클러스터 환경에서의 부하 분산 시스템 설계 및 구현

박은지(EUN-JI PAK) 최민(MIN CHOI)

박동근(DONG-GEUN PARK)

한국과학기술원 전산학과 컴퓨터 구조 연구실

2003년 7월 22일

요약

클러스터는 두 대 이상의 컴퓨터를 마치 하나의 시스템처럼 행동하도록 연결하여, 막대한 양의 계산이 필요하거나, 중단 없는 서비스를 수행하고자 할 때 사용하는 시스템이다. 연결하는 노드나, 네트워크의 종류에 따라 가격과 성능은 많이 다르지만, 슈퍼 컴퓨터에 비해 상대적으로 저렴한 비용으로 필요한 컴퓨팅 파워나 기능을 구현할 수 있다.

이런 클러스터 시스템에서는 동일 시간에 한 노드에서 많은 작업이 수행되는 경우 한 노드에만 부하가 몰리게 되고, 이런 부하집중 현상은 전체 시스템의 성능을 저하시킬 수 있다.

따라서 시스템의 상태를 파악하여 부하를 적절히 각 노드에 분배해줄 필요가 있는데, 사용자가 처음 작업을 시작하고자 할 때 가장 적합한 노드를 골라서 수행시켜 주거나, 필요한 경우 로컬 노드에 있는 작업을 다른 노드로 옮겨서 수행하는 방식으로 부하를 분배할 수 있다. 이를 위해 부하 분산 시스템을 설계, 구현하여 부하를 각 노드에 균등하게 분배함으로써 전체 시스템의 성능을 향상시키는 방향으로 작업을 수행하게 하였다.

목차

제 1 장 서론

제 2 장 관련 연구

2.1 부하분산 시스템의 개요

2.2 다른 부하 분산 시스템들

제 3 장 설치 및 실행환경

3.1 설치

3.2 실행환경 및 실행방법

제 4 장 부하 분산 시스템의 설계

4.1 전체 시스템의 구성

4.2 서버 쪽 부하 분산 시스템의 구조

4.3 클라이언트 쪽 부하 분산 시스템의 구조

제 5 장 부하 분산 시스템의 구현

5.1 서버 쪽 부하 분산 시스템의 각 부분의 역할

5.2 클라이언트 쪽 부하 분산 시스템의 각 부분의 역할

5.3 부하 정보의 관리

5.4 작업의 수행

5.5 작업정보의 관리

제 6 장 부하 분산 시스템의 성능 측정

6.1 성능 측정 환경

6.2 LINPACK 벤치마크

6.3 성능 측정 결과

제 7 장 결론 및 향후 과제

제 8 장 참고 문헌

제 1 장

서론

클러스터 시스템에서 높은 성능을 얻고, 전체 자원을 효과적으로 사용하기 위해서는 각 노드의 자원 상태를 파악하고, 현재 수행중인 작업을 잘 분배할 필요가 있다. 각 노드의 자원상태를 파악하고, 작업을 잘 분배하는 것은 부하 분산의 기본 기능이며 이를 위해 부하 분산 시스템에서는 전체 클러스터 시스템의 자원 정보를 관리하고, 현재 수행중인 작업들에 대한 정보를 관리함으로써 가장 적합한 노드에서 작업을 수행할 수 있게 함으로써 전체 클러스터 시스템의 성능을 향상시키는 것을 목표로 하고 있다. [1]

시스템의 자원을 항상 감시해서 가장 최근의 자원 정보를 기반으로 하여 사용자가 수행하고자 하는 작업이나, 현재 각 노드에서 수행중인 작업의 정보를 살피고, 적절히 작업을 노드에 분배해 주고, 필요한 경우 작업들을 노드 사이에서 이동시켜 수행할 필요가 있다. 이를 위해 각 노드의 자원 정보를 가지고 작업을 분배하게 되는 부하 분산 시스템을 설계 및 구현하였다.

본 보고서의 구성은 다음과 같다. 2장에서는 부하 분산 시스템의 기본적인 개요와 다른 부하 분산 시스템들이 갖는 특성에 대해 설명하고, 3장에서는 설치 및 실행환경에 대해 설명한다. 4장에서는 부하 분산 시스템의 전체적인 구조에 대해 설명하고, 5장에서는 좀 더 자세하게 어떻게 부하 분산 시스템이 구현되었는지에 대해 설명한다. 6장에서는 벤치마크를 사용한 성능 분석 결과를 살펴보고, 7장에서 결론을 맺는다.

제 2 장

관련 연구

2.1 부하분산 시스템의 개요

클러스터 시스템에서 필요한 부하 분산 기능을 제공하는 시스템은 그 방식에 따라 정적 부하 분산과 동적 부하 분산으로 나눌 수 있다. 정적 부하 분산은 작업을 분배할 때 과거의 작업 정보나 자원 정보를 사용하는 것이고, 동적 부하 분산은 가장 최근에 업데이트된 자원 정보와 작업 정보를 이용해서 작업을 분배하는 것이다. 정적 부하 분산 시스템은 최근의 작업 정보를 않기 때문에 작업의 잘못된 분배로 인한 성능의 저하를 가져올 수 있고, 따라서 자원이나 작업 정보를 업데이트 하는 데 쓰이는 오버헤드를 감안하고도 동적 부하 분산을 많이 사용한다.

이런 동적 부하 분산 시스템을 구현하는 데에는 상황에 따라 여러 가지 방식을 선택할 수 있는데 먼저 각 노드에서 어떤 식으로 자원이나 작업에 대한 정보를 얻어오고, 각 노드들이 어떤 식으로 그 정보를 주고받을지에 대한 load communication policy가 있고, 어떤 정보를 기준으로 작업을 분배할지, 예를 들어서 현재 수행하고자 하는 작업을 노드에 분배할 때, load average값이 작은 노드로 분배할 것인지, CPU를 사용하는 작업의 수가 가장 작은 노드로 분배할 것인지에 대한 load metric policy, 또 필요에 의해 수행중인 작업을 옮기고자 할 때, 어떤 노드에서(location selection policy) 수행중인 어떤 작업(candidate selection policy)을 옮길지에 대한 여러 가지 방식들이 존재한다. [2]

이렇게 클러스터 환경 하에서 서로 다른 방식을 사용해서 분배한 부하 분산 시스템들은 Condor, PBS, LoadLeveler, CODINE 등이 있으며 각각의 부하 분산 시스템들은 서로 다른 방식의 부하 분산을 제공하고 있다. [3, 4, 5]

2.2 다른 부하 분산 시스템들

1) NQS (Netwok Queueing System)

batch 작업을 스케쥴링하기 위한 툴로서 NASA에서 유래된 시스템으로서 작업의 특성에 따라 자원을 제한하는 'multiple batch queue'의 정의를 허용한 시스템으로 batch작업을 큐에 넣고 FIFO(First In First Out)순으로 수행시킨다.

1996년 개발된 Generic NQS는 발전된 버전으로 SMP프로세서 집합들에게 작업을 묶어주거나, 작업을 특정한 프로세스에 예약하는 것이 가능하다.

2) Condor

오랫동안 수행하는 batch작업에 대해 workstation의 사용하지 않는 CPU를 사용하기 위해서 University of wisconsin에서 개발한 프로그램이다.

사용자가 작성한 프로그램을 따로 고치지 않아도 실행이 가능하지만, Condor 라이브러리에 다시 링크시켜야 하는 단점이 있다. 또한 실행중인 노드가 fail되었을 때는 체크포인팅된 시점으로부터 다시 수행할 수 있지만, 이렇게 작업을 체크포인팅 하기 위한 디스크 공간의 오버헤드가 크다. [6]

3) DQS (Distributed Queueing System)

Supercomputer Computations Research Institute에서 각 사용자들이 공정하게 자원들을 사용하면서 최적으로 전체 자원들을 활용할 수 있게 하기위해 public-domain package로 개발하였다.

작업의 throughput을 최대화 하기 위해서 현재의 load average와 연결되어 있는 노드들의 작업 처리 능력을 고려하여 작업을 분배한다. 다기능의 GUI를 제공하며 현재는 java 버전이 개발중에 있다.

4) CODINE (Computing in a Distributed Network Environment)

독일의 Genias Software GmbH에서 개발했고, “DQS의 상업적 가용성 버전” 이라고 할 만큼 기능과 디자인면에서 매우 유사하지만, DQS에 비해 문서화가 잘 되어있다.

QUI를 사용하여 쉬운 접근이 가능하고, 관리자 기능을 없앰으로서 end-user가 더 친근하게 느낄 수 있도록 해준다. DQS와 같이 multiple queue를 제공하며 migration을 통해서 동적 부하 분산을 수행하지만, 체크포인팅을 제공하지는 않는다. 또한 interactive작업들을 사용하게 하기 위해 특별한 명령어를 제공하고 있다.

5) Load Leveler

IBM에서 개발한 Load Leveler는 Condor를 베이스 모델로 하고 있으며 디자인과 기능면에서 유사한 특성을 계속 유지시켜 왔다.

사용자 편의에 디자인의 중점을 두어서 문서가 읽기 쉽고 화면 구성 등이 잘 되어 있지만, 자원 사용에 관한 정보 묘사가 다소 제한되어 있다. 또한 Uniform한 유저 공간과 AFS[7]나 NFS[8]와 같은 파일 시스템을 요구한다. 다른 시스템들과 달리 작업을 수행하고자 할 때 작업이 요구하는 것들을 제시할 수 없으며, 이런 요구 사항이 사용자의 작업 스크립트에 내장되어야 한다.

Load Leveler에서는 부하 분산을 달성하기 위해서 현재의 load average와 현재 클러스터에 연결되어 있는 노드들의 idle time을 고려한다. 그렇지만 전체 메모리 사용량은 고려하지 않는데 이것은 사용자의 작업과 시스템 전체에 많은 성능 저하를 야기할 수 있다. [9]

6) PBS (Portable Batch System)

NASA Ames의 연구 프로젝트에서 고성능 컴퓨팅의 필요성을 역설하기 위해 처음에 개발한 시스템으로 batch작업과 interactive작업의 관리에 높은 신뢰성이 있는 제품으로 사용자의 편의를 위해 Posix 1003.2d 표준을 완전히 따르고 있다.

작업을 submit, hold, release, monitor, delete하는 명령어뿐만 아니라, signal, select, return, alter등의 능력도 제공하고 있고, NQS의 작업 스크립트를 PBS의 형태로 바꾸어주는 명령어까지 제공하고 있으며 GUI를 통해서 이런 명령어의 set에 쉽게 접근할 수 있도록 제공해준다.

그렇지만 PBS에서는 사용자 레벨에서 체크포인팅하는 라이브러리나 migration을 제공하지는 않는다.

제 3 장

설치 및 실행환경

3.1 설치

현재 개발된 부하 분산 시스템은 모든 시스템이 유저 레벨에서 개발되고 테스트 되었다. tar.gz로 묶여있는 파일을 풀게 되면 Load_balancing_system/ 디렉토리가 생성된다. 이 디렉토리는 전체 시스템의 정보를 관리하고 작업을 적절히 분배해주는 역할을 하는 central manager 소스가 있는 Cent_manager/ 디렉토리, 작업이 수행될 때 어느 노드에서 수행할지 metric을 가지고 결정하는 scheduler 소스가 있는 Sched/ 디렉토리, 작업이 수행되는 노드에서 주기적으로 정보를 수집하여 central manager에게 보내는 역할을 하고, central manager가 수행하도록 명령한 작업을 수행하게 하는 monitor의 소스가 있는Monitor/ 디렉토리로 구성되어 있다. 그림 3.1은 부하 분산 시스템의 전체 디렉토리 구조를 나타낸다.

전체 시스템이 커널이 아닌 유저 레벨에서 개발되었기 때문에, 따로 커널 패치나 컴파일 없이 Cent_manager 디렉토리에서 make한 결과 생성되는 ‘Cent’ 프로세스와 ‘Jsub’ 프로세스와 Monitor 디렉토리에서 make한 결과 생성되는 ‘Monitor’ 프로세스만을 이용하여 작업 분배 시스템을 수행시킬 수 있다.

3. 2 실행환경 및 실행방법

부하 분산 시스템을 실행하고, MPI 작업을 실행시키기 위해서는 MPICH같은 MPI작업 수행 프로그램이 설치되어 있어야 하고, 클러스터 내에는 전체 노드들이 파일을 공유할 수 있는 AFS나 NFS와 같은 파일 시스템이 설치되어 있어야 한다.

```
Load_balancing_system/
      |-- Cent_manager/
               |-- makefile
               |-- header_cent.h
               |-- var_cent.h
               |-- cent_mgr.c
               |-- CM_management.c
               |-- execute_job.c
               |-- get_info.c
               |-- information.c
               |-- job_submit.c
               |-- queue_adt.c
               |-- receive_info.c
               |-- software_interface.c
               |-- submit.c
       |-- Sched/
               |-- kill.c
               |-- metric_sched.c
               |-- migration.c
               |-- parallel.c
               |-- scheduler.c
       |-- Monitor/
               |-- makefile
               |-- var.h
               |-- monitor.c
               |-- collect_info.c
               |-- get_execute_command.c
               |-- migration_command.c
               |-- MONITOR_management.c
               |-- send_info.c
```

[그림 3.1] 부하 분산 시스템의 전체 디렉토리 구조

Cent_manager 디렉토리에 있는 serv_host 라는 파일에 서버로 사용하게 될 노드의 도메인 네임을 적어준다. (부하 분산 시스템은 서버 1대와, 클라이언트들로 이루어지며, 클러스터에 있는 노드 중 한대만 서버로 사용되고, 나머지가 클라이언트로 사용된다)

실행할 때는 서버 노드로 선택한 노드에서 'Cent' 프로세스를 띄우고, 클라이언트로 선택한 노드들에서 'Monitor' 프로세스를 띄운다. 이렇게 되면 부하 분산 시스템이 작동하게 된다. 이렇게 시스템을 실행시킨 후에 클라이언트 노드들에서 정보를 수집하여 서버 노드로 보내게 되고, 사용자는 이 정보를 직접 확인할 수 있다.

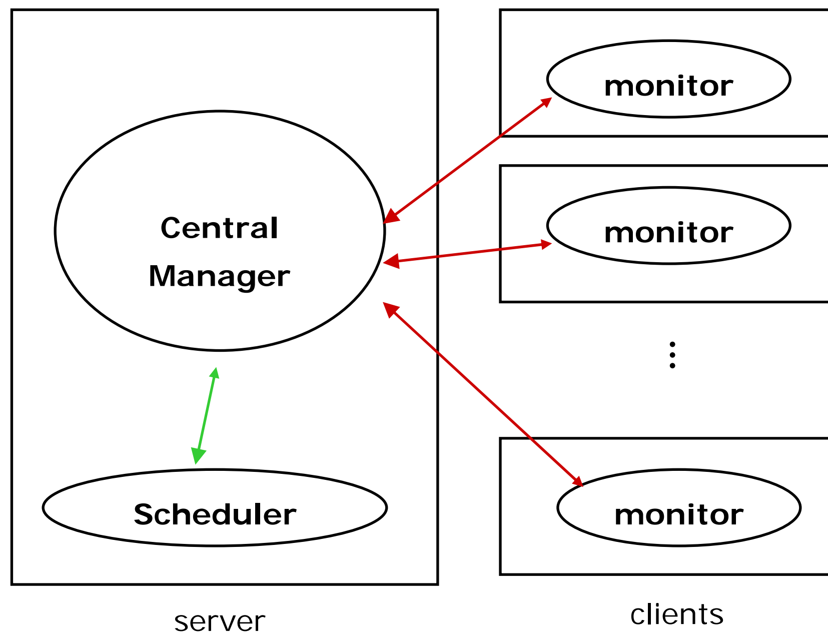
작업을 실행할 때는 'Jsub' 프로세스를 사용하며 실행하고자 하는 작업에 대한 정보를 담은 파일을 먼저 작성해야 한다. 이 파일에는 수행하고자 하는 작업의 이름이나 수행하려는 작업이 있는 디렉토리, input, output 파일 등의 정보가 포함되어야 하며 이렇게 작성된 파일을 이용해서 "Jsub [파일이름]" 의 형태로 실행시키면 작업을 수행할 수 있다. 이때 옵션으로 metric값을 주어줄 수 있는데 제시하지 않으면 기본으로 load average를 metric으로 하여 작업을 분배하게 된다. 작업 파일과 metric에 대한 자세한 내용은 5장에서 언급하겠다.

제 4 장

부하 분산 시스템의 설계

4.1 전체 시스템의 구성

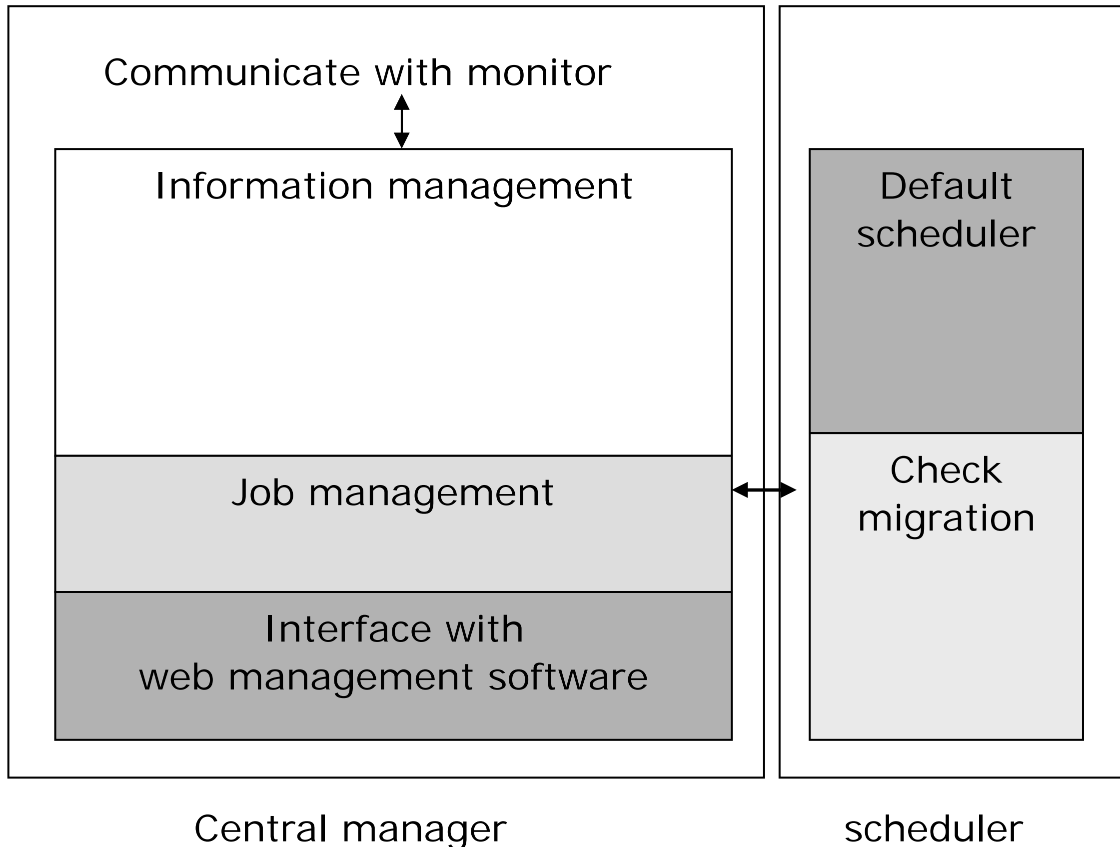
부하 분산 시스템은 크게 서버와 클라이언트로 구성된다. 부하 분산 시스템을 사용하고자 하는 클러스터 중에서 한 노드가 서버가 되고 나머지가 클라이언트가 된다. 서버에는 central manager와 scheduler 데몬이 실행되고, 클라이언트에는 각각 monitor 데몬이 실행되며, central manager는 scheduler와 통신함으로서 작업을 어떻게 수행할 것인지에 대한 정책을 결정하고, monitor와의 통신을 통해 각 자원 정보를 관리하거나, 작업을 수행하고 작업 정보를 관리한다. 전체 구조는 그림 4.1과 같다. [10]



[그림 4.1] 전체 시스템의 구조

4.2 서버 쪽 부하 분산 시스템의 구조

서버의 시스템의 구조는 그림 4.2와 같다.

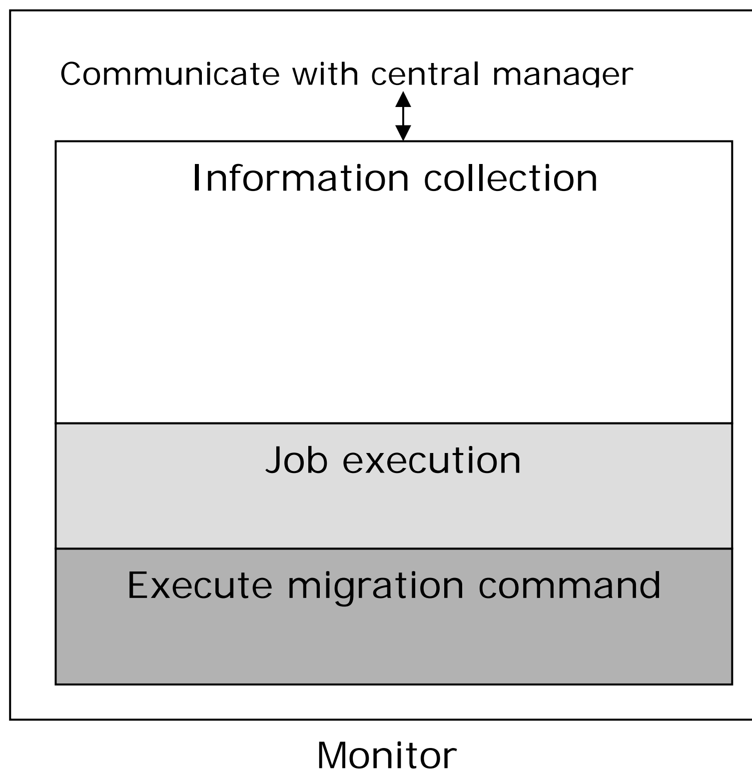


[그림 4.2] 서버 쪽의 부하 분산 시스템의 구조

서버는 크게 central manager와 scheduler로 이루어져 있으며, central manager는 클라이언트들로부터 각 노드의 자원 정보를 수집하고 이것을 관리하는 information management와 작업을 사용자가 수행하고자 할 때 적합한 노드에서 수행시켜 주는 job management 부분, 그리고 web에서 작업을 관리할 수 있게 만든 KCMS (ganglia를 기본으로 하여 웹에서 클러스터에 대한 자원 모니터링뿐만 아니라 작업을 수행할 수 있도록 제공하는 시스템이다.)와의 인터페이스를 지원하는 interface with web management software부분이 있다.

Scheduler는 크게 기본적인 작업 스케쥴을 담당하는 scheduler가 있는데 이것은 사용자가 작업을 수행하려고 할 때, 현재의 자원 정보를 파악하고 사용자 작업의 특성을 파악한 후에 작업을 수행할 노드를 선택하여 central manager에게 알려주는 역할을 한다. 그리고 각 노드의 부하 정보를 주기적으로 체크하여 한 노드에 부하가 집중되었을 경우 그 노드에서 다른 노드로 작업을 옮길 수 있게 해줌으로써 부하를 균등하게 분배하는 process migration부분이 있다.

4.3 클라이언트 쪽 부하 분산 시스템의 구조



[그림 4.3] 클라이언트 쪽 부하 분산 시스템의 구조

클라이언트 쪽에는 monitor라는 프로세스가 항상 데몬으로 실행된다. Monitor 의 구조는 그림 4.3 과 같다.

클라이언트에서 수행되는 monitor는 현재 노드의 정보를 수집하여 이것을 central manager에게 보내주는 역할을 하는 information collection 부분이 있고, central manager가 작업을 수행하기를 요청할 때 그 요청을 받아서 처리하는 job execution 부분, 그리고 migration 을 수행할 때 그것을 수행하는 execute migration command 부분으로 이루어져 있다.

제 5 장

부하 분산 시스템의 구현

먼저 1절과 2절에서는 앞에서 언급한 서버 노드와 클라이언트 노드에서의 부하 분산 시스템에서 각 부분들이 어떤 역할을 하는지에 대해 좀 더 자세히 언급한다.

부하 분산 시스템이 동작하게 되면 크게 전체 클러스터 시스템의 부하 정보를 관리하는 부분과, 이 부하 정보를 기반으로 하여 사용자가 요구하는 작업을 수행하는 부분으로 나눌 수 있는데. 먼저 시스템의 부하 정보를 관리하는 부분에 대해 3절에서 설명하고, 사용자가 작업을 수행하고자 할 때 어떤 식으로 이루어지는지에 대한 구현 부분에 대해 4절에서 설명하도록 하겠다. 또한, 작업을 수행할 때 서버와 클라이언트에서 각각 작업에 대한 정보를 관리하는 부분에 관하여 5절에서 설명하도록 하겠다.

5.1 서버 쪽 부하 분산 시스템의 각 부분의 역할

1) information management

작업 분배를 위해서 central manager는 현재 부하 분산 시스템에 연결되어 있는 클라이언트들의 정보를 알고 있어야 한다. 이를 위해 부하 분산 시스템의 클라이언트에 수행되고 있는 monitor가 현재의 자원 정보를 얻어서 5초마다 한번씩 주기적으로 central manager에게 소켓(TCP)을 통해 정보를 전송하도록 되어있다.

Central manager는 이 정보를 받아서 현재 부하 분산 시스템에 연결되어 있는 클라이언트들의 접속 여부를 체크하고 그 클라이언트 노드 각각의 부하 정보들을 자료 구조를 작성하여 저장하게 된다. 이때, 클라이언트에서 보내주는 노드의 자원 정보는 CPU, memory, 그리고 load average값이 된다. [11, 12]

2) Job management

job management 부분에서는 사용자가 작업을 수행하고자 할 때, 사용자가 작성한 작업 파일을 받아서 이 안에 있는 정보들을 자료 구조로 저장한다. 그리고 나서 이 작업 정보를 scheduler에게 보내면 scheduler쪽에서는 가지고 있는 노드들의 자원 정보와 작업 정보를 사용하여 작업을 수행하기에 적절한 노드를 결정하여 다시 central manager에게 알려준다. 이렇게 해서 받은 정보를 사용하여 monitor와의 통신을 통해 작업을 수행할 수 있도록 한다.

3) Interface with web software

현재 작성한 부하 분산 시스템은 웹에서 작업을 수행할 수 있고, 다른 노드로 옮길 수 있고, 현재의 부하 정보를 모니터링 할 수 있게 해주는 시스템을 제공하고 있다. 이 시스템과의 인터페이스를 위한 부분으로서 웹에서 소켓을 사용하여 오는 요구들을 적절히 상황에 맞게 판단하여 정보를 웹으로 보내주거나, 혹은 작업을 수행하는 일을 담당하는 부분이다.

4) default scheduler

scheduler에서는 central manager로부터 받은 작업 정보와 현재 수집되어 있는 자원 정보를 이용하여 현재 사용자가 수행하고자 하는 작업이 어느 노드에서 수행될 때 가장 좋은 성능 향상을 가져올지를 결정하게 된다.

일단 central manager로부터 받은 작업 정보를 사용하여 현재 사용자가 수행하고자 하는 작업이 serial 작업인지, 아니면 여러 프로세스를 필요로 하는 parallel 작업인지를 판단한다.

그리고 나서 현재 자원 정보를 이용하여 metric을 계산하게 되는데 현재 시스템에서는 세 가지 metric을 제공한다. 이 metric을 이용하여 현재 작업을 수행하기에 적합하

다고 판단되는 노드를 결정한다. 이렇게 결정한 것들을 자료 구조로 만들어서 저장하고 이를 다시 central manager에게 보내준다.

5) Process migration

부하가 한 노드에 몰리는 현상을 방지하기 위한 부분으로 주기적으로 현재의 자원 정보를 검사한다. 한 노드에 부하가 몰렸다는 것이 감지되면 그 노드에서 현재 수행중인 작업 중에서 적절한 작업을 골라서 다른 노드에서 수행될 수 있도록 하기 위해 자료 구조로 노드 정보와 작업 정보를 central manager에게 넘겨준다. 그러면 central manager는 이 정보를 받아서 수행중인 작업을 다른 노드로 옮기게 된다.

5.2 클라이언트 쪽 부하 분산 시스템의 각 부분의 역할

1) Information collection

이 부분에서는 현재의 노드 정보를 수집한다. /proc 디렉토리 밑에 있는 loadavg 파일, meminfo파일을 읽어서 현재의 loadavg값과 memory 정보를 수집하고, 각 pid에 해당하는 /proc/[pid]/stat 파일을 읽어서 현재 노드에서 수행중인 프로세스 정보를 함께 수집한다. 이렇게 얻은 노드 정보를 5초마다 한번씩 central manager에게 보낸다.

2) Job execution

이 부분에서는 사용자가 수행하고자 하는 작업을 central manager가 요청하면, 작업을 수행하는 역할을 한다. 작업을 수행하기 시작하면서 그 작업에 대한 프로세스 아이디를 central manager에게 알려주며, 작업이 끝나면 작업을 수행하는데 사용한 자원 정보를 central manager에게 알려준다.

3) Execute migration command

필요에 의해 현재 수행중인 작업을 한 노드에서 다른 노드로 옮길 때, 두 노드간의 통신을 이용해서 작업을 옮기는 일이 수행된다. 이 부분은 작업을 원활하게 한 노드에서 멈추고, 다른 노드에서 재 시작할 수 있도록 구현된 부분으로서 작업을 새로 시작하고자 하는 노드에서 수행된다.

5.3 부하 정보의 관리

부하 분산 시스템에서 작업이 적합한 노드에서 수행되도록 하기 위해서는 부하 정보를 항상 central manager가 관리하고 있도록 해야 한다. 이를 위해 central manager쪽의 information management 부분[그림 4.2] 과 monitor쪽의 information collection 부분[그림 4.3] 이 서로 소켓(TCP)을 이용하여 통신하게 된다.

1) 클라이언트에서의 부하 정보 관리

클라이언트에서 수행되는 monitor 프로세스는 부하 정보와 자원 정보를 관리하기 위해 node_state 라는 구조체를 가지고 있으며, 그림 5.1과 같이 선언되어 있다.
클라이언트에서 수행되는 monitor 프로세스에서는 State_to_server() 라는 함수를 호출하는데, 이 함수는 5초에 한번씩 노드 정보를 얻어서 node_state 구조체에 저장하고 이를 서버에게 보내준다. 좀 더 자세히 살펴보면 다음과 같다.

5초에 한번씩 클라이언트에서 수행되는 monitor 프로세스는 /proc 디렉토리 밑에서 파일들의 내용을 읽어 node_state 라는 구조체로 그 정보를 저장한다. /proc 디렉토리 밑의 loadavg 파일은 현재의 load average값 (최근 1분 동안 CPU를 사용하여 수행되고 있는 평균 프로세스의 개수를 나타내는 값이다.) 을 나타내며, meminfo 파일은 메모리의 정보를 나타낸다. 또한 /proc/[pid]/stat 파일에서는 현재 수행중인 프로세스 정보를 얻어올 수 있으며, 이 정보들은 process_state 구조체에 저장되게 된다.

```
struct node_state{
	int command;
	// 해당되는 노드의 이름 ex)can52.kaist.ac.kr
	char node_name[30];
	// 해당되는 노드의 load average값
	   float loadavg;
	// 해당되는 노드의 메모리 정보로서 전체 메모리양, 유저 메모리양,
	// 현재 사용 가능한 메모리양, 스왑 메모리 양 등이 포함된다.
	   long int total_memory;
	   long int user_memory;
	   long int free_memory;
	   long int total_swap;
	   long int user_swap;
	   long int free_swap;
	// 해당되는 노드에서 현재 수행중인 프로세스들의 정보를 담고 있다.
	// struct process_state 는 아래 정의되어 있다.
	   struct process_state Procinfo[50];
};
// 현재 수행중인 프로세스들의 정보를 담기 위한 구조체로서 프로세스 아이디, 이름, 현재
상태, 세션, 플래그, 수행시간 등의 정보를 가지고 있다.
struct process_state{
	int pid;
	char name[30];
	char state[1];
	long int session;
	long int flags;
	long int utime;
	long int stime;
	long int cutime;
	long int cstime;
	long int starttime;
	long int vsize;
	long int rss;
};
```

[그림 5.1] node_state 구조체 선언

이렇게 해서 monitor는 현재 노드의 정보를 얻고, 이것을 서버에게 소켓(TCP)을 통하여 보내게 된다.

2) 서버에서의 부하 정보 관리

central manager는 클라이언트에 있는 monitor로부터 연결 요청이 오기를 기다렸다가, 클라이언트에서 접속하면 클라이언트 하나당 하나씩 쓰레드를 생성한다. 생성된 쓰레드는 각각 해당하는 클라이언트로부터 주기적으로 노드 정보를 받아서 서버에 있는 node_state 구조체에 저장하게 된다. Node_state 구조체는 정보를 받아올 때마다 최근의 정보로 업데이트되며, 클라이언트로부터의 접속이 끊어지면 해당되는 노드의 정보를 삭제한다.

이런 식으로 서버는 현재 접속하고 있는 클라이언트들의 자원 정보와 부하 정보를 관리하고, 이것을 기반으로 하여 사용자 작업을 수행할 때 적합한 노드를 선택한다.

5.4 작업의 수행

1) 작업 정보를 구조체에 저장하고, scheduler 호출

사용자가 작업을 수행하고자 할 때 사용자는 작업 실행을 위한 파일을 하나 생성하여야 한다. 파일에 저장된 정보는 job_desc라는 구조체에 저장되며, job_desc는 그림 5.2과 같이 선언되어 있다.

선언된 구조체 중에서 이탤릭체로 표시된 정보들이 작업 실행을 위해 생성된 파일에 포함되어야 한다.

사용자가 이렇게 작성한 파일을 가지고 Jsub 프로세스를 수행시켜 작업의 수행을 요청하면 central manager는 Jsub 프로세스와 통신을 하여, 작업 정보 파일의 경로를 받아오고, 파일에 저장된 정보를 job_desc라는 구조체에 저장한 후 scheduler() 함수를 호출한다.

```
struct job_desc{
    int job_id;
    // 작업의 아이디로, 부하 분산 시스템에서 작업을 효율적으로 관리하기 위한 값으로
    // 사용자가 작업을 수행하려고 할 때 주어지는 값이다.
    char job_name[40];  // 수행하고자 하는 작업의 이름
    int argnum;
    char arguments[100];  // 인자가 있다면 인자들의 배열
    int type;    // 작업의 type 0:serial,1:batch,2:parallel
    char inputfile[40];
    char outputfile[40];
    char errori[40];
    char executable[255];  // executable의 위치
    int user_priority;  // 사용자가 임의로 정할 수 있는 작업의 우선순위
    char initialdir[100];  // 작업을 수행하고자 하는 기본 디렉토리
    int mem_size;  // 작업 수행시 요구하는 메모리 크기

    int tasks_per_node;
    int total_tasks;  // 작업 수행시 원하는 프로세스의 수

    double rss_limit;
    int restart_from_ckpt; //0:no,1:yes

    char user_name[30];
    char pre_defined_node[30];
};
```

[그림 5.2] job_desc 구조체 선언

2) load metric의 계산

Scheduler 함수는 먼저 현재의 부하 정보를 가지고 metric을 계산한다. 이때 metric은 작업을 어느 노드에서 수행할지 결정하는 데 가장 중요한 역할을 하며, 부하 분산 시스템에서는 전체 성능에 영향을 줄 수 있는 값이다. 현재 부하 분산 시스템에서는 metric을 세가지 종류로 구현하였는데, loadavg값, CPU length, 그리고 'Mem+CPU'로 구현하였다.

Loadavg는 load average로서 클라이언트에서 5초마다 주기적으로 보내오는 정보에 포함되어 있는 값으로 최근 1분 동안 CPU를 사용하는 평균 작업의 개수다.

CPU length 는 현재 수행중인 프로세스의 개수로서 이것도 클라이언트가 보내주는 부하 정보에서 프로세스의 개수를 계산하여 얻는다.

'Mem+CPU' 는 메모리와 CPU를 둘 다 고려하여 계산한 metric인데, 현재 사용 가능한 자원들에 대해 최대한 고려하는 방식으로 접근한 것이라고 할 수 있다. 이 값은 현재의 load average 값에 memory_rate값을 곱한 것이다. Memory_rate값은 전체 유저에게 할당된 메모리 값을 현재 사용 가능한 메모리 값으로 나눈 것으로 메모리가 현재 어느 정도의 비율로 사용되고 있는지를 판단할 수 있는 값이다. [13]

이렇게 metric을 계산하고 나면, scheduler는 사용자가 수행하고자 하는 작업의 type을 검사하고, 각 type에 따라 적절한 수행 방법을 central manager에게 알려준다.

3) 작업 수행에 관한 결정

scheduler는 작업의 type을 검사한다.

만약 사용자가 수행하고자 하는 작업이 serial 이거나 batch 작업인 경우, 현재 계산된 metric을 사용하여, 가장 부하가 적게 걸린 노드로 작업을 수행할 수 있도록 central manager에게 알려준다.

사용자가 수행하고자 하는 작업이 parallel 작업인 경우에는 사용자가 요구하는 프로세스의 개수를 보고, 현재 연결되어 있는 클라이언트의 수가 사용자가 요구하는 프로세스 수보다 작은 경우에는 최대한 많은 클라이언트를 사용하기 위해 현재 연결되어 있는 클라이언트 모두에서 parallel 작업을 수행하도록 한다.

만약 사용자가 요구하는 프로세스의 수가 연결되어 있는 클라이언트의 수보다 작으면 프로세스 수만큼 metric이 작은 클라이언트부터 하나씩 골라서 central manager에게 알려준다.

Scheduler는 이외에도 각 노드의 부하를 주기적으로 검사하여 부하의 집중 현상이 생겼을 때 (부하가 가장 작은 노드와 가장 큰 노드의 차이가 5 이상일 때), 작업을 현재 노드에서 부하가 작은 노드로 옮기기로 결정하고 이것을 central manager에게 알려주는 일도 한다.

4) 작업 수행 시작

이렇게 해서 결정된 사항을 scheduler가 central manager에게 알려주면 central manager는 연결된 monitor에게 작업을 수행하라는 명령을 보낸다. 이때, serial 이나 batch 작업일 경우 한 노드에게 작업을 수행하라는 명령을 보내게 되고, parallel 작업일 경우에는 central manager에서 MPI 수행 명령어를 사용하여 작업을 수행하게 된다. 이렇게 central manager가 보낸 정보에 따라 monitor 쪽에서는 작업을 수행하게 되고, 작업이 시작되었을 때와 끝날 때 central manager에게 알려준다.

5) 작업의 수행

monitor 프로세스에서는 쓰레드를 생성하여, central manager와 소켓(TCP)으로 연결하여, 작업 요청이 들어오는지를 계속 확인한다. 만약 central manager로부터 작업의 요청이 들어오면 이 작업을 fork() 명령어를 이용하여 프로세스를 생성한 뒤 수행한다. 작업의 수행이 끝나면 작업이 수행되는 동안 사용된 자원 정보를 central manager에게 알려주고, central manager는 이때 작업이 끝났다는 사실을 알게 된다.

만약 수행하고자 하는 것이 작업을 현재 수행중인 노드에서 다른 노드로 옮기는 일인 migration 일 경우에는 현재 수행중인 노드와 다른 노드, 그리고 서버 노드, 이렇게 세 노드가 소켓(TCP)을 통해 통신함으로써 작업을 수행한다. 현재 수행중인 노드에서 수행 중이던 작업을 중단시키고, 새로 작업이 수행될 노드에 알려주면, 노드에서는 작업을 재 시작하는 방식으로 migration이 이루어진다.

5.5 작업정보의 관리

작업을 수행할 때, 좀 더 효율적으로 작업을 관리하기 위해서 서버는 사용자가 수행한 모든 작업의 정보를, 그리고 클라이언트는 각 노드에서 수행한 작업의 정보를 유지한다. 이를 통해 작업을 관리하고 사용자는 작업이 끝났을 때를 알 수 있고, 작업을 수행하는 데 얼마나 많은 시간이 걸렸는지, 혹은 얼마나 많은 자원들이 사용되었는지 알 수 있다.

서버가 작업 수행을 클라이언트에게 요청할 때, 혹은 작업을 수행할 때는 네 가지 경우가 있다. 사용자가 serial 작업을 수행하고자 할 때, parallel 작업을 수행하고자 할 때, 부하의 집중 현상으로 인한 migration을 수행하고자 할 때, 사용자가 현재 수행중인 작업에 대한 중단 요청을 했을 때(kill)로 크게 나눌 수 있다.

네 가지 경우에 대해서 서버와 클라이언트가 동작하는 방식에는 약간의 차이가 있지만 기본적으로 서버는 작업을 수행하기 이전에 현재 수행하고자 하는 작업의 정보를 CM management라는 구조체에 추가한다. 클라이언트에게 작업 수행 요청을 하고 나서 클라이언트에서 작업을 시작하면 그 작업을 수행하는 프로세스의 아이디를 클라이언트에게 받아서 CM management에 업데이트해주고, 작업이 끝나면 클라이언트로부터 작업 수행시 사용한 자원의 정보와 작업이 끝났다는 사실을 CM management에 기록해준다.

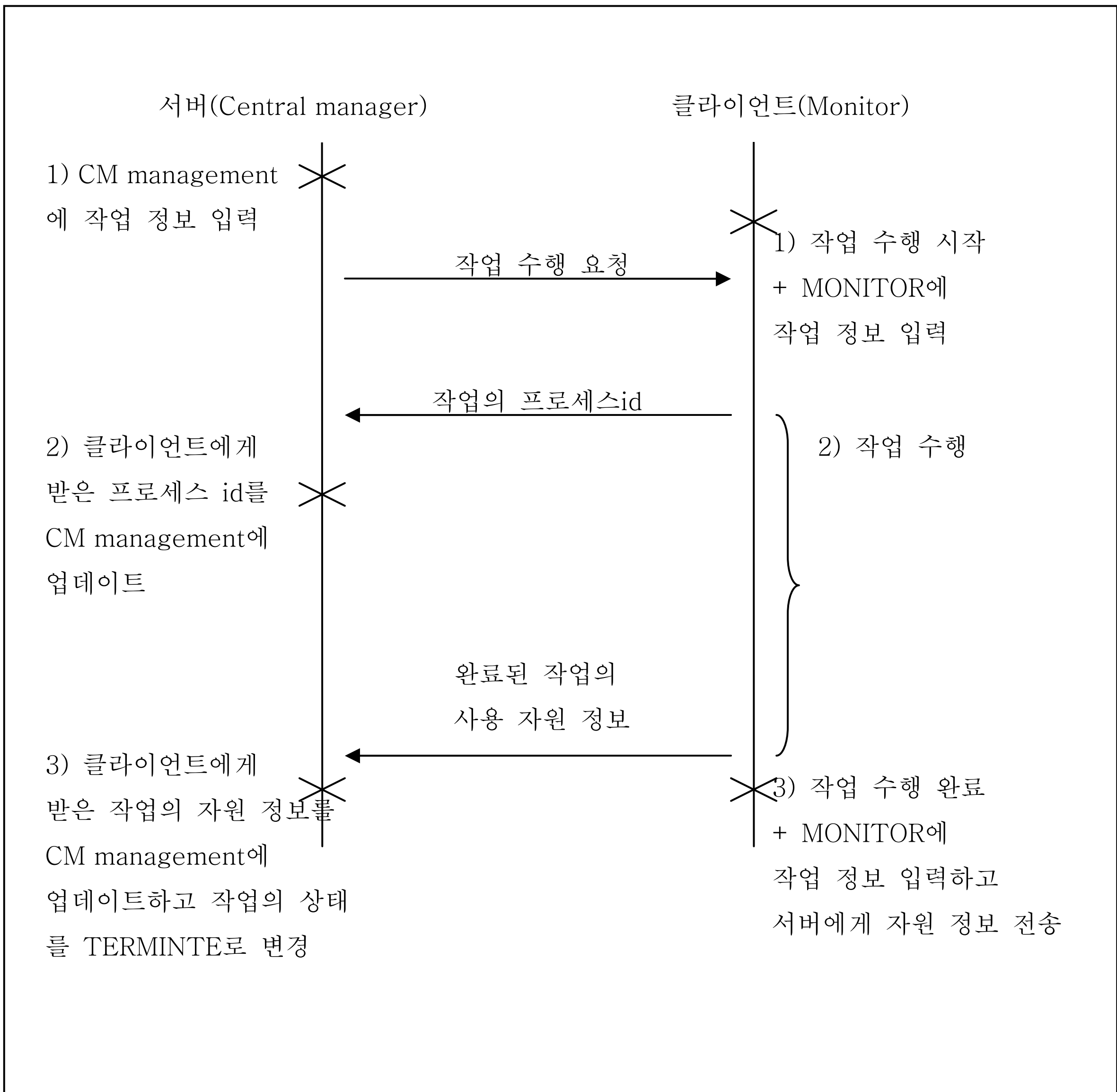
그림 5.3에는 작업 정보를 관리하기 위한 서버 쪽 구조체의 선언이, 그림 5.4에는 클라이언트 쪽 구조체의 선언이, 그림 5.5, 5.6, 5.7, 5.8 에서는 서버가 작업을 수행하는 네 가지 경우에 대해서 작업 정보가 어떤 식으로 관리되는지에 대해 표현되어 있다.

```
struct CM_management{
      int job_id;  // 작업의 아이디로 클러스터 시스템에서 unique한 값이다.
      int job_type; // 작업의 type: PARALLEL|SERIAL|BATCH|MIGRATED
      char node[500]; // 현재 작업이 수행중인 클라이언트의 이름
      char job_name[100]; // 사용자가 지정한 작업의 이름

      long starttime;
      long endtime;  // 작업의 시작시간과 끝나는 시간을 기록한다.

      int status; // 작업의 현재 상태: RUNNING(수행중)|TERMINATE(완료)
      int migration; //migration 된 작업인지의 여부

      int pid; // 작업이 수행되는 노드에서의 프로세스의 아이디
      char job_owner[30]; // 작업 수행을 요청한 사용자

      struct rusage resource_usage;
      // 작업이 끝나고 난 후에 기록되며 작업이 수행 중 사용한 자원의 정보
};
```

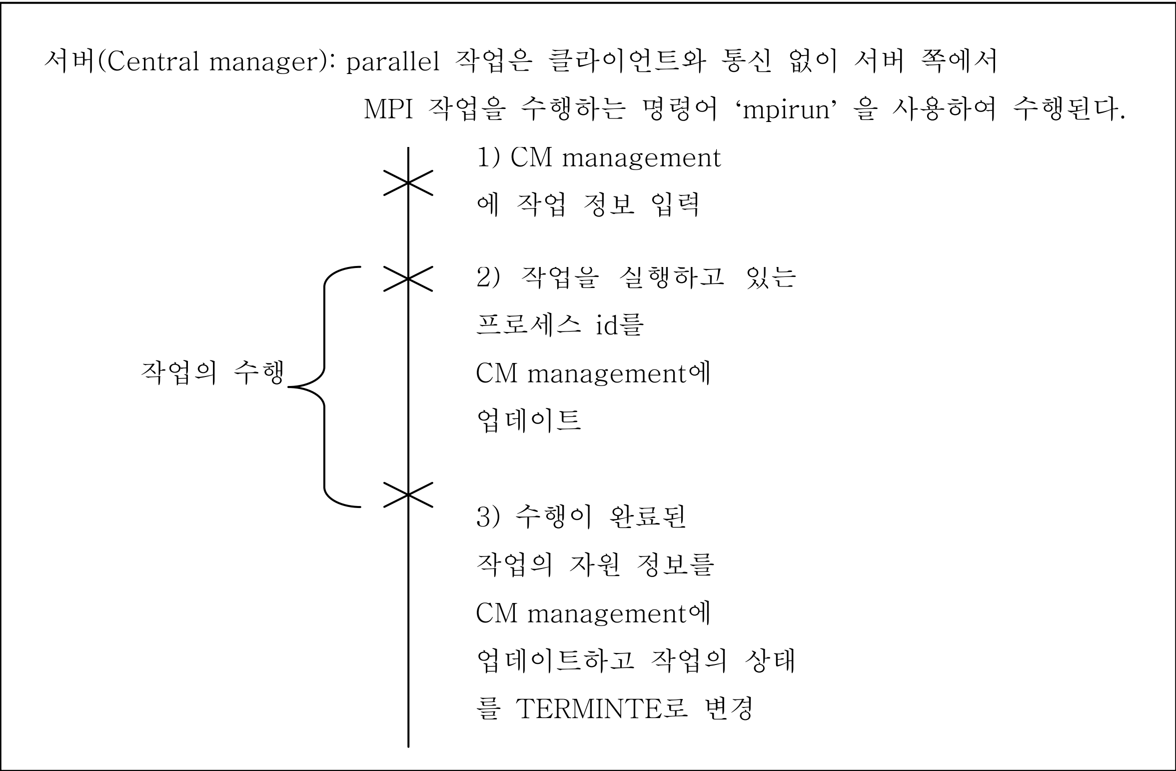
[그림 5.3] 작업 정보 관리를 위한 서버 쪽 구조체

```
struct MONITOR_management{
      int M_job_id;  // 작업의 아이디로 클러스터 시스템에서 unique한 값이다
      int M_pid; // 작업 수행 시 생성된 프로세스의 아이디
      int M_status;
       // 현재 작업의 상태: 수행 중인지(RUNNING), 종료된 작업인지(TERMINATE),
          migration 된 작업인지(MIGRATED) 표현한다.
      char M_job_name[100]; // 사용자가 지정한 작업의 이름
      char M_job_owner[30]; // 작업 수행을 요청한 사용자
```

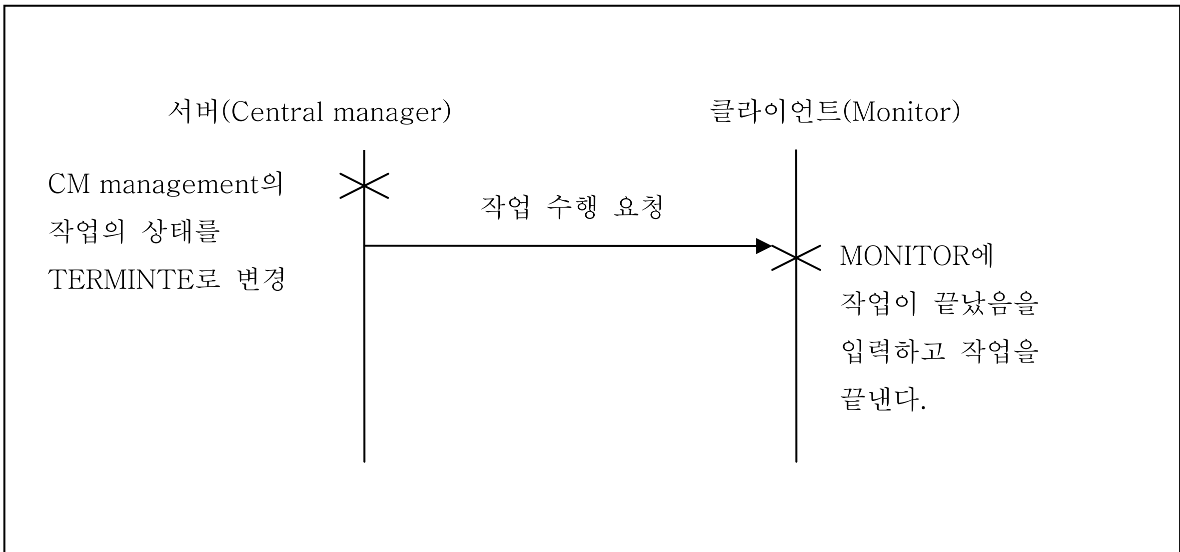
[그림 5.4] 작업 정보 관리를 위한 클라이언트 쪽 구조체



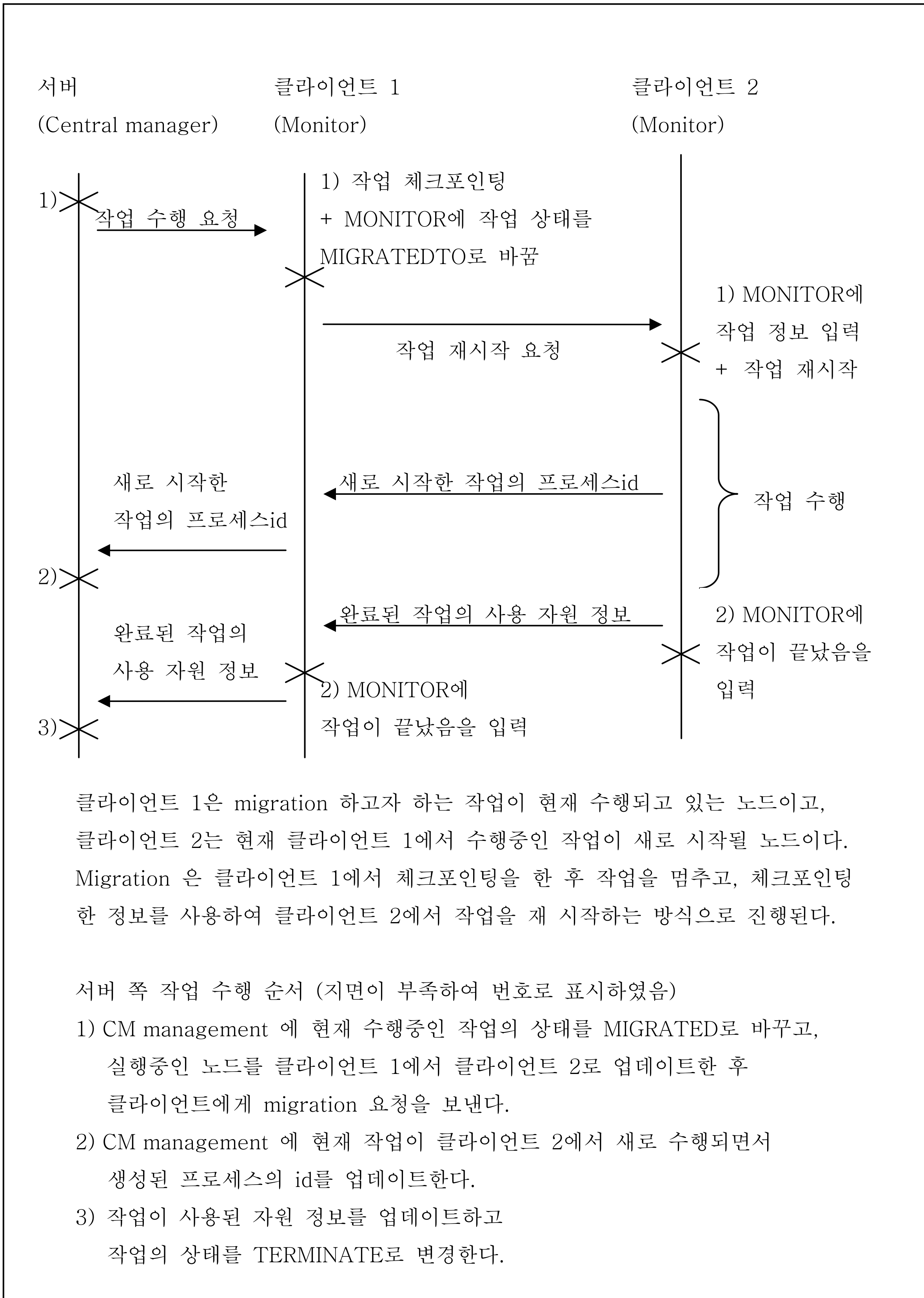
[그림 5.5] 작업의 수행과 작업 정보의 관리 – serial 작업의 수행



[그림 5.6] 작업의 수행과 작업 정보의 관리 – parallel 작업의 수행



[그림 5.7] 작업의 수행과 작업 정보의 관리 – kill 작업의 수행



클라이언트 1은 migration 하고자 하는 작업이 현재 수행되고 있는 노드이고, 클라이언트 2는 현재 클라이언트 1에서 수행중인 작업이 새로 시작될 노드이다. Migration 은 클라이언트 1에서 체크포인팅을 한 후 작업을 멈추고, 체크포인팅 한 정보를 사용하여 클라이언트 2에서 작업을 재 시작하는 방식으로 진행된다.

서버 쪽 작업 수행 순서 (지면이 부족하여 번호로 표시하였음)

1) CM management 에 현재 수행중인 작업의 상태를 MIGRATED로 바꾸고, 실행중인 노드를 클라이언트 1에서 클라이언트 2로 업데이트한 후 클라이언트에게 migration 요청을 보낸다.
2) CM management 에 현재 작업이 클라이언트 2에서 새로 수행되면서 생성된 프로세스의 id를 업데이트한다.
3) 작업이 사용된 자원 정보를 업데이트하고 작업의 상태를 TERMINATE로 변경한다.

[그림 5.8] 작업의 수행과 작업 정보의 관리 – migration 작업의 수행

제 6 장

부하 분산 시스템의 성능 측정

6.1 성능 측정 환경

성능 측정을 위해서 8개의 노드로 구성된 클러스터에서 부하 분산 시스템을 수행하였다. 노드 각각의 사양은 다음과 같다. 서버 노드를 can52.kaist.ac.kr로 설정하고 CPU의 클럭 수가 다른 두 클라이언트 집단으로 “can1/can2/can3/can4”와 “can49/can50/can51”을 클라이언트 노드로 사용하였다.

Server node: can52.kaist.ac.kr	
CPU	Pentium Ⅲ (700MHz)
Memory	512MB
OS	Linux kernel 2.4.18

Client nodes (set 1): can1/can2/can3/can4.kaist.ac.kr	
CPU	Pentium Ⅲ (500MHz)
Memory	512MB
OS	Linux kernel 2.4.2
Client nodes (set 2): can49/can50/can51.kaist.ac.kr	
CPU	Pentium Ⅲ (700MHz)
Memory	512MB
OS	Linux kernel 2.4.2

성능 측정에는 SCALARPACK 벤치마크를 사용하였으며, 이 중에서 사용된 LINPACK은 Argonne Nat'l Lab에서 개발된 것으로 부동점 연산 능력을 행렬 계산을 통해 측정하는 벤치마크 프로그램으로 현재 세계 슈퍼 컴퓨팅 랭킹을 정하는 기본이 되고 있다. 6.2장에서 LINPACK 벤치마크에 대해 좀 더 자세히 설명하도록 하겠다.

6.2 LINPACK 벤치마크

LINPACK은 주로 선형 방정식과 선형 최소 제곱 법 문제를 푸는 포트란 서브루틴들로 구성된 수치 해석 패키지의 하나로서 컴퓨터의 연산속도를 측정하는 벤치마크 프로그램으로 이용되기도 한다. [14, 15]

SCALAPACK 은 선형대수 의 해를 구하는 패키지로 많은 부분이 부동소수점 연산으로 구성되어 있다. LINPACK 벤치마크 에서 중점적으로 사용되는 루틴들은 Gauss 소거법을 이용한 N 개의 선형방정식 의 해를 구하는 것으로 BLAS (Basic Linear Algebra Subprograms) 에 포함되어 있다. BLAS 는 LINPACK 벤치마크 에서 가장 기본이 되는 라이브러리로써 기본적인 선형대수 연산함수 들을 구현해놓은 집합이다. 이것은 Fortran으로 짜여 있으며 BLAS 라이브러리 내의 각 함수들은 연산자와 연산결과가 Vector냐 Matrix냐에 따라 계산 레벨이 나뉘어 진다.

6.3 성능 측정 결과

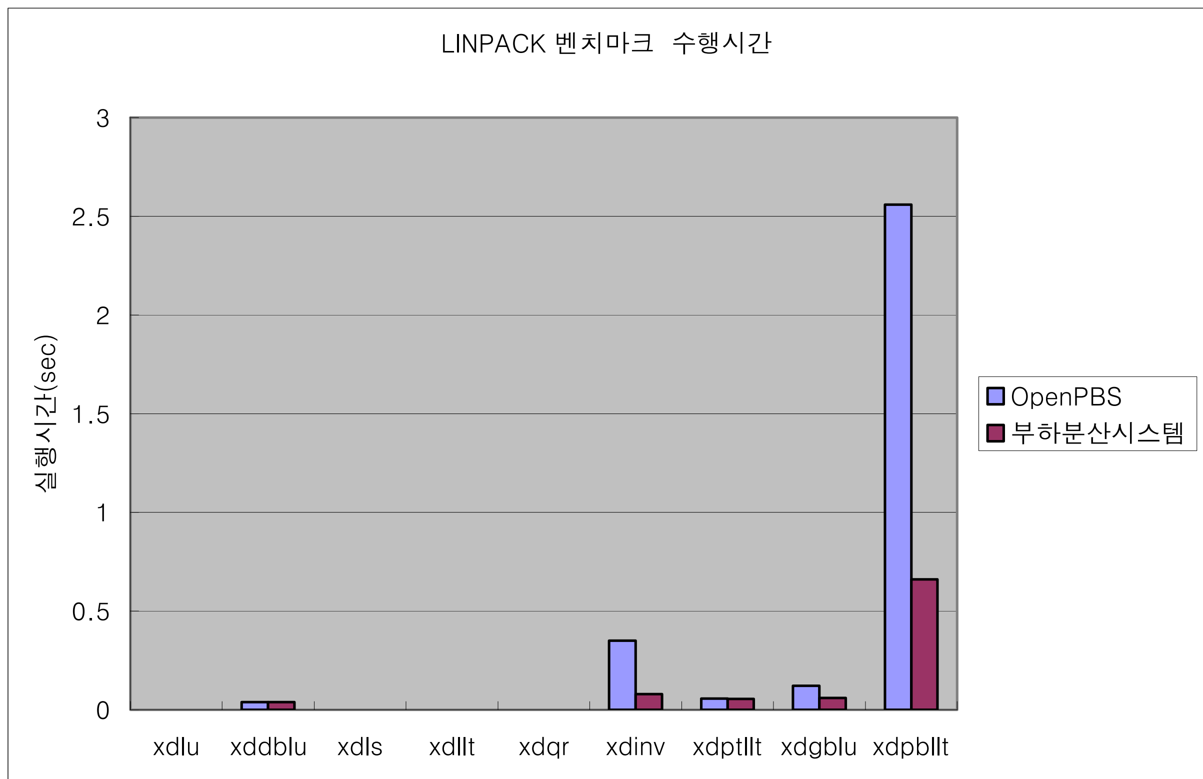
성능 측정을 위하여 앞서 2장에서 설명한 PBS라는 프로그램을 설치하여 수행하였다. 8노드에 OpenPBS(PBS의 공개된 버전이다) 을 설치하여 벤치마크 프로그램을 수행시키고, 또 구현한 부하 분산 시스템에서 벤치마크 프로그램을 수행시켜 수행시간을 계산하였다. 부하 분산의 성능측정을 위해서 클라이언트 노드 7개중 두 곳에 프로그램을 수행시켜 loadavg값을 임의로 4로 조정해 둔 상태에서 성능 측정을 하였다.

아래 표6.1은 벤치마크의 수행 결과 OpenPBS에서와 부하 분산 시스템에서의 수행 시간

을 표로 나타낸 것이고, 그림 6.1은 이 표를 그래프로 나타낸 것이다.

프로그램	OpenPBS에서의 실행시간(sec)	부하 분산 시스템에서의 실행시간(sec)
xdlu	0	0
xddblu	0.0383	0.0383
xdls	0	0
xdllt	0	0
xdqr	0	0
xdinv	0.35	0.08
xdptllt	0.0562	0.0552
xdgblu	0.1214	0.0607
xdpbllt	2.56	0.6609

[표 6.1] OpenPBS와 구현된 부하 분산 시스템에서의 LINPACK벤치마크 수행시간



[그림 6.1] OpenPBS와 구현된 부하 분산 시스템에서의 LINPACK벤치마크 수행시간

전체 9개의 프로그램을 수행하였을 때 OpenPBS에서는 평균 약0.347초가, 부하 분산 시스템에서는 평균 약0.099초가 걸렸음을 알 수 있고, 1.8배의 성능 효과를 나타낸 것을 알 수 있다.

9개의 프로그램을 수행하였을 때 그 중 6개의 프로그램에서는 수행하는 데 같은 시간이 걸린 반면 3개의 프로그램에서는 부하 분산 시스템이 좋은 성능을 나타내는 것을 알 수 있다. MPI작업을 수행하는 데 있어서 각 노드에 걸리는 부하가 다른 환경 하에서 작업을 수행할 때, 현재 부하가 많은 노드와 부하가 적은 노드 사이에서 프로세스가 통신하면서 작업을 수행해야 하기 때문에 더 많은 성능의 저하를 가져옴을 알 수 있다.

현재의 시스템은 OpenPBS보다는 좋은 성능을 나타내었지만, MPI작업을 노드에서 실행하는 데 있어서는 좀 더 많은 부분을 추가하는 것이 필요하다.

제 7 장

결론 및 향후 과제

본 보고서에서는 클러스터 환경 하에서 작업을 수행할 때 전체 클러스터 시스템의 성능을 높이기 위해 설계, 구현한 부하 분산 시스템에 대하여 소개하였다. 클러스터 환경 하에서 부하의 불균형 현상은 전체 시스템의 성능에 좋지 않은 영향을 줄 수 있기 때문에 부하를 적절히 분배하는 방식으로 작업을 실행시키는 것이 중요하다. 이렇게 작업을 분배하기 위해서는 현재 클러스터 시스템에 포함되어 있는 노드들의 정보를 알고, 가장 부하가 적은 노드로 작업을 수행시키고, 필요에 따라 작업을 부하가 많은 노드에서 적은 노드로 옮기는 역할을 하는 부하 분산 시스템이 필요하다.

Central manager를 클러스터 시스템에 하나 두고, 작업을 수행시킬 각 노드에 monitor를 하나씩 두어 부하 분산 문제를 해결한다. Central manager는 최근의 노드의 부하 정보를 monitor에게 받아서 저장하고 관리하며, 이 정보를 바탕으로 사용자가 작업을 수행하고자 할 때 작업의 type이나 요구 사항을 파악해서 적절한 노드에서 수행될 수 있도록 한다. 서버에 있는 scheduler가 이런 역할을 하며, 주기적으로 노드의 상태를 확인해서 필요한 경우 부하가 높은 노드에서 낮은 노드로 작업을 이동하기도 한다. 또한, 이렇게 수행하거나 이동한 작업의 정보를 central manager와 작업이 수행중인 노드의 monitor에서 기록하고 관리한다.

이렇게 해서 구현한 부하 분산 시스템을 OpenPBS와 비교하기 위해서 노드 8개에 부하 분산 시스템과 OpenPBS를 각각 설치하여 SCALARPACK 벤치마크를 사용하여 성능을 측정한 결과 9개의 프로그램 중에서 3개의 프로그램에서 더 나은 성능을 보였고, 나머지 6개에서는 같은 수행시간을 보였다. 그리고 평균 1.8배의 수행시간 향상을 보였다.

응용 프로그램의 데이터 크기나 상태를 변화시켜 가면서 좀 더 많은 성능 측정을 수행하

고 MPI작업을 수행하는 데 있어서 지금보다 더 자세한 알고리즘을 구현할 예정이다. 또한 현재 많이 사용하고 있는 backfill방식을 사용하여 구현해보고, 여기서 좀 더 나아가 성능 향상을 가져올 수 있는 부분에 대해 살펴볼 생각이다. [16, 17]

또한 부하 정보를 현재는 5초마다 받아오고 있는데, 이 부분에서 작업이 5초 사이에 많이 들어오게 되면 부하 정보가 잘 반영되지 않고 있기 때문에, central manager에서 관리하는 작업 정보를 사용하여 이 부분을 개선하여 볼 생각이다.

제 8 장

참고 문헌

[1] Buyya, R., ed. 1999. High Performance Cluster Computing: Architectures and Systems. Vol. 1

[2] Narendran Ganapathy and Sunil Kumar, Process migration and Load Balancing. Term paper. Dept of Computer Science, Univ of New Hampshire

[3] Mary Papakhian, 1998. COMPARING JOB-MANAGEMENT SYSTEMS: THE USER'S PERSPECTIVE. IEEE COMPUTATIONAL SCIENCE & ENGINEERING

[4] Jean Suplick and Richardson, January 1994. An Analysis of Load Balancing Technology

[5] S. Zhou, "A trace-driven simulation study of dynamic load balancing," Rept. No. UCB/CSD 86/, University of California, Berkeley, June 1986.

[6] Litzkow, M., Livny, M. and Mutka, M. ''Condor — A Hunter of Idle Workstations''. Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif. June 1988

[7] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, and R.N. Sidebotham, Scale and performance in a distributed file system, ACM Transactions on Computer Systems, 6(1):55-81, February 1988.

[8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Serverless Network File Systems. In Proceedings of the 15th Symposium on Operating System Principles. ACM

[9] IBM LoadLeveler for AIX, Using and Administering, Version 2 Release 2

[10] Chin Lu, John C.S. Lui, Peter W.K. Lie, M.K. Tang, S.Y. Lau, H.K. Li, Distributed Scheduling Framework A Load Distribution Facility on Mach

[11] Sau-Ming LAU, Qin LU, Kwong-Sak LEUNG, Dynamic Load Distribution Using Anti-Tasks and Load State Vectors

[12] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle, UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. 1992. Software - Practice and Experience

[13] X. Zhang, Y. Qu, and L. Xiao, Improving distributed workload performance by sharing both CPU and memory resources, Proceedings of 20th International Conference on Distributed Computing Systems, (ICDCS'2000), Taipei, Taiwan, April 10-13, 2000.

[14] J.Dongarra. The LINPACK Benchmark: An Explanation. Supercomputing, Spring 1998.

[15] Jack Dongarra, Jim Bunch, Cleve Moler and Pete Stewart, "LINPACK", http://www.netlib.org/linpack/

[16] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In 12th International Parallel Processing Symposium, pages 542–546, April 1998.

[17] H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In Proceedings of SC99, Portland, OR, November 1999. IBM Research Report RC21559.