선형 시스템의 동기 병렬 연산을 위한 응용 수준의 무정지 연산 시스템

박필성*
*수원대학교 컴퓨터학과
e-mail:pspark@suwon..ac.kr

An Application-Level Fault Tolerant System For Synchronous Parallel Linear System Solver

Pil-Seong Park*
*Dept of Computer Science, University of Suwon

요 약

많은 수의 CPU를 사용해 오랜 시간 계산하는 초대형 연산의 경우, 일부 노드나 통신회선의 장애로 연산 실패를 종종 겪는데, 이를 위해 응용 수준의 무정지 연산 시스템의 구현이 중요하다. 본 논문에서는 비동기 알고리즘을 사용한 이전 시스템의 약점을 보완하여, 동기 알고리즘에도 적용가능한 새로운 응용수준의 무정지 연산 시스템을 제안하고 선형시스템의 해법에 적용하였다.

1. 서론

대규모 병렬 시스템을 이용한 연산은, 연산 도중 노드의 장애로 인한 연산 실패를 종종 겪으므로 사용자 입장에서는 응용 프로그램 수준의 무정지(fault tolerance) 연산 시스템의 구현이 중요하다 [4,5]. 그러나 병렬 프로그래밍의 표준인 MPI(Message Passing Interface) [3]는 이를 위한 대안을 제시하고 있지 않아, 비표준인 다양한 무정지형 MPI 라이브러리들이 개발되고 있다 [2,6]. 그러나 이들을 사용한 병렬 코드는 이식성이 떨어지며, 일반적인 복구 전략을 사용하므로 복구에 시간이 많이 걸린다.

[1]에서는, 매스터-슬레이브-백업 모델과 비동기 연산을 도입하여, MPI 표준함수만으로 무정지 선형시스템의 해법을 제안하였다. 그러나 이는 동기화가 필수적인 문제에는 직접적인 적용이 불가능하다는 단점이 있다 [7]. 본 논문에서는 이를 개선하여 동기화가 필요한 연산에서도 적용가능한 무정지 해법을 제시한다.

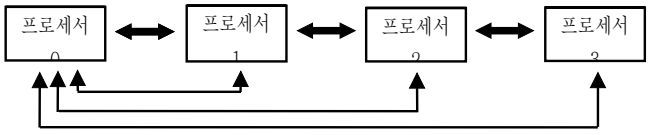
2. 일반적인 동기 연산 방식

자연과학 및 공학 문제에서는 종종 격자를 사용하여 격자점에서의 값을 구하는 연산을 수행하는데, 대략적인 동기 알고리즘의 구조는 다음과 같다.

Algorithm 1

1. 문제 및 초기치를 셋업 한다.
2. 만족스런 해가 얻어질 때까지 다음을 반복한다.
   1) 담당 영역의 격자점의 값을 1회 업데이트한다.
   2) 좌우측 프로세서와 경계치를 교환한다.
   3) 수렴 판단에 필요한 부분 정보를 각기 계산하여 대표 프로세서에게 보내고, 대표 프로세서는 이를 종합하여 연산의 계속 여부를 결정하고 다른 프로세서들에게 알린다.

이 알고리즘은 2.1)에서 대부분의 시간을 소모한다. 만일 4개의 프로세서를 사용하면, 개념적으로 (그림 2)와 같이 배열되며, 데이터 교환의 대부분을 차지하는 2.2)는 굵은 화살표의 경로로, 전체 컨트롤 관련 정보 교환 2.3)은 가는 화살표로 표시되어 있다.



(그림 1) 4개의 프로세서를 사용하는 경우의 메시지 패싱

이런 연산 구조는 동기화로 인하여 하나의 프로세서의 장애는 전체의 교착상태(deallock)를 야기한다.

3. 이전의 비동기적 연산 구조 및 문제점

[1]에서는, 매스터가 슬레이브들간의 데이터 전달을 중계하고 각 슬레이브의 장애 여부를 감시하며, 어느 슬레이브의 장애 발생시 백업 프로세서가 대신하여 연산을 수행토록 하였으나, 다음과 같은 가정 및 약점을 가지고 있다.

1) 매스터와 백업에는 장애가 발생하지 않는다.
2) 비동기적인 연산이 불가능한 문제도 존재한다.

3) 정상적인 경우에도 슬레이브간의 송수신은 반드시 매스터를 거치므로 시간이 많이 걸린다.

4. 교착상태 방지를 위한 전략

병렬 연산에서의 교착상태는 송수신이 쌍을 이루지 못하여 연산을 진행하지 못해 발생한다. 약간의 비동기성을 가미한 다음의 전략으로 교착상태를 피하고자 한다.

- 매스터는 슬레이브들로부터의 수신 순서를 미리 정하지 않고 전달된 순서대로 수신한다. 이는 대기시간의 감소에 따른 성능 향상 효과도 기대할 수 있다.
- 일정 시간이 지나도록 정해진 수의 송수신이 완료되지 않으면 그 송수신을 취소하고 상대 프로세서에게 장애가 발생한 것으로 간주하고 복구를 수행한다.

MPI의 블로킹(blocking) 함수는 상대에게 장애가 발생하면 교착상태에 빠지게 되나, 비블로킹(non-blocking) 함수는 송수신이 백그라운드에서 이루어지므로 다른 작업을 계속할 수 있고, 송수신 완료 여부는 통신 요청 핸들(communication request handle)로 확인할 수 있고 미완의 송수신을 취소할 수도 있다. 본 논문에서 핵심적으로 사용하는 MPI 함수는 다음과 같다.(실제 함수의 인수의 수는 이보다 많으나 중요한 것만을 나타내기로 한다).

- MPI_Irecv(sender, data, &request)
  sender에게 data의 수신 준비를 알리고 data가 전송되면 수신한다. 일찍 사용할수록 효율적이다.
- MPI_Isend(receiver, data, &request)
  receiver에게 data를 송신하는 비블로킹 함수이다.
- MPI_Iprobe(source, &status)
  어느 프로세서가 어떤 데이터를 전송해올 지 알 수 없는 경우, 이를 처리하기 위한 함수이다.
- MPI_Testall(count, &request[], &flag, &status[])
  이전의 count개의 request[]와 관련된 모든 송수신의 완료 여부를 확인한다.
- MPI_Cancel(request)
  request와 관련된 송수신을 취소한다.

한편 알고리즘의 각 주체 및 데이터는 다음과 같이 요약하도록 한다.

- MASTER, SLAVE : 매스터 혹은 슬레이브.
- GV(grid value) : Algorithm 1의 2.1)에서 계산된 값
- BV(boundary value) : Algorithm 1의 2.2)에서 슬레이브들 간에 교환할 경계치
- CI(convergence info) : Algorithm 1의 2.3)에서 계산된 수렴판단을 위한 (부분) 정보
- GO_ON, STOP, ERR_SLAVE, ERR_MASTER는 MASTER가 SLAVE들에게 계산을 계속/중지 명령, 또는 오류 발생한 SLAVE 또는 MASTER의 프로세서 번호를 의미한다.

또한 MY, L/R(left/right), OTHER라는 접두어를 추가로 사용하여 구분을 더욱 명확히 표시하기로 한다.

5. 무정지형 동기 연산 시스템

본 논문에서는 어느 순간에 단지 하나의 프로세서에게만 장애가 발생하는 것을 가정하며, (그림 2)와 같은 dual MASTER-SLAVE 모델을 제안한다.

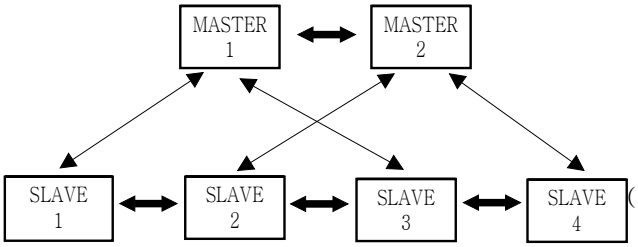


그림 2) 동기 연산의 dual MASTER-SLAVE 모델

1) SLAVE간의 BV 전송은 당사자간 직접 하도록 하되, 연산 실패에 대비하여 MASTER에게 정기적으로 담당 격자점의 최신 값(GV)을 보고한다.
2) 각 MASTER는 각기 다른 SLAVE들로부터 보고를 받아 두 MASTER는 상호 정보를 교환하여 수렴여부를 판단하고 SLAVE들의 중간 계산값을 저장한다.
3) MASTER는, 어느 SLAVE의 장애가 파악되면, SLAVE와 다른 MASTER에게도 알리고 MASTER 중 하나는 장애 SLAVE의 역할을 대신 수행한다.
4) MASTER에게 장애가 발생하면 다른 MASTER는 이후 모든 SLAVE를 관리한다.
5) 장애로 연산구조를 바꿀 경우, 각자 적절히 상대의 rank를 바꿈으로써 쉽게 재구성한다.

Check_comm()은 Wtime 간격으로 MaxTry 회수만큼 req[]와 관련된 송수신의 완료를 확인하고, 실패가 발견되면 그 원인 프로세서 번호 ErrPrcs를 파악한다.

```
int Check_comm(count, req[], MaxTry, Wtime)
{
    For i=1 to ,MaxTry, {
       MPI_Testall(count, req[], flag, status[]);
       if (flag)   return 0;    /* OK */
            else   sleep Wtime;
    }
    Identify ErrPrcs using status[];
    Cancel the uncleared communication
            using MPI_Cancel();
    return ErrPrcs;  /* nonzero */
}
```

Secure_exchange()는 두 매스터 간의 데이터 교환을 위한 것으로, 실패시 상대 MASTER의 장애를 가정한다. 한편 이와 유사하게 Secure_send()는 단지 MPI_Isend()만 있는 함수로 가정한다.

```
int Secure_exchange(receiver,data,MaxTry,Wtime)
{
    MPI_Irecv(OTHER_MASTER,data,,&req[0]);
```

```
        MPI_Isend(OTHER_MASTER,data,,&req[1])
        Err=Check_comm(2,req[],MaxTry,Wtime);
        return Err;
    }
```

Probe_and_receive()는 MASTER가 무작위로 count 개의 SLAVE들로부터 data를 수신하며, 실패시 장애 발생 SLAVE를 파악한다.

```
Probe_and_receive(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;
            /* data is identified by msgkind */
            MPI_Recv(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 ID.
}
```

이상의 함수를 사용한 각 SLAVE 및 MASTER의 알고리즘은 다음과 같다.

Algorithm 2 (SLAVE)

```
Repeat until MY_MASTER sends STOP signal,
1. MPI_Irecv(L_SLAVE, L_BV, &reqBV[0])
   MPI_Irecv(R_SLAVE, R_BV, &reqBV[1]).
2. Compute MY_GV.  (Algorithm 1 step 2.1))
3. MPI_Isend(L_SLAVE, MY_BV, &reqBV[2]).
   MPI_Isend(R_SLAVE, MY_BV, &reqBV[3])
4. At every n-th turn,
   MPI_Isend(MY_MASTER, MY_GV, &reqBV[4]).
5. Using Check_comm(reqBV[]), check
       if all previous communications are successful.
   If failed, { Start recovery;    continue; }
6. Compute MY_CI.
7. MPI_Isend(MY_MASTER, MY_CI, &reqCI[0]);
   MPI_Irecv(MY_MASTER, msg, &reqCI[1]);
8. Using Check_comm(reqCI[]), check
       if all previous communications are successful.
   If failed, { Start recovery;    continue; }
9. if (msg==STOP) {
       MPI_Send(MY_MASTER, MY_GV,);
       MPI_Finalize();
   }
```

Algorithm 3 (MASTER)

```
Repeat until satisfactory,
1. At every n-th turn,
  1) Receive the whole GV from
       all MY_SLAVEs  by Probe_and_receive();
     If failed,   { Start recovery;  continue; }
  2) Secure_exchange(OTHER_MASTER, CI,
       MaxTry, Wtime);
     If failed,  { Start recovery;   continue; }
2. Receive CI from all MY_SLAVEs by
       Probe_and_receive();
   If failed,   { Start recovery;  continue; }
3. Secure_exchange(OTHER_MASTER, CI,
       MaxTry, Wtime);
   If failed,   { Start recovery;   continue; }
4. Decide whether to continue or stop.
   Send GO_ON or STOP to all MY_SLAVES
       by Secure_send().
5. If decided to STOP, MPI_Finalize();
```

Algorithm 2의 4와 Algorithm 3의 1은 일정 주기로 SLAVE들의 최신 값을 MASTER에게 보내어 연산실패에 대비한다. 한편 어느 프로세서의 장애가 인지되면 반드시 MASTER로부터 받은 정보로 복구하도록 한다. 다음은 위의 Start recovery의 과정을 나타낸 복구 알고리즘이다.

Algorithm 4 (Recovery)

```
If I am a MASTER {
  1) If OTHER_MASTER failed,
     ① Set ERR_MASTER=OTHER_MASTER.
     ② Send ERR_MASTER to all SLAVEs..
  2) If ERR_SLAVE failed,
     ① Mark the time TIME_MARK.
     ② Exchange ERR_SLAVE and TIME_MARK
       with OTHER_MASTER by Secure_exchange().
     ③ By comparing TIME_Marks, decide which
       SLAVE really failed.
     ④ Send ERR_SLAVE to all MY_SLAVEs.
     ⑤ Send GV to all my SLAVEs for roll-back.
     ⑥ If I am the second MASTER,
        - Set  MY_MASTER=OTHER_MASTER,
               L_SLAVE=ERR_SLAVE-1;
               R_SLAVE=ERR_SLAVE+1;
        - Begin computation as ERR_SLAVE.
}
else {          /* If I am a slave */
  1) Receive info about failed processor
     by MPI_Iprobe(ANY_SRC, ANY_DATA)
  2) If MY_MASTER failed,
     set MY_MASTER=OTHER_MASTER.
  3) If OTHER_MASTER failed, nothing to do.
  4) If ERR_SLAVE failed, and
     ① if it is my L_SLAVE,
               set L_SLAVE=2nd_MASTER.
     ② if it is my R_SLAVE,
               set R_SLAVE=2nd_MASTER.
}
```

5. 구현, 성능 평가 및 결론

실험에는 [1]과 동일한 3,000×3,000의 2차원 격자 문제에서 파생된 거대 선형 시스템 문제를 사용하였으며, 수원대학교의 Hydra 클러스터의 6개 노드를 사용하여 (그림 2)처럼 구현하였다. 특정 노드의 장애는, 50회의 연산을 시행한 후 특정 노드가 무한 루프를 돌도록 함으로써 시뮬레이션하였다.

복구의 기능이 없는 Algorithm 1을 사용한 경우, 잔차의 크기가 10^{-10}에 도달할 때까지는 203.73초가 결렸다.

그러나 장애에 대비한 본 알고리즘을 사용하되 장애가 발생하지 않는 경우는 208.16초로서 장애에 대비한 연산으로 인하여 연산 시간은 약간 증가하였으나 예측한대로 같은 결과를 얻었다.

표 1은 MASTER 또는 SLAVE의 장애시, MaxTry와 Wtime에 따른 복구에 소요된 시간을 나타내는데, MaxTry*Wtime에 비례하여 오래 걸렸음을 알 수 있다. 즉 장애로 판정 후 복구하여 연산이 재시작할 때까지 걸린 시간에 비해 장애 판정까지의 시간이 대부분을 차지함을 알 수 있다. 물론 이를 너무 짧게 하면 정상적인 경우를 장애로 판정할 수도 있으므로 얼마가 적절한가 하는 것은 장애 판정에서의 정책적인 문제이다. 그리고 매스터의 장애보다 슬레이브의 장애시 복구에 시간이 더 소요되었는데, 이는 MASTER 2가 장애 발생한 슬레이브의 역할을 떠맡기 위해 추가적으로 소요되는 시간 때문이다.

<표 1> 장애발생부터 연산 재시작까지 소요된 시간

	MASTER 장애시			SLAVE 장애시		
MaxTry	5	10	5	5	10	5
Wtime	1	1	0.5	1	1	0.5
시간(sec)	5.85	10.62	3.47	6.17	11.02	3.93

한편 SLAVE에 장애 발생시 롤백(roll-ack)을 위해 정기적으로 각 SLAVE는 자신의 최신 계산치 GV를 보고하나, 그 때마다 MASTER는 길이 900만의 벡터 값을 전송받으므로 그 부담이 상당하다. 그렇다고 너무 드물게(즉 Algorithm 2와 3의 n-th turn의 n값이 클 경우) n단계 이전으로 롤백하므로 연산이 상당히 후퇴하였다.

반면, 모든 SLAVE는 롤백하지 않고 MASTER 2는 자신이 가진 값을 그대로 사용하여 장애 SLAVE의 역할을 대신하여도 그리 나쁘지 않은 결과를 얻었다.

한편 어느 노드에 장애가 발생하는 경우, 처음에는 Algorithm 4와는 달리, 각 노드가 스스로 복구하는 방법을 시도하였으나 실패하였다. 그 이유는, 장애 발생 노드의 인접 노드도 장애에 빠진(실제로는 장애가 발생한 것이 아니라 송수신을 완료하려고 오랜 시간을 소모하므로) 것으로 주변 노드들이 오해를 하여 전체적으로 과연 어느 노드에 문제가 생겼는지 판단하기 곤란하였다. 따라서 장애 발생시 어느 노드에 문제가 생겼는지는 MASTER만이 결정하도록 수정하였고, 두 개의 MASTER는 위와 같은 이유로 각기 다른 SLAVE에 장애가 발생하였다고 주장하는 경우도 있어, 장애 발생 파악 직후 시스템 시간을 체크하여 TIME_MARK의 값을 비교하여 판정하도록 하였다.

본 논문에서는 MPI-1 표준 함수만을 사용하여 구현하여 두 개 이상의 프로세서에 동시에 장애가 발생하는 경우는 고려하지 않았다. 물론 대기 프로세서를 더 사용하면 해결할 수 있는 문제이나, MPI-2의 MPI_Spawn() 등의 동적 프로세스 관리 함수를 사용하면 더 일반적인 것으로 개선할 수 있을 것이다.

참고문헌

[1] 박필성, "MPMD 방식의 비동기 연산을 이용한 응용수준의 무정지 선형 시스템의 해법", 한국정보처리학회 논문지 12-A(5):421-426, 2005.

[2] 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.,

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

[4] I. T. Foster, "Designing and building parallel programs," Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

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

[6] 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.

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