ISBN 978-89-6211-453-7 98560

대용량 데이터 가시화의 효율적 데이터 공유를 위한 분산 메모리 기술

(Distributed Memory Techniques for sharing data on Massive Data Visualization)

김 민 아 (petimina@kisti.re.kr)

한 국 과 학 기 술 정 보 연 구 원
Korea Institute of Science & Technology Information

목차

표 차례

그림 차례

소스 차례

1. 개 요

대용량 데이터를 처리하고 만들어내는 계산과학 응용에 있어 데이터의 처리 방법은 매우 중요한 이슈이다. 때로 데이터들은 지리적으로 떨어진 곳에 분산되어 위치하기도 하며, 지리적으로 떨어져 있진 않더라도 그 양의 방대함으로 인해 데이터 저장소와 계산처리가 다른 클러스터에서 이루어지기도 한다. 기후 모델링과 예측, 의학 이미지들, 고에너지 물리, 항공우주 시뮬레이션 등 오프라인으로 처리하기에도 힘든 테라, 페타 바이트 단위의 데이터를 이들 모델링 도구들은 인터렉브하게 처리하여야 한다. 이러한 data-intensive한 응용들에 있어 성능의 병목은 데이터 스토리지와 원격 데이터 접근을 위한 지연이다. NASA의 기후 모델링, 분석, 예측(MAP)을 위한 프로젝트의 경우 실행시간의 25~50% 가 입출력을 기다리는 idle 시간이었다[2]. 실시간 data-intensive 응용들은 데이터의 접근과 사용에 대한 새로운 관점의 변화를 요구하였다.

- 로컬 스토리지 시스템이나 지리적으로 분산된 스토리지에 저장된 데이터에 대한 접근 지연의 최소화
- 여러 스토리지 시스템에 대한 병렬 접근
- 테라 바이트, 페타 바이트 수준의 데이터 처리
- 병렬 처리 시스템간의 데이터 공유

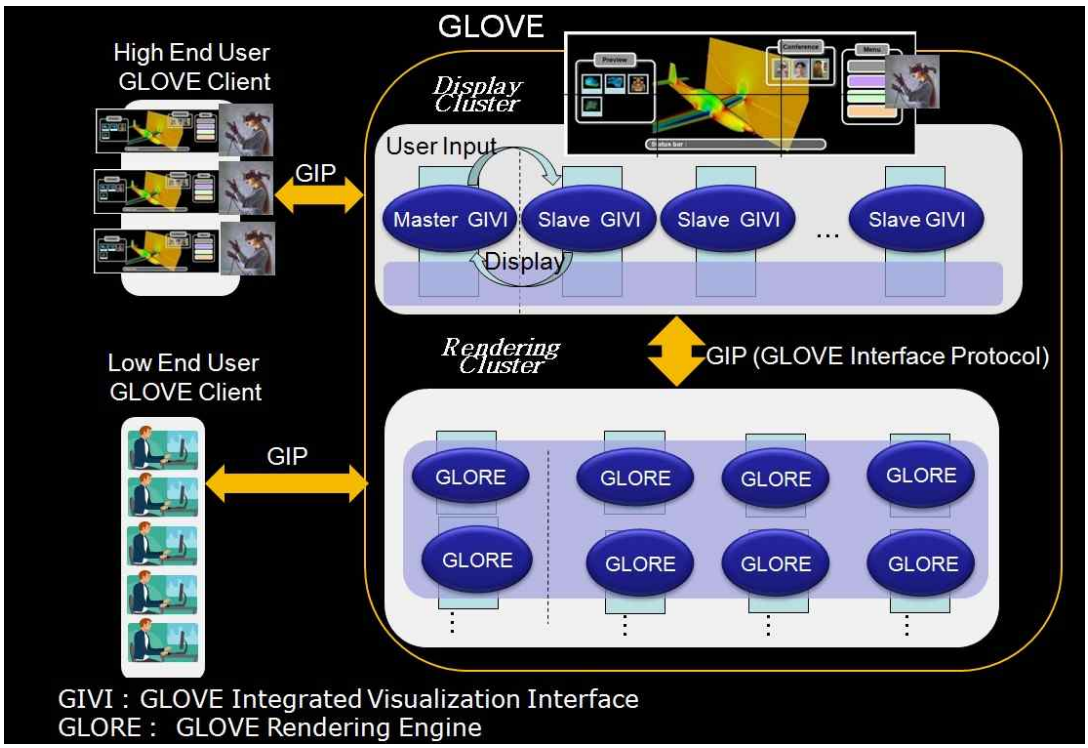
대용량 데이터 가시화 또한 데이터의 입출력 패턴이 이들 계산과학의 응용들과 다르지 않다. 이에 더하여, 가시화는 대부분의 경우 계산과학이 부분적으로 만들어낸 데이터를 한 scene 내에 표현하여야 한다. 이러한 일을 수행하기에 한 시스템내의 메모리는 턱없이 부족하다. 또한, 대용량 데이터의 가시화는 데이터 입출력 뿐 아니라 이미 메모리에 읽어 들인 데이터로 하나의 scene을 렌더링 하는데에 걸리는 시간 지연도 또 하나의 병목이다. 이를 위해 여러 대의 클러스터가 렌더링을 나누어 수행하는 병렬 렌더링 기법을 사용한다. 병렬 렌더링 시, 데이터는 어떻게 나누어 공유할 것인지는 대용량 데이터 가시화의 또 다른 이슈이다.

본 문서에서는 먼저 이러한 대용량 가시화 시스템을 설계할 때 부족한 메모리와 입출력지연 때문에 발생할 수 있는 문제에 대해 기술하고, 이러한 문제들을 해결하기 위한 여러 가지 분산 메모리 기술과 그 해결책에 대해 기술한다.

2. 대용량 데이터 가시화 시스템의 입출력과 데이터 공유 문제

본 절에서는 대용량 가시화 시스템이 데이터를 다룰 때 생길 수 있는 문제에 대해 KISTI에서 설계한 GLOVE 시스템의 예를 통해 좀 더 상세히 기술한다.

[그림 1-1]은 KISTI에서 설계한 대용량 가시화 시스템 GLOVE의 구조를 보여준다. GLOVE는 16개의 디스플레이 노드와 93개의 렌더링 노드를 가지는 시스템이다.

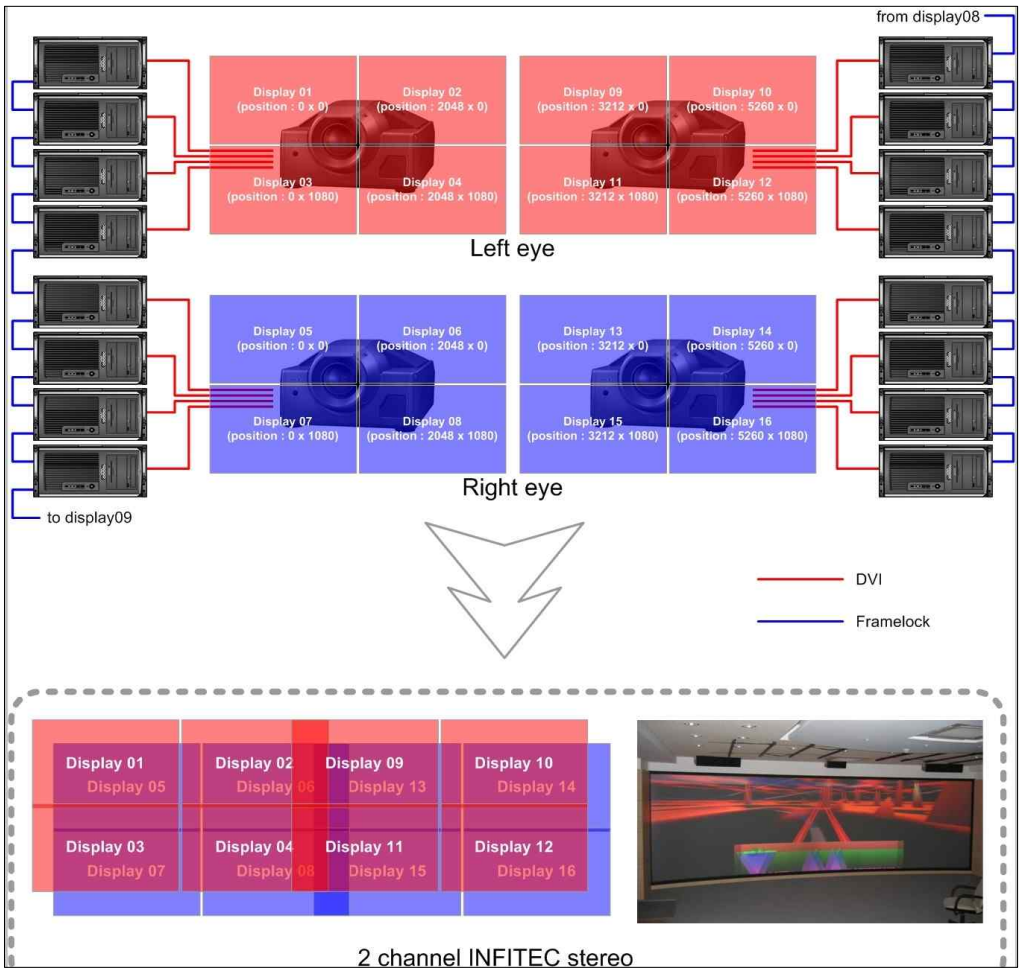


[그림 1-1 GLOVE HW Architecture]

GLOVE의 하드웨어는 상하 4개씩 8개의 화면으로 나뉘는 tiled-display를 제공한다. 입체영상의 지원을 위해 하나의 화면 당 좌우 2개씩 16개의 디스플레이 노드가 각각 디스플레이를 담당하고 있다. 이들은 모두 동일한 하나의 scene을 각각 렌더링 하며, 자신의 viewport에 맞는 부분만 화면에 보여준다.

이러한 시스템에서 대용량 데이터의 입출력과 공유에 대해 생각해 보자. 사용자 입력을 통해 데이터를 읽어 들일 경우, 16개의 노드에서 모두 데이터를 읽어 들인다. 또한 각자 렌더링을 수행하여 전체 scene을 구성한다고 할 경우 이 데이터가 모두 16번 렌더링 되어야 한다. 좀 더 효율적으로 시스템을 구성하여 렌더링을 위해 렌더링 노드를 사용하여 병렬 처리를 하고 그 결과만 전송한다고 해도,

16개의 디스플레이 노드 모두에 렌더링의 결과를 보내 주어야 한다.



[그림 1-2 GLOVE Display Nodes]

먼저, 스토리지로부터 16개의 노드가 데이터를 읽는 과정부터 보자. 동일한 데이터를 16개의 노드가 읽을 경우 스토리지 시스템에 병목이 발생한다.

두 번째로, 캐쉬 메모리를 보자. 캐쉬 메모리는 이미 접근하였던 데이터를 메모리에 유지하여 다음에 그 데이터를 접근할 때 하드디스크를 접근하지 않고, 메모리를 접근함으로써 하드디스크 입출력을 위한 대기 시간을 줄이기 위한 기법이다. [표 1-1]과 [표 1-2]는 캐쉬 메모리를 사용할 때와 그렇지 않을 때, 데이터를 로딩하는 시간을 보여준다. 비교적 데이터양이 적은 블레이드 데이터의 경우 캐쉬를 사용할 때와 그렇지 않을 때 15배에서 20배의 시간 차이가 난다. 그러나, 데이터양이 많은 필드데이터를 읽을 경우 캐쉬를 사용할 때와 그렇지 않을 때, 시간은 차이가 거의 없다. 즉, 시스템의 캐쉬 메모리를 초과하는 양의 데이터를 읽을 경우, 캐쉬는 전혀 효력을 발생하지 못한다.

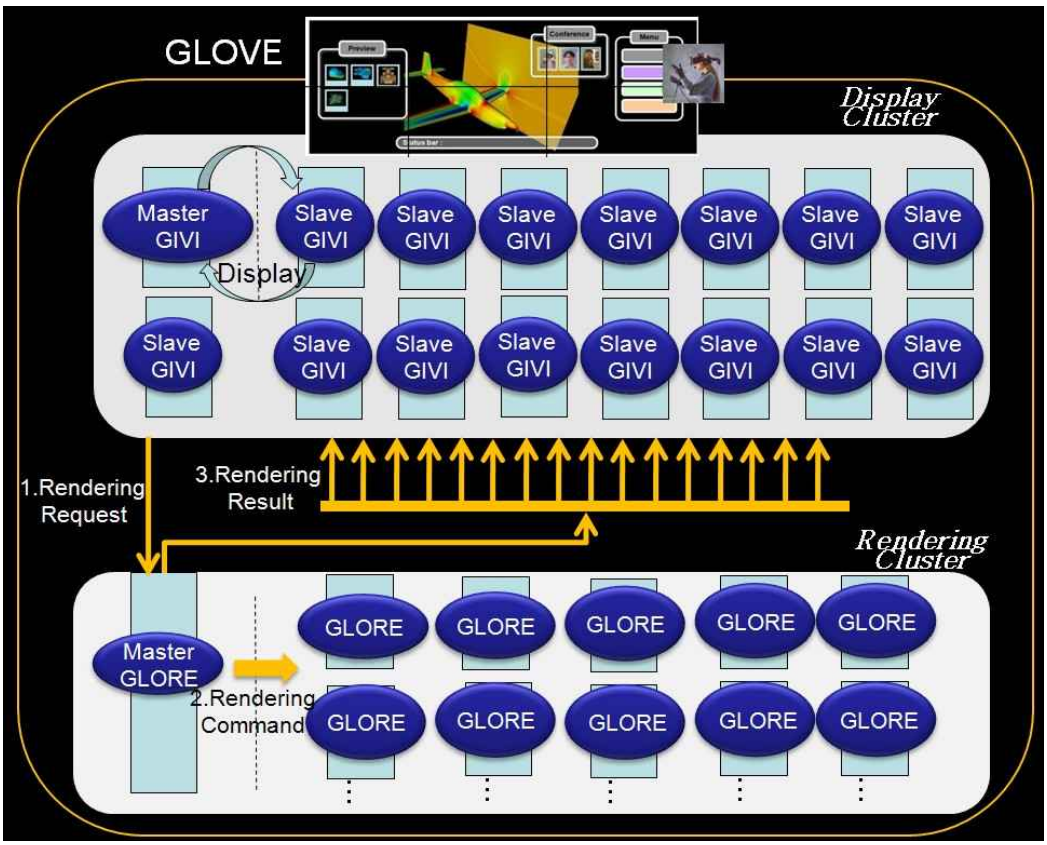
[표 1-1 Blade data loading time in sec]

Number of vertices	Cache	Ascii	Binary
35360	X	0.54	0.219
	O	0.16	0.016
1208064	X	22.84	15.345
	O	6.408	1.318

[표 1-2 Field data loading time in sec]

Number of vertices	Cache	Ascii	Binary
1070124	X	138.79	10.31
	O	134.13	3.31
218049216	X	1591.41	129.91
	O	1533.59	48.54

세 번째로, 16개의 노드가 렌더링 노드에 렌더링을 요청하는 과정을 보자. [그림 1-3]은 그 과정을 보여 준다. 만일 16개의 노드가 모두 렌더링을 요청할 경우, 동일한 요청이 16번 보내지며, 이에 대한 결과도 16번 전송되어야 한다. 또한, 렌더링 노드들 사이에도 렌더링 할 데이터의 공유가 이루어져야 한다. 결과는 결국 한 노드에서 통합되어 전송되어야 한다. 설계가 잘 이루어져 한 번의 요청으로 16번의 결과를 전송한다고 할지라도 데이터의 공유는 이루어져야 한다. 데이터 공유 문제뿐 아니라 계산을 위해 한 번에 할당할 수 있는 메모리도 로컬 메모리 영역으로는 부족하다.



[그림 1-3 Rendering Request Processing]

GLOVE의 경우 로터 블레이드 시뮬레이션 데이터의 360도 회전 데이터를 모두 올릴 경우 64GB의 로컬메모리가 부족하였다.

이러한 여러 가지 문제를 어떻게 해결할 것인가? 먼저 스토리지 시스템의 병목 현상은 병렬 입출력 파일 시스템으로 해결한다. 병렬 입출력 파일 시스템은 동일한 파일을 여러 노드에서 접근할 경우 한 번의 하드 디스크 접근만 수행하고 나머지는 그 데이터를 브로드캐스팅 한다.

두 번째, 데이터의 양은 많고 로컬 캐쉬 메모리는 부족한 현상으로 나타나는 문제는 분산 캐쉬 메모리 기법인 LambdaRAM을 사용하여 해결한다. LambdaRAM은 동일한 클러스터에 속하는 노드들 뿐 아니라 지리적으로 떨어진 곳에 위치한 클러스터 노드들의 메모리도 하나의 캐쉬 메모리처럼 사용할 수 있게 해 준다.

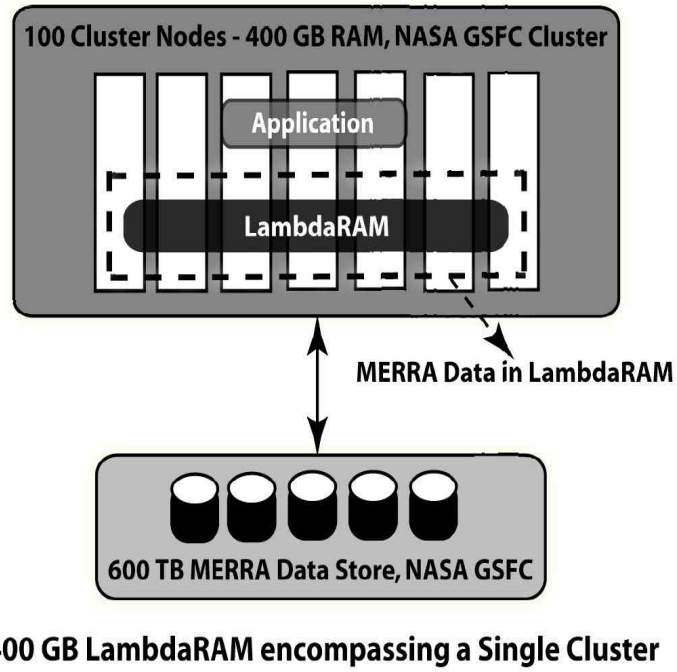
세 번째, 렌더링 클러스터와 디스플레이 클러스터 사이, 렌더링 클러스터 내의 노드와 노드 사이의 데이터 공유 문제는 Global Address 메커니즘을 가지는 분산 공유 메모리 기법을 통해 해결한다. 현재 노드당 64GB 의 메모리를 가지고 있는 GLOVE 시스템의 경우 모든 렌더링 노드와 디스플레이 노드를 분산 공유 메모리로 묶을 때 총 6TB의 메모리를 사용할 수 있게 된다.

3절과 4절에서는 이상에서 언급한 해결책인 LambaRAM과 분산 공유 메모리 기술들에 대해 알아본다.

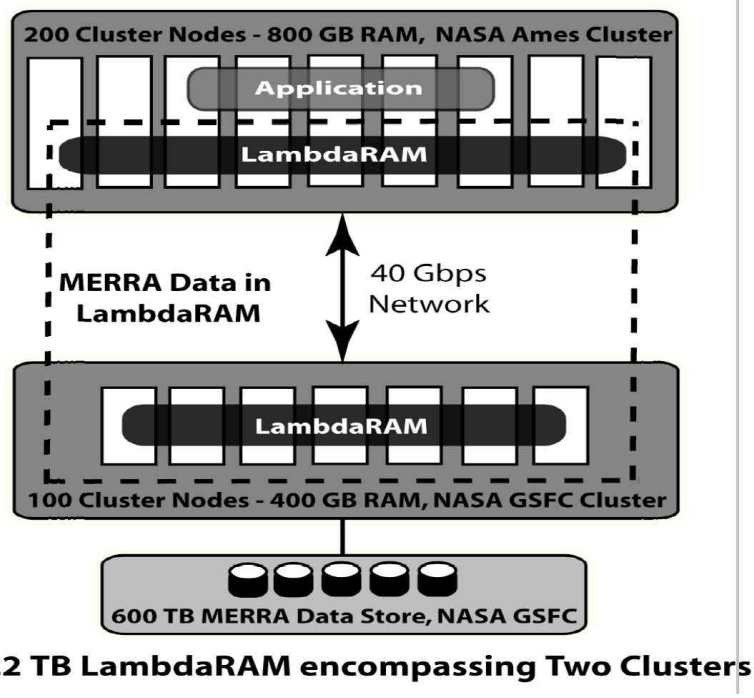
3. LamdaRAM

가. LambdRAM의 개념

LambdaRAM은 지리적으로 분산되어 있는 데이터에 대한 빠른 접근을 위해 고성능 네트워크로 연결된 클러스터 시스템의 메모리를 하나의 대용량 캐쉬 메모리처럼 사용할 수 있게 해주는 고성능 multi-dimensional 분산 캐쉬 메모리이다. LambdaRAM의 물리적 구성은 크게 local cluster 모드와 client-server cluster 모드, hierarchical cluster 모드 세가지 모드로 나뉜다. Local cluster 모드는 전체 클러스터 노드들의 메모리를 하나의 메모리처럼 확장하는 모드이다. [그림 2-1]은 하나의 클러스터를 LambdaRAM으로 사용하는 예를 보여준다. NASA GSFC 클러스터 노드에서 실행되고 있는 응용이 NASA GSFC 스토리지에 저장된 데이터를 입력으로 읽을 때, 그 캐쉬 메모리로 LambdaRAM을 사용하였다. 각 노드의 4GB 메모리를 400GB 메모리로 확장한 효과를 가진다.



[그림 2-1 LambdaRAM encompassing a single cluster]

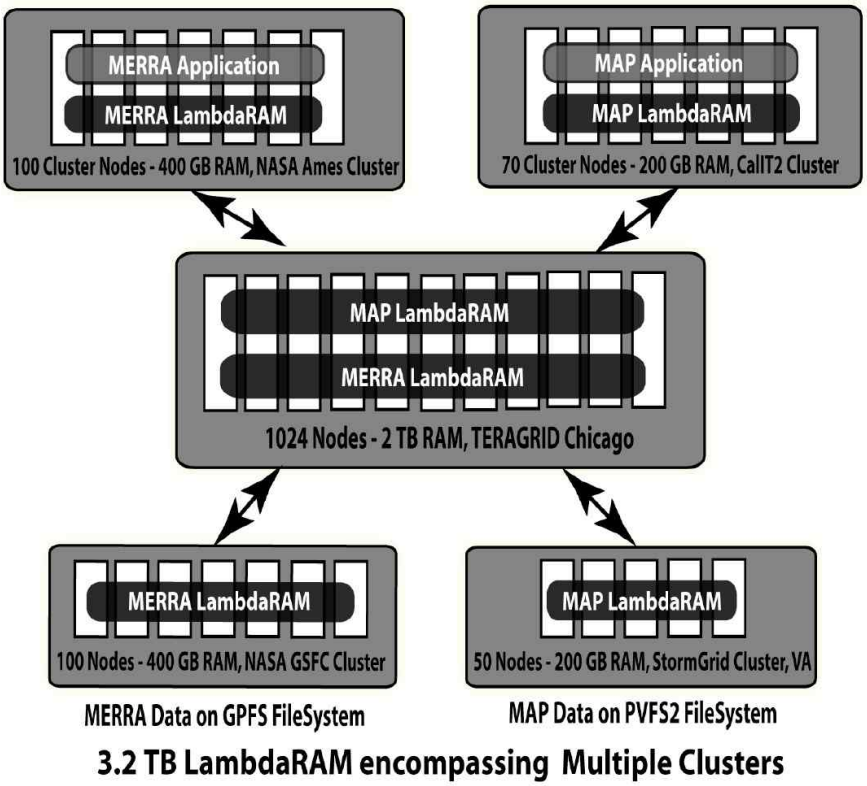


[그림 2-2 Client-Server LambdaRAM Configuration]

[그림 2-2]는 client-server 모드의 LambdaRAM의 물리적 구성을 보여준다. 지리적으로 떨어진 NASA GSFS 클러스터와 NASA Ames 클러스터 간에 client

-server 모드로 LambdaRAM을 구축함으로써, 총 1.2TB의 캐쉬 메모리를 확보하였다.

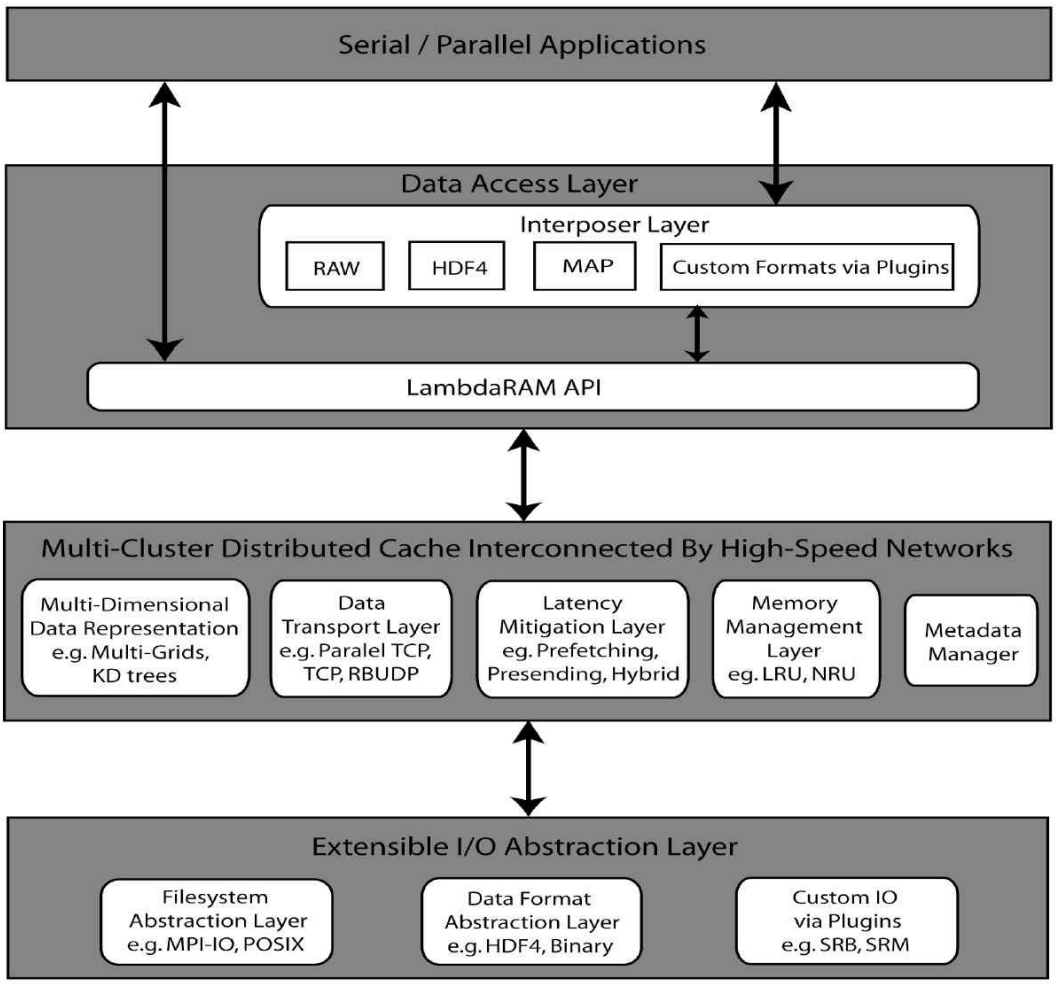
Hierarchical 모드는 다양한 데이터 집합과 응용을 수용하는 구조로 대용량 데이터 가시화 시스템이 여러 응용을 수용할 경우 유용한 구조이다. [그림 3-3]은 Hierarchical Cluster 모드를 보여 준다.



[그림 2-3 Hierarchical cluster LambdaRAM configuration]

나. Functional Architecture

[그림 2-4]는 LambdaRAM의 functional architecture를 보여준다. 응용프로그램은 데이터를 접근하는 API나 혹은 interposer layer를 통해 LambdaRAM 내의 데이터에 접근할 수 있다. 데이터 접근 API는 multi-dimensional dataset으로 분산되어 존재하는 데이터를 다룰 수 있다. 가시화 데이터의 경우 대부분 좌표계의 위치와 그 위치에서의 값을 표현하는 경우가 많으므로 배열 형태의 multi-dimensional 형태로 데이터를 다루는 LambdaRAM은 매우 유용한 구조이다.



[그림 2-4 LambdaRAM Architecture]

Interposer layer는 이미 구현된 응용이 소스의 변경 없이 LambdaRAM을 사용할 수 있는 기능을 제공하는 layer이다. 데이터를 접근하는 응용의 call을 가로채 LambdaRAM에 관한 접근이면, LambdaRAM API를 부른다.

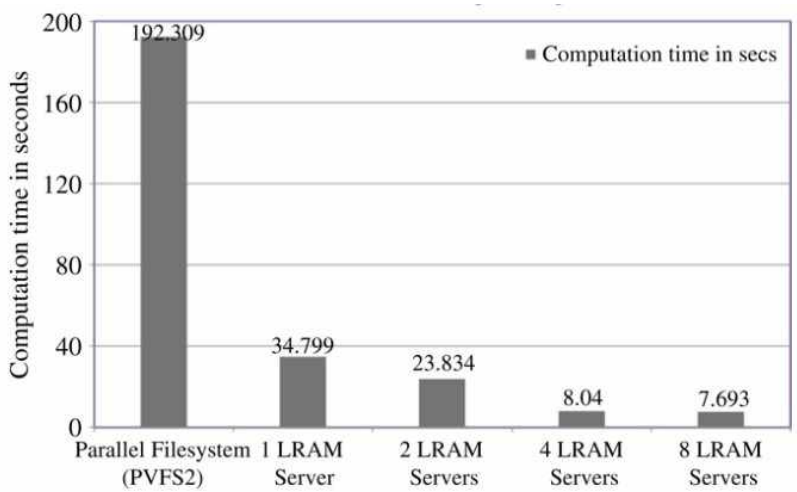
여러 대의 클러스터에 분산된 데이터를 네트워크로 전송하여 내부 데이터로 가져 오기 위해서는 많은 기능의 지원이 필요하다. 데이터를 표현하는 방법과 네트워크 전송, 메모리 관리 분산된 데이터에 대한 메타 데이터 관리 등이 그것이다.

Multi-Dimensional Data Representation layer에서는 데이터의 표현방법에 대해 다룬다. Data transport layer는 병렬 TCP 등 대용량 데이터의 효율적 전송에 대한 프로토콜 레벨의 기능을 지원한다. Latency Mitigation Layer에서는 지연을 줄이기 위한 prefetching과 presending 등을 응용에 맞는 heuristic을 사용하여 수행한다. Memory Management Layer는 메모리 관리 메커니즘을 다룬다. 메타 데이터 관리자는 분산된 데이터의 메타 데이터를 관리한다. Extensible I/O Abst

raction Layer는 실제 하드디스크 입출력에 관련된 기능을 지원한다.

다. LambdaRAM의 성능

캐쉬 메모리를 확장했을 경우 어느 정도의 성능향상을 기대할 수 있을까? [그림 3-5]의 그래프는 LambdaRAM 사용 시 응용의 계산 시간을 보여 준다.



[그림 2-5 Wind shear computation for the Atlantic basin over Local Area Network]

결과는 4대의 LambdaRAM 서버를 사용할 경우 20배의 속도 향상을 있다는 것을 보여 준다. 네 대 서버 이상일 경우 병목현상은 client node의 1Gbps 네트워크 성능의 제약 때문에 발생한다.

LambdaRAM 서버의 성능 비교는 100대 이상의 서버일 경우에 대해 기술된 바가 없다. 대용량 데이터 가시화 시스템의 경우 LambdaRAM 서버는 100 노드 이상이 될 것이며, 이를 위한 네트워크의 성능도 10Gbp가 될 것이다. 이를 위한 성능이 그만큼 향상 될 것인지는 테스트를 거쳐야 할 것이다. 또한 가시화 시스템은 한 번 읽은 데이터에 대한 접근 보다는 매번 새로이 생성된 데이터에 대한 접근이 더 많을 것이다. 따라서, LambdaRAM은 현재 버전의 경우, 대용량 데이터 가시화 시스템에서 16대의 디스플레이 노드가 하나의 데이터 파일에 접근할 때 캐쉬 메모리의 역할로 로딩 타임을 줄이는데 사용 가능하다.

라. LambdaRAM을 사용한 간단한 Parallel Programing

이상에서 언급한 LambdaRAM으로 구현한 병렬처리 프로그램의 예를 살펴보자.

```
#include "LRAM.h"
#include “mpi.h”
using namespace LRAM;

int main (int argc, char** argv)
{
        // Initialize MPI
        MPI_Init();

        // Initializing LambdaRAM with the relevant configuration file
        LRAM::initialize(lram_conf_file);

        // Open a Dataset with relevant dataset configuration file
        int datasetID = LRAM::open(dataset_conf_file);

        // Read the multi-dimensional array
        // In this case, we are reading a 3D data with starting extents
        // from st_ext with a length of len in each dimension into
        // a 1D buffer buffer
        long long st_ext[3], len[3];
        unsigned char* buffer;

        nread = LRAM::read (dsid, st_ext, len, buffer);

        // Close the Dataset
        LRAM::close(datasetID);

        // Finalize LambdaRAM
        LRAM::finalize())

        // Finalize MPI
        MPI_finalize();

        return 0;

}
```

[소스 1-1 Parallel MPI Program using LambdaRAM API]

LambdaRAM의 API는 매우 간단하다. initialize로 초기화하고, 파일을 open 하듯, open()함수를 사용하여 open()한 다음 read() 함수로 원한 크기만큼의 데이터를 읽는다. close()로 닫아 주고, finalize()로 종료한다. [소스 1-1]의 코드를 보면 LambdaRAM은 데이터 공유를 위해서 사용할 수 있는 것처럼 보인다. 그러나, LambdaRAM은 캐쉬일 뿐이다. 즉, dataset_conf_file에 의해 기술된 스토리지에 저장된 데이터를 한 번 더 접근할 때 유용한 구조이다. 다시 말해 한번은 하드디스크에 저장하여야 쓸 수 있는 구조이다. 대용량 가시화 시스템에서 메모리에서만 다루어지는 중간 생성 데이터를 공유하기 위해서 우리는 다른 방안을 찾아보아야 한다.

다음 절에서는 이러한 데이터 공유와 대용량 데이터를 메모리 내에서만 다룰 수

있도록 한 노드의 메모리를 전체 노드의 메모리로 확장해 주는 분산 공유 메모리 기술들에 대해 알아본다.

4. Distributed Shared Memory

Distributed Shared Memory(DSM)은 각각의 클러스터에 존재하는 제한된 로컬 메모리를 하나의 공유 메모리처럼 접근할 수 있게 해 주는 하드웨어적 소프트웨어적 구현을 총칭한다. 하드웨어적 구현은 특별한 시스템을 요하므로 본 문서의 범위에서 다루지 않는다.

소프트웨어적 DSM 시스템은 오퍼레이팅 시스템에 의해 혹은 프로그래밍 라이브러리에 의해 구현된다. 오퍼레이팅 시스템에서 구현된 DMS은 일반적으로 Virtual Memory의 확장으로 생각할 수 있으며, 사용자에게 완전히 숨겨진 형태로 transparent 하다. 반면, 프로그래밍 라이브러리로 구현된 DSM은 개발자가 프로그램을 달리 해야 한다. 그러나, 이들 시스템은 다른 시스템으로의 이식성이 높다. 소프트웨어적 DSM은 공유 메모리 영역을 관리하기 위해 서로 다른 방법을 사용한다. 오퍼레이팅 시스템이 메모리를 다루는 방법인 페이지 단위의 접근은 고정된 크기의 페이지로 공유 메모리를 관리한다. 반면 오브젝트에 기반을 둔 접근 방법은 공유할 오브젝트 단위로 공유 메모리를 관리하므로, 관리 대상의 크기는 오브젝트 크기에 따라 달라질 수 있다.

대용량 데이터 가시화 시스템은 대부분 클러스터로 구성된다. 이들 클러스터 시스템들은 환경에 맞는 오퍼레이팅 시스템을 설치하여 운영한다. 본 문서에서는 대용량 가시화를 위해 일반적으로 사용하고 있는 Unix System의 공유 메모리 개념을 차용하는 분산 공유 메모리 시스템들에 대해서 알아본다.

가. Global Address Space

분산 공유 메모리는 computer science에서 Distributed Global Address Space(DGPS)로도 불린다. 로컬 메모리를 관리하기 위해 오퍼레이팅 시스템에서 사용하는 주소공간은 프로그래밍 언어와 컴파일러에 의존적이다. 그러나, 분산 공유 메모리시스템에서의 주소공간은 클러스터 노드 전체 혹은 다른 클러스터에 있는 노드들도 포함한 주소 공간이 존재해야 해야 한다. 대부분의 DSM은 이러한 Global Address Space를 관리하는 방안을 가지고 있다.

나. DIPC (Distributed Inter Process Communication)

DIPC는 리눅스 환경 하에 분산 프로그래밍을 위해 IPC를 제공하는 유일한 소프트웨어적 솔루션이다. DIPC는 두 개의 파트로 구성된다. 첫 번째는 Unix System V IPC 메카니즘 (shred memory, queue, semaphore)에 대한 접근을 가로채는 커널 파트이고 두 번째는 분산 시스템의 관리와 네트워크 전송에 대한 책임을 지고 있는 데몬 프로세스이다. DIPC의 DSM은 데이터 일관성을 위해 데이터를 읽을 때, 가장 최근에 쓴 값을 읽는다.

다. SCASH

SCASH는 communication layer 로 MPI를 사용한 소프트웨어적 분산 메모리 공유 시스템이다. 대부분의 존재하는 소프트웨어적 DSM 시스템들은 보통 communication layer로 이더넷 상에 TCP나 UDP 기반으로 시스템 전용의 프로토콜을 사용한다. SCASH 는 하나의 thread 가 MPI를 사용한 원격 메모리 통신을 지원한다. 페이지 단위로 관리하며, 페이지 폴트가 일어날 때까지 페이지 전송은 일어나지 않으나, prefetch를 사용하여 페이지 전송 지연을 줄인다.

라. TreadMarks

TreadMarks는 오퍼레이팅 시스템 레벨이 아닌 완전히 사용자 레벨에서 구현된 DSM솔루션이다. TreadMakrs는 malloc(), free() 와 같은 함수를 이용하여 공유 힙 메모리를 제공한다. 또한 이들 공유 메모리의 동기화를 위해 barrier와 lock을 제공한다. 메모리는 페이지 단위로 접근하며, 다른 노드들로부터 메시지는 SIGIO 핸들러를 사용한다. 사용자 레벨의 DSM 솔루션으로 가장 많이 사용된다.

5. 결론

대용량 데이터 가시화 시스템은 대용량 데이터를 다루어야 하는 응용의 특징 때문에 tiled-display와 클러스터라는 하드웨어적 지원이 필요한 시스템이다. 이 때문에 여러 노드가 동시 하나의 데이터를 접근하거나, 여러 노드가 병렬 처리를 위해 데이터를 공유해야 하는 데이터 접근 방식의 새로운 요구가 발생하였다. 이러한 요구를 지연 없이 빠른 시간 안에 사용자의 요구에 따라 인터렉티브하게 처리하기 위해 본 문서에서는 대용량 가시화 시스템이 가지는 특징을 분석하여 데이터 접근 시 발생할 수 있는 지연과 메모리 부족 현상을 해결하기 위한 해결책들에 대해 기술하였다.

그러나 이러한 기술들을 적용하기 위해서는 오퍼레이팅 시스템의 특징과 네트워

크의 특성을 고려해야 한다. 오퍼레이팅 시스템이 DSM이 요구하는 커널을 지원하지 않을 수도 있고, 10Gbps 네트워크 카드가 LambdaRAM이 사용하는 UDP를 사용할 경우 치명적인 성능저하를 가져 올 수도 있다.

따라서, 대용량 데이터 가시화 시스템을 위한 분산 메모리 기술들은CI 적용하고자 하는 가시화 시스템에 맞게 선택되어야 하며, 환경에 맞는 수정과 확장 개발이 필요할 수도 있다.

6. 참고문헌

[1] E. A. Anderson and J. M. Neefe, "An exploration of network RAM", Tech.Rep. UCB/CSD-98-1000, Dec.9, 1994

[2] Vishwanath, V., "LambdaRAM: A High-Performance, Multi-Dimensional, Distributed Cache Over Ultra-High Speed Networks", Dec.15, 2008.

[3] General Parallel File System, IBM Corporation,

http://www-03.ibm.com/systems/cluster/software/gpfs/index.html

[4] M.J.Feeley, W.E.Morgan, F.H.Pighin, A.R.Karlin, H.M.Levy, and C.A.Tekkath, "Implementing global memory management in a workstation cluster", ser. Operating System Review, vol.29(50), Dec.3-6, 1995

[5] Yoshinori Ojima, Mitsuhisa Sato, Taisuke Boku and Daisuke Takahashi,

"Design of a Software Distibuted Shared Memory System using an MPI communication layer", Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks, 2005.

[6] Mohsen Sharifi, Kamran Karimi, "DIPC: The Linux Way of Distributed Programming", Linux Journal, JAN.1, 1999