이식성을 고려한 사용자기반 MPI 체크포인터의 설계 및 구현

(Design and Implementation of a User-based MPI Checkpointer for Portablility)

안 선 일 [†]   한 상 영 [††]
(Sun-Il Ahn)   (Sang-Yong Han)

요 약 MPI 체크포인터는 MPI 응용 프로그램에 체크포인팅을 통해 결함내성을 제공하는 툴이다. 네트워크의 개방성이 확대되고 GRID에 대한 활용이 증가함에 따라 MPI 체크포인터가 다양한 플랫폼과 MPI 구현들로 쉽게 이식되어야 한다는 요구가 커지고 있다. 기존의 MPI 체크포인터들은 자동 체크포인팅과 복구 기능에 초점을 맞추었고 이식성에 대한 고려가 없었기 때문에, 다른 플랫폼과 MPI 구현들로 이식되기 어려웠다. 본 논문에서는 사용자기반 MPI 체크포인터인 STFT를 개발하면서 이식성을 위해 고려하였던 설계 및 구현 이슈들에 대해 설명한다.

STFT는 MPI 체크포인터의 이식성을 위해 첫째로 단일 프로세스 체크포인터들에 대한 추상화 인터페이스를 제시한다. 둘째로 사용자기반 체크포인팅 방법에서 사용자가 체크포인팅할 수 있는 지점을 제한하여 메시지 체크포인팅을 회피한다. 셋째로 네트워크 연결을 재생성하기 위해 MPI_Init가 다른 랭크를 가진 프로세스들과 항상 고정된 순서대로 연결을 생성하도록 강제한다. 이를 통해 STFT는 다양한 플랫폼과 MPI 구현들로 쉽게 이식 가능할 것으로 기대되며, 우리는 프로토타입의 구현을 통해서 STFT가 LAM과 MPICH/P4의 두 MPI 구현들로 쉽게 이식 가능함으로 확인하였다.

키워드 : MPI, 체크포인팅, 이식성

Abstract An MPI Checkpointer is a tool which provides fault-tolerance through checkpointing. The previous researches related to the MPI checkpointer have focused on automatic checkpointing and recovery capabilities, but they haven't considered portability issues. In this paper, we discuss design and implementation issues considered for portability when we developed an MPI checkpointer called STFT.

In order to increase portability, firstly STFT supports the abstraction interface for a single process checkpointer. Secondly, STFT uses a user-based checkpointing method, and limits possible checkpointing places a user can make. Thirdly, STFT lets the MPI_Init create network connections to the other MPI processes in a fixed order. With these features, we expect STFT can be easily adaptable to various platforms and MPI implementations, and confirmed STFT is easily adaptable to LAM and MPICH/P4 with the prototype implementation.

Key words : MPI, checkpointing, portability

1. 서 론

분산된 대용량 병렬처리 시스템에서 결함내성은 매우 중요하다. 이것은 대용량 병렬처리 시스템에서 실행되는 응용 프로그램들이 많은 CPU 자원과 장시간 실행되는 것을 요구하지만, 여러 시스템으로 구성되어서 짧은 MTBF(mean time between failure)를 갖기 때문이다.

체크포인팅(checkpointing)은 이러한 단점을 극복하기 위해 사용될 수 유용한 툴이다. 체크포인팅은 실행되고 있는 프로그램의 상태를 영구 저장소에 저장한다. 그래서 어떠한 이유(하드웨어 혹은 소프트웨어적인 실패)로 프로그램이 종료되는 경우, 프로그램의 상태를 가장 최근의 체크포인트(checkpoint)로부터 복구하여, 시스템에 실패가 발생하지 않은 것처럼 재시작할 수 있다.

† 정 회 원 : 한국과학기술정보연구원 연구원
sunilahn@paran.com
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
syhan@pplab.snu.ac.kr
논문접수 : 2003년 3월 17일
심사완료 : 2005년 9월 29일

MPI[1]는 대용량 병렬 처리 시스템에서 고성능을 목표로 1994년에 MPI 포럼에 의해 정의된 메시지 전송을 위한 표준이다. MPI 표준은 MPI 응용 프로그램에 오류가 발생한 경우 기본적으로 응용 프로그램을 즉시 종료하도록 정의하고 있다. 그러므로 대부분의 MPI 구현들[2-4]은 프로세스의 실패와 같은 심각한 상황이 발생하는 경우, 계산에 참여한 모든 프로세스들의 실행을 종료한다.

MPI 체크포인터는 MPI 응용 프로그램에 체크포인팅을 통해 결함내성을 제공하는 툴이다. 분산 대용량 병렬 처리 시스템에서 결함내성이 점점 중요해짐에 따라 많은 연구[5-10]들이 MPI 응용 프로그램에 결함내성을 제공하기 위해 체크포인팅 방법을 사용하였다. 네트워크의 개방성이 확대되고 GRID[11]에 대한 활용이 증가함에 따라 MPI 체크포인터가 다양한 플랫폼과 MPI 구현들로 쉽게 이식되어야 한다는 요구가 커지고 있다. 그러나 기존의 MPI 체크포인터들은 사용자의 간섭 없이도 자동으로 체크포인팅과 복구가 이루어지는 시스템수준 체크포인팅의 연구에 초점을 맞추었고 이식성에 대한 고려가 없었기 때문에, 다른 플랫폼과 MPI 구현으로 이식되기 어려웠다. MPI 체크포인터의 이식성이 크면 사용자는 특정 플랫폼이나 MPI 구현에 제한되지 않고 다양한 플랫폼과 MPI 구현을 선택할 수 있다. 본 논문에서 설명하려는 MPI 체크포인터는 이식성을 고려한다는 면에서 기존의 MPI 체크포인터 연구들과 차별성이 있다.

본 논문에서는 사용자기반 MPI 체크포인터인 STFT (Semi-Transparent Fault-Tolerance for MPI)를 개발하면서 이식성을 위해 고려하였던 설계 및 구현 이슈들을 설명한다. STFT는 MPI 체크포인터의 이식성을 위해 첫째로 단일 프로세스 체크포인터들에 대한 추상화 인터페이스를 제시한다. 둘째로 사용자기반 체크포인팅 방법에서 사용자가 체크포인팅할 수 있는 지점을 제한하여 메시지 체크포인팅을 회피한다. 셋째로 네트워크 연결을 재생성하기 위해 MPI_Init가 다른 랭크를 가진 프로세스들과 항상 고정된 순서대로 연결을 생성하도록 강제한다. 이를 통해 STFT는 다양한 플랫폼과 MPI 구현들로 쉽게 이식 가능할 것으로 기대되며, 우리는 프로토타입의 구현을 통해서 STFT가 LAM[2]과 MPICH/P4[3]의 두 MPI 구현들로 쉽게 이식 가능함으로 확인하였다.

본 논문의 내용은 쉽게 이식 가능한 MPI 체크포인터에 대한 것이다. STFT는 MPI 체크포인터 사용자가 다양한 플랫폼과 MPI 구현을 선택하는 것을 가능하게 하며, 그 효용성은 GRID와 같이 다양한 플랫폼을 활용하는 환경에서는 더욱 증가될 것으로 기대된다. 또한 PVM 등 다른 병렬프로그램 환경의 체크포인터들에도 동일한 아이디어를 확장하여 적용하는 것도 가능하리라 기대한다.

본 논문의 나머지는 다음과 같이 구성된다. 2장에서는 기존의 연구들에 대해 소개하고, 3장에서는 MPI 체크포인터의 이식성을 위한 여러 설계 및 구현 이슈들에 대해 설명한다. 4장에서는 STFT의 프로토타입 구현 환경과 성능을 분석하고, 5장에서는 본 논문을 결론짓는다.

2. 관련 연구

Cocheck[5]는 MPI에 결함내성을 제공한 최초의 연구들 중 하나로써, Condor[12]라는 단일 프로세스 체크포인팅 툴을 확장하였다. Cocheck는 MPI 응용 프로그램을 일관성 있는 상태[13]로 저장하기 위해 싱크앤스탑(sync and stop) 프로토콜[14]을 사용한다. 체크포인팅을 담당하는 프로세스가 다른 모든 프로세스들에게 체크포인팅 메시지를 전송하면 다른 프로세스들은 하던 일을 멈추고 체크포인팅을 수행한다. 체크포인팅은 프로세스의 이미지와 네트워크로 전송 중인 메시지를 저장하는데, Cocheck는 RM(ready message)을 사용하여 네트워크로 전송 중인 메시지가 목적지에 도착하는 것을 기다린다. MPI 표준은 프로세스 사이의 메시지의 전송은 선입선출(FIFO) 방식으로 동작한다고 정의하므로, 모든 프로세스들은 다른 프로세스들에게 RM메시지를 보내고, 모든 프로세스들로부터 RM을 받은 후에는 네트워크로 전송중인 모든 메시지가 목적지에 도착했다는 것을 보장할 수 있다. 그런 후 각 프로세스들은 프로세스의 이미지와 전송된 메시지들을 체크포인트로 저장하였다가, 나중에 실패가 발생하면 가장 최근의 체크포인트로 돌아가서 재시작할 수 있다.

Cocheck는 이식성을 위해 MPI 위에서 동작하는 것을 목표로 하였으나 실제적으로는 몇가지 이유로 tuMPI라는 MPI 구현의 소스 코드를 수정하여 구현되었고, 이 때문에 이식성이 떨어진다. 그 첫째 이유는 MPI_Init 함수가 여러 번 호출되는 것에 대해 MPI 위에서 적절하게 처리할 수 없었으며, 둘째로 네트워크로 전송 중인 메시지를 체크포인팅하는 것에 대해 MPI 위에서 적절하게 처리할 수 없었기 때문이다. STFT는 Cocheck와 달리 완전히 MPI 위에서 설계 및 구현되어서, 쉽게 다양한 플랫폼과 다양한 MPI 구현에 적용되어 사용될 수 있다. *Cocheck*의 이식성 문제점들에 대한 STFT의 해결 방안은 3.2절과 3.3절에서 다시 설명된다.

Starfish[6]는 Ensemble[15] 분산 시스템에 기반을 둔 정적 혹은 동적인 MPI 응용 프로그램들의 실행을 위한 환경이다. Starfish는 여러 결함내성 방법을 제공한다. 첫째는 체크포인팅과 재시작을 사용하는 방법이다. 둘째는 이벤트 모델을 사용하여, 동적으로 클러스터

환경이 변경되는 이벤트가 발생하는 경우 정상적으로 동작하는 프로세스들이 작업을 다시 재분배하여 실행하는 방법이다.

CLIP[7]은 Intel Paragon을 위한 체크포인팅 툴이다. CLIP은 시스템에 의한 체크포인팅이 구현에 따라 정확성이 보장되지 않고 성능을 저하시킬 수 있기 때문에 본 논문의 STFT처럼 사용자에 의한 체크포인팅 방법을 사용한다. FT-MPI[16]는 MPI-2[17]의 DPM(dynamic process model)을 지원하여 사용자에 의해 결함내성이 이루어지도록 하는 방법을 사용한다. MPICH-V[8], Egida[9], MPI/FT[10]는 프로세스들의 이미지를 체크포인팅하는 동시에, 전송되는 메시지에 대한 로그를 기록한다. 이를 통해 실패한 프로세스가 체크포인트로부터 다시 시작할 때 저장된 메시지들의 로그를 이용함으로써 정상적인 프로세스들의 실행을 방해하지 않는다는 장점을 갖지만, 로그를 관리하는 비용이 크다.

기존 MPI 체크포인터들[5-10]은 Cocheck와 비슷한 이유로 완전히 혹은 부분적으로 특정 MPI의 구현이나 플랫폼에 종속적이어서 이식성이 크지 않다. 이에 반해 본 논문에서 제안하는 STFT는 플랫폼이나 MPI의 구현에 종속되지 않고 MPI 위에서 설계 및 구현되었기 때문에 이식성이 크다.

3. 이식성을 위한 STFT 설계 및 구현

사용자기반 MPI 체크포인터인 STFT는 높은 이식성을 제공하여 다양한 플랫폼과 MPI 구현들로 이식되는 것을 목표로 하였다. 이를 위한 STFT 시스템은 그림 1과 같이 구성된다. STFT는 특정 MPI 구현에 대한 종속성을 피하기 위해 MPI 표준 위에서 설계되고 구현되었다. 그림1의 구성요소들 중 "단일프로세스 체크포인터(이하 SPC, Single Process Checkpointer)"는 실행 중인 프로세스의 이미지를 체크포인팅 하는 역할을 한다. 기존의 SPC로는 Condor[12], Ckpt[18], LibCkpt[19] 등이 있으며, STFT는 기존의 SPC들을 활용하여 MPI 응용 프로그램의 체크포인팅 기능을 수행한다. "SPC 인터페이스"는 SPC를 추상화한 것으로써, MPI 체크포인터가 다른 플랫폼으로 이식되어 다른 SPC를 채용할 필요성이 생기더라도, MPI 체크포인터가 수정되는 것을 최소화시킨다. "STFT 라이브러리"는 사용자기반 체크포인팅 기능을 지원하기 위해 "MPI_Ckpt" 함수와 체크포인팅 프로토콜을 제공하며, 응용 프로그램이 "MPI_Ckpt" 함수를 호출하면 프로세스들의 이미지가 체크포인트 파일로 기록된다. "STFT 재시작 프로그램"은 체크포인트 파일들에 저장된 MPI 응용프로그램을 복구하여 재시작 하고 종료된 MPI 네트워크 연결을 재생성하는 역할을 한다. 본 장의 나머지에서는 이식성 향상이라는 점에 초점을 맞추어 STFT의 각 구성요소들에 대해 설명한다.

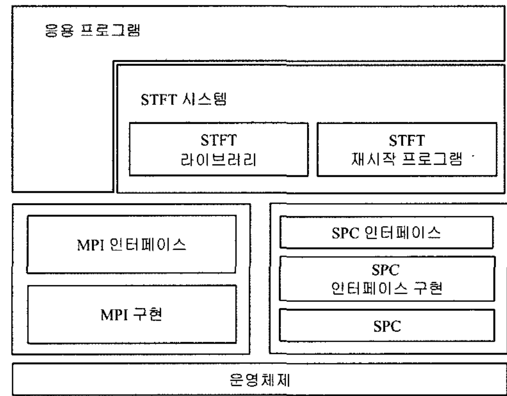

그림 1 STFT 시스템 구성도

3.1 SPC와 SPC 인터페이스

MPI 체크포인터는 실행중인 MPI 프로세스들의 이미지를 영구저장소에 저장하는 기능을 지원해야 한다. STFT는 LibCkpt, Condor, CKPT와 같은 기존에 이미 개발된 SPC를 그대로 활용하여 프로세스 이미지 체크포인팅 기능을 제공한다. 실행되고 있는 프로세스의 상태를 저장하는 작업은 플랫폼에 종속적이고, 플랫폼마다 새로 개발하는 것은 부담이 큰 작업이기 때문에 STFT는 기존의 SPC를 최대한 활용한다.

이식성을 확보하는 위해 Cocheck는 대부분의 유닉스 운영체제에 이식 가능한 것으로 알려진 LibCkpt를 활용하였다. 그러나 이 방법은 LibCkpt에 종속되며, LibCkpt의 계속된 지원이 없다면 MPI 체크포인터의 이식성에도 한계가 따른다.

이식성을 향상시키기 위해 STFT는 MPI 체크포인터에게 추상화된 SPC 인터페이스를 제공하는 방법을 사용한다. 그림 2는 SPC 인터페이스에 대한 개념도이다. SPC 인터페이스는 SPC를 추상화하여 MPI 체크포인터가 SPC의 내부 구조에 대해서 모르더라도 SPC 인터페이스를 통해 SPC를 활용하게 한다. 이를 위해서는 각 SPC마다 MPI 체크포인터를 위한 SPC 인터페이스 구현을 제공할 필요가 있다. STFT는 SPC 인터페이스를 통해 MPI 체크포인터가 다른 플랫폼에 이식되어 새 SPC를 채용할 필요성이 생기더라도, MPI 체크포인터 자체는 수정할 필요 없이 새 SPC에 맞추어 인터페이스만 구현하면 되도록 하였다. 이를 통해 플랫폼이 변경되었을 때 SPC가 변경되는 것에 따른 MPI 체크포인터의 변경을 최소화할 수 있다.

그림 3은 CKPT를 위해 SPC 인터페이스를 구현한

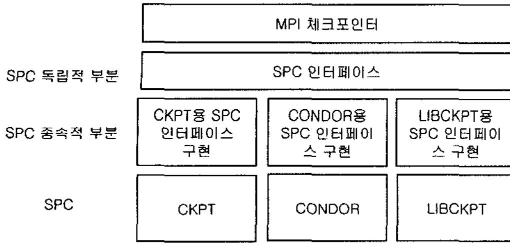

그림 2 SPC 인터페이스의 개념도

```
/* 체크포인트 파일명 명시 */
void STFT_SET_FILENAME(char* str) {
  OPTIONS.ckpt_filename = str ;
}

/* 체크포인트 전에 실행할 일 명시 */
void STFT_BEFORE_CKPT() {  }

/* 체크포인트 후에 실행할 일 명시 */
void STFT_AFTER_RESTART() {  }

/* SPC를 이용한 체크포인팅 기능 구현 */
void STFT_DO_CKPT()    {  do_ckpt();  }

/* SPC를 이용한 재시작 기능 구현 */
void STFTR_RESTART() {  restart();  }
```

그림 3 CKPT용 SPC 인터페이스 구현

코드이다. 이 작업은 CKPT의 사용 방법을 배우는 것과 약 10 라인의 소스 코드를 추가하는 것만을 요구하였다. 그러므로 다른 플랫폼으로 MPI 체크포인터를 이식하기 위해 새 SPC를 채용하는 것은 많은 노력을 요구하지는 않을 것으로 기대된다.

3.2 STFT 체크포인팅 라이브러리

"STFT 라이브러리"는 사용자기반 체크포인팅 기능을 지원하기 위해 "MPI_Ckpt" 함수를 제공한다. 사용자기반 체크포인팅은 사용자가 코드에 하나 이상의 MPI_Ckpt 함수를 추가하여 체크포인팅 지점을 명시하도록 한다. 그러므로 사용자가 STFT가 제공하는 체크포인팅 기능을 사용하기 위해서는 MPI_Ckpt 함수를 코드에 추가하고, 작성된 코드를 STFT 라이브러리와 링크하여야 한다. MPI_Ckpt함수가 호출되면 프로세스는 SPC에서 제공하는 기능을 활용하여 자신의 프로세스 이미지를 체크포인트 파일로 저장한다. 나중에 응용프로그램이 재시작되면 MPI_Ckpt가 호출된 바로 다음 위치에서부터 다시 시작된다.

STFT는 체크포인팅을 위해 모든 MPI 프로세스들을 동기화하며, 한 프로세스의 MPI_Ckpt 함수의 호출에 대해 다른 프로세스들도 해당하는 MPI_Ckpt 함수의 호출을 갖는 것을 가정한다. 그러므로 모든 프로세스들이 MPI_Ckpt를 호출하는 것은 하나의 체크포인트 버전을 생성하는 것이며, 이 버전 단위로 MPI 응용프로그램은 체크포인팅되고 재시작된다.

MPI 응용프로그램을 체크포인팅하기 위해서는 한 MPI 프로세스가 전송했지만, 다른 MPI 프로세스가 전송받지 못한 메시지들까지 저장해야한다. MPI 구현에 따라 내부적인 메시지 전송 프로토콜이 상이하기 때문에 메시지를 체크포인팅하는 것은 MPI의 구현에 종속적이다. Cocheck의 경우도 네트워크로 전송 중인 메시지를 체크포인팅하는 것에 대해 MPI 위에서 적절하게 처리할 수 없었기 때문에, tuMPI라는 MPI의 구현에 종속될 수밖에 없었다.

이러한 종속성을 제거하고 이식성을 향상시키기 위해 STFT는 (1) 사용자기반 체크포인팅 방법을 사용하고, (2) 응용프로그램에서 MPI_Ckpt 함수를 호출할 수 있는 위치를 제한하고, (3) 체크포인팅 후 프로세스들을 동기화하여 메시지 체크포인팅을 회피한다. 시스템수준 체크포인팅의 경우에는 체크포인팅 시점에 전송중인 임의의 시점에 체크포인팅이 이루어지므로 전송중인 메시지가 없다는 것을 보장할 수 없다. 그러나 사용자가 체크포인팅 시점을 결정한다면 전송중인 메시지가 없는 시점을 결정하는 것은 어려운 일이 아니다.

그림 4는 사용자기반 체크포인팅의 몇가지 예이다. STFT는 그림 4(a)와 (b)의 경우처럼 MPI_Send와 매칭하는 MPI_Recv가 체크포인팅 시점(MPI_Ckpt) 이전이나 이후에 함께 위치해 있을 것을 강제한다. 그리고 체크포인팅 후 모든 프로세스들을 동기화한다. 이를 통해 MPI_Ckpt 함수가 호출되었을 때에는 전송중인 메시지가 없음을 보장할 수 있고 STFT는 메시지 체크포인팅을 회피할 수 있다. 위의 조건이 만족되었을 때 전송중인 메시지가 없음을 보이기 위해 여기에서는 메시지를 송부하는 MPI_Send 함수가 체크포인팅 시점 이전에 있는 경우와, 체크포인팅 시점 이후에 있는 경우로 나누어 생각하자. 먼저 MPI_Send가 MPI_Ckpt 이전에 있는 그림 4(a)와 같은 경우는, 수신측에 해당하는 MPI_Recv가 MPI_Ckpt 이전에 있을 것을 STFT가 강제하므로, MPI_Send에 의해 송신된 메시지는 MPI_Recv에 의해 수신되고 체크포인팅 시점에서는 전송중인 메시지는 없음을 알 수 있다. 두번째로 MPI_Send가 MPI_Ckpt 이후에 있는 그림 4(b)와 같은 경우는, MPI_Send가 송부한 메시지가 상대편 MPI_Ckpt 이전에 도착할 가능성이 있으며, 이 경우 전송중인 메시지가 있다. STFT는 이러한 경우를 방지하기 위해 그림 5처럼 MPI_Ckpt의 끝에서 모든 MPI 프로세스들을 동기화한다. 이 경우 모든 프로세스들이 체크포인팅을 마친 후

MPI_Send가 호출되는 것이므로, 체크포인팅 시점에서 전송중인 메시지는 없음을 알 수 있다. 즉, MPI_Send가 MPI_Ckpt 이전에 있거나 혹은 이후에 있어도 STFT는 사용자의 적절한 체크포인팅 시점의 선택과, 체크포인팅 후 동기화를 통해 메시지 체크포인팅을 회피할 수 있다. 또한 MPI_Irecv, MPI_Isend와 같은 비동기전송인 경우에는 MPI_Ckpt를 호출하기 전에 사용자는 MPI_Wait 함수 등을 호출하여 전송중인 메시지가 없음 것을 보장해 주어야 한다.

그림 4(c), (d)의 경우 STFT는 올바른 체크포인팅을 보장하지 않는다. 그림 4(c)의 경우에는 MPI_Send가 체크포인팅 시점 이전에 위치하고 MPI_Recv가 체크포인팅 시점 이후에 위치하므로 전송중인 메시지가 없다는 것이 보장될 수 없다. 그림 4의 (d)의 경우에는 MPI_Send가 체크포인팅 시점 이후에 위치하고 MPI_Recv가 체크포인팅 시점 이전에 위치하므로 데드락 문제를 유발한다.

STFT 라이브러리의 체크포인팅 과정은 그림 5와 같다. STFT_BEFORE_CKPT 함수를 통해 체크포인트 전에 필요한 작업을 수행하고, STFT_SET_FILENAME 함수를 통해 체크포인트 파일명을 설정한 후, STFT_DO_CKPT 함수를 이용하여 SPC를 이용한 체크포인팅을 수행한다. 그런 후 호출되는 STFT_AFTER_RESTART 함수는 체크포인트 후에 필요한 작업을 수행하고, STFT_Barrier 함수는 내부적으로 MPI_Barrier 함수를 사용하여 모든 프로세스들을 동기화한다. STFT_Barrier 함수는 모든 프로세스들을 동기화하므로 프로그램 상에서 MPI_Ckpt는 MPI_Barrier 함수처럼 모든 프로세스들의 동기화가 가능한 지점에 위치해야 한다.

STFT가 사용자기반 체크포인팅 방법을 사용하고 MPI_Ckpt 호출 위치를 제한하는 것은 코드를 수정할 필요가 없는 시스템수준 체크포인팅에 비해 프로그래머에게 부담을 줄 수 있다. 그러나 STFT를 활용한 우리의 경험에 의하면 그 부담은 매우 작다. 우리는 NPB[20] 벤치마크 프로그램들에 MPI_Ckpt 함수를 삽입하였는데, 우리가 NPB 벤츠마크의 코드를 이해하고 있지 않음에도 불구하고, 큰 범위(coarse-grained)의 이터레이션(iteration) 끝이나, 통신단계(communication phase) 전 후에서 MPI_Ckpt 함수를 삽입할 수 있는 적합한 위치를 찾는 것은 어렵지 않았다. 또한 STFT의 체크포인팅 방법은 메시지를 체크포인팅할 필요가 없기 때문에 시스템수준 체크포인팅에 비해 체크포인트 파일의 크기가 작으며, 체크포인팅에 소요되는 시간이 짧다는 장점도 갖는다.

STFT는 사용자기반 체크포인팅 방법을 사용하고, MPI_Ckpt가 호출될 수 있는 위치를 제한하고, 체크포인팅 후 프로세스들을 동기화하여, 메시지 체크포인팅을 회피한다. STFT 체크포인팅 프로토콜은 SPC 인터페이스와 다른 프로세스들과의 동기화를 위한 MPI 함수들만을 사용하므로, MPI 구현에 종속되지 않는다.

```
void MPI_Ckpt() {
  ...
  STFT_BEFORE_CKPT();   /* 체크포인트 선작업 실행 */

  /* 체크포인트 파일이름 설정 */
  ....
  STFT_SET_FILENAME(_ck_file) ;

 /* SPC를 이용한 체크포인트 실행 */
  STFT_DO_CKPT();

 /* 체크포인트 후작업 실행 */
  STFT_AFTER_RESTART();

  STFT_Barrier();
}
```

그림 5 STFT 라이브러리의 체크포인팅 과정

3.3 STFT 재시작 프로그램

SPC는 프로세스 이미지의 체크포인팅과 재시작을 지원할 뿐, 네트워크 연결에 대한 체크포인팅은 지원하지 않는다. "STFT 재시작 프로그램"은 MPI 프로그램의

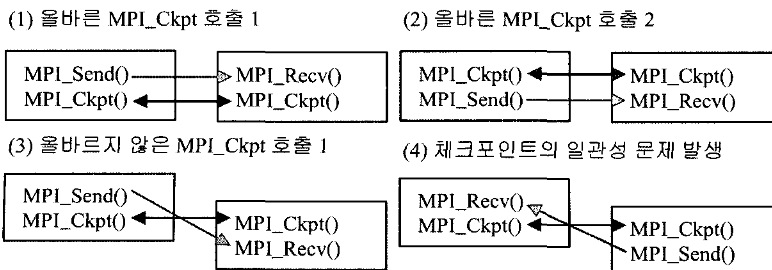

그림 4 사용자기반 체크포인팅의 예

네트워크 연결을 재생성하고, SPC를 활용하여 체크포인트 파일에 저장된 프로세스의 이미지를 복구하여 재시작하게 하는 역할을 한다.

MPI 프로세스들은 시작할 때 각 프로세스를 구분할 수 있는 랭크(rank)를 할당 받으며, 다른 랭크를 갖는 프로세스들과의 통신을 위한 네트워크 연결을 관리한다. MPI 응용프로그램의 실패는 프로세스들간 네트워크 연결이 종료됨을 의미하므로, 재시작하는 과정은 다른 랭크를 갖는 프로세스들과의 네트워크 연결을 재생성하는 과정을 포함한다. SPC에 의해 체크포인팅된 프로세스 이미지 파일은 실패 전의 랭크별 네트워크 연결 정보를 가지고 있으며, 올바른 네트워크 연결 재생성을 위해서는 새로운 네트워크 연결을 생성하고, 체크포인트 파일로부터 재시작된 프로세스에 새로 생성된 네트워크 연결 정보를 적용하여야 한다.

Cocheck의 경우 SPC를 통한 프로세스의 재시작 후 MPI_Init 함수를 호출하여 네트워크 연결을 재생성한다. 이 방법은 프로세스가 처음에 MPI_Init를 호출하였고, 재시작된 프로세스가 MPI_Init를 한 번 더 호출하는 것이지만, MPI 표준은 MPI_Init 함수가 여러 번 호출되는 것을 정의하지 않으며, MPI_Init 함수가 여러 번 호출되는 것을 허용하는 MPI 구현들이 없다. 이 때문에 Cocheck는 tuMPI라는 MPI의 구현을 수정하여 MPI_Init를 여러 번 호출할 수 있도록 하였고, 재시작 후 새로운 네트워크 연결을 생성하고 기존 프로세스가 가진 네트워크 연결 정보를 대체하여 올바른 네트워크 연결이 되도록 하였다. 이 때문에 Cocheck는 MPI 구현에 대한 종속이 불가피하였다. 다른 연구들[6-10]도 새로 생성된 네트워크 연결정보를 재시작된 프로세스에 적용하기 위해 특정 MPI 구현에 대한 의존이 불가피하였다.

이식성을 높이기 위해 STFT는 MPI 위에서 동작하는 네트워크 연결 재생성 방법을 제시한다. STFT는 먼저 MPI_Init 함수를 호출하여 네트워크 연결을 재생성한 후, SPC를 통해 체크포인트 파일로부터 프로세스를 재시작 시킨다. 그림 6은 이 방법이 적용된 STFT 재시작의 주요 과정을 보여준다. STFT의 재시작 프로그램은 MPI_Init 함수를 통해 재시작 프로그램들 사이에 네트워크 연결을 생성한다. 그 후 자신의 랭크를 파악하고 SPC의 재시작 기능을 이용하여 자신의 랭크에 해당하는 체크포인트 파일로부터 재시작한다.

STFT가 제시한 네트워크 연결 방법의 문제점은 재시작 프로세스가 만든 새로운 네트워크 연결 정보는 체크포인트 파일로부터 재시작된 프로세스 이미지에 의해 덮어 써져서 사라진다는 것이다. 이 문제에 대해 TCP 소켓기반의 MPI 구현의 예를 들어 보자. TCP 소켓기반 MPI 구현들은 다른 랭크를 가진 MPI 프로세스들과

```
int main(int argc, char *argv[]) {
  ....
  /* MPI_Init를 통한 네트워크 연결 재생성 */
  MPI_Init( &argc, &argv );

  /* 랭크를 알아내어 자신의 랭크에 해당하는 체크포
  인트 파일로부터 재시작 */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  /* 랭크에 따라 프로세스가 재시작될 체크포인트 파
  일명 설정 */
  ...
  STFT_SET_FILENAME(_ck_file) ;

  /* SPC 기능을 활용한 재시작 */
  STFT_RESTART();
  ....
}
```

그림 6 재시작 프로그램의 주요 과정

의 연결이 파일서술자(file descriptor)에 의해 이루어지며, 네크워크 연결정보는 바로 파일서술자 정보이며 랭크별로 관리된다. 파일 서술자는 운영체제에 의해 관리되는 프로세스별 파일 테이블(file table)에 대한 색인(index)이며, 프로세스는 네트워크 연결을 생성할 때마다 정해진 알고리즘에 의해 새로운 파일 서술자를 할당받는다. 이 때 세 MPI 프로세스가 있고, 랭크 0 프로세스가 MPI_Init에 의해 랭크 1과 네트워크 연결을 맺어 파일서술자 5번을 할당한 후, 랭크 2와 네트워크 연결을 맺어 파일서술자 6번을 할당했다고 하자. 이 MPI 프로그램이 실행 중 실패하였고, 재시작 프로그램이 MPI_Init를 호출했을 때 랭크 2와 먼저 네트워크 연결을 맺어 파일서술자 5번을 할당한 후, 랭크 1과 네트워크 연결을 맺어 파일서술자 6번을 할당한다고 하면, 운영체제에 있는 랭크 0 프로세스의 파일테이블은 파일서술자별로 이 정보를 유지하고 있을 것이다. 재시작 프로그램이 MPI_Init 호출하여 새로운 연결을 생성하고 SPC를 통해 프로세스를 재시작하면, 재시작된 랭크 0 프로세스는 재시작 프로그램이 새로 생성한 네크워크 연결 정보를 알지 못하므로 실패전 연결정보를 기반으로 랭크 1에게 메시지를 보내기 위해 파일서술자 5번을 사용하지만, 운영체제는 파일서술자 5번이 랭크 2라고 알고 있으므로, 메시지가 잘 못 전달되는 문제가 발생한다.

STFT가 이 문제를 해결하기 위한 전략은 MPI_Init가 다른 랭크를 가진 프로세스들과 연결을 생성할 때 항상 고정된 순서대로 연결을 생성하도록 강제하는 것이다. 위에서 들었던 예를 보면 파일 서술자가 나타내는 랭크가 실패전과 재시작 후에 일치한다면 올바르게 네트워크 연결 상태가 복구된 것으로 볼 수 있으며, 이 것

은 MPI_Init 함수가 항상 고정된 순서로 네트워크 연결을 생성함으로써 가능하다.

MPI의 구현에 따라서는 우리가 둔 제한을 만족시키기 위해 코드를 수정할 필요가 있을 수 있다. 그러나 우리의 경험에 의하면 대부분의 MPI 구현들[2,3]에서는 코드를 수정할 필요가 없다. 실제로 얼마만큼의 코드 수정이 필요한지를 알아보기 위해 우리는 가장 많이 사용되는 MPI 구현인 LAM[2]과 MPICH/P4[3]에 이 방법을 적용하였다. 첫 번째로 LAM MPI는 MPI_Init 함수가 호출될 때 모든 네트워크 연결이 생성되고, 다른 랭크를 가진 프로세스들과의 연결이 일정한 순서대로 생성된다. 그러므로 STFT에서 제안하는 방법은 LAM MPI의 구현에 아무런 수정 없이 적용 가능하다. 두 번째로 MPICH/P4는 MPI_Init가 호출되었을 때 네트워크 연결이 생성되지 않고, 메시지를 전송할 필요가 생길 때 실시간으로 네트워크 연결을 생성한다. 이 경우 고정된 순서대로 연결을 생성하지 않으므로 재시작 후 올바른 네트워크 연결이 보장되지 않는다. 이 문제를 해결하기 위해 우리는 MPI_Init 함수의 프로파일(profile) 인터페이스를 사용하여 MPI_Init 함수가 원래의 기능을 수행한 후 정해진 순서대로 다른 랭크들과 메시지를 주고받도록 하였다. MPI_Init 다음에 주고받는 메시지는 최초로 주고받는 메시지이고, 이 때 다른 랭크를 갖는 프로세스들과의 네트워크 연결이 정해진 순서에 의해 생성될 수 있다. 정해진 순서대로 다른 랭크들과 네트워크 연결을 생성하기 위한 MPICH/P4의 MPI_Init 프로파일 인터페이스 구현 내용은 그림 7과 같다. 그림 7에서 사용된 알고리즘은 자신보다 작은 랭크를 가진 프로세스들로부터 메시지를 받은 후에 자신보다 큰 랭크를 가진 프로세스들에게 메시지를 보내며, 가장 작은 랭크를 가진 프로세스를 시작으로 메시지를 전송하므로 항상 고정된 순서의 메시지 전송을 보장한다. 이 알고리즘은 최선의 알고리즘은 아니며 향후 보완이 필요하다.

STFT는 네트워크 연결의 재생성을 위해 MPI_Init가 다른 랭크를 가진 프로세스들과 연결을 생성할 때 항상 고정된 순서대로 연결을 생성하도록 하는 방법을 사용한다. 이 방법의 사용은 널리 사용되는 MPI 구현들[2,3]에 대한 수정을 요구하지 않으며, 수정이 필요하다 하더라도 기존의 MPI 체크포인터들이 사용하는 방법들에 비하면 그 수정되는 부분은 MPI_Init의 일부분으로 한정되므로 그 이식성이 크다. 본 논문에서는 TCP 소켓을 사용하는 환경에 대해서 주로 설명하였고, 다른 환경에 대해서는 STFT에서 제안하는 내용을 그대로 혹은 발전하여 적용할 수 있을 것이며, 이에 대해서는 추가적인 연구가 필요하다. TCP 소켓 환경만을 가정하였다 하더라도 현재 대다수의 클러스터나 그리드가 사용하는 MPI 구현들이 TCP 소켓 기반의 MPI 구현들이라는 점을 고려하면, STFT가 제공하는 이식성은 그 적용 범위가 작지 않으므로 그 의의가 크다.

MPI 프로그램에서 MPI_Init 함수가 호출되기 이전에 "파일 열기"처럼 프로그램의 외적인 상태를 바꾸는 사건이 있다면 올바른 네트워크 연결이 복구되지 않을 수 있다. MPI 표준은 MPI_Init 함수 호출 이전에 프로그램이 할 수 있는 일에 대해 명시하지 않고, MPI 구현들[2,3]에서는 MPI_Init 함수 호출 이전에 프로그램의 외적인 상태를 바꾸는 것을 피할 것을 권장한다. 그러므로 우리는 MPI 프로그램이 MPI_Init 함수를 호출하기 전에 프로그램의 외적인 상태를 바꾸는 것을 고려하지 않았다.

4. STFT 프로토타입 구현 환경 및 성능 분석

STFT 프로토타입 시스템은 리눅스 운영체제와 CKPT 환경에서 구현되었다. 현재 많은 클러스터들이 개방형 운영체제인 리눅스를 활용하고 있기 때문에 리눅스가 선택되었고, CKPT는 리눅스 플랫폼에서 안정적으로 동작하며 무료로 사용가능하기 때문에 선택되었다. STFT

```
int MPI_Init(int *pargc, char ***pargv){
  ...
  PMPI_Init(pargc,pargv);
  MPI_Comm_size( MPI_COMM_WORLD, & mysize );
  MPI_Comm_rank( MPI_COMM_WORLD, & myrank );
  for ( i = 0 ; i < myrank ; i++)
    MPI_Recv ( & aaa, 1, MPI_INT, i, MYTAG, MPI_COMM_WORLD, &stat );
  for ( i = myrank + 1 ; i < mysize ; i++)
    MPI_Send ( & aaa, 1, MPI_INT, i, MYTAG, MPI_COMM_WORLD);
  ...
}
```

그림 7 정해진 순서대로 다른 랭크들과 네트워크 연결을 생성하는 MPI_Init 프로파일 인터페이스

프로토타입 시스템은 STFT 체크포인팅 라이브러리, CKPT 체크포인팅 라이브러리, MPICH/P4용 라이브러리(고정된 순서의 네트워크 연결 생성), STFT 재시작 프로그램으로 구성된다.

STFT는 가장 널리 사용되는 MPI 구현들인 MPICH/P4와 LAM MPI 구현들로 이식되었다. 3장에서 언급하였던 대로 MPICH/P4와 LAM MPI 구현들로 이식하기 하기 위해 MPI 구현들에 대한 수정은 필요하지 않았다. 3.3절에서 언급한 고정된 순서로 네트워크 연결을 생성하는 코드 부분을 제외하면, MPICH/P4와 LAM MPI로 이식하기 위한 STFT의 수정은 불필요하였다.

그림 5에서처럼 체크포인팅 과정의 STFT의 성능은 SPC의 체크포인팅 시간과 한번의 MPI_Barrier 함수 호출 시간에 의해서 결정된다. 그러므로 STFT 체크포인터의 성능은 SPC의 성능과 MPI 구현의 성능에 의존하며, 이는 전적으로 외부적인 요인들이다. 또한 얼마나 많은 MPI_Ckpt 함수를 코드에 추가하는 가에 따라 체크포인팅에 따른 부담(Overhead)이 결정되며, MPI_Ckpt 함수를 코드에 추가하지 않으면 부담은 전혀 없다. 그림 6에서처럼 재시작 과정도 MPI_Init 후 SPC의 도움만으로 체크포인팅을 진행한다. 그러므로 STFT 재시작 과정의 성능도 역시 거의 SPC에 의존한다. 결과적으로 STFT의 성능은 SPC의 성능에 의존하므로, STFT의 성능 향상을 위해서는 고성능의 SPC를 채용하여 사용해야 할 것이다.

5. 결론과 미래의 연구

GRID가 확산되고 다양한 플랫폼이 분산 대용량 병렬처리 시스템에 적용됨에 따라 이식성 높은 MPI 체크포인터에 대한 요구가 커지고 있다. 이러한 요구에 맞추어 본 논문에서는 MPI 체크포인팅 시스템인 STFT에서 이식성을 향상시키기 위한 설계 및 구현 이슈들에 대해 설명하였다. STFT는 MPI 체크포인터의 이식성을 향상시키기 위해 첫째, 기존의 SPC들에 대한 인터페이스를 제공한다. 둘째, STFT는 사용자기반 체크포인팅 방법을 사용하며, MPI_Ckpt 함수가 호출될 수 있는 위치를 제한하여 메시지 체크포인팅을 회피한다. 셋째로 MPI_Init 함수가 고정된 순서로 네트워크 연결을 생성하도록 강제한다. 이를 통해 STFT는 다양한 플랫폼과 MPI 구현들로 쉽게 이식 가능할 것으로 기대한다.

본 논문의 내용은 쉽게 이식 가능한 MPI 체크포인터에 대한 것이다. STFT는 MPI 체크포인터 사용자가 다양한 플랫폼과 MPI 구현을 선택하는 것을 가능하게 하며, 그 효용성은 GRID와 같이 다양한 플랫폼을 활용하는 환경에서는 더욱 증가될 것으로 기대된다. 또한 PVM 등 다른 병렬프로그램 환경의 체크포인터들에도 동일한 아이디어의 적용이 가능할 것으로 기대한다.

참 고 문 헌

[1] MPI Forum, "MPI: A message-passing interface standard," International Journal of Supercomputer Applications, 8(3/4):pp.165-414, 1994.

[2] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," In Proc. Of Supercomp. Symp., 1994.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," Parallel computing, Vol. 22, No. 6, pp.789-828, Sep 1996.

[4] MPI Software Technology, Inc., "MPI/Pro," http://mpi-softtech.com/, 1999.

[5] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," Proc. Of the International Parallel Processing Symposium, IEEE Computer Soc. Press, pp.526-531, 1996.

[6] A. Agbaria, and R. Friedman, "Starfish: Fault-tolerant Dynamic MPI programs on cluster of workstations," Eighth IEEE International Symposium on High Performance Distributed Computing, 1999.

[7] Y. Chen, J. S. Plank, and Kai Li, "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs," Proceedings of the ACM/IEEE conference on Supercomputing, 1997.

[8] George Bosilca, et al., "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," In Proceedings of SC2002. IEEE, 2002.

[9] Sriram Lorenzo Alvisi, and Harrick M., "Egida: An Extensible Toolkit For Low-overhead Fault-Tolerance," Symposium on Fault-Tolerant Computing, 1999.

[10] Rajanikanth Batchu, et al., "MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing," 1st International Symposium on Cluster Computing and the Grid, 2001.

[11] Ian Foster, and Carl Kesselman, The Grid: Blueprint for a New Computing Infrastructure, MK Publications, 1999.

[12] T. Tannenbaum, and M. Litzkow, "Checkpointing and migration of Unix processes in the Condor distributed system," D. Dobbs Journal, pp.40-48, Feb. 1995.

[13] K. M. Chandy, and L. Lamport, "Distributed snapshots: Determining global states of distributed system," ACM Trans. On Computer Systems, 3(1):pp.63-75, Feb. 1985.

[14] J.S. Plank, "Efficient Checkpointing on MIMD Architectures," Ph.D. thesis, Princeton University,

June 1993.
[15] M, Hayden, "The Ensmble System," Doctoral dissertation, Cornell University, Dept. Computer Sciences, 1997.
[16] G.F. Fagg, and J.J. Dongara, "FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world," EuroPVM/MPI User's Group Meeting 2000, Springer-Verilag, pp.346-353, 2000.
[17] W. Gropp, S. Husss-Lederman, et al., "MPI-The Complete Reference, Vol-2, The MPI Extensions," ISBN, MIT Press, 1998.
[18] Victor C. Zandy, Barton P. Miller, and Miron Livny, "Process Hijacking," The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99), pp.177-184, August 1999.
[19] J. S. Plank, M. Beck, G. Kingsley, and K. Li., "Libckpt: Transparent Checkpointing under Unix,'' In Usenix Winter 1995 Technical Conference, pp.213-223, January, 1995.
[20] David Baile, et al., "The nas parallel benchmarks 2.0," Technical Report, NSA-95-020 Ames Research Center, December 1995.

안 선 일

1997년 전남대학교 전산학과(학사). 1999년 서울대학교 전산과학과(석사). 1999년~현재 서울대학교 컴퓨터공학부 (박사과정). 2000년~2004년 (주)클루닉스. 2004년~2005년 정보통신연구진흥원. 2005년~현재 한국과학기술정보연구원 연구원

한 상 영

1972년 서울대학교 응용수학과(학사). 1977년 서울대학교 전산과학과(석사). 1977년~1978년 울산대학교 공과대학 전임강사. 1983년 미국 텍사스오스틴 전산학과(박사). 1984년~현재 서울대학교 컴퓨터공학부 교수