동기 병렬연산을 위한 응용수준의 결함 내성 연산시스템

An Application-Level Fault Tolerant System For Synchronous Parallel Computation

박 필 성*
Pil-Seong Park

요 약

대규모 병렬 시스템의 MTBF(mean time between failures)는 아주 짧아 겨우 수 시간 단위에 불과하여 장시간의 연산 도중 연산 실패로 끝나 소중한 계산 시간이 낭비되는 경우가 많다. 그러나 현재의 MPI(Message Passing Interface) 표준은 이에 대한 대안을 제시하지 않고 있다. 본 논문에서는, 비표준의 결함 내성 MPI 라이브러리가 아닌 MPI 표준 함수들만을 사용하여, 일반적인 동기 병렬 연산에 적용할 수 있는 응용 수준의 결함 내성 연산 시스템을 제안한다.

Abstract

An MTBF(mean time between failures) of large scale parallel systems is known to be only an order of several hours, and large computations sometimes result in a waste of huge amount of CPU time. However, the MPI(Message Passing Interface), a de facto standard for message passing parallel programming, suggests no possibility to handle such a problem. In this paper, we propose an application-level fault tolerant computation system, purely on the basis of the current MPI standard without using any non-standard fault tolerant MPI library, that can be used for general scientific synchronous parallel computation.

☞ Keyword : 결함 내성(fault-tolerant), MPI(Message Passing Interface), 동기 알고리즘(synchronous algorithm), 체크포인팅/롤백(checkpointing/rollback)

1. 서 론

Grand Challenge [1] 같은 문제는 연산에 수 천 개의 노드를 사용한다. 이런 시스템은 일부 노드의 장애가 자주 발생하며, MTBF(mean time between failures)는 수 시간에 불과하므로 결함 내성(fault tolerant) 시스템이 절실하다 [2,3,4].

그러나 메시지 패싱 프로그래밍의 표준인 MPI(Message Passing Interface) [5]에는 결함 내성에 대한 기준이 없어 비표준 결함 내성 MPI 라이브러리들이 개발되고 있으나([4,6] 참고), 이들은 이식성이 떨어지며, 일반적인 복구 전략을 사용하는 한편 수동적으로 재시작해야 하는 시스템도 있어 복구에 시간이 많이 걸린다.

본 논문은 [7]에서 제안한 MASTER-SLAVE -BACKUP 연산 구조를 널리 사용하는 동기 병렬 연산에도 적용이 가능하도록 수정하고, 어느 프로세스의 장애가 전체의 교착을 가져오는 문제의 해결책을 제시한다. 한편 재시작에 필수적인 데이터만 체크포인팅하며, 여분의 BACKUP 프로세스가 대기하다가 어느 프로세스의 장애가 파악되는 즉시 그 역할을 떠맡아 연산을 재시작하되 다른 프로세스들은 단순히 이전의 값으로 롤백함으로써 전체적으로 다시 연산을 계속할 수 있는 응용 수준의 결함 내성 연산 시스템을 제안한다.

2. 관련 연구

2.1 MPI와 결함 내성 MPI 라이브러리들

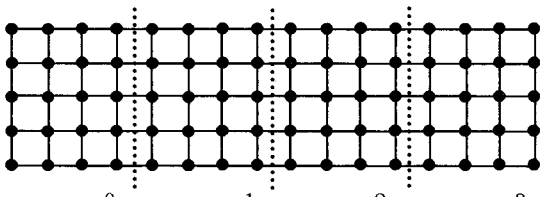
* 종신회원 : 수원대학교 컴퓨터학과 교수
pspark@suwon.ac.kr
[2008/07/07 투고 - 2008/07/09 심사 - 2008/07/21 심사완료]

MPI는 메시지 패싱 라이브러리의 표준이나 결함 내성 연산을 지원하지 않아, 다양한 결함 내성형 MPI 라이브러리가 개발되고 있다. 이들은 체크포인트/롤백(checkpoint/rollback)을 사용하는 것과 메시지나 프로세스의 복제를 사용하는 두 종류로 나눌 수 있다.

Co-Check MPI, MPI-CH V, LA-MPI가 전자에 속하며, MPI/FT 및 MPI-FT는 후자의 예이다. 그러나 비교적 적은 오버헤드를 가지는 FT-MPI라도 자동으로 복구되지 않고 사용자가 응용 프로그램 내에서 구현하여야 한다 [6,8]. 한편 BLCR (Berkeley Lab Checkpoint/Restart) [9]와 같은 오픈 소스 체크포인팅 유틸리티들이 있으나, 적용가능한 운영체제 및 MPI는 제한적이다 [10].

2.2 격자 문제와 동기 병렬 알고리즘의 구조

공학 및 자연과학에서는 FDM(finite difference method)과 같은 기법을 적용하여 격자점의 값을 구하는 수치 모델을 많이 사용한다. 예를 들어, 그림 1은 영역 분할하여 4개의 프로세스를 사용하는 경우, 각 프로세스가 계산할 격자점들을 나타낸다. 점선 좌우 격자점의 값(경계치)은 인접 프로세스의 연산에 필요하므로 메시지 전달을 통해 서로 교환해야 한다.



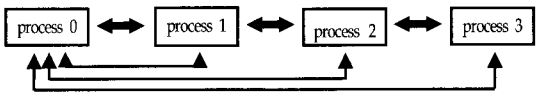
(그림 1) 각 프로세스가 담당할 격자점들

이런 연산을 수행하는 동기 병렬 알고리즘 구조는 Algorithm 1과 같다.

Algorithm 1

1. 초기 해 벡터를 적절히 선택한다.
2. 수렴할 때까지 다음을 반복한다.
   1) 각 프로세스는 담당 영역 내의 모든 격자점의 값을 순차적으로 업데이트한다.
   2) 좌우측 프로세스와 경계치를 교환한다.
   3) 수렴 판단에 필요한 부분 정보를 각기 계산하여 Process 0에게 보낸다.
   4) Process 0은 수렴여부를 판단하고 연산의 계속 여부를 다른 프로세스에게 알린다.

Algorithm 1은 2.1)에서 대부분의 시간을 소모한다. 그림 2처럼 4개의 프로세스를 사용하면, 경계치 교환 2.2)는 굵은 화살표로, 컨트롤 정보 교환 2.3) 및 2.4)는 가는 화살표 경로로 이루어진다. 동기화는 매 단계 수행되므로 한 프로세스의 장애는 전체를 교착상태(deadlock)에 빠뜨린다.



(그림 2) 메시지 패싱 패턴

2.3 이전의 비동기 연산 구조의 문제점

[7]에서는 비동기 연산 [11]을 도입, 결함내성 시스템을 제안하였는데 다음과 같은 단점이 있다.

- 비동기 연산을 적용할 수 없는 문제가 많다.
- SLAVE들간의 데이터 교환(Algorithm 1의 2.2)은 모두 MASTER를 통해 수행하며, MASTER의 장애에 대비하여 BACKUP MASTER도 같은 데이터를 수신하므로 네트워크 부하가 크다.
- 두 MASTER가 각기 모든 SLAVE를 담당하므로 부담이 크다.
- 별도의 BACKUP MASTER와 BACKUP SLAVE를 둔다.

2.4 체크포인팅 기반의 롤백 복구

병렬 연산의 체크포인팅 기법은 uncoordinated checkpointing(UC), coordinated checkpointing(CC), communication-induced checkpointing (CIC)로 분류된다 [4]. UC는 각 프로세스가 독립적으로 체크포인팅을 수행하며, 상태 정보량이 적을 때 수행해 효율적일 수도 있으나, 도미노 효과(domino effect) [12]로 작업 결과를 잃을 수도 있다. CC는 국지적 체크포인팅이 전체 체크포인팅에 모순이 없게 조정되므로 도미노 효과 같은 부작용이 없고 단 하나의 체크포인트만 유지하므로 복구가 간단하나 대기시간이 크다. CIC는 다른 프로세스로부터 받은 정보를 바탕으로 각 프로세스가 개별적으로 로컬 체크포인팅을 하며, 시스템 전체의 일관 상태 유지를 위해 때때로 추가적인 강제적 체크포인팅을 수행해 도미노 효과를 방지한다.

3. 동기 연산을 위한 전략

3.1 고려할 문제들

1) 교착의 원인

MPI는 송수신이 쌍을 이루어 전송이 되도록 할 뿐, 성공적으로 완료될 것을 가정하고 TCP/IP에 완전히 맡겨 버린다. 따라서 동기 병렬 알고리즘에 있어서, 어느 프로세스의 장애는 상대 프로세스의 송수신 함수를 완료할 수 없도록 하여 교착 상태에 빠뜨리고, 그 효과는 순식간에 파급되어 전체가 교착 상태에 빠지게 된다.

2) 메시지 송수신의 비동기화

동기 연산은 수시로 동기화하며 일정 순서로 같은 수의 반복을 수행하나, 비동기 연산은 프로세스의 성능에 따라 다른 회수의 반복을 허용한다. 비동기 연산은 점차 응용분야가 넓어지고 있으나 모든 문제에 적용되지는 않는다 [11].

비록 동기 연산이 필수적이라도 메시지 전송과 같이 연산 결과에 영향이 없는 작업은, 교착을 피하기 위해 비동기적 요소를 가미할 필요가 있다.

3) 체크포인팅/재시작 방법

자연과학 문제의 경우, 재시작에 필요한 데이터는 비교적 적다. 보통의 체크포인팅/재시작의 경우, 장애 발생 프로세스 뿐 아니라 모든 프로세스를 재시작한다. 그러나 본 논문에서는, 장애 발생 프로세스는 BACKUP 프로세스가 맡아 재시작하더라도 다른 프로세스들은 재시작하지 않고 이전 값으로 롤백하여 연산을 계속하도록 한다.

3.2 동기연산을 위한 해결책

1) 교착 방지 및 복구 도구

비블로킹(non-blocking) MPI 함수는 송수신이 백그라운드에서 이루어지므로 다른 작업을 계속할 수 있고, 통신 요구 핸들(communication request handle)로 완료 여부를 확인가능하며 미완의 송수신을 취소할 수도 있다. 핵심적으로 사용할 MPI 함수 및 주요 인자는 다음과 같다.

- MPI_Irecv(sender, data, &request)
  sender에게 수신 준비를 알리고 data가 전송되면 수신한다. request는 통신 요구 핸들이다.
- MPI_Isend(receiver, data, &request)
  receiver에게 data를 송신하는 함수이다.
- MPI_Iprobe(source, &status)
  source로부터의 메시지가 있는지 확인한다. source로 MPI_ANY_SOURCE를 사용하면 임의의 소스로부터의 메시지를 확인할 수 있다. status로 프로세스 및 메시지 종류를 파악할 수 있다.
- MPI_Testall(count,&request[],&flag, &status[])
  이전의 count개의 request[]와 관련된 모든 송수신의 완료 여부를 확인한다.
- MPI_Cancel(request)
  request와 관련된 미완의 송수신을 취소함.

교착 방지를 위해, 매 송수신 직후 일정 시간 간격으로 송수신 완료를 반복하여 확인하고, 실패하면 관련 송수신을 취소할 수 있어야 한다.

Check_comm()은 Wtime 간격으로 MaxTry 회수만큼 req[]와 관련된 count개의 송수신의 완료를 확인하고, 실패가 발견되면 장애 발생 프로세스의 랭크를 반환한다.

```
Check_comm(count,req[],MaxTry,Wtime) {
 FailedNode=99999;
 for i=1 to ,MaxTry, {
   MPI_Testall(count, req[], flag, status[]);
   if (flag)  return 7777;
      else  sleep Wtime; }
 Identify FailedNode using status[];
 Cancel uncleared communication by MPI_Cancel();
 return FailedNode;
}
```

Sure_send()는 메시지를 보내고 Check_comm()으로 전송 완료를 확인하며, 만일 실패하면 상대 프로세스의 번호를 알아낸다. 이와 유사하게 수신을 위한 Sure_recv()도 있으나 생략하기로 한다.

```
Sure_send(receiver,data,MaxTry,Wtime) {
  FailedNode=7777;   /* some number */
  MPI_Isend(receiver,data,,&req)
  FailedNode=Check_comm(1,req,MaxTry,
                        Wtime);
  return FailedNode;
}
```

만일 송수신에 실패하면 상대 프로세스 번호를 MASTER에게 보고하고 복구를 위한 지시를 받는데, 다음과 같이 MPI_Iprobe()를 사용하여 확인 후 수신한다.

```
flag=FALSE;
do {
  MPI_Iprobe(MPI_ANY_SOURCE,flag,&status); } while
(!flag);
Identify sender(status.MPI_SOURCE.)
   and receive command for recovery.
```

2) 체크포인팅을 위한 메시지 전송

SLAVE들의 계산치를 체크포인팅하며, 프로세스의 장애를 파악해 전체 연산이 롤백하여 재시작하는 컨트롤러로서 MASTER를 별도로 둔다. MASTER는 SLAVE로부터 전송되어 오는 순서대로 데이터를 수신함으로써 교착도 피하고 대기시간도 약간 감소시킬 수 있다.

```
Probe_and_recv(count,data,MaxTry,Wtime) {
  received=0;
  Set all members of success[] to FALSE;
  for i=1 to MaxTry,
    for j=1 to count, {
      MPI_Iprobe(ANY_SRC,ANY_DATA,
                 flag,&status);
      if (flag) {
         sender = status.MPI_SOURCE ;
         msgkind = status.MPI_TAG;
         MPI_Irecv(sender, data);
         success[sender]=TRUE;
         received++;
         if (received==count) return 0;  }
    sleep Wtime; }
  Identify which slave did not send data.
  Return the failed slave's rank.
}
```

Probe_and_recv()는 count개의 SLAVE로부터 무작위로 수신하며, 실패시 WTime 시간 간격으로 MaxTry 회수만큼 시도하나, 여전히 실패하면 장애 발생 SLAVE의 랭크를 반환한다.

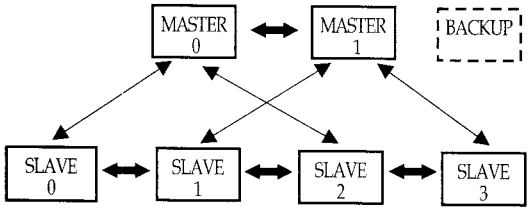
4. 결함 내성 동기 연산 시스템

4.1 연산 시스템의 구조

본 연구는 어느 순간 단 하나의 프로세스에게만 장애가 발생하는 것을 가정하며, 그림 3과 같은 MASTER-SLAVE-BACKUP 모델을 제안한다.

SLAVE들은 그림 2의 프로세스와 역할이 거의 같아 대부분의 연산을 수행하되, 주기적으로 스스로 체크포인팅하며, 또한 자신에게 장애가 발생하면 다른 프로세스가 역할을 대신하도록 MASTER

에게도 보내 체크포인트하도록 한다.



(그림 3) dual MASTER-SLAVE-BACKUP 모델

요약하면, SLAVE끼리 직접 메시지를 교환하고 두 개의 MASTER를 사용하여 MASTER의 부하를 줄이고, BACKUP이 MASTER나 SLAVE의 역할을 떠맡을 수 있으며, 엄격히 동기화하여 연산을 수행한다는 점이 [7]과는 다르다.

4.2 각 프로세스의 역할

이하 다음의 약어를 사용하기로 한다.

- GV(grid value) : 격자점에서의 값
- BV(boundary value) : Algorithm 1의 2.2)에서 SLAVE들 간에 교환할 경계치
- CI(convergence info) : Algorithm 1의 2.3)에서 계산된 수렴판단을 위한 (부분) 정보
- GO_ON, STOP : MASTER가 SLAVE들에게 계산을 계속/중지하라는 명령
- ERR_PRCS : 장애가 발생한 프로세스 번호
- LR(left and right), MY 같은 접두어는 의미를 명확히 하기위해 사용한다.

Algorithm 1의 2.2) 및 2.4) 단계 직후 동기화가 이루어지며, MASTER는 모든 SLAVE가 해당 메시지를 보내오는지 아닌지로 장애를 판단한다.

SLAVE

1) SLAVE간의 BV 교환은 당사자간 직접 수행한다(그림 3의 굵은 화살표).
2) 전체 연산의 수렴판정에 필요한 CI를 계산하여 MY_MASTER에게 보내고 연산의 계속 여부에 대한 지시를 받는다.
3) 주기적으로 별도 배열에 자신의 최신 값(GV)을 복사하며 MY_MASTER에게도 보낸다.
4) 상대의 장애를 발견하면 MY_MASTER에게 즉각 보고하고 지시를 기다린다.

MASTER

1) MY_SLAVE들로부터 CI를 받아, MASTER끼리 그 값을 교환하여 수렴 여부를 판단하고, 계속 여부를 MY_SLAVE들에게 알린다.
2) MY_SLAVE들로부터 정기적으로 GV를 수신하고 MASTER끼리 교환하여 각기 완전한 모든 SLAVE들의 GV를 유지 체크포인팅한다.
3) SLAVE의 장애가 파악되면, 다른 MASTER와 BACKUP에게 알려 장애 SLAVE의 역할을 대신토록 한다. MASTER는 MY_SLAVE에게 ERR_PRCS를 알려 체크포인팅 된 값으로 롤백하고 메시지 전달 구조를 바꾼다.
4) 다른 MASTER가 장애에 빠지면 BACKUP이 그 역할을 대신하도록 지시한다.

BACKUP

1) 장애 발생 SLAVE 또는 MASTER를 대신하기 위하여 두 MASTER로부터 오는 메시지가 있는지 대기하며 계속 Probe_and_recv()한다.
2) MASTER의 지시에 따라 장애가 발생한 SLAVE 또는 MASTER의 역할을 수행한다.

MASTER는 SLAVE의 장애를 감시하며 Probe_and_recv() 방식으로 SLAVE로부터 CI와 BV를 받는다. BACKUP이 SLAVE 역할을 대신하려면 MASTER로부터 GV를 받아야 하며, MASTER의 역할을 대신할 경우는 추가적으로 자신이 담당할 SLAVE들의 랭크도 얻어야 한다. SLAVE에 장애가 발생하면 모든 SLAVE가 롤백해야 하나, MASTER는 장애가 발생하더라도 SLAVE들은 롤백할 필요가 없고 그 사실만 알도록 하면 된다.

어느 프로세스든 장애가 발생하면 메시지 전달구조가 바뀌게 되는데, 이는 상대 프로세스의 랭크를 변경하면 해결된다.

Algorithm 2 (SLAVE)

Repeat until MY_MASTER sends STOP,
1. 1) MPI_Irecv() LR_BV from SLAVEs.
   2) Update MY_GV. (Alg. 1 step 2.1))
   3) MPI_Isend() MY_BV to SLAVEs.
   4) Check if al send/recv are cleared.
      If not, start SRP#1 and continue.
2. At every n-th turn, /* checkpointing*/
   1) Checkpoint my current state.
   2) Sure_send() MY_GV to
      MY_MASTER for checkpointing.
      If failed, start SRP#2.
3. Compute MY_CI.
4. Sure_send() MY_CI to MY_MASTER.
   If failed, start SRP#2.
5. Probe_and_recv() a signal(GO_ON or STOP) from MY_MASTER.

SLAVE's Recovery procedure #1(SRP#1)

1. 1) Cancel the failed communication.
   2) Identify the rank of the failed process.
   3) Report the rank to MY_MASTER.
2. 1) Get a message from MY_MASTER.
   2) Change message passing structure,
      if necessary.
3. Roll back using my checkpointed data.

SLAVE's Recovery procedure #2(SRP#2)

1. Take BACKUP for MY_MASTER.
2. Sure_send() the data again to MY_MASTER.

Algorithm 3 (MASTER)

Repeat until satisfactory,
1. 1) Probe_and_recv() CI from SLAVEs.
      If some SLAVE does not send CI
      or some SLAVE sends ERR_PRCS,
      start MRP#1.
   2) Exchange CI with other MASTER.
      If other MASTER does not respond,
      start MRP#2.
   3) Check convergence and decide whether to GO_ON or STOP.
   4) Send the signal to all MY_SLAVEs.
2. At every n-th turn, /* checkpointing */
   1) Probe_and_recv() GV from SLAVEs.
      If some SLAVE does not send GV,
      start MRP#1.
   2) Exchange GV with other MASTER.
      If other MASTER does not respond,
      start MRP#2.

MASTER's Recovery procedure #1(MRP#1)

1. Decide which SLAVE failed.
2. Send the info to all MY_SLAVES and other MASTER to start recovery.

MASTER's Recovery procedure #2(MRP#2)

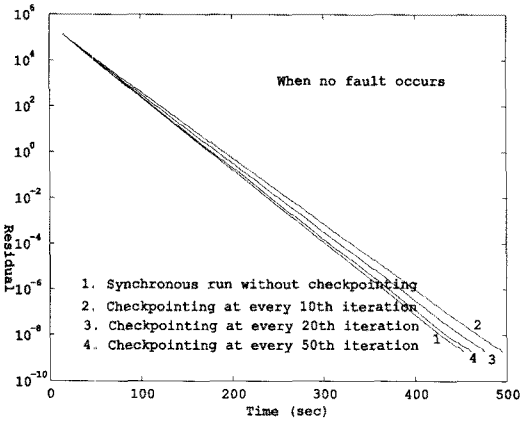
1. Direct BACKUP to take over the role of other MASTER.

Algorithm 4 (BACKUP)

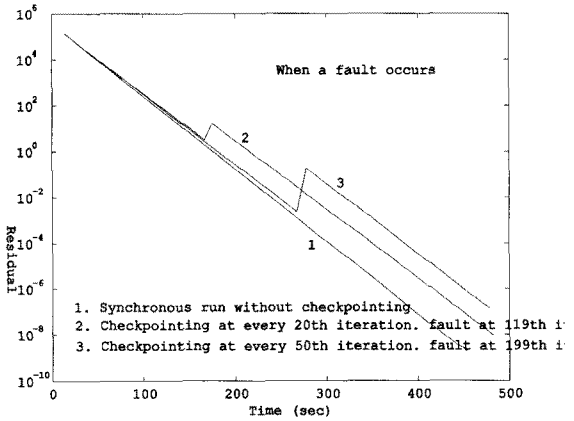
1. Probe_and_recv() a signal.
2. If the signal says to take over some SLAVE's role,
   1) Get the SLAVE's GV from MASTER.
   2) Start working as the SLAVE.
3. If the signal says to take over some MASTER's role,
   1) Get all SLAVE's GV from MASTER.
   2) Start working as one MASTER.

5. 성능 평가

[7]과 유사한 3,000×3,000의 2차원 격자 문제를, 수원대학교의 Hydra 클러스터의 7개 노드를 사용하여 성능 실험하였다. 노드의 장애는, Algorithm 1을 지정된 수만큼 반복한 후 무한 루프를 돌게 하여 시뮬레이션하였다.



(그림 4) 장애가 발생하지 않는 경우의 수렴 패턴



(그림 5) 장애가 발생하는 경우의 수렴 패턴

그림 4는 장애가 발생하지 않는 경우의 수렴 패턴이다. 1은 보통의 동기 알고리즘이며, 2, 3, 4는 각기 매 10회, 20회 및 50회 반복마다 체크포인팅을 수행한 것이다. 표 1은 5회 시행의 평균으로, 체크포인팅 당 약 1.2-1.4초가 소요되었다. 물론 이 시간은 Check_comm() 같은 확실한 전송을 위한 다양한 조처를 포함하나, 이들은 거의 추가적 시간을 소모하지 않음을 짐작케 한다.

(표 1) 장애가 발생치 않는 경우의 체크포인팅

		체크포인팅 수	시간(초)	체크포인팅당 소요된 시간(초)
체크포인팅 없는 동기연산		0	452.6	-
체크포인팅 간격	10 반복당	35	495.6	1.23
	20 반복당	17	475.8	1.35
	50 반복당	7	460.6	1.28

NetPIPE v.3.6.2 [13]으로 측정한 결과, MB 단위 패킷의 전송속도는 대략 600Mbps 정도로, MASTER가 체크포인팅을 위해 수신하는 데이터는 $9\times10^6\times8$bytes=72MB(배정밀도 연산)이므로 표 1의 결과에서 유추된 값과 유사하다 하겠다.

Wtime은 정상 상태에서 Algorithm 1의 1회 반복 시행시간의 평균으로 택하였는데, 대략 1.25초 정도로 추정되었다.

그림 5는 SLAVE 2에 장애가 발생한 경우이다. 2와 3은 각기 20/50번의 반복마다 체크포인팅하고 119/199번째 반복에서 장애가 발생(따라서 각기 100번째와 150번째 반복의 연산결과로 롤백)한 경우를 나타낸다.(MaxTry는 3으로 설정)

체크포인팅 비용을 C, 장애발생률을 m이라 하고 포아송(Poisson) 프로세스를 따른다면 최적 체크포인팅 간격은 $T_{opt}=\sqrt{2C/m}$로 주어진다 [12]. 본 실험에서는, 2의 경우처럼 자주 체크포인팅하는 것이 더 나은 결과를 보였는데 이는 체크포인팅 비용이 비교적 낮음을 의미한다 하겠다.

(표 2) 연산 재시작까지 소요된 평균 시간

	SLAVE 장애시			MASTER 장애시		
MaxTry	1	3	5	1	3	5
Wtime	1.25 sec					
복구 시간(초)	3.12	5.72	7.96	4.37	6.64	9.06

표 2는 매 20회 반복마다 체크포인팅 할 경우, SLAVE 2 또는 MASTER 1의 장애를 파악하여 연산 재시작까지 소요된 시간을 나타낸다(5회 평균). 당연히 MaxTry가 증가하면 복구에 걸리는 시간은 길어진다. 그러나 MaxTry*Wtime에 비해, SLAVE의 장애시는 대체로 1.7-2.3초, MASTER의 장애시는 2.2-3.1초 정도 더 소요되었는데, 이는 장애 프로세스를 파악하고 BACKUP을 기동하며

다른 프로세스들을 롤백시키는 데 걸린 시간이다. 한편 MASTER의 장애시는 시간이 약간 더 걸렸는데, 이는 BACKUP이 모든 SLAVE들의 GV를 다른 MASTER로부터 전송받고 나서야 새로운 MASTER로 역할을 시작하기 때문이다.

6. 결론 및 제안

본 논문에서는 표준 MPI 함수만을 사용하여 이식성 있는 결함 내성 연산 시스템을 제안하고 3,000×3,000의 2차원 격자 문제에 적용하였다. [7]과 다른 점은, MASTER-SLAVE-BACKUP 모델을 사용하되 두 개의 MASTER를 두어 부하를 분산시키고 더욱 안정적인 연산이 가능케 하였다.

처음에는 각 노드가 장애 노드를 파악하는 방법을 시도하였으나 실패하였다. 그 이유는, 장애가 발생하지 않았음에도 장애 노드와의 송수신 완료를 위해 시간을 소모하는 장애 노드의 인접 노드를 장애에 빠진 것으로 주변 노드들이 잘못 판단하여 실제 어느 노드에 문제가 생겼는지 판단하기 곤란하였다. 따라서 장애 발생 노드의 판단은 MASTER만이 수행하도록 하였다.

Hydra 클러스터에는 별도의 복구 메커니즘이 없어 비교해 볼 결과가 없으나, 2,677,324 크기의 conjugate gradient 해법을 16개의 프로세스로 계산할 경우 69초의 복구시간(전체 연산 시간의 6%)이 소요되었다고 보고된 예가 있다 [6]. 이에 비해 제안한 방법은 MaxTry를 5로 택한 경우, 복구시간은 전체 시간의 2% 정도로 빠른 복구가 이루어지나, 메커니즘이 복잡한 것이 단점이다.

MaxTry를 줄이면 복구시간도 감소하겠으나, 일시적 지체를 장애로 판정할 수도 있어, 최적치의 선택은 아직 미해결의 문제이다.

본 논문에서 장애 프로세스는 단순히 병렬 연산에서 제외하고 두 번째 장애가 발생할 경우는 고려치 않았다. 물론 BACKUP 프로세스를 여러 개 대기시켜 해결할 수도 있으나 MPI-2의 MPI_Spawn() 같은 함수를 사용해 개선할 수 있을 것이다. 한편 전용 시스템이 아닌 경우, 노드의 일시적 부하로 인해 일정 시간 반응을 못하여 장애가 발생한 것으로 간주될 수도 있는데, 이 경우 단순히 그 프로세스를 연산에서 제외시킬 것이 아니라 BACKUP 프로세스 역할로 전환한다면 더욱 유연한 결함 내성 시스템이 될 것이다.

참 고 문 헌

[1] http://www.powua.com/wiki/index.php/ Grand_challenge_problem.

[2] L. Alvisi, "Scalable Fault Tolerance through Compiler-Driven Communication-Induced Check pointing" *in* CSRI 2003 Annual Report, Report SAND2006-0001, pp.13-16, 2006.

[3] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations", J. Supercomputer Applications, 15(3), 2001.

[4] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The LAM/MPI Checkpoint /Restart Framework: System-Initiated Checkpointing", International Journal of High Perfornamce Computing Applications, Vol.19, pp.479-493, 2005.

[5] MPI Forum. 1995. MPI: A Message- Passing Interface standard.

[6] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovsky, & J. J. Dongarra, "Fault tolerant communication library and applications for high performance computing", Proceedings of the Los Alamos Computer Science Institute Symposium 2003, Santa Fe, NM.,

[7] 박필성, "표준 MPI 환경에서의 무정지형 선형 시스템 해법", 한국인터넷정보학회 논문지 6(6):23-34, 2005.

[8] R. Subramaniyan, V. Aggarwal, A. Jacobs, and A. George, "FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems", Proc. of International Conference on Embedded Systems & Applications(ESA), Las Vegas, NV, June 26-29, 2006.

[9] http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml.

[10] http://www.lam-mpi.org/

[11] D. B. Szyld, "Different models of parallel asynchronous iterations with overlapping blocks," Computational and Applied Mathematics, Vol.17, pp.101-115, 1998.

[12] E. Elnozahy, D. Johnson and Y. M. Wang, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", TR CMU-CS-96-181, School of Computer Science, Carnegie-Mellon Univ., Oct. 1996.

[13] http://www.scl.ameslab.gov/Netpipe/

◐ 저 자 소 개 ◑

박 필 성(Pil-Seong Park)

1977년 서울대학교 해양학과(이학사)
1984년 미국 Old Dominion Univ. 대학원 계산 및 응용수학과(이학석사)
1991년 미국 Univ. of Maryland, 대학원 응용수학과(이학박사)
1978년~1982년 KIST 해양연구소 연구원
1991년~1995년 한국해양연구원 선임연구원(전산실장, 수치모델그룹장 역임)
1995~현재 수원대학교 컴퓨터학과 교수
관심분야 : 고성능 컴퓨팅, 병렬처리, 수치해석, etc.
E-mail : pspark@suwon.ac.kr