MPI-IO의 CrownFS 지원 방안

*조미옥° *강봉직 *최경희 **정기현
*아주 대 학 교 정보 및 컴퓨터 공학부, **아주대학교 전자 전기 공학부
jjomi@cesys.ajou.ac.kr, bjkang@orient.dytc.ac.kr, {khchoi, khchung}@madang.ajou.ac.kr

Supporting for CrownFS in MPI-IO

*Mi-Ok Jo° *Bong-Jik Kang *Kyung-Hee Choi **Gi-Hyun Jung
*The Professional Graduate School for Information&Communication Technology,
Ajou Univercity,
** Division of Eletronical & Electronics Engineering Ajou University

요 약

가장 느린 서브시스템인 I/O의 성능이 전체적인 컴퓨터 시스템의 성능을 결정짓게 된다. 따라서 전반적인 시스템의 성능 향상을 위해서는 I/O의 성능이 높아져야 한다. 분산병렬환경에서 I/O의 성능을 높이기 위해서 parallel I/O를 사용한다. 하위 레벨에서 최적화된 병렬 파일시스템을 사용하고, 어플리케이션 레벨에서 병렬 어플리케이션의 개발을 쉽게 해줄 수 있는 인터페이스를 사용하면 더 효과적인 parallel I/O를 구현할 수 있다. 본 논문에서는 MPI에서 병렬 파일 시스템인 CrownFS를 지원하도록 하기 위해서 MPI-IO에 CrownFS를 추가하여 병렬환경에서 높은 성능을 나타낼 수 있는 parallel I/O 환경을 구현한다.

1. 서론

가장 느린 서브시스템인 I/O의 성능이 전체적인 컴퓨터 시스템의 성능을 결정짓게 된다. 따라서 전반적인 시스템의 성능 향상을 위해서는 I/O의 성능이 높아져야 한다. Multiprocessor나 clustering환경에서 여러 개의 프로세서를 통해 동시에 실행되는 병렬 어플리케이션의 경우 하나의 대용량 디스크나 RAID 시스템으로 제공할 수 있는 I/O의 성능은 상당히 제한적일 뿐만 아니라 전체적인 시스템의 성능저하를 초래할 수 있다. 이런 병렬 환경에서의 I/O 시스템의 성능 향상을 위해서 제안된 것이 parallel I/O이다. Parallel I/O는 데이터를 분할 저장하여 여러 머신의 입출력 능력을 모아 전체 입출력 전송률을 높이는 병렬 파일시스템을 사용한다. 병렬파일시스템은 하나의 디스크나 RAID 시스템보다 높은 I/O 성능을 나타낸다. 그러나 사용자에게는 물리적으로는 분산되어 있는 파일로 인식하지 않고, 하나의 이미지 파일로 보게 됨으로써 어플리케이션 개발시 편의성을 높여준다.[1]

MPI(Message Passing Interface)는 병렬 어플리케이션을 개발하는데 공통된 인터페이스를 제공하기 위한 라이브러리 표준안이다. MPI는 I/O부분에서도 머신들간에 protability와 flexibility를 제공하면서도 입출력의 효율성을 높이기 위해 프로세스간에 파일 데이터를 분할 하거나, 프로세스들의 입출력 요구를 모아서 처리하는 collective I/O 등의 기능을 지원함으로써 병렬 어플리케이션들의 데이터 억세스 패턴에 최적화되어 있다.[2]

현재 MPI에서는 유서 레벨 병렬 파일시스템으로 PVFS (Parallel File System for Linux Clusters)를 제공하고 있다. PVFS는 LINUX 클러스터 환경을 지원하지만, file locking과 nonblocking I/O 등을 제공하지 않아서 MPI-IO에 정의되어 있는 shared file pointer와 같은 함수들을 구현하지 못한다. CrownFS 파일 시스템은 워크스테이션 및 리눅스 클러스터 환경에서 병렬 입출력이 가능한 유저레벨 파일 시스템이다. 그리고 PVFS 파일 시스템에 비해 파일의 물리적인 분산 정보인 메타 데이터를 관리하는 서버가 여러 개로 구성되어 있어서 더 안정적이며, file locking을 제공함으로써 더 많은 MPI-IO 함수들을 지원할 수 있다.

본 논문에서는 MPI에서 병렬 파일 시스템인 CrownFS 파일 시스템을 지원하도록 하기 위해서 MPI-IO에 CrownFS 파일시스템을 추가해서 병렬환경에서 높은 성능을 지원할 수있는 parallel I/O 환경을 구현하고자 한다. 2장에서는 관련 연구로 MPI-IO의 특징과 ROMIO의 구성, ROMIO의 PVFS 파일 시스템에 대해서 살펴보고, 3장에서는 ROMIO에 파일 시스템을 추가하는 방법과 CrownFS 파일 시스템의 특징, 그리고 CrownFS 파일 시스템을 ROMIO에 추가하는 구현 방법에 대해서 논하고자 한다.

2. 관련연구

2.1 ROMIO- ADIO(Abstract-Device Interface for I/O)

open, lseek, read, write, close와 같은Unix의 기본 I/O 함수들은 Unix와 유사한 거의 모든 운영체제에서 제공하고 있지만 이 함수들 만으로는 MPI-IO에서 제공하고자 하는 기능들을 모두 지원할 수 없다.

첫번째로 Unix의 기본 I/O함수들은 모두 blocking 함수들로 nonblocking 함수들은 파일 시스템별로 다른 함수들을 사용하고 있다.

두번째로 Unix의 기본 I/O 함수들이 지원하는 파일 사이즈는 2GB로 제한되어 있다. MPI-IO에서 큰 파일 사이즈 지원에 관한 것은 언급되고 있지 않지만 높은 이식성을 위해서는 파일 사이즈 제한에 관한 것이 해결되어야 한다.

세번째로 파일 시스템마다 MPI-IO에서 제공하는 기능을 지원하기도 하지만 지원하지 않는 경우도 있다. 예를 들어 IBM PIOFS나 Intel PFS와 같은 파일 시스템에서는 사용자가 파일의 striping을 제어할 수 있고, 파일의 억세스 모드 - atomic or nonatomic-를 선택하는 기능을 제공한다.

네번째로 Unix의 기본 I/O 함수들을 파일 시스템마다 모두 제공하기는 하지만 항상 이들의 사용을 권장하지는 않는다. 예를 들어 Intel Paragon 같은 경우 cread, cwrite 함수를, SGI IRIX 6.5 같은 경우는 pread64, pwrite64 와 같은 함수를 사용할 것을 권장하고 있다.

이와 같은 제약점에 의해 MPI-IO를 POSIX I/O의 인터페이스 위에서 구현하고자 하는 대안도 나왔으나, 아직 많은 파일 시스템에서 POSIX I/O를 제한적으로 지원하고 있기 때문에 이도 적합치 않다.

따라서 파일 시스템별로 높은 성능과 완전한 기능성을 제공하는 MPI-IO를 구현하기 위해서는 각각의 파일 시스템에서 고유하게 지원하는 기능 및 함수들을 이용해야 한다. 대부분의 MPI 구현 라이브러리에서 I/O 부분으로 사용하고 있는 ROMIO는 MPI-IO 함수들의 인터페이스를 구현해놓은 파일 시스템 독립적인 부분과 직접 시스템콜들을 호출하는 코드를 포함하는 파일 시스템에 의존적인 부분으로 함수들을 구성하는 이중 구조 방식을 취하고 있다. 따라서 새로운 파일 시스템을 ROMIO에서 지원하도록 하기 위해서는 파일 시스템 의존적인 부분만 다시 그 파일 시스템의 특성에 맞게 최적화 시켜서 구현해서 추가하면 된다. ROMIO에서 사용하는 이런 구조를 ADIO라고 한다. ADIO는 다양한 파일 시스템을 지원하는 이식성이 높은 병렬I/O APIs를 구현하기 위해서 디자인된 메커니즘이다.

ADIO는 파일 시스템에서 제공하는 시스템콜들을 호출하는 머신 의

존적인 부분과 ADIO 위에서 병렬 I/O의 구현을 위 한 인터페이스 부분인 머신 독립적인 부분으로 구성되어 있다. 파일 시스템 의존적인 기본 함수들을 파일 시스템별로 따로 구현하므로, ADIO는각각의 파일 시스템 별로 최적화될 수 있다.[3]

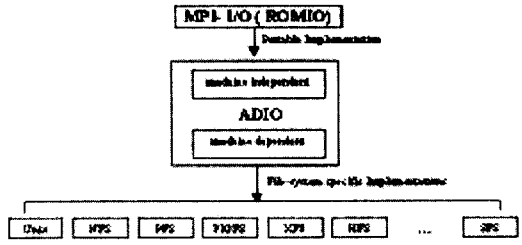

그림 1 ROMIO 와 ADIO 의 구조

2.2 CrownFS와 PVFS의 비교 분석

2.2.1 PVFS(Parallel Virtual File System for Linux Cluster)

PVFS는 병렬 어플리케이션들이 파일 데이터를 높은 속도로 억세스할 수 있도록 하기 위해 구현된 병렬 파일 시스템이다. 병렬 파일 시스템의 기본적인 기능외에도 데이터의 스트라이핑을 유저가 제어할 수 있고, 유저 레벨 파일 시스템으로 구현되어 커널을 수정할 필요가 없다.

초기에는 유닉스 클러스터링 환경에서 구현되었던 PVFS가 PC 클러스터링 환경을 지원하기 위해서 linux cluster환경을 지원할 수 있도록 porting 되었다.

많은 프로세스 또는 쓰레드들이 동시에 파일을 읽고 쓸 수 있으며 PVFS 파일에 대해서 UNIX의 일반적인 shell command인 cp, ls, rm 등을 지원하고, UNIX I/O API 들을 이용해서 개발된 어플리케이션들이 재컴파일 없이 PVFS 파일들을 억세스 할 수 있도록 했다.

PVFS는 크게 Manager Daemon, I/O Daemons, Client Library, APIs들로 구성이 된다.

하나의 Manager Daemon이 PVFS 파일 시스템의 전체 메타데이터를 관리한다. 메타데이터를 이용해서 파일을 생성, open, close, delete 하는데 필요한 퍼미션 체크를 하고 어플리케이션에서 필요로 하는 I/O Daemon에 대한 정보를 알려준다. 메타데이터는 파일에 관련된 퍼미션, owner, group, 파일 데이터의 물리적인 분산에 대한 정보를 가지고 있다. 파일의 분산 저장에 대한 정보는 파일이 위치하고 있는 디스크의 위치와 클러스터 내에서 디스크의 위치 정보를 포함한다.

I/O Daemon은 클러스터 노드 하나로 구성되는 각각의 I/O node에서 하나씩 실행된다.각각의 I/O 데몬은 로컬 파일 시스템 내에서 PVFS 파일이 저장되어 있는 위치에 대한 정보 를 가지고 있다.

Client Library는 Manager Daemon을 통해서 필요로 하는 I/O Daemon의 정보를 알아온 후 I/O Daemon과 직접 연결해서 데이터를 요청하게 된다. 클라이언트로부터 데이터 요청이 들어오게 되면 I/O Daemon이 가지고 있는 파일 데이터의 위치 정보에 따라서 이를 억세스 할 수 있는 디스크위치를 부여받는다.

마지막으로 PVFS는 native API, UNIX/POSIX API, MPI-IO등 여러 API 를 지원하고, native API의 경우 I/O 효율을 높이기 위해서 simple striping, logical view를 제공한다. [4][5]

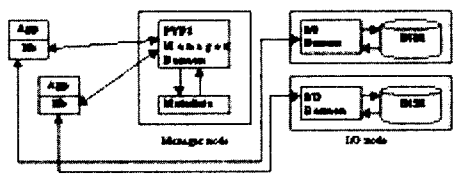

그림 2 PVFS 의 구성

2.2.2 CrownFS

CrownFS는 라이브러리들과 Message Passing Library(MPL), File System Daemon(FSD), I/O Daemon(IOD) , Metadata Server(MS), 그리고Buffer Cache 모듈로 구성된다.

FSD는 파일시스템의 제어를 담당하며 노드 내부의 통신을 위한 기능들을 제공한다. 그리고 버퍼 할당 및 캐시를 관리한다.

MS는 메타 데이터를 저장하며, 각 노드마다 실행이 되어서 시스템 오류에 따른 전체 시스템 작동 중지를 막는다. 또한 fault manager를 통해서 MS간의 우선순위와 메타데이터 업데이트를 제어하고, lock manager를 통해서 FSD를 관리한다.

IOD는 실제적인 데이터를 저장관리한다. IOD에서는 다수의 저장 미디어를 하나의 저장장치로 간주하며 처리하게 되고, 모든 동작은 FSD와 MS에 의해 제어를 받는다.

Buffer cache는 접근 패턴에 따른 prefetching과 재사용을 위한 caching을 하게 된다.

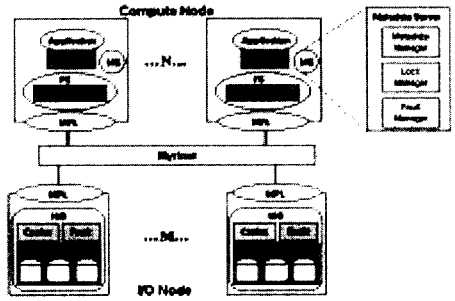

그림 3 CrownFS 의 구성

2.2.4 PVFS와 CrownFS의 비교

PVFS는 TCP/IP 소켓을 사용하기 때문에 추가적인 메시지 패싱 라이브러리가 필요 없다.

또한 여러 인터페이스를 지원하므로 binary들을 수정할 필요가 없다. 그러나 PVFS는 유저레벨의 파일 시스템을 구현하기 위해서 리눅스의 LD_PRELOAD 변수를 이용해서 libc의 system calls wrapper 함수들을 PVFS의 wrapper함수들로 바꾼다. PVFS syscall wrapper함수는 요청된 시스템콜이 PVFS파일을 access하는 시스템 콜인 경우 PVFS I/O 라이브러리로 연결해준다. 그러나 이방식의 경우 기존의 어플리케이션들을 재컴파일 없이 그대로 사용할 수 있다는 장점을 가지나, 시스템 라이브러리가 바뀌는 경우 PVFS 라이브러리도 계속 변경되어야 한다.

또한 유저레벨로 구현된 파일 시스템이므로 exec()를 call할경우 유저 공간에서 관리되던 PVFS 파일에 대한 정보가 모두 사라진다

CrownFS는 CrownFS가 제공하는 시스템콜을 사용해야 하므로 CrownFS의 파일들을 사용하기 위해서는 binary의 수정이 필요하다. 그러나 CrownFS의 시스템콜을 사용하므로써 PVFS처럼 시스템 라이브러리가 수정될 때마다 업데이트를 해야하는 불편함이 없다.

또한 CrownFS는 여러 개의 메타데이터 서버가 동작하므로써 하나의 메타데이터 서버가 죽어도 시스템의 유지가 가능하다는 장점을 가지고 있다. 그리고 PVFS에서는 파일 locking을 지원하지 않는데 비해 CrownFS는 클러스터내의 lock이 필요한 작업을 관리해주는 lock manager를 제공한다. fault된 모듈을 체크를 해주는 fault manager와 데이터 복구를 위한 미러링 기능도 제공한다.

3.본론

3.1 ROMIO, ADIO 분석

3.1.1 Collective I/O 및 Strided I/O의 구현 방식 분석

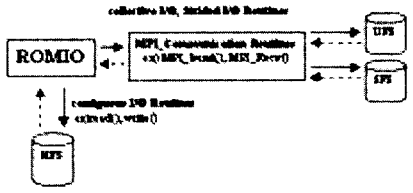

그림 4 ROMIO 의 I/O 루틴의 구성

ROMIO(ADIO)에서는 파일 데이터가 하나의 저장공간에 연속적으로 저장되어 있는 경우 read(), write()와 같은 system call을 직접 호출해서 데이터를 억세스 한다.

그러나 collective I/O, strided I/O 함수들의 구현에 있어서는 직접적으로 socket과 같은 통신 system call을 직접 호출하지 않고, MPI에서 정의된 communication Routine들을 사용한다. 따라서 새로운 파일 시스템을 추가

하는 경우 system call을 직접 호출하는 연속적으로 저장되어 있는 파일을 억세스 하는 루틴만 다시 구현해주면 된다. Collective I/O와 strided I/O에 관련된 부분은 MPI의 communication 루틴들을 사용하므로 ADIO에서 현재 구현되어 있는 공통된 루틴을 사용해서 지원이 가능하다[6][7]

3.1.2 ROMIO 루틴의 구성 방식

ROMIO의 함수들의 구성 방식은 호출하는 MPI-IO 함수, ADIO함수, system call등의 계층에 따라서 크게 5가지의 유형으로 나눌 수 있다.

가. 파일 시스템 고유의 system call을 호출하는 경우

파일 시스템별로 구현되어 있는 ADIO의 머신 의존적인 함수 계층의 함수들을 호출하는 루틴들이 속한다.

MPI-IO의 함수가 머신 독립적인 ADIO 함수들을 호출한다. 호출된 *ADIO*함수는 파일 시스템의 종류에 따라 머신 의존적이고, 파일 시스템별로 구분된 ADIO함수를 호출한다. 파일 시스템 별로 구현된 ADIO함수들은 함수명 내에 파일 시스템의 이름을 포함시켜 구분한다.
호출된 해당 파일 시스템의 ADIO 함수는 다시 system call을 호출하거나 아니면 *ADIO*에서 구현해놓은 공용 루틴을 호출하는 방식으로 나누어진다.

먼저 *system call*을 호출하는 방식은 파일 시스템 별로 다른 형식으로 제공하거나 함수의 처리 과정이 다른 경우, 기본 Unix의 I/O 함수들보다 더 좋은 기능을 제공하는 최적화된 함수가 지원되는 경우가 속한다.

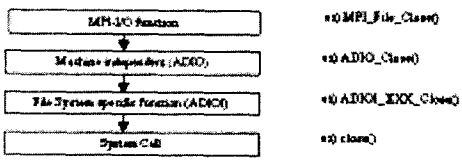

그림 5 *system call* 을 호출하는 *ADIOI* 함수 구성방식

나. 공통적으로 구현되어 있는 ADIOI루틴을 사용하는 경우

보통 해당 파일 시스템에서는 제공되지 않는 collective I/O나 파일을 나누어서 저장하는(strided I/O)경우, 파일의 힌트(file info)기능 등을 제공하기 위해서 ADIO에서 구현해 놓은 공용 루틴을 사용하게 된다. 그러나 앞의 system call을 호출하는 경우와 마찬가지로 파일 시스템별로 다른 기능이나 다른 조건에서 처리되어야 한다면 별도로 구현되어야 한다.

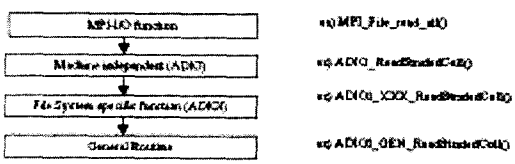

그림 6 공용 루틴을 호출하는 ADIOI 함수 구성 방식

다. 모든 파일 시스템에서 공통적인 system call을 이용하는 경우

*ADIO*루틴에서 *system call*을 호출 하지만 *system call*의 형식이 파일 시스템에 구분 없이 동일하게 지원되는 경우 ADIO 머신 독립적인 루틴에서 바로 *system call*을 호출한다. 그러나 현재까지 이런 구성방식을 취하는 함수는 unlink()를 호출하는 MPI_File_delete() 밖에 없다.

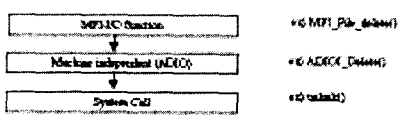

그림 7 ADIO 루틴에서 공통된 system call 을 호출하는 구성 방식

라. MPI-IO 루틴과 ADIO 루틴만으로 구성되는 경우

MPI-IO 함수에서 머신 독립적인 ADIO 함수를 호출하는 경우이다. 호출된 머신 독립적인 ADIO함수에서 기능을 다 구현하는 경우도 있고, 또는 다시 다른 ADIO 함수들을 호출하거나 아니면 다른 MPI함수들을 호출 하기도 한다.

파일을 직접 억세스 하는 함수들보다는 파일의 상태 정보를 설정하거나, ADIO와 MPI-IO의 기능들을 제공하기 위한 내부 루틴들이 주로 속한다. 또는 파일의 상태정보를 알기 위해서 다른 *I/O* 부분이 아닌 다른 MPI 루틴들을 호출는 경우도 이런 방식에 속한다.

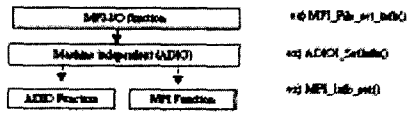

그림 8 MPI-IO 루틴과 ADIO 루틴만으로 구성되는 방식

마. MPI-IO 루틴만으로 구성이 되는 경우

MPI 함수만으로 구성되는 방식이다. 파일의 간단한 상태 정보를 알려주는 함수들이 주로 속한다. 내부적으로 다른 MPI함수들을 호출하기도 한다.

그림 9 MPI-IO 루틴만으로 구성되는 방식

3.2 MPI-IO 함수의 구현을 위해 필요한 system calls

MPI-IO 함수들을 지원하기 위해서는 기본적으로 파일 시스템에서 제공되어야 하는 시스템 콜들이 있다. 먼저 file을 open하고 close하는 open()과 close()가 필요하다.

Blocking I/O 함수들을 지원하기 위해서는 read(), write(), lseek() 함수가 필요하다. lseek()의 경우 SEEK_SET, SEEK_CUR, SEEK_END가 모두 제공되어야 한다.

Nonblocking I/O 함수들은 시스템에서 제공하는 asynchronous 시스템 콜을 사용한다. POSIX Realtime Extensions, IEEE Std 1003.1b에 정의되어 있는 함수들을 예로 들면 aio_return(), aid_error(), aio_suspend(), aio_read(), aio_write()가 제공되어야 한다. 만약 이와 같은 함수들을 파일 시스템에서 제공하지 않는다면 nonblocking I/O함수들을 사용하기 위한 인터페이스는 제공되지만 실제적으로는 blocking I/O 함수들을 사용한다.

Shared file pointer 관련 함수들은 파일의 locking을 이용해서 구현되기 때문에 fcntl()의 F_SETLK, F_SETLKW가 제공되거나 아니면 시스템별로 파일을 locking할 수 있는 함수가 제공되어야 한다.
이외에도 unlink(), fsync(), ftruncate() 함수들이 필요하다.

4. 결론 및 향후과제

ROMIO에서 병렬파일시스템인 CrownFS를 지원함으로써 I/O의 성능을 향상시킬 수 있을 뿐만 아니라, MPI-IO의 병렬 어플리케이션의 개발을 쉽게할 수 있는 어플리케이션 레벨의 환경도 구축이 가능하다.

현재 개발중인 CrwonFS가 제공하는 *system calls*을 토대로 *ADIO* 내부 루틴들을 CrownFS에 최적화 시키는 방안에 대해 분석 후 실제로 구현을 할 예정이다. read(), write()를 비롯한 가장 기본적인 몇 개의 함수들만 있으면 MPI-IO의 모든 기능들이 구현 가능하다. 하지만 CrownFS에서 사용을 권장하는 고유의 함수들을 사용하는 것이 더 높은 성능을 나타낼 것으로 예상된다.

5. 참고 문헌

[1]. Apratim Purakayastha,``Characterizing and Optimizing Parallel File Systems", May 1996.

[2]. Message Passing Interface Forum, "MPI-2 : Extensions to the Message Passing Interface, Ch.9 I/O", July 1997, p.209-270

[3]. Rajeev Thakur, William Gropp, and Ewing Lusk, ``On Implementing MPI-IO Portably and with High Performance," in Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, May 1999, pp. 23--32.

[4]. W. B. Ligon III and R. B. Ross, ``An Overview of the Parallel Virtual File System", Proceedings of the 1999 Extreme Linux Workshop, June, 1999.

[5]. W. B. Ligon III, R. B. Ross, and R. Thakur, ``PVFS: A Parallel File System For Linux Clusters" , submitted to ICS 2000, December, 1999

[6]. Rajeev Thakur, William Gropp, and Ewing Lusk, ``Data Sieving and Collective I/O in ROMIO," in Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation, February 1999, pp. 182--189.

[7]. .Rajeev Thakur, William Gropp, and Ewing Lusk, ``An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces ," in Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp. 180-187.