 성능 비교

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