2006년 정보보호 연구발표

클러스터를 이용한 슈퍼컴퓨터 구축 및 암호분석 (JBCERT)

팀원 : 손문성
강래훈
이율환
이광인
이영준

중부대학교 정보보호학과

PROLOGUE

우리가 이 프로젝트를 생각하게 된 것은 영화 해커즈2에서 나온 보안 시설을 보고“어떻게 저렇게 많은 컴퓨터들이 연동이 되는 것일까“부터 궁금증을 유발하기에 충분했다. 이 영화는 케빈미트닉이라는 최고의 해커를 배경으로 만들어지는 것인데 미트닉이 다른 프로그래머가 만들어 놓은 프로그램이 암호화 되어 있다는 것을 알고 어느 대학에 슈퍼컴퓨터를 이용하게 되다. 그렇게 하면서 일반 컴퓨터로는 몇 년이 걸리는 작업을 몇 일 만에 암호화된 프로그램을 풀고 영화는 끝나게 되는데 우리로써는 충분히 감탄할 만한 영화였다 물론 과장된 부분이 있었다고는 하나 슈퍼컴퓨터는 현시대에서도 많은 분야에서 사용되어지고 있다. 이러한 기술을 우리도 사용해 보고 싶었기에 우리는 이번 프로젝트로 클러스터를 이용한 슈퍼 컴퓨터를 만들어 보기로 하고 시작한지 1년이 지난 지금 이렇게 문서를 작성하게 되었습니다.

클러스터링을 이용한 슈퍼컴퓨터는 컴퓨터 과학을 넘어서 자연과학, 공학 분야와 설계, 그래픽, 기상예측 같은 우리 생활에 가까운 분야까지 넓어졌고 지금도 계속확장 되고 있습니다. 하지만 그에 대한 자료는 많지 않아 경험자에게 물어보거나 일일이 인터넷, 참고서적 등을 뒤져 필요한 부분을 스스로 찾는 정도에 그치고 있다. 이런 점을 우리는 보안하기 위해여 어떠한 운영체제가 좋고 그 운영체제에 대해 설치 매뉴얼을 만들고 MPI프로그래밍을 이용한 암호 분석하는 프로그래밍을 해 보려고 한다. 이보고서가 우리 정보보호학과 후배들과 전공자들에게 클러스터를 구축하는데 있어 조금이나마 도움이 되었으면 좋겠고 이보고서를 보는 모든 이들에게 도움이 되었으면 좋겠다. 이 보고서 다루는 많은 내용은 꼭 클러스터와 관계없는 이들에게도 유용한 할 것이고 , 리눅스에 관심이 있는 모든 이들에게도 많은 참고가 될 수 있으리라 기대한다.

우리는 이 프로젝트를 하면서 우리 전공분야에서 공부했던 거의 모든 부분에 대해서 다시 한 번 공부하게 되는 배경이 되었다 특히 허브를 연결하여 컴퓨터를 연결하는데 있어 라우터 기능을 하게 만들기 하는 헤더노드와 계산노드들 간의 많은 연동부분에 있어 많은 고심을 하였고 하드웨어적으로 연결하는데 있어 무려 한 달이라는 시간동안 이론적인 배경과 구축을 하는데 끙끙대기도 했다. 이보고서는 클러스터링에 대한 배경과 구축방법과 MPI프로그래밍에 대한 내용을 담고 있으니 이 보고서를 보는 이들은 이러한 잘 못을 하지 않기를 바란다.

보고서를 마치면서 가장 먼저 우리들이 클러스터를 다루는 내내 많은 귀찮게 해 드렸음에도 불구하고 지도와 조언을 아끼지 않으신 저희 학과 교수님들에게 감사드립니다.

이 보고서를 보시는 이들의 기대에 많이 모자라지만, 이 보고서를 보시는 분들에게 그만한 가치가 있기를 바란다.

<표 차례>

<그림 차례>

1 서론

1.1. 클러스터란 ?

클러스터는 여러 개의 시스템이 하나의 거대한 시스템으로 보이게 만드는 기술이다. 이렇게 하는 기술에는 여러 가지가 있기 때문에 각 기술의 특징을 잘 이해하는 것이 클러스터링 기술을 잘 활용할 수 있다.

1.2. 클러스터의 구성요소

클러스터는 노드와 관리자로 구성되어 있다. 노드는 프로세싱 자원을 제공하는 시스템이고, 클러스터 관리자는 노드를 서로 연결하여 단일 시스템처럼 보이게 만드는 로직을 제공한다.

클러스터노드 - 클러스터의 실질적인 작업을 처리하는 것을 말한다. 일반적으로 클러스터 노드는 클러스터 노드는 클러스터에 속하도록 구성해야 한다. 클러스터의 역할과 업무에 따라 해당 소프트웨어는 독특하게 제작된 것일 수 있고, 일반적인 소프트웨어 일 수도 있다. 역할에 따른 특정 소프트웨어라면 공학 계산을 위한 각 노드에 맞는 프로그램일 수도 있으며, 일반적인 소프트웨어는 로드벨런싱을 위한 클러스터라면 아파치 같은 소프트웨어를 들 수 있을 것이다.

클러스터관리자 - 리눅스의 커널이 모든 프로세서에 대한 스케쥴과 자원관리를 하는 것 처럼 클러스터 관리자 역시 이런 관리자의 역할로써 각 노드에 대한 자원 분배 및 관리를 할 수 있는 기능을 가지고 있다. 기본적으로 하나의 관리자가 필요하지만 때에 따라서는 클러스터노드가 클러스터 관리자 기능을 하기도 하며, 대규모 환경의 경우에는 여러 대의 클러스터 관리자가 있기도 한다.

1.3. 클러스터의 종류

클러스터의 주요 목적은 CPU자원을 공유하거나, 여러 컴퓨터간의 부하 조정, 가용성이 높은 시스템을 구축, 주 시스템이 다운되었을 때를 대비한 Fail-Over기능을 제공하는 것이다.

공유 프로세싱 : HPC(High Performance Computing)

일반적으로 리눅스 클러스터링이라고 불리는 클러스터링이다. beowulf 프로젝트로도 유명하다. beowulf는 여러 시스템의 프로세싱 능력을 조합하여 대용량의 프로세싱 능력을 갖는 하나의 시스템을 제공한다.

이것은 원래 과학용 시스템이나 CPU위주의 용도로 설계된 것인데, 이 시스템에서는 API에 따라 특별히 작성된 프로그램만 자신의 작업을 여러 시스템 사이에 분배할 수 있다. 즉, 프로그래밍을 별도로 해야할 경우가 많아진다는 것이다.

부하조정 : Load Balancing

최근에 대규모 웹사이트 구축에 필수적으로 사용되는 기술로 여러 대의 웹서버 노드를 두고 중앙의 관리툴에서 부하를 분산하게 해주는 기술이다. 이 기술의 특징은 노드 간 통신이 필요 없다는 것이다. 부하 조정을 이용하면 각 노드는 자신의 용량이나 로드에 맞게 요청을 처리할 수 있기도 하고, 클러스터 관리자에서 할당한 양의 프로세스를 처리할 수도 있다.

fail-Over

fail-over는 부하조정과 비슷하다. 하지만 조금 틀린 것이 잇는데 부하조정의 경우에는 모든 노드가 한꺼번에 동작을 하는 것이고, fail-Over의 경우에는 평소엔 동작을 하지 않고 Primary 서버가 문제가 발생했을 시에 백업서버로써 가동을 하는 것이다. 부하조정을 응용하면 부하조정과 fail-Over기능을 동시에 하게 할 수가 있다.

높은 가용성

아무리 시스템 관리자가 뛰어난 실력을 지니고, 부지런하다고 해도, 서버는 기계이기 때문에 간혹 문제를 발생하기도 한다. 어떤 경우에는 관리자도 어떻게 할 수 없는 상황이 발생하기도 한다. 이런 경우에 최대한 가용성을 높이기 위한 방법이 바로 클러스터링이다. 이런 높은 가용성을 만들기 위해 위의 fail-Over기술을 사용한다.

1.4. 왜 클러스터인가?

무엇보다도 특수한 하드웨어 대신 값싼 일반 컴퓨터에 사용하는 하드웨어를 사용하기 때문에 비슷한 성능의 전용 시스템에 비해 매우 비용이 절감된다.

예를 들면 대전을 KAIST 내에 있을 SERI에 있는 국내 최초의 슈퍼컴퓨터 Cray-2는 약 4백억 원 정도에 도입되었다. 지금도 전문적인 슈퍼컴퓨터는 몇 십~몇백억 정도의 가격으로 대학이나 기업, 연구소 등에서 도입되고 있다.

그에 비해, 현재 국내에서 가장 빠른 슈퍼컴퓨터로 기록된 클러스터 시스템은 약 20억 원 정도의 비용으로 만들어 졌다. 따라서 같은 성능을 1/10 이하의 가격으로 얻을 수 있기 때문에 클러스터가 인기를 얻게 되었고, 이러한 경향은 앞으로도 계속될 것이다.

또 다른 장점은 필요한 하드웨어와 프로그램을 쉽게 구할 수 있고 사용자 스스로 클러스터를 만들 수 있다는 것이다. 주변에서 쉽게 구입할 수 있는 PC나 서버용 하드웨어와 네트워크 장비, 리눅스와 윈도우 등 일상에서 쓰이는 운영체제만 갖고 있으면 된다. 그리고 클러스터를 위한 전문 프로그램과 설치 방법, 참고 등도 인터넷 등에서 쉽게 찾을 수 있으므로 컴퓨터에 대한 지식만 있다면 스스로 클러스터를 만들기 위해 필요한 모든 것을 쉽게 구해서 시작할 수 있다.

리눅스 기반의 베오울프 클러스터가 지금처럼 많이 쓰이는 이유로는 사용자가 스스로 쉽게 클러스터를 만들 수 있고 또 클러스터에 맞도록 프로그램을 개발하고 운영체제를 개조할 수 있었다는 것이 큰 이유가 된다.

다만 클러스터는 네트워크가 중요한 구성 요소 중 하나이므로 네트워크의 성능에 의해 효율이 많이 좌우되고 클러스터의 규모가 커질수록 효율적으로 성능을 높이기 어렵다.

또 클러스터의 관리는 시스템뿐만 아니라 내부 네트워크의 관리도 필요하므로 보통 단일 시스템보다 유지 보수가 복잡한 단점도 있다.

여기까지 보면 클러스터는 매우 거대한 시스템이고 전문적인 분야에서 전문가들만 사용하는 것처럼 보일 수도 있다. 또 직접 PC로 클러스터를 만들어도 그걸로 뭘 할 수 있을까 하는 생각도 들게 된다.

최신 게임이나 그래픽, 동영상 프로그램이 실행이 되지 않는다. 스스로 프로그램을 만들지 않는 한 아직 클러스터는 집에서 편리하게 쓸 만한 시스템이 아니다.

하지만 정반대로 클러스터는 컴퓨터를 전공한 전문가들만이 쓰는 시스템이라고 생각할 수도 있지만, 그렇지 않다. 이제는 슈퍼컴퓨터를 많이 필요로 하고 또 직접 사용하는 사람들은 물리, 화학, 생물, 의학 등의 연구자들, 기사예측이나 기계설계 및 모의실험을 하는 기술자들 등 오히려 컴퓨터 자체에 대한 비전공자들이 많다.
클러스터를 구축하여 오랜 시간이 걸리는 예를 스스로 프로그래밍해서 자신의 클러스터에서 실행하는 것도 재미있는 일이 될 수 있다.

1.5. 클러스터는 어떻게 발전 할 것인가?

클러스터를 구성하는 네트워크는 클러스터를 만드는 쪽에서 구축했으며, 이러한 시스템에 노드를 확장해서 더 큰 클러스터 시스템을 만들려고 하면 역시 네트워크의 하드웨어와 논리 구조도 그에 맞추어 확장해야 한다.
그렇지만 이러한 방식으로 클러스터의 규모를 무한정 확장하는 것은 한계가 있다. 단순히 클러스터만 사용하는 네트워크 안에서 클러스터를 확장해 나가는 것은 네트워크의 규모 뿐 아니라 시스템이 자리하게 되는 공간상의 문제와 관리의 어려움 등 여러 가지 문제를 갖게 된다. 클러스터는 그에 속하는 컴퓨터가 네트워크를 통해 하나의 시스템을 구성하는 구조이므로, 한 장소에 모여 있는 컴퓨터의 집합이라는 개념을 뛰어넘어서 여러 곳에서 분산되어 있는 컴퓨터도 네트워크로 서로 연결해서 클러스터로 만들 수 있지 않을까 하는 생각에서 새로운 클러스터의 개념이 만들어지고 있다. 이때 여러 곳에 분산되어 있는 컴퓨터를 서로 연결해 주는 네트워크는 바로 인터넷이다. 즉, 지금의 인터넷을 이용해서 새로운 형태의 클러스터를 구축하려는 노력이 차세대 클러스터의 개념으로서 많은 곳에서 시도되고 있다. 이런 방식으로 만들어지는 새로운 시스템을 그리드 시스템이라고 한다. 그리드는 전 세계의 컴퓨터 자원을 격자 형태로 연결해서 원하는 사람이 장소에 관계없이 필요한 만큼의 컴퓨터 자원을 쓸 수 있도록 하는 개념이다.
인터넷에서 사용자들은 주로 웹서버에서 웹페이지를 보거나 멀티미디어를 감상하고 필요한 파일을 받아와 자신의 컴퓨터에서 실행한다. 즉 인터넷을 통해 공유되는 컴퓨터 자원은 지금은 주로 데이터에 한정되어 있다. 인터넷에서 볼 때 이것을 클러스터와 같은 프로세서 등 다른 요소도 공유할 수 있도록 확장한 개념이 그리드다.
클러스터와 같은 그리드에서 프로세서 자원도 공유할 수 있다면 그리드 컴퓨팅으로 할 수 있는 것도 클러스터와 같은 고성능, 고가용성, 결함 허용 등 다양할 것이다. 물론 그리드는 이런 것뿐만 아니라 여기에 지금 인터넷의 데이터 자원 공유를 더해서 인터넷을 통한 하나의 거대한 컴퓨터를 구현하는 것을 목표로 하고 있다.
그리드를 응용 목적에 따라 분류하면, HPC와 같은 계산 그리드, HA와 같이 같은 시간에 더 많은 수의 계산을 처리하는 초생산성(high throughput) 그리드가 있다. 이 밖에 그리드로 구현할 수 있는 것을 생각해 보면 다음과 같다

♦ 다국적 기업이나 은행에서 쓰이는 자료나 과학기술 데이터베이스 등 전 세계에 분산되어 있는 엄청난 양의 데이터베이스 검색 또는 자료를 처리하는 그리드를 데이터 그리드라고 한다. 사용자가 단일 그리드 주소로 들어와서 데이터베이스 검색 또는 분석을 요청하면 전 세계에 잇는 그리드 기반요소가 자신에게 할당된 작업만큼 검색해서 사용자의 요청에 맞는 결과를 돌려주게 된다.

1.6. 병렬프로그램이란?

복수의 프로세서를 이용하여 같은 결과를 얻는 프로그램을 말하며 목적은 N개의 프로세서를 이용하여 N배 빠른 프로그램 방법으로는 모든 프로세서들 사이에 부하를 균등화하고 통신과 계산 사이에 부하를 균등화 하는 것이다.
클러스터와 같은 MPP머신에서 병렬프로그램은 프로세서마다 각자의 메모리에 저장된 데이터를 참조해서 실행됩니다. 여기서 메모리에 있는 데이터와 프로세서에서 실행되는 프로그램에 따라 병렬 프로그램은 다음과 같은 네 가지로 분류된다.
SISD(Single Instruction, Single Data)는 모든 프로세서에서 같은 프로그램이 실행되고 모든 메모리에도 같은 데이터가 있다.. 예를 들면 모든 노드에서 1부터 100까지 더하는 프로그램을 똑같이 실행하는 것과 같다. 따라서 실제로 병렬 프로그래밍 모델이라고 할 수 없다.

SIMD(Single Instruction, Multiple Data)는 모든 노드에서 같은 프로그램이 실행되는데 노드마다 다른 데이터를 처리한다. 예를 들면 -mpirun -np 2 sum.exe으로 두 노드에서 똑같이 sum.exe라는 프로그램을 실행하는데 두 노드는 1~20, 51~100과 같이 다른 값을 더해서 결과를 1~100까지 합으로 돌려주는 식으로, MPI-1은 이 모델을 기본으로 한다.

MISD(Multiple Instruction, Single Data)는 모든 노드의 메모리에 있는 같은 데이터를 프로세서마다 다른 프로그램이 처리하는 것으로 현제 이 모델에 맞는 병렬 알고리즘은 없다.

MIMD(Multiple Instruction, Mutiple data)는 노드마다 다른 데이터를 프로세서마다 다른 프로그램이 서로 도우면서 실행하는 모델입니다. 이것은 PVM과 MPI-2의 실행 모델로 하나 노드에 master.exe, 다른 노드에 slave1.exe, slave2.exe , …가 있고 master.exe를 실행하면 이것이 다른 노드의 slave1.exe, …를 실행시켜서 병렬처리를 합니다.

SIMD는 모든 노드가 똑같은 프로그램을 실행하므로 프로그래밍은 간단해지지만 모든 프로세스가 어느 순간에는 같은 동작을 하므로 프로세스 수가 많아지면 효율이 떨어집니다. 또 노드마다 다른 작업을 처리하는 복잡한 프로그램을 만들 때 SIMD모델은 프로그램 자신이 모든 루틴을 내장하고 있어야 하므로 프로그램 크기도 커지고 복잡해진다.

예를 들어 compute-0-0, compute-0-1와 compute-0-2,compute-0-3에서 서도 다른 행렬을 곱하는 프로그램이 있다면 SIMD 모델에서는 compute-0-0, compute-0-1와 compute-0-2, compute-0-3에서 같은 프로그램이 실행되므로 프로그램 안에 두 행렬을 만들어내는 루틴을 포함하고 있어야 하지만 MIMD모델에서는 compute-0-0, compute-0-1와 compute-0-2, compute-0-3에서 서로 다른 종속 프로그램을 실행한다.

MIMD모델에서는 몇 사람이 동시에 다른 종속 프로그램을 만들어서 그것으로 크 병렬 프로그램을 쉽게 만들 수 있고 마스터의 규칙에 맞으면 새로운 기능의 종속 프로그램을 쉽게 붙일 수 있다. 그리므로 MIMD모델은 큰 규모의 프로그래밍에 보다 적합하고 확장성도 뛰어나다.

2. 본론

2.1. 구축환경-클러스터를 만들기 위한준비?

2.1.1. H/W의 고려사항

클러스터 제작에 사용할 하드웨어를 선택하는 것은 중요하다. 특히 특정한 프로그램을 돌리기 위한 전용 클러스터를 구축하고자 한다면 더욱 중요하다. 다음은 일반적 몇가지 선택에 대한 조언이다.

-프로세서 (CPU)

클러스터를 구축하여 원하는 컴퓨팅 파워를 얻기 위한 두 가지 고려사항이 있다. 저성능의 컴퓨터를 다수 연결할 것인가 고성능의 컴퓨터를 소수 연결할 것인가를 고려해야 한다. 일반적으로 노드의 수가 증가할수록 프로세서간의 통신이 증가하므로 네트워크 비용이 증가하게 되고 동기화에 필요한 시간이 증가하므로 병렬화 효율 역시 줄어들게 된다. 따라서 고성능 컴퓨터를 소수 연결하는 것이 효율면에서는 좋다. 그러나 고성능 컴퓨터의 가격은 그 성능에 비해 비싸기 때문에 타협을 하여야 한다.

ALPHA 프로세서는 floating 계산 능력에 있어서 가장 앞선다. 동일클럭의 intel 머신에 비해 약 2.5 배 정도 빠르며 대부분의 수치모델들이 floating 연산이 주를 이루는 것을 감안하면 수치모델을 위한 클러스터 구축에 적합하다. 또한 64bit 아키텍쳐이므로 그에 따른 몇가지 장점이 있으며 그중 하나로 리눅스에서의 최대 파일사이즈 2G의 제한이 없다. 비록 리눅스 커널 2.4버전에서는 intel역시 2G제한이 없어졌으나 32bit 기반이므로 2개의 레지스트리가 주소 참조 시 사용되므로 여전히 알파가 빠른 주소결정을 할 수 있다. 그러나 가격이 비싸다는 단점이 있다.

INTEL의 Pentium 시리즈로 대변되는 인텔 프로세서들은 저가에 클러스터를 구성할 수 있으며 다양한 H/W와 S/W의 지원을 받을 수 있는 장점이 있다.

AMD 프로세서 역시 저가에 구성할 수 있으며 각종 벤치마킹에 의하면 Intel 프로세서에 비해 우수하다. 그러나 지원되는 하드웨어가 부족 한 것이 단점이며 듀얼(dual)보드가 아직 지원되지 않는 것 역시 단점이다. 참고로 듀얼보드를 사용할 경우 동일 보드상의 프로세서간의 통신이 빠르고 가격/프로세서 비용을 낮출 수 있다는 장점이 있다.

- 네트워크 장치 (랜카드, 허브)

가장 일반적인 경우 100Mbps 스위칭 허브와 랜카드를 사용하여 구성된다. 100Mbps 네트워크 장치를 사용하여 제작한 클러스터는 많은 사용자들에 의해 충분히 사용할 만한 장비라는 것이 입증되었다. 만약 더 좋은 네트워크 장치를 사용하기를 원한다면 미리넷과 기가비트이더넷을 고려 할 수 있으나 이들 장비는 매우 고가이다. 특히 미리넷은 전용 프로토콜을 사용할 경우 패킷전송시 지연시간(latency)이 매우 작은 클러스터 전용 네트워크 장치이지만 TCP가 지원되지 않는 문제점은 있다.

최근 TCP를 에뮬레이션 할 수 있는 방법이 나와 있으나 아직 안정화 되어 있지 않으며 성능에도 문제가 있다.

- 기타 주변장치

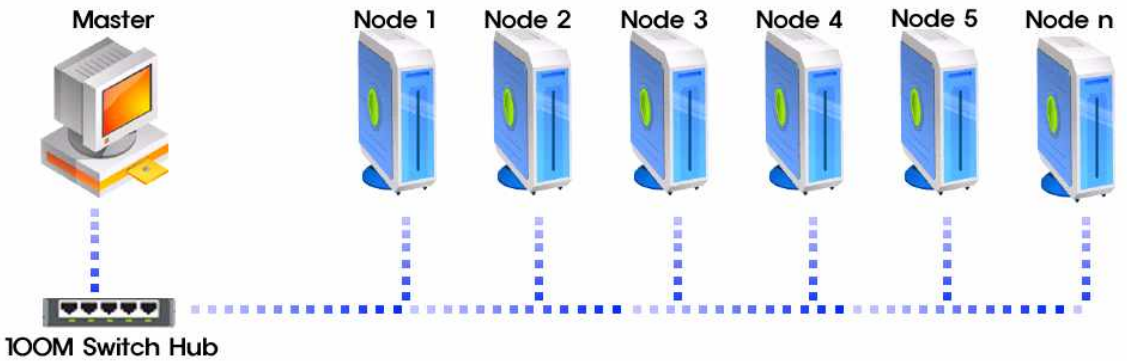
메모리에 있어서는 1 bit 에러 보정능력이 있는 ECC램과 일반램이 고려될 수 있다. 좀 더 신뢰성 있는 시스템을 구축하기 위해서는 ECC램을 사용하는 것이 좋으나 가격은 좀 더 비싸다. 보드의 선택에 있어서는 성능보다는 안정성에 중점을 두어서 선택해야 하는 것이 바람직하다. 클러스터는 다수의 컴퓨터로 구성되므로 하드웨어가 문제를 일으킬 확률이 그만큼 증가하게 되기 때문이다. Storage(저장장치)의 경우 일반적으로 IDE방식과 SCSI방식의 HDD가 고려될 수 있다. 이는 전적으로 어떤 프로그램을 수행할 것인지에 따라 결정될 수 있다. I/O가 빈번이 발생하는 프로그램을 수행한다거나 동시에 다수의 프로세스에 의해서 I/O가 이루어진다면 SCSI을 고려해야 할 것이다. 그리고 더 나은 안정성과 속도를 원하다면 고가의 RAID역시 고려될 수 있다. 한편 다수의 하드디스크를 이용하여 소프트웨어적으로 RAID를 구성하는 방법도 있다.

2.1.2. S/W의 고려사항

- OS에 대한 고려사항

가장 널리 사용되는 OS로는 역시 리눅스를 들 수 있다. 현재 리눅스를 이용하여 클러스터를 구축하기 위한 많은 기술문서와 라이브러리, 모니터링 툴, 큐잉시스템 등이 공개되어 있으며 심지어는 클러스터링을 위한 커널이미지, 배포판 등도 나와 있다.
한편 리눅스뿐만 아니라 UNIX, NT등에서도 클러스터를 구축할 수 있다.

2.2. 구축 설계



<이번 보고서를 위해 직접 구현한 클러스터 슈퍼컴퓨터의 개념도>

이번 보고서에서 클러스터 제작 기법은 고성능 계산을 제공하기 위한 HPC 클러스터를 구축하기 위한 것이다.

N대의 컴퓨터를 클러스터링 할 것이며 이후 필요한 장비는 다음과 같다.
컴퓨터 N대, 각 컴퓨터는 CPU, RAM, BOARD, HDD, VGA, Lan Card등의 기본적인 장치를 갖고 있다. 추가로 Lan Card 1개, 스위치허브 1개, 모니터 1개, 마우스 1개, 키보드 1개, IP한 개가 필요하다.

클러스터 구축 방법은 컴퓨팅 노드의 하드디스크 유무에 따라 아래와 같이 두 가지로 구분될 수 있다. 첫째 모든 컴퓨터가 물리적인 하드디스크를 갖고 있다. 그러나 /home을 위한 물리적 공간을 가지고 있지 않다. 단지 하나의 리눅스박스에만 /home을 위한 물리적 공간을 준다. 나머지 컴퓨터에서는 이 디렉토리를 네트워크를 통하여 공유하여 사용하는 것이다. 이 방식이 확장성과 편리성에 있어서 좋다.

둘째 하나의 리눅스 박스에만 하드디스크가 있고 나머지 리눅스박스들은 하드디스크를 가지고 있지 않는 방식이다. 따라서 디스크가 없는 컴퓨터들은 부팅 디스켓 또는 부트롬을 사용해서 최소한의 부팅을 한 후 NFS ROOT를 사용하여 다른 컴퓨터의 하드디스크 일부를 자신의 하드디스크처럼 사용한다. 이 방식은 하드디스크를 위한 비용을 절약 할 수 있으며, 이후에 노드를 추가하기 매우 쉬운 장점이 있다. 그러나 역시 확장성과 편리성에는 제약이 따르게 된다.

이번 보고서에서는 첫 번째 방식으로 구현할 것이다.

한편 준비된 n대의 컴퓨터 중 어떤 컴퓨터를 헤드 노드로 사용할 것인지를 결정해야 한다. 클러스터 는 하나의 시스템이다. 클러스터가 비록 여러 대의 컴퓨터로 구성되어 있다고 할지라도 전체적인 관리는 한 곳에서 이루어져야 한다. 우리는 이렇게 관리 작업을 할 컴퓨터를 헤드 노드라고 부르자. 클러스터가 구축된 후에 사용자들은 모두 헤드 노드로 접속하여 작업을 수행하게 될 것이며 관리자 역시 헤드 노드에서 클러스터의 관리를 수행하게 될 것이다. 따라서 다른 컴퓨터 보다 성능이 좋은 컴퓨터를 헤드노드로 사용하는 것이 좋다. 최소한 동일한 성능의 컴퓨터를 헤드 노드로 사용하면 된다. 나머지 컴퓨터들을 계산 노드라고 부르자.

Master Node	Node 1~9
- 펜티엄4 CPU 2.6G	- 펜티엄4 CPU 2.6G
- RAM 1G	- RAM 1G

☩ Node N개에 따라 클러스터의 성능이 N배 증가한다는 이론에 맞추어 Node 1개부터 9개 까지 구축함으로써 9배 빠른 계산 속도를 테스트 한다.

2.3. 클러스터 구축

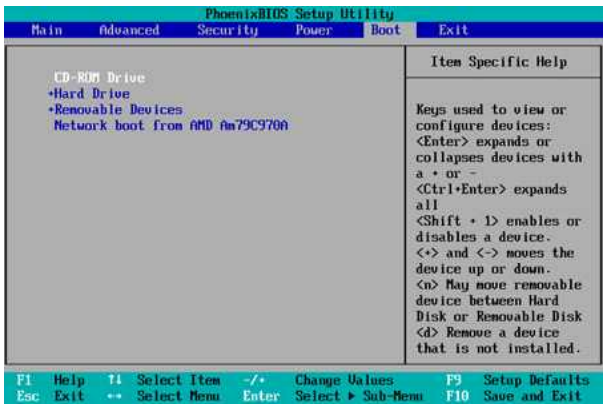
이 Rocks라는 리눅스를 사용하게 된 이유는 클러스터를 구축하고 관리하는데 Rocks를 사용함으로써 얻는 이점은 명확하기 때문이다. 클러스터의 과정은 상당히 단순한 작업이지만, 그것의 소프트웨어에 대한 관리는 상당히 복잡해질 수 있다. 이 복잡성은 클러스터의 설치 및 확장 시 가장 관리하기 어려운 부분이 되곤 한다. Rocks는 이러한 클러스터의 설치 및 확장 과정에서의 복잡성을 제어하기 위한 방법과 성능 모니터링 도구를 제공한다. 이 보고서에서는 클러스터를 구축하고, 그것의 소프트웨어를 설치하는 과정을 설명한다.

2.3.1. 헤더노드에 리눅스 설치

이 번에는 Rocks Base CD 와 HPC Roll CD를 사용하여 Rocks 클러스터의 헤더노드를 설치하는 방법을 시작하겠다.

- 헤더노드 컴퓨터를 켜서 BIOS모드로 들어간다. (키는 보통 F2 와 DEL 를 누르면 됨)

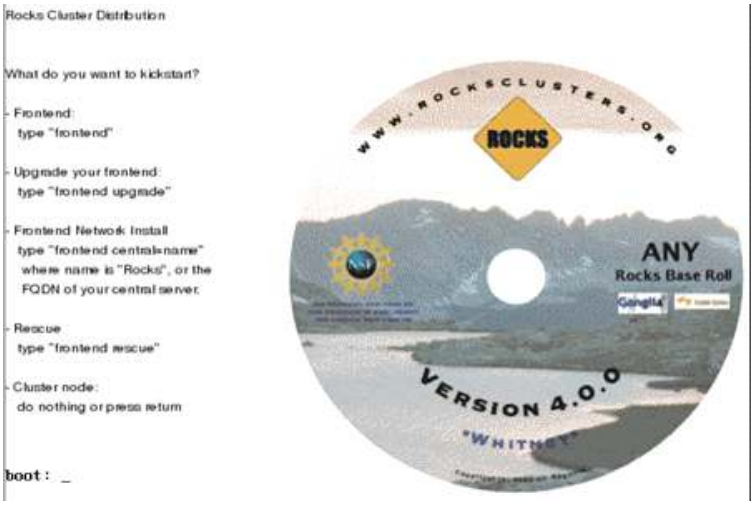
- 부팅 순서를 CD가 가장 먼저 부팅이 되도록 설정해 준다.



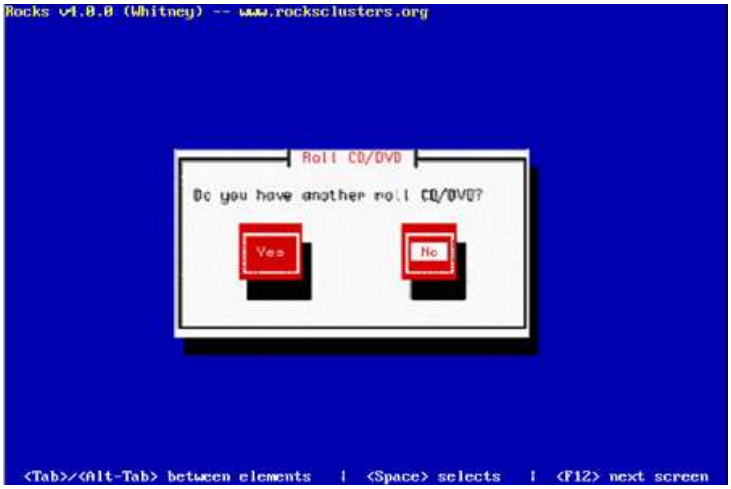
- Rocks Base CD를 헤드노드에 넣고, reboot시킨다.

- 헤더 노드가 CD로 부팅하면, 아래화면과 같이 부팅 스크린을 볼 것이다. “frontend"를 타이핑하면 된다.

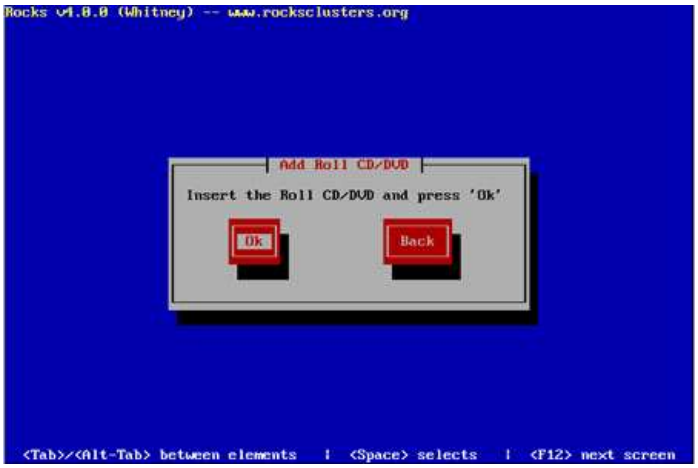
✝ "boot":프롬프트가 보인후 재빨리 화면이 바뀐다. 따라서 위의 화면을 놓치기가 쉽다. 아무런 입력이 없을 때, 기본적으로 계산 노드로 가정한다. 따라서 이 경우 헤드노드 설치는 되지 않을 것이며, 반드시 리부팅한 후 설치과정을 다시 시작해야한다.



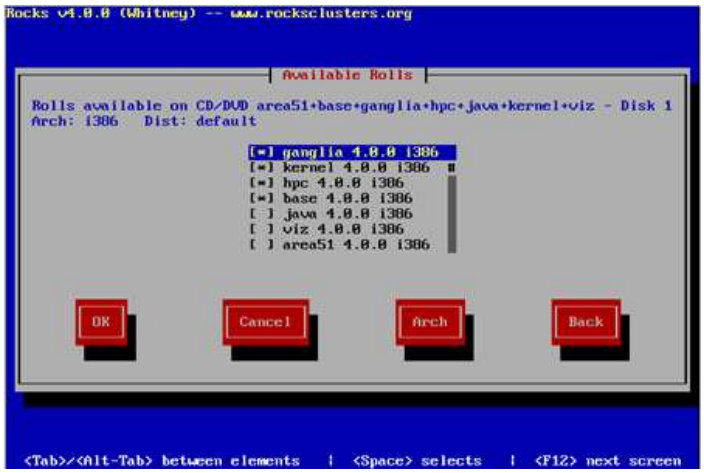
frontend를 타이핑하고 나면, RedHat installer (anaconda)가 시작될 것이다. 그리고나서 곧 다음과 같은 화면을 볼 것이다. HPC roll CD/DVD를 사용하고자 한다면 스페이스바를 눌러 'YES'를 선택해야 한다.



- CD/DVD 드라이버가 Rocks Base 미디어를 밖으로 보내면, HPC roll CD 를 그 드라이브에 삽입한다. 그리고 스페이스바를 눌러서 'OK'를 선택하라.

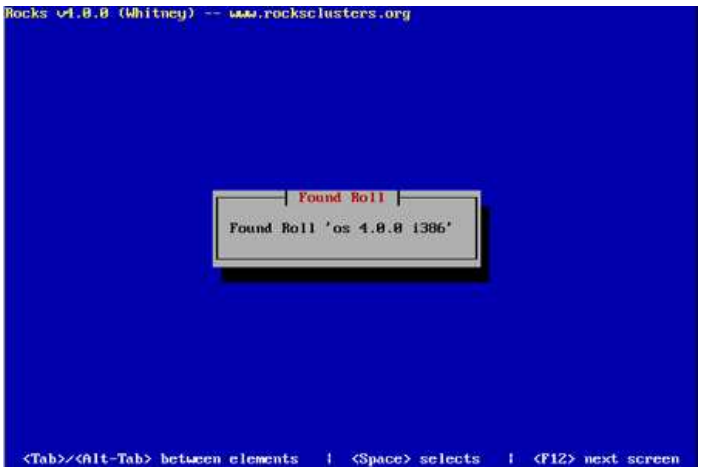


- roll을 추가하는 화면이 볼일 것이다.

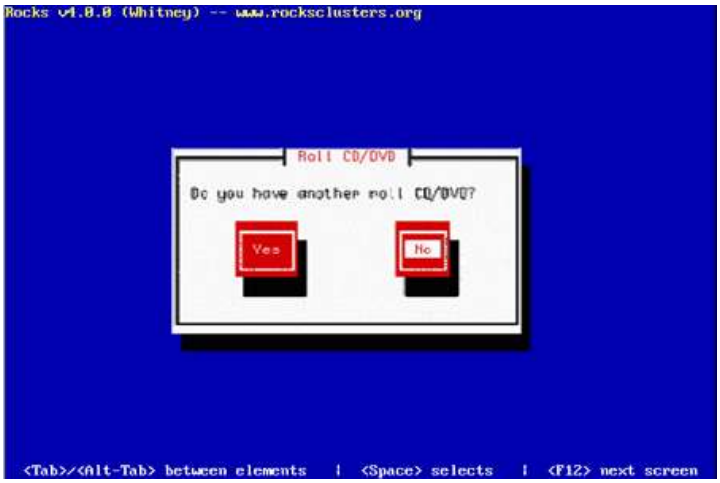


원하는 roll을 방향키, 탭키, 스페이스바키를 사용하여 추가하고 'OK'를 스페이스바키로 선택하라.

- OS 미디어가 발견되면, 다음의 화면이 보일 것이다.



-또 다른 roll 이 있는지를 물어올 올 것이다.



'YES'를 선택한 후 Kernel Roll CD를 사용하여 위의 과정을 반복하여 자신이 원하는 모든 Roll을 추가한다. 이를 다 추가 했다면 탭키와 스페이스바키를 사용하여 “NO”를 선택하라.

- Cluster Information 스크린을 볼 것이다. 이 정보는 Ganglia 가 설치할 클러스터의 기본 정보로 사용할 것이며, Fully Qualified Hostname을 제외한 다른 정보의 입력은 선택사항이다. 입력을 마치면 ‘OK'를 눌러라.

✝ Fully Qualified Hostname을 지정하는 것이 중요하다. 즉 이 이름은 alias를 통해 다른 이름으로 변경되어 사용될 수 없다. 여기에서 설정한 hostname은 헤더노드 및 계산노드의 여러 파일들에 쓰여지며, 이 hostname을 변경하면, SGE, Globus, AuteFS 그리고 Apacheemddml 클러스터 관련 서비스를 사용할 수 없게 될 것이다.

- 디스크 파티셔닝 화면을 통해 '자동' 혹은 '수동'으로 파티셔닝을 할 수 있다. '자동'으로 파티셔닝이 되게 하기 위해서는 Autopartition 버튼을 눌러야 한다. 이것은 헤더노드를 아래의 표와 같이 파티셔닝할 것이다. 수동으로 헤더노드를 파티션하기 위해서는, Disk Druid 나 Fdisk를 선택해야 한다. 자동 파티셔닝이 default 이며, 특별한 이유가 없는 한 자동 파티셔닝을 추천한다.

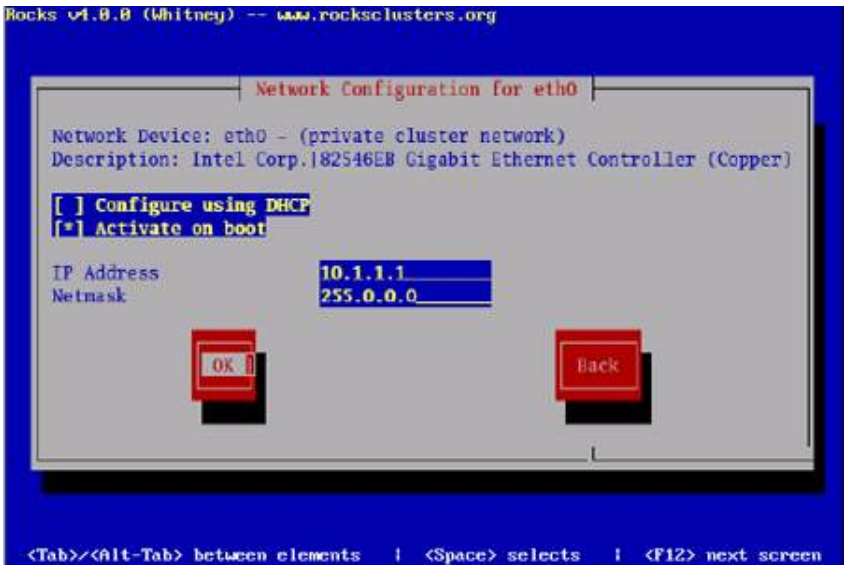
[표]-default partition

Partition Name	Size
/	6 GB
/swap	1 GB
/export	root 디스크의 나머지

✐ 수동 파티션을 할 경우, 루트 파티션에 6 GB 이상을 할당해야 하며, 또한 /home 대신 /export 파티션을 만들어야 한다. 설치가 끝나면 /exprot/home 이 /home 으로 automount

된다.

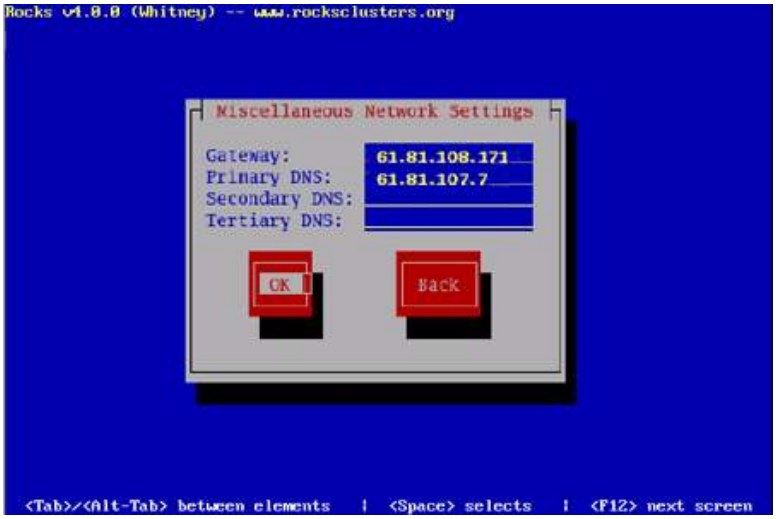
- private 클러스터 네트웍 설정화면에서 헤더노드와 계산노드를 연결하는 네트웍의 setup을 할 수 있다.



- public 클러스터 네트웍 설정 화면에서 헤드노드와 외부 네트웍(예를 들어 인터넷)과의 연결을 위한 네트웍 변수를 setup할 수 있다.

고정 IP주소를 지정하고자 한다면, 'Use bootp/dncp' 옵션을 사용하지 말아야 하며, 당연히 'Activate on boot' 옵션은 사용해야 하며, 그 아래의 변수를 설정하면 된다. 위에 예는 하나의 헤드노드의 대한 외부 네트웍 설정을 보여준다.

- 헤드노드의 Gateway와 DNS를 설정하라.

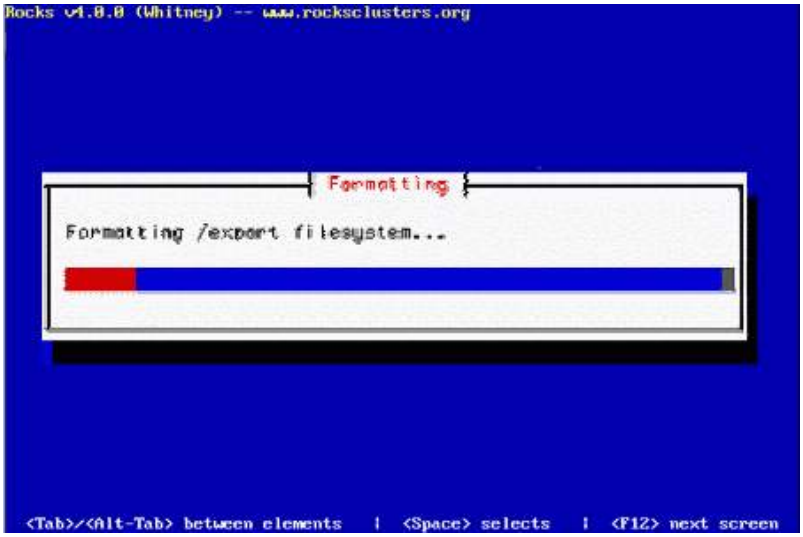


-시간 설정 부분에서는 Korea/Seoul

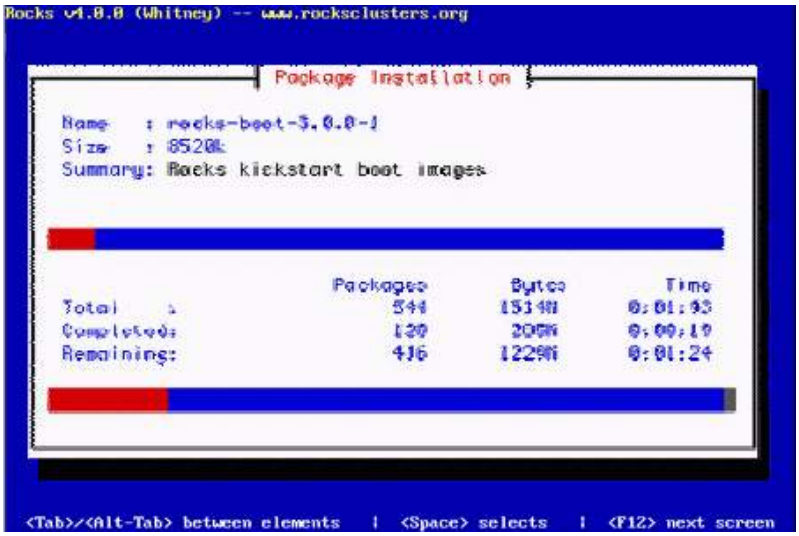
-root 패스워들 입력한다.

[이후의 설치과정에서는 자동을 설치된다.

- 헤드노드의 파일시스템을 포맷한 후 , 기본적인 패키지를 설치한다.



-기본적인 패키지들이 local 하드 디스크로 복사될 것이다.



installer는 Rocks Base CD의 내용을 헤더노드의 local 하드 디스크에 한다.

-installer는 앞에서 추가한 roll CD들을 요청할 것이다. 적절한 roll CD를 CD 드리이버에 넣은후 'OK'를 누르면 된다.

2.3.2 네트워크와 서비스 설정

2.3.2.1. 랜 카드 설정 확인

두 개의 랜카드에 다음과 같은 주소를 부여한다. eth1에 부여되는 주소에 대해서는 그대로 따라하면 된다. 내부네트워크에서 사용될 주소이므로 어떠한 값이 사용되어도 상관없다.

eth0 : IP(10.1.1.1) hostname : JBCERT.local
eht1 : IP(61.81.108.171) hostname : JBCERT.cluster.domain

[root@JBCERT ~]# ifconfig 명령을 사용하여 제대로 설치 됐는지를 확인해 볼 수 있다.

```
[root@JBCERT ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:04:75:BF:DC:5E
          inet addr:10.1.1.1  Bcast:10.1.1.255  Mask:255.255.255.0
          inet6 addr: fe80::204:75ff:febf:dc5e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1080482 errors:0 dropped:0 overruns:1 frame:0
          TX packets:1598680 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:100056318 (95.4 MiB)  TX bytes:1666700905 (1.5 GiB)
          Interrupt:10 Base address:0xe400

eth1      Link encap:Ethernet  HWaddr 00:E0:7D:FA:47:DD
          inet addr:61.81.108.171  Bcast:61.81.108.255  Mask:255.255.255.0
          inet6 addr: fe80::2e0:7dff:fefa:47dd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

RX packets:244671 errors:0 dropped:0 overruns:0 frame:0

TX packets:37545 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:51332245 (48.9 MiB) TX bytes:9461468 (9.0 MiB)

Interrupt:11 Base address:0xe000

lo Link encap:Local Loopback

.....

2.3.2.2. route 명령을 통한 설정확인

한편 라우팅은 아래와 같이 설정해 주어야 한다. xwindow 환경에서 netcfg 명령을 사용하여 쉽게 설정할 수 있다.

[root@JBCERT ~]# route

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
255.255.255.255	*	255.255.255.255	UH	0	0	0	eth0
61.81.108.0	*	255.255.255.0	U	0	0	0	eth1
10.1.1.0	*	255.255.255.0	U	0	0	0	eth0
169.254.0.0	*	255.255.0.0	U	0	0	0	eth1
224.0.0.0	*	240.0.0.0	U	0	0	0	eth0
default	61.81.108.1	0.0.0.0	UG	0	0	0	eth1

2.3.2.3. hosts 파일 설정 확인

클러스터를 이루고 있는 각 노드의 주소와 이름을 정의해 준다. 이것은 헤드노드와 모든 계산노드에서 동일하게 설정해 준다.

[root@JBCERT ~]# vi /etc/hosts

#

Do NOT Edit (generated by dbreport)

#

127.0.0.1 localhost.localdomain localhost

61.81.108.171 JBCERT.cluster.domain

10.1.1.1 JBCERT.local JBCERT # originally frontend-0-0

10.1.1.254 compute-0-0.local compute-0-0 c0-0

10.1.1.253 compute-0-1.local compute-0-1 c0-1

10.1.1.252 compute-0-2.local compute-0-2 c0-2

10.1.1.251 compute-0-3.local compute-0-3 c0-3

10.1.1.250 compute-0-4.local compute-0-4 c0-4

2.3.2.4. /home을 공유하게 설정

다음과 같이 exports 파일을 편집하자.

[root@JBCERT ~]# cat /etc/exports

/export 10.1.1.0/255.255.255.0(rw) /*

그리고 다음의 명령을 사용하여 nfs를 다시 시작해주자.

[root@master /etc]# /etc/rc.d/init.d/nfs restart

마스터 노드에 nfs 데몬이 제대로 떠있는지 확인해보자. 아래와 비슷하게 나와야 한다..

[root@JBCERT ~]# ps -auex |grep nfs

root 2503 0.0 0.0 0 0 ? S Oct15 0:00 [nfsd]

root 2504 0.0 0.0 0 0 ? S Oct15 0:00 [nfsd]

. . . . .

nfs 데몬이 떠 있지 않다면 nfs는 RPC(Remote Procedure Call)를 사용하기 때문에 포트 매퍼(Port mapper)라는 데몬이 먼저 떠 있어야 한다. 이를 확인해 보자.

[root@JBCERT ~]# rpcinfo -p

program vers proto port

100000 2 tcp 111 portmapper

100000 2 udp 111 portmapper

. . . . . .

이 데몬이 떠 있지 않다면 다음과 같은 명령을 사용하여 portmapper를 먼저 띄우고 nfs를 재시작 해보자.

2.3.2.5. ssh를 사용하는 MPICH

ch_p4방식으로 MPICH를 설치하려면 원격으로 명령을 실행할 수 있는 도구인 rsh가 꼭 필요하다. 하지만 rsh는 보안에 취약하기 때문에 인터넷에 노출된 네트워크에서는 사용자 인증과 데이터를 암호화해서 전송하는 ssh를 대신 사용한다. 클러스터도 내부 네트워크가 외부에 노출되어 있어서 노드를 직접 볼 수 있으면 노출된 rsh데몬이 스니핑이나 패킹 표적이 될 수 있으므로 ssh를 대신 사용한다.

ssh는 사용자 기반 인증을 사용하기 때문에 ssh로 다른노드에 명령을 내릴 때마다 암호를 입력한다. 따라서 클러스터에서 rsh 대신 ssh를 사용하려면 ssh의 인증방법을 바꾸어야 한다.

ssh2는 password, publickey, hostbased 세 가지 인증 방법을 사용한다.

password인증은 암호를 입력해서 사용자를 인정하는 것으로 기본 값이다.

publickey 인증은 자신과 원격 호스트가 갖고 있는 키로 접속을 요청한 상대를 검사합니다. 그리고 hostbased는 rsh와 같이 사용자의 .shosts 파일이나 /etc/ssh/shosts.equi파일에 있는 호스트를 신뢰한다고 생각하고 암호 없이 접속을 허용한다.

2.3.2.6. ssh를 호스트 기반 인증으로 바꾸기

⊙헤드 노드의 디렉토리를 복사해서 계산 노드 디렉토리를 만들었으면 계산 노드의 ssh키로 헤드 노드와 같다. 이것은 올바른 설정이 아니므로 노드마다 자신의 고유키를 만들도록 /etc/ssh 밑의 ssh_host로 시작하는 파일을 전부 삭제한다. 그리고 ssh데몬을 재시작하거나 시스템을 재부팅하면 새로운 키를 자동을 만든다.

⊙openssh는 ssh1과 ssh2를 모두 지원하는데 기본으로 ssh2를 사용한다.. 그리고 ssh2가 사용하는 기본 키는 ssh_host_rsa_key이다.

ssh hostbased 인증은 rsh와 같이 전체 사용자에게 적용할 때에는 /etc/ssh/shosts.equiv, 사용자마다 적용하려면 $HOME/.shosts 파일을 사용한다. 파일의 형식은 rsh의 것과 같다. 단 단축 이름이 아니라 JBCERT.cluster.domain같이 /etc/hosts에 있는 전체 이름을 쓰도록 한다.

⊙ssh가 hostbased인증에서 shots.equiv에 있는 신뢰하는 호스트로부터 연결 요청을 받으면 그 호스트의 공개키가 자신이 갖고 있는 키 목록에 있는 확인한다. 키가 일치하면 신뢰하는 호스트가 틀림없다고 판단해서 암호 없이 접속을 허용한다. 그리프로 hosts.equiv에 있는 이름만을 확인하는 rsh보다 보안을 높일 수 있다.

그러므로 ssh서버는 신회하는 호스트 목록과 신뢰하는 호스트의 키 목록을 같이 갖고 있어야 한다. 이 파일은 /etc/ssh/ssh/known_hosts 또는 /etc/ssh/ssh_known_hosts2이다. 둘 중 하나만 있으면 된다.

이것은 신뢰하는 호스트의 /etc/ssh/ssh_host_rsa_key.pub에 있는 공개키 모두를 하나의 파일로 합쳐 놓은 것이다.

[root@JBCERT ~]# vi /etc/ssh/ssh_host_rsa_key.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAr1APCJrn8pqvwWG9qQdceBa47MkpUCcffGlCABJI630U7gYyqcgssN+ XtwDFEoCKBibyp9dJbwnnQLgl+ o/asKhyrrtNcnDWCnyKQ4KbyHOmdUfnL9DV8hQPE2TFYriigq0y9o/ALge2TU8U//oMa4gZjdaotq9VqtkRfbhisxk=

노드마다 파일 내용을 단순히 하나로 합치면 어떤 것이 어떤 노드의 키인지 알 수 없으므로 ssh_known_hosts 파일은 앞에 다음과 같이 노드 이름이 추가된다.

[root@JBCERT ~]# head -2 /etc/ssh/ssh_known_hosts2

JBCERT.cluster.domain,JBCERT,10.1.1.1 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAr1APCJrn8pqvwWG9qQdceBa47MkpUCcffGlCABJI630U7gYyqcgssN+ XtwDFEoCKBibyp9dJbwnnQLgl+ o/asKhyrrtNcnDWCnyKQ4KbyHOmdUfnL9DV8hQPE2TFYriigq0y9o/ALge2TU8U//oMa4gZjdaotq9VqtkRfbhisxk=

compute-0-0.local,compute-0-0,10.1.1.254 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA3V6Et6EFqEJXN74RCp7MmjJpKFqO+ kRwdK3XxG3T91kn10B0P4Qqben5HyWbTxEGXAZLuZ9Vgb7oVD39aTkLTkq56CLNjiaEjKj7P8jbekgg98WWSucvPxaOPQK/YP+ QADf1cfqhJdIkPs/PIppzmBJjF+ 0WHF0r3n3YW39RnQk=

각 노드를 “ssh-keyscan -t rsa 계산노드명“을 이용해서 헤드노드의 /etc/ssh/ssh_known_hosts2의 넣는다.

⊙설정 파일인 /etc/ssh/ssh_config와 /etc/ssh/sshd_config파일을 편집합니다. 파일의 내용은 다름 부분은 변동 없이 다음 한 줄만 no에서 yes로 변경한다.

HostbasedAuthentication yes

일반 계정사용하려면 ssh_config에 다음과 같이 키워드를 추가한다.

EnableSSHKeysign yes

⊙앞에서 편집한 ssh_config와 sshd_config, /etc/ssh/shost.equiv와 /etc/ssh/ssh_known_host2 파일을 모든 노드에 복사하고 노드에서 ssh데몬을 재시작하면 hostbased 인증을 쓸 수 있다.

2.3.3 계산노드의 리눅스 설치

- 헤드노드에 root로 로그인 한다.
- 계산노드의 DHCP 요청을 cupture 해서, Rocks MySQL database에 입력하기 위해 다음을 실행하라.

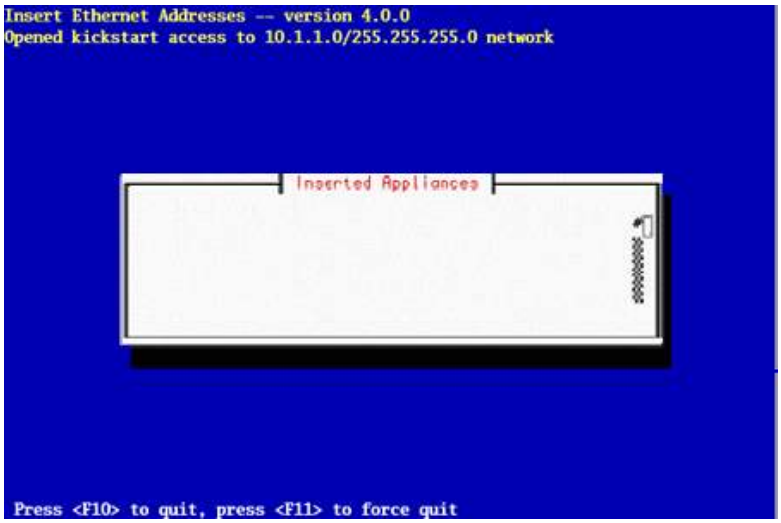
#insert-ethers

<Insert-Ethers>

✝ insert-ethers 명령은 managed 스위치를 위한 DHCP 요청을 capture 하여 이것을 Ethernet switch로 설정하며, 이 정보를 헤더노드의 MySQL database 에 저장한다.

초기값, 즉 'Compute'를 선택하여 계산 노드에 대한 설치를 시작한다..

- 다음과 같은 화면이 나올것이다.



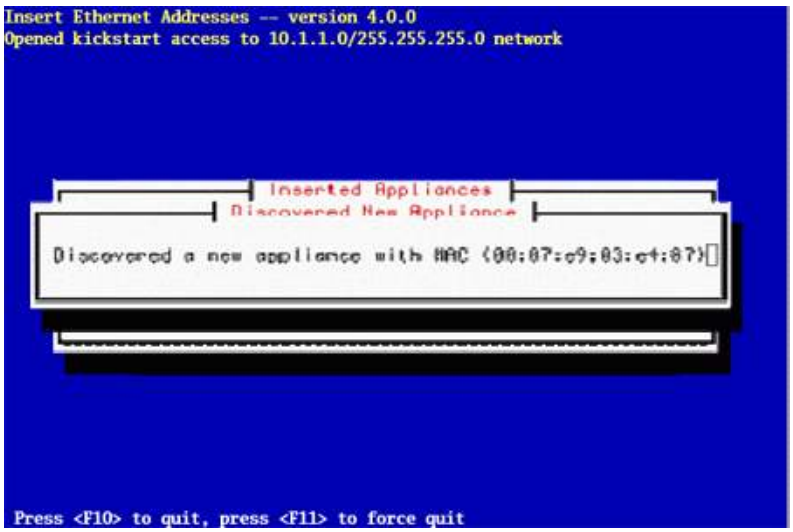
이것은 insert-ethers가 새로운 계산 노드를 기다리고 있음을 보여준다

- Rocks base CD를 설치하고자 하는 첫 번째 계산노드에 넣는다. 클러스터의 계산노드의 IP 주소는 10.1.1.254로부터 역순으로 주어진다.

✝ 만일 계산노드에 CD 드라이버를 갖고있지 않다면, PEX 또는 Boot 플로피를 이용하여야 한다.

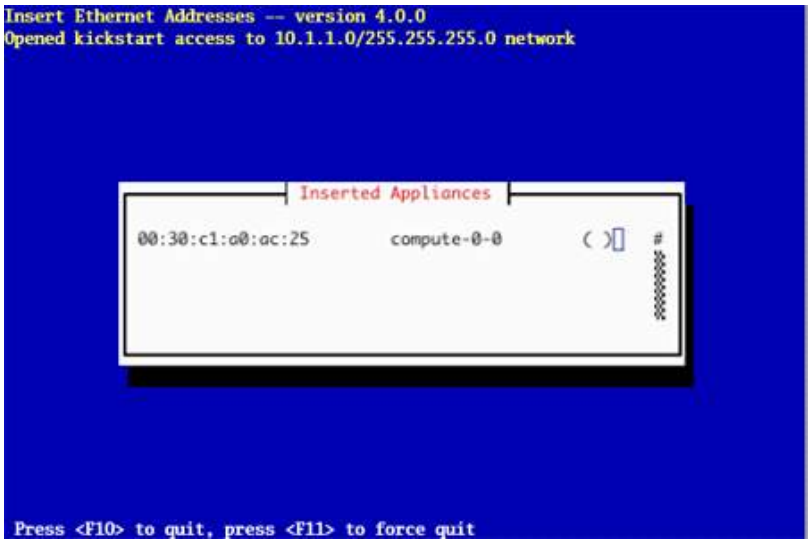
- 설치하고자 하는 첫 번째 계산노드를 켠다

- 헤더노드가 DHCP요청을 방금 전에 켠 계산 노드로부터 받는다면 다음과 같은 화면을 볼 것이다.



이것은 inert-ethers가 DHCP 요청을 계산 노드로부터 받고 이 계산노드에 대한 기본정보(hostname, Mac address, node type 등)을 MySQl database에 저장하고 나서 관련된 모든 설정파일(예로 /etc/hosts, /etc/dhcp.conf, batch system f 파일, 그리고 NIS 설정 파일)을 수정함을 의미한다.

앞의 하면이 몇 초간 보이고 나서 다음화면으로 바뀐다. 첫 번째 계산 노드의 이름이 compute-0-0로 자동으로 주어진다.



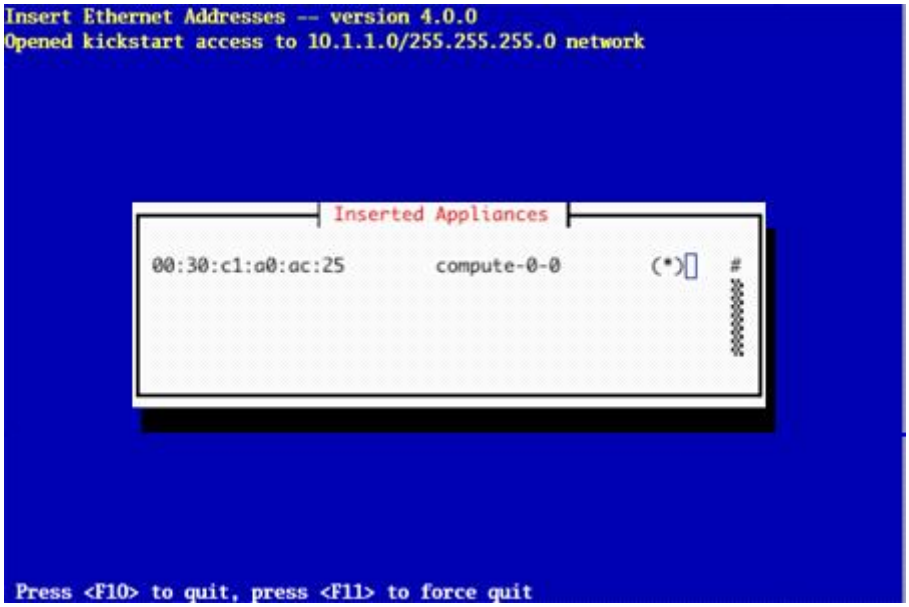
Insert-ethers가 하나의 계산노드를 발견했음을 위의 그림은 보여준다. 계산노드의 이름 즉, compute-0-0 옆의 ()는 해당 노드가 아직 Kickstart 파일을 요청하지 않고 있음을 보여준다.

이렇게 insert-ethers가 계산노드를 발견할 때 마다 볼 수 있다.

다음과 같은 과정을 통해 인증의 효과를 얻을 수 있다. insert-ethers가 실행중일 때 헤더노드는 자신이 속한 localnetwork 상에 등장한 새로운 노드를 받아들일 것이다.

insert-ethers는 새 노드가 kickstart파일을 요청할 때 까지 실행 중이어야 한다. 또 위의 kickstart 파일 요청이 있을 때까지 사용자가 기다리도록 요청할 것이다. 만일 insert-ethers가 꺼져 있다면, 알려지지 않은 이미의 노드가 kickstart 파일을 얻을 수는 없다.

☥ kickstart 파일은 plain text 형태로 가지고 있기 때문에, 이것은 암호화된 후 네트워크 상에 보내져야 한다. 부가적으로 오직 "인식된" 노드만이 이 kickstart파일을 요청하는 것이 허용된다. Insert-ethers는 새로운 노드를 추가하는 도구로 사용되기 때문에 주의해서 사용할 필요가 있다. 보안이 문제가 될 수 있는 상황이면, insert-ethers상에 보이는 "미지의" MAC address에 대해 의심해 볼 필요가 있다.



위에 그림은 계산노드가 성공적으로 kickstart파일을 헤더노드로부터 요청했음을 보여준다. 더 이상의 계산노드가 없다면, insert-ethers를 중지시킬 수 있다. Kickstart 파일은 https를 통해 보내지므로, 전송 중에 error가 발생하면, HTTP 에러를 "(*)"대신 보게 될 것이다.

- 이제부터 telnet을 통해 계산 노드의 설치 과정을 모니터할 수 있다. 즉 다음과 같은 명령을 헤더노드에서 실행하면 된다. (hostname 은 파일의 insert-ethers에서 주어진 이름을 사용하면 된다.).

#ssh compute-0-0 -p 2200

- 계산노드의 설치 과정이 끝나면 CD가 나올 것이다. CD를 꺼내어 다음 계산노드에 넣은 후 설치 과정을 계속하면 된다.

- 모든 계산 노드의 설치가 끝나면 F10키를 사용하여 insert-ethers를 중지시킨다.

- 만약 cabinet을 사용한다면 cabinet에 모든 계산 노드를 설치하고 다음 cabinet에 설치할 때에는 다음과 같이 구분해 주면 된다.

#insert-ethers --cabinet=1

이렇게 하면, 설치될 새로운 계산노드의 이름은 e compute-1-0, compute-1-1, ... 과 같

이 될 것이다.

2.3.4 방화벽 설정

1. ssh 설정을 통해서 접근할 수 있는 유저를 제한한다.
[root@JBCERT ~]#/etc/ssh/sshd_config
AllowUsers 접근을허용할유저1 접근을허용할유저2
[root@JBCERT ~]#/etc/rc.d/init.d/sshd restart

2. ssh 포트를 변경한다.
[root@JBCERT ~]#/etc/ssh/sshd_config
#port 22 라인의 주석을 해제 후 자신이 원하는 포트로 변경
[root@JBCERT ~]#/etc/rc.d/init.d/sshd restart

3. tcp/wrapper를 이용하여 ssh 접근을 허용할 네트웍을 제한하다.

[root@JBCERT ~]# vi /etc/hosts.deny

sshd: ALL (모든 ssh 접속을 막는다.)

[root@JBCERT ~]# vi /etc/hosts.allow

sshd: 접근을허용할아이피 접근을허용할네트웍

4. iptables 방화벽을 이용하여 ssh접근을 허용할 네트웍을 제한한다.

##

[root@JBCERT ~]# vi /etc/sysconfig/iptables

*nat

-A POSTROUTING -o eth1 -j MASQUERADE

COMMIT

*filter

:INPUT ACCEPT [0:0]

:FORWARD DROP [0:0]

:OUTPUT ACCEPT [0:0]

Preamble

-A FORWARD -i eth1 -o eth0 -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT

-A FORWARD -i eth0 -j ACCEPT

-A INPUT -p tcp -s 10.1.1.2 --dport 22 -j ACCEPT 22

```
-A INPUT -p tcp -s 61.81.108.3 --dport 22 -j ACCEPT 22
-A INPUT -i eth0 -j ACCEPT
-A INPUT -i lo -j ACCEPT

# Allow these ports
-A INPUT -m state --state NEW -p tcp --dport ssh -j ACCEPT
# Uncomment the lines below to activate web access to the cluster.
#-A INPUT -m state --state NEW -p tcp --dport https -j ACCEPT
#-A INPUT -m state --state NEW -p tcp --dport www -j ACCEPT

# Standard rules
############################################################
```

2.4. 암호 분석 프로그래밍

2.4.1. MPI 프로그래밍 기초(Message Passing Interface)

병렬프로그래램 데이터를 나누고 결과를 모아 오는 과정만 일반 프로그래밍에 추가하면 된다. 그러므로 어떤 병렬 프로그래밍도 평소에 쓰는 알고리즘으로 코드를 짠 다음 이것을 쉽게 병렬 프로그램으로 바꿀 수 있습니다.

2.4.1.1 MPI 프로그램의 기본구조

MPI-1은 SIMD모델을 기본으로 하므로 노드마다 같은 프로그램을 실행한다. 프로그램은 다음과 같은 기본 구조로 되어 있다.

```
preprocessor 선언
int main(int argc. char *argv[])
{
        변수 선언;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &innode);

        실행 본문;
        for(;;)
        {
                병렬화된 실행 본문;
        }
```

```
        실행본문;
        MPI_Finalize();
}
```

프로그램에서 다른 노드에 나누어 실행될 부분은 MPI_Init()로 시작해서 MPI_Filnalize()끝난다. 여기서 알아둘 것은 클러스터 노드에 동일한 실행 코드를 가진 프로그램을 실행시키면 그 프로그램은 모든 노드에서 똑같이 실행된다는 것이다.

간단한 1부터 100까지 더하는 프로그램이 있다면, 더하는 부분을 병렬로 처리하기 위해서 처음과 끝부분을 MPI_init()와 MPI_Finalize()로 선언하기만 하고 프로그램을 컴파일패서 실행 한다. 어떤 결과가 나올까?

```
compute-0-0 : 1+2+3+…+100=5500   -------┐
compute-0-1 : 1+2+3+…+100=5500   ------ └--JBCERT : 5500+5500+5500
compute-0-2 : 1+2+3+…+100=5500   -------┘          +5500=22000
compute-0-3 : 1+2+3+…+100=5500   -------┘
```

이 프로그램을 노드 4개를 써서 실행하면 모든 노드가 똑같이 프로그램 코드를 실행한다. 즉 모든 노드가 1부터 100까지 합을 구해서 결과를 돌려주기 때문에 노드의 결과를 모으면 5050의 4배가 된다. 즉, 프로그램이 노드마다 서로 다른 일부분씩 작업을 처리하도록 하는 것은 프로그래머가 직접 해야한다.

프로그래머가 프로그램이 작업을 나누어 실행하게 하려면 무엇을 알아야 할까? 먼저 작업을 몇 개 노드에서 실행할 것인지 알아야 하고 프로그램 스스로가 자신이 어떤 노드에서실행되고 있는지 실행 중에 판단할 수 있어야 한다.

즉 1부터 100까지 병렬로 실행하려면 실해하는 노드 수를 얻어서 수를 합하는 작업을 1~10, 11~20, …등으로 어떻게 나눌 것인지 결정한 후, 소러 다른 노드에 이것을 중복 되지 않도록 분배합니다. 프로그램은 모든 노드에서 동일하게 실행되므로 실행 중간에 자신이 어떤 노드에서 실행되고 있는지 검사해서 자신이 노드1에서 실행하면 된다. 1~10, 노드2에서는 실행되면 11~20, …등으로 자신이 계산할 작업 분량을 스스로 결정하게 하면 모든 노드에 중복 없이 작업ㅇ을 분배할 수 있다. mpirun으로 프로그램이 시작하면 MPI는 사용할 노드의 이름과 숫자를 저장한 통신자(communicator)라는 변수를 시작할 때 정의한다. 프로그램중간에 MPI의 모든 함수는 통신자로부터 노드수의 자신의 순번(rank) 같은 병렬 실행에 필요한 정보를 얻을 수 있다.

통신자 MPI_Init()에서 정의되고 MPI_Finalize()에서 해제 된다. 그러므로 병렬로 실행되는 부분은 반드시 두 함수 상에 있어야한다.

MPI는 기본으로 mpirun에서 정해진 노드 정보를 MPI_COMM_WORLD라는 이름으로 미리 정의된 통신자에 저장한다. 그러므로 대개 기본 통신자로 이 변수만 사용해도 충분하다.

그러면 프로그램이 노드 수와 자신의 순번을 사용할 수 있으니 그것을 상요해서 앞의 1부터 100까지 합 프로그램을 다시 프로그래밍하면 다음과 같이 된다.

```
compute-0-0 : 1+ 2+ 3+ …+ 25=325      -------┐
compute-0-1 : 26+ 27+ …+ 50=950       ------- └--JBCERT : 425+ 950+ 1275+ 2200
compute-0-2 : 51+ 52+ …+ 75=1275      -------┘                =5500
compute-0-3 : 56+ 57+ …+ 100=2200     -------┘
```

MPI함수로 노드 수와 순번을 얻고 프로그램 단계에서 순번에 따라 다른 값을 합하여도록 하면 수열 전체를 노드가 나누어 합을 구할 수 있다.

2.3.1.2. MPI 프로그래밍 함수

MPI에서는 120개가 훨씬 넘는 무수히 많은 함수들을 정의해 놓고 있다. 이들 대부분은 6개의 기본적인 함수들의 조합으로 구현될 수 있으며 이들 함수는 아래와 같다.

MPI_Init(&argc, &argv)

프로그램 실행 인수들과 함께 mpi함수들을 초기화 해준다.

MPI_Finalize()

mpi함수들을 종료한다. 모든 mpi함수들은 MPI_Inti() 과MPI_Finalize() 사이에서 호출되어야 한다.

int MPI_Comm_rank(MPI_Comm comm, int *rank)

comm 커뮤니케이터에서 자신의 프로세스 id를 얻는다.

int MPI_Comm_size(MPI_Comm comm, int *size)

comm 커뮤니케이터에서 실행되는 프로세스의 개수를 얻는다.

int MPI_Send(void *message, int count, MPI_Datatupe datatype, int dest, int tag, MPI_Comm comm)

dest로 메시지를 보낸다.
message는 보내고자 하는 메시지를 저장하고 있는 버퍼
count 는 보낼 메시지 개수
datatype는 보낼 메시지 타입
dest는 보내고자 하는 프로세스 id
tag는 보내는 메시지에 대한 꼬리표
comm은 dest와 자신의 프로세스가 속해있는 커뮤니케이터

int MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

source로부터 메시지를 받는다.
message는 받은 메시지를 저장할 버퍼
count는 받을 메시지 개수(받는 메시지 개수보다 작으면 에러 발생)
datatype은 받을 메시지 타입
source는 메시지를 보내주는 프로세스 id
tag는 받은 메시지를 확인하기 위한 꼬리표(MPI_Recv에서의 tag와 MPI_Send에서의 tag가 같아야 한다)
comm은 source와 자신의 프로세스가 속해있는 커뮤니케이터
status는 실제받은 데이터에 대한 정보(source와 tag)

2)Collective Communication을 위한 함수

Communicator란 병렬 실행의 대상이 되는 프로세스 그룹으로서 정의된다. 기본적으로 MPI프로그램에서는MPI_COMM_WORLD라는 기본 Communicator가 생성되는데 이는 동시에 수행되는 모든 병렬 프로세스를 포함한다. 그러나 사용자는 필요에 따라 임의의 프로세스로 구성된 새로운 Communicator를 생성할 수 있기도 하다. 이러한 Communicator에서 정의되는 프로세스간의 통신을 위한 함수들을 다음에 보였다.

int MPI_Bcast(void *message, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

comm으로 정의된 Communicator에 속한 모든 프로세스들에게 동일한 메시지를 전송한다.

root는 메시지를 뿌려주는 프로세스id

root의 message에만 보내고자 하는 데이터가 들어있고, 다른 프로세스의 message는 받은 메시지를 저장할 공간이다.

int MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

모든 프로세스들이 MPI_Reduce를 수행하면 모든 operand가 op에 따라 연산을 수행하고 그 결과가 root 프로세스의 result에 저장된다.

operand는 연산이 적용될 피연산자

result는 연산결과가 저장될 버퍼

op는 어떤 종료의 연산이 수행될 것인지를 알려주는 OP-CODE

root는 연산결과가 저장될 프로세스 id

int MPI_Barrier(MPI_Comm comm)

comm Communicator에 속한 모든 프로세스들이 MPI_Barrier를 호출할때까지 block시킴으로서 동기화를 시킨다.

int MPI_Gather(void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, int root, MPI_comm comm)

root프로세스는 각 프로세스들이 보내준 자료(send_buf)를 프로세스 id 순으로 root의 recv_buf에 차곡차곡 쌓는다.

int MPI_Scatter(void *send_buf, int send_count, MPI_Datatype send _type, void *recv_buf, int recv_count, MPI_Datatyp recv_type, int root,MPI_Comm comm)

send_buf의 내용을 send_count의 크기로 잘라서 모든 프로세스들의 recv_buf로 순서대로 전송한다. MPI_Gather와는 반대의 역할

int MPI_Allgather(void *send_buf, int send_count, MPI_Dataltype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, MPI_comm comm)

MPI_gather와 마찬가지로 모든 프로세스의 send_buf의 내용을 모으나, 모든 프로세스의 recv_buf에 저장하는 것이 다르다.

int MPI_Allreduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op, MPI_Comm comm)

MPI_Reduce와 마찬가지로 모든 프로세스의 operand에 op연산을 적용하지만 그 결과를 모든 프로세스의 result가 가지게 된다.

3) Communicators and Topologies

다양한 커뮤니케이터를 만들고 프로세서들의 위상을 선언할 수 있는 함수들로 구성된다.

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

comm에 포함된 프로세스 그룹을 얻어온다.

int MPI_Group_incl(MPI_Group old_group, int new_group_size, int *ranks_in_old_group , MPI_Group *new_group)

old_group에서 rank_in_old_group의 id를 가지고 있는 new_group_size개수의 프로세스들로 새로운 그룹 new_group을 만든다.

int MPI_Comm_creat(MPI_Comm old_comm, MPI_Group new_group, MPI_Comm *new_comm)

그룹을 커뮤니케이터로 만들어준다. (이로써 서로간의 통신이 가능해진다.) 이것은 MPI_Comm_group, MPI_Group_incl와는 달리 Collective operation 단독으로 실행되서는 안되고, 반드시 모든 프로세스가 동시에 수행되야 하는 명령이다.

int MPI_Comm_split(MPI_Comm old_comm, int split_key, int rank_key, MPI_Comm *new_comm)

동일한 split_key로 호출된 프로세스들을 묶어서 동일한 커뮤니케이터로 만든다. rank_key는 프로세스 rank를 결정할 때 작은 순서대로 작은 id를 부여 받는다. 예를 들면, 프로세스 0, 1, 2번은 split_key 1로 호출하고, 프로세스 3, 4, 5번은 split_key 2로 호출했을 때, 동일한 이름(new_comm)의 두 개의 커뮤니케이터가 생성된다. 만약 0번 프로세스에서 new_comm으로 MPI_Bcast를 호출하면 0, 1, 2번으로 전송되고, 3번 프로세스에서 new_comm으로 MPI_Bcast를 호출하면 3, 4, 5번으로 전송된다.

int MPI_Cart_create(MPI_Comm old_comm, int number_of_dims, int *dim_sizes, int *periods, int reorder, MPI_Comm *cart_comm)

새로운 커뮤니케이터 cart_comm을 생성해준다. old_comm과 cart_comm에 속해있는 프로세스들은 같지만 그들의 id는 같지 않다. cart_comm의 프로세스들은 number_of_dims차원의 매트릭스(dim_size *dim_size ...)로 재구성되며, 그에 맞게 새로운 id를 갖게 된다. periods는 각 dimention이 circular인지 linear인지의 flag.

reorder는 재구성을 하면서 프로세스 id변환을 허용하는 flag.

int MPI_Cart_rank(MPI_Comm comm, int *coordinates, int *rank)

coordinates[]의 정보를 가지고 그에 해당하는 프로세스의 rank를 구한다. 예를 들어 coordinates[]={2,4} 이면 프로세스의 매트릭스에서 2행4열의 프로세스 id를 리턴한다.coordinates[]는 몇열 몇행의 정보를 가지고 있다.

int MPI_Cart_coords(MPI_Comm comm, int rank, int number_of_dims, int *coordinates)

rank를 가지고 그에 해당하는 coordinates[]를 구한다. 예를 들어 2행 4열에 위치한 프로세스 id가 6일때, rank=6로 함수를 호출하면 coordinates[]={2,4}를 리턴 한다.

int MPI_Cart_sub(MPI_Comm grid_comm, int varying_coords, MPI_Comm *comm)

2 by 2 매트릭스인경우 2개의 커뮤니케이터를 만든다. MPI_Cart_sub(grid_comm, {0,1}, comm) 을 사용하여 각 열을 동일한 커뮤니케이터로 묶어준다. varying_coords[]는 각 dimention이 comm에 속하는지를 알려주는 Boolean형이다.

2.4.2 MPI 프로그래밍-암호분석

2.4.2.1. crypt()함수를 이용한 shadow파일 만들기

```
#include <string.h>
#include <stdio.h>

int main()
{
        char crp1[255];
        char salt[3];
        char filename[15];
        FILE *shadow;

        printf("Enter shadow filename : ");
        scanf("%s",filename);
        printf("₩nEnter passphrase : ");
        scanf("%s",crp1);
        printf("₩nEnter 2 character cryption key : ");
        scanf("%s",salt);
        shadow=fopen(filename,"wt");
//fopen한 파일에 crypt()를 이용해 crp1 배열을 이용하여 DES로 암호화 하고 저장한다.
        fprintf(shadow,"%s",crypt(crp1,salt));
        fclose(shadow);
        printf("₩n₩nFile %s is successfully created.₩n",filename);
        return 0;
}
```

소스 컴파일

[SMS@JBCERT ~]$ gcc makeshadow.c -o makeshadow -lcrypt

2.4.2.2. MPI 프로그래밍을 이용한 암호분석

◆적당한 길이의 문자열 배열과 그 길이와 같은 정수 배열을 준비한다. 정수 배열이 문자열에 대응되는 카운터 역할을 한다.

◆시작할 길이를 정하고 그것을 저장할 변수를 지정한다. 그리고 그 길이만큼 정수 배열을 1로 채운다.

◆배열 첫 자리를 1부터 62까지 반복해서 채운다. 그리고 만들어진 정수 배열의 자리마다 해당하는 숫자를 문자열 배열에 대응해서 문자로 넣는다.

◆배열 첫 자리에서 62가 끝나면 뒷자리에 1을 더한다. 그리고 배열 요소마다 값을 검사한다. 63이 된 값이 있으면 뒷자리에 1을 더하고 그 자리는 1로 바꾼다. 이것을 정해진 배열 길이까지 반복한다.

◆정해진 배열 길이의 마지막에 저장된 값이 63이 되면 배열 길이를 1 크게 하고 그 길이까지 1로 채운다. 그리고 위 3번째와 4번째를 반복한다. 이렇게 하면 문자열은 000, 100, 200, …, xzz, yzz, zzz, 0000, 1000, …식으로 만들어지게 된다.

```
#include <unistd.h>
#include <string.h>

#include <stdio.h>
#include <mpi.h>
#define sizeof 256
int main(int argc, char *argv[])
{
//변수를 선언하고 문자열 배열을 초기화 한다.
        char salt[3];
        char str[sizeof];
        char crp1[sizeof];
        char crp2[sizeof];
        int count[sizeof];
        int i,j,nc=62,len=3;
        FILE *shadow;
        int iproc,nproc;
        int stopflag=0,tot_stopflag=0,loopcount=0;
//MPI 병렬 프로그램을 시작한다.
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&nproc);
        MPI_Comm_rank(MPI_COMM_WORLD,&iproc);

        for(i=0;i<3;i++) salt[i]='₩0';
        for(i=0;i<sizeof;i++)
        {
                str[i]='₩0';
                crp1[i]='₩0';
                crp2[i]='₩0';
        }
//인자로 받은 파일 이름으로 파일을 열고 암호문을 읽어 들인다.
        if(argv[1]=='₩0')
        {
                if(iproc==0) printf("usage : decrypt filename₩nexcute again with
filename.₩n");
                MPI_Finalize();
```

```
                exit(1);
        }
        if(iproc==0)
        {
                shadow=fopen(argv[1],"rt");
                fgets(crp1,sizeof,shadow);
                strncpy(salt,crp1,2);
                fclose(shadow);
        }
        MPI_Bcast(salt,3,MPI_CHAR,0,MPI_COMM_WORLD);
        MPI_Bcast(crp1,sizeof,MPI_CHAR,0,MPI_COMM_WORLD);

//정수 배열을 초기화하고 3자리 문자열 000부터 시작한다.
        for(i=0;i<sizeof;i++) count[i]=0;
        for(i=0;i<=len-1;i++) count[i]=1;

        for(;;)
        {
//배열 첫 자리를 1부터 62까지 채운다.
                for(i=iproc+1;i<=nc;i+=nproc)
                {
                        count[0]=i;
//정수 배열의 모든 자리를 문자열 배열에 문자로 바꾸어 저장한다.
                        for(j=0;j<=len-1;j++)
                        {
                                str[j]=count[j]+47;
                                if(count[j]>=11) str[j]=count[j]+54;
                                if(count[j]>=37) str[j]=count[j]+60;
                        }
                        str[len]='₩0';
//만들어진 문자열을 암호화해서 암호문과 검사한다.
                        strcpy(crp2,crypt(str,salt));
                        if(!strcmp(crp1,crp2))
                        {
                                printf("₩nfind passphase!₩n");
                                printf("passphase is %s₩n₩n",str);
                                stopflag=1;
                        }
                }
//배여 첫 자리 대입이 끝나면 뒷자리를 다음 문자로 바꾼다.
                count[1]+=1;
```

```
//배열을 검사해서 64(z)보다 큰 값이 있으면 다음 문자로 바꾼다.
                for(j=1;j<=len-2;j++)
                {
                        if(count[j]>nc)
                        {
                                count[j+1]+=1;
                                count[j]=1;
                        }
//문자열의 마지막 문자가 z 다음이 되면 길이를 하나 늘려주고 다시 시작한다.
                        if(count[len-1]>nc)
                        {
                                count[len]=1;
                                count[len-1]=1;
                                len+=1;
                                if(iproc==0) printf("Begin decrypting passphase of
length %i₩n",len);
                        }
                }
//첫 자리 외에 다른 자리를 바꾸는 부분은 무한 반복문에 속해 있으므로  MPI_Barrier()
등 메시지 전달을 적당한 간격으로 해 주어야 하므로 loopcount라는 카운터 변수를 넣어
1천 번 반복될 때마다 메시지 전달을 하도록 했다.
                if(loopcount>1000)
                {
                        MPI_Barrier(MPI_COMM_WORLD);

MPI_Allreduce(&stopflag,&tot_stopflag,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
                        if(tot_stopflag)
                        {
                                MPI_Barrier(MPI_COMM_WORLD);
                                MPI_Finalize();
                                if(iproc==0)        printf("₩nprogram        succesfully
finished.₩n");
                                exit(1);
                        }
                        loopcount=0;
                }
//문자열의 길이가 80이 넘어가면 강제로 종료한다.
                if(len>8)
                {
                        printf("Error!      cannot      find      passphase.₩n₩nprogram
terminated.₩n");
                        MPI_Finalize();
```

```
                        exit(1);
                }
        loopcount+=1;
        }
}
```

암호분석 프로그램 MPI 컴파일

[SMS@JBCERT ~]$ /opt/mpich/gnu/bin/mpicc de2-mpi.c -o de2-mpi -lcrypt

2.4.3. 쉘 프로그래밍을 이용한 User Interface

2.4.3.1. Shell 프로그램이란?

Unix Shell이 처리 가능한 명령문들을 파일로 작성하여 Shell이 일관 처리 방식으로 수행하며 Shell Procedure, Shell File, Shell, Shell Script 라고도 한다. 그리고 일반적으로 복잡한 일련의 명령어의 사용을 자동으로 처리하게 하거나 일반적인 유닉스 명령과 특정 쉘에서만 유효한 명령으로 구성되어 있다.

Shell 프로그램의 실행 : 쉘이 쉘 스크립트파일을 한 줄 단위로 읽어 들여 해석하여 실행한다. (인터프리터 방식)

2.4.3.2. Shell 프로그램의 종류

일반적인 언어로는 Bourne Shell 프로그래밍 언어, C Shell 프로그래밍 언어 등이 있으며 쉘 프로그램 전역 해석기는 Perl, TCL 등이 있다

-이번 보고서에서 사용한 Shell 프로그래밍

<셀 프로그램을 이용한 User Interface Main화면>

위 프로그램을 만들기 위해서 사용된 셀프로그래밍 소스는 다음과 같다.

```
[SMS@JBCERT shadow]$ vi main.sh
```

```
#암호분석 프로그램 로고밑 메뉴
#!/bin/sh
clear
sh ./LOGO.sh
echo "Welcome to JBCERT Cluster"
for i in `seq 1 100`;
do
        echo "1)Find passwd (MPI programing)   2) EXIT "
        echo input:
        read SMS;
        clear
        case $SMS in
                1) ./cluster.sh;;
                2) exit;;
                *) echo "Not Number.";;
        esac
done
```

<Main.sh 소스 출력 화면>

```
[SMS@JBCERT shadow]$ vi cluster.sh
echo "1)make shadow file" ### 화면 출력부
echo "2)find shadow file"
echo "3)view shadow file"
echo "4)undo shadow file"
echo "*)previous"
read number
case $number in             ### 번호별 출력 프로그램
        1)./makeshadow;;
        2)/bin/ls -al;;
        3)./viewshadow.sh;;
        4)./mpich.sh;;
        *)exit;;
esac
```

<Cluster.sh 소스 출력 화면>

```
[SMS@JBCERT shadow]$ vi mpich.sh
#암화된 파일을 불러들여 mpirun을 이용해 암호분석
echo "Input the modification of the node which it will use:"
read nodenum
echo "Node Number $nodenum ? 1)YES 2)NO"
```

```
read yes
case $yes in
        1) echo "input makeshadow file name"
        read makeshadow
        echo "Node name $makeshadow ? 1)YES 2)NO"
        read yes1
        case $yes1 in
                1)echo `time /opt/mpich/gnu/bin/mpirun -machinefile nodelist -np
$nodenum de-mpi.out $makeshadow`;;
                2)exit;;
                *)echo "Dot't Input";;
        esac;;
        2) exit;;
        *) echo "Don't Input";;
esac
```

<mpich.sh 소스 출력 화면>

```
[SMS@JBCERT shadow]$ vi viewshadow.sh
#암호화한 파일이 암호화가 이루어 졌는지를 보여주는 프로그램
echo "Input the name of makeshadow files which you made : "
read makeshadow
echo `cat ./$makeshadow`
```

<viewshadow.sh 소스 출력 화면>

2.5. 분석결과-계산노드수의 따른 암호분석 결과

노드 수별로 분석한 결과입니다.

⦿compute 1개로 돌린결과 shadow파일

```
[SMS@JBCERT ~]$ time /opt/mpich/gnu/bin/mpirun -machinefile nodelist -np 2
de2-mpi shadow.5
Begin decrypting passphase of length 4
Begin decrypting passphase of length 5

find passphase!
passphase is 2k3nl



program succesfully finished.

real    57m53.128s
```

```
user    57m34.768s
sys     0m6.468s
```

⦿compute 2개로 돌린결과 shadow파일

```
[SMS@JBCERT ~]$ time /opt/mpich/gnu/bin/mpirun -machinefile nodelist -np 3
de2-mpi shadow.5
Begin decrypting passphase of length 4
Begin decrypting passphase of length 5

find passphase!
passphase is 2k3nl


program succesfully finished.

real    39m24.124s
user    39m11.086s
sys     0m5.531s
```

⦿compute 3개로 돌린결과 shadow파일

```
[SMS@compute-0-0 ~]$ time /opt/mpich/gnu/bin/mpirun -machinefile nodelist3
-np 4 de2-mpi shadow.5
Begin decrypting passphase of length 4
Begin decrypting passphase of length 5

find passphase!
passphase is 2k3nl


program succesfully finished.

real    29m52.528s
user    29m43.292s
sys     0m4.207s
```

⦿compute 5개로 돌린결과 shadow파일

```
[SMS@JBCERT ~]$ time /opt/mpich/gnu/bin/mpirun -machinefile nodelist3 -np 6
de2-mpi shadow.5
Begin decrypting passphase of length 4
Begin decrypting passphase of length 5
```

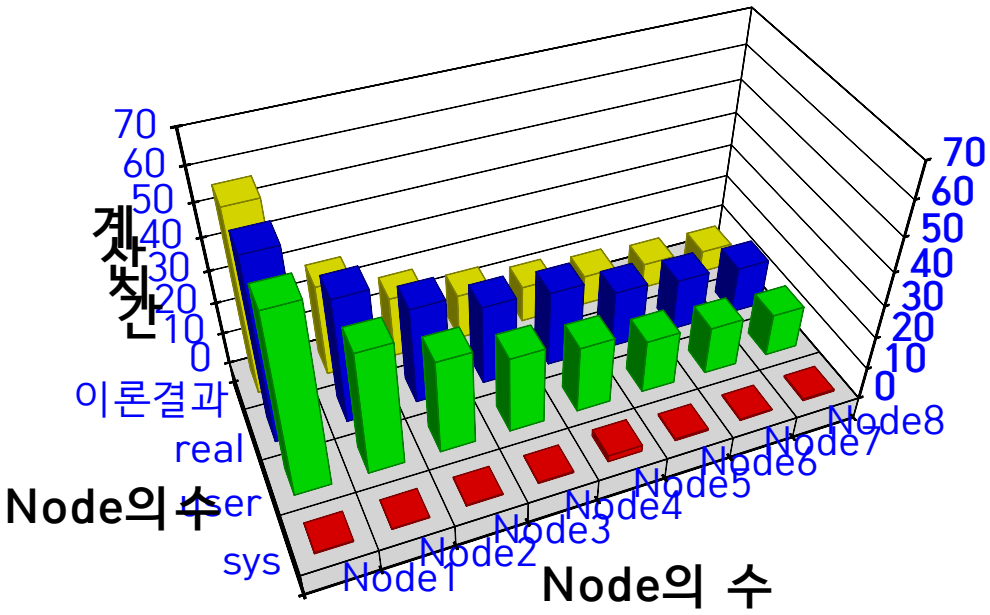
```
find passphase!
passphase is 2k3nl

program succesfully finished.

real    20m38.562s
user    20m27.146s
sys     0m7.027s
```

-Time 프로그램 각 시간의 뜻

real : 시작과 종료까지 모든 시간
user : user 레벨에서의 수행시간
sys : 커널 레벨에서의 수행시간

Time / Node N	Read	User	Sys	이론결과
Node 1	56.650s	55.980s	0.328s	56.65
Node 2	38.835s	38.111s	0.278s	28.32
Node 3	29.976s	29.108s	0.356s	18.88
Node 4	24.743s	23.737s	0.429s	14.16
Node 5	23.445s	20.789s	2.009s	11.33
Node 6	17.894s	16.595s	0.539s	9.44
Node 7	16.065s	14.846s	0.388s	8.09
Node 8	14.651s	13.162s	0.502s	7.08

<Node수의 따른 계산 시간 표>



<노드수의 따른 계산속도 그래프>

◆ 결론

본 보고서에서는 클러스터를 구축하고 계산 노드수의 따른 암호분석 시간의 대한 연구를 수행해 보았다. 위 그래픽에서 보듯이 프로세서 수의 따른 계산 처리 속도가 어떻게 변하는지를 봄으로서 클러스터링의 유용성을 입증한 것으로 하였다. 또한 클러스터 구축에 있어서도 여러 가지 프로젝트(beowulf 프로젝트로도 유명하다.)들이 있어 그렇게 어렵지 않음을 알 수 있고 우리가 이제는 일상 컴퓨터 작업에 있어서도 클러스터를 사용할 수 있을 것으로 기대된다.

그리고 프로젝트를 하면서 Node N개에 따라 클러스터의 성능이 N배 증가한다는 이론에 맞추어 Node 1개부터 9개 까지 구축함으로써 9배 빠른 계산 시간을 단축 할 수 있을 것으로 기대를 했으나 그 만큼에 기대 효과를 얻지는 못하였다. 이론과 같이 처음에 2대까지는 2배에 기대 효과를 얻었으나 그 후 계산 노드 추가부터는 2배에 기대 효과를 얻지 못하고 위의 분석 결과 그래프와 같은 반비례 포물선 그래프가 나왔다. 이 원인 네트워크 장애 및 여러 가지 여러 가지 시스템의 원인 인 것으로 생각된다.

이를 보완하기 위해서는 아래와 같이 연구 과제를 안고 있다.

더운 빠른 클러스터를 구축을 실현하기 위해서는 조금 더 연구해야 할 분야는 채널본딩 기법이나 이더넷 카드와 이더넷 스위치를 1GB로 바꾸어 봄으로써 네트워크 속도에 따른 계산 시간이 얼마나 단축 되는지 연구해 볼 필요성이 있다. 또한 헤더노드의 시스템을 더욱

빠른 것으로 업그레이드 함으로서 어느 정도 계산 시간이 단축 되는지도 해볼만한 과제다.
사용자의 입장에서 좀 더 편리하고 사용하기 쉽고 강력한 기능을 갖는 운영 도구의 개발을 필요로 한다. 개발자 관점에서는 고속 통신을 지원하는 네트워크, 자원을 통합 관리하는 클러스터를 위한 미들웨어, 입출력 속도를 향상시킬 수 있는 병렬 I/O, 사용자 중심의 간편한 프로그래밍 환경, 저장 장치, 소프트웨어 공학 등의 기술 지원들이 앞으로 더욱 연구되어야 할 것이다.
또한 이 연구를 하면서 발생한 문제였던 한 계산 노드가 속도가 떨어짐으로써 다른 계산 노드에도 영향을 주는 경우가 있었다. 이에 대한 해결 방안을 연구해 볼 필요성이 있다.

[참고 자료]

*서적

[1] 리눅스 & 윈도우 클러스터로 도전하는 슈퍼컴퓨터 구축과 활용 혜지원 2006년
[2] Linux System & Shell Programming - 윤성철 영진.com 2003년
[3] Mark Baker, Rajkumar Buyya, High Performance Cluster Computing: Architectures and Systems, Prentice Hall, 1999.
[4] Luis M. Silva, Rajkumar Buyya, High Performance Cluster Computing: Programming and Applications, Prentice Hall, 1999.

*참고 사이트

[1]Rocks Clusterrs-[http://www.rocksclusters.org/wordpress/]
[2]KLDP Wiki:응용프로그램 -
[http://wiki.kldp.org/wiki.php/%C0%C0%BF%EB%C7%C1%B7%CE%B1%D7%B7%A5#s-1.1]
[3] MPI Forum, at http://www.mpi-forum.org/docs/docs.html
[4] The beowulf project at CACR, at http://www.cacr.caltech.edu/beowulf/index.html
[5] Beowulf class Project, at http://www.beowulf.org/