제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

인터넷기반 분산 클러스터 환경에서의 결함허용을 위한 중복된 작업할당 기법

최인복*, 이재동*
*단국대학교 컴퓨터과학및통계학과
e-mail : pluto612@cs.dankook.ac.kr

A Task Duplication Scheme For Fault-Tolerance In Internet Based Distributed Clustering Systems

In-Bok Choi*, Jae-Dong Lee*
*Division of Information and Computer Science, Dan-Kook University

요 약

최근 인터넷이 발달하면서 TCP/IP 프로토콜 기반의 분산 클러스터환경의 구축이 가능해졌다. 이렇게 서로 다른 네트워크를 통해 연결된 분산 클러스터 시스템에서는 기존의 클러스터 시스템과는 다르게 네트워크의 지연이나 노드의 결함등에 유연히 대처해야 한다. 따라서, 본 논문에서는 이러한 인터넷 기반의 분산 클러스터 환경에서 메시지 전달방식을 이용한 고성능 클러스터 컴퓨팅 작업 시 네트워크나 노드의 결함에 효과적으로 대처할 수 있도록 중복된 작업할당 기법을 통한 결함허용 기법을 제시한다. 중복된 작업할당 기법을 적용하기 위해 Send, GSS, WF 와 같은 기존의 부하공유 알고리즘에 대한 공통적인 스케줄러를 설계하였으며, 이 스케줄러를 이용한 TDS_for_FT 알고리즘을 작성하였다.

본 논문에서 제시한 중복된 작업할당 기법이 효과적임을 보이기 위하여 게이트웨이를 통해 연결된 두 개의 네트워크를 구성하여 분산 PC 클러스터 환경을 구축하고, PVM 을 이용한 행렬의 곱셈 프로그램을 통하여 실험하였다. 클러스터를 구성하는 임의의 한 노드에 일정시간의 delay 를 적용한 결과, 중복된 작업할당 기법을 통하여 결함허용성 보장이 가능함을 보였다.

1. 서론

인터넷이 발달하면서 대부분의 컴퓨터는 TCP/IP 프로토콜에 의해 네트워크에 연결되어 있다. 따라서, 추가적인 네트워크의 구성없이 지역적으로 분산되어 있는 컴퓨터들을 인터넷 상에서 연결하여 분산된 클러스터를 구성할 수 있게 되었다. 현재, 대부분의 클러스터 시스템은 보다 나은 효율을 위해 Myrinet, SCI, 또는 Gigabit Ethernet 과 같은 고속 네트워크에 연결하거나, VIA 나 MyrinetGM 등과 같이 사용자 수준의 특수한 프로토콜을 이용하고 있다. 하지만 이러한 방법들은 지역적으로 한정된 특별한 네트워크를 구성해야 하므로 추가적인 비용이 필요하고 지역적으로 한정되어 구성해야 하는 확장성에 문제가 있다.

이기종 환경에서의 클러스터 연구는 대부분 NOW 에서 수행되어져 왔다. NOW 나 COW 와 같은 기존의 클러스터 컴퓨팅 환경에서는 노드들의 안전성을 보장하지 않았다. 그 이유는 NOW 나 COW 는 잘 정비된 지역적 네트워크 환경이기 때문이다. 따라서, 이러한 환경에서 설계된 기존의 부하공유 알고리즘 역시 네트워크나 노드의 고장에 대한 고려를 하지 않았다. 하지만, 인터넷 상의 분산 클러스터 컴퓨팅 환경에서는 네트워크의 문제로 인한 전송지연이나 연결단절이 있을 수 있고, 프로그램 수행도중 노드의 고장으로 인한 결함이 발생할 가능성이 높다.

따라서, 본 논문에서는 인터넷 기반의 분산 클러스터 환경에서 메시지전달방식을 이용한 고성능 클러스터 컴퓨팅 작업 시 결함허용성이 보장되도록 하기위한 작업의 중복할당 기법을 제안하고, Send, GSS,

WF 알고리즘과 같은 기존의 부하공유 알고리즘에 이 기법을 적용한 실험을 통하여 작업의 중복할당 기법이 결함허용성에 효과적임을 보인다.

본 논문은 다음과 같이 구성된다. 2 장에서는 기존의 대표적인 부하공유 알고리즘들에 대한 연구들을 소개하고, 3 장에서는 작업 중복할당 기법에 대하여 설명하다. 4 장에서는 인터넷기반의 분산 클러스터 환경에서 실험을 통하여 본 논문에서 제시한 기법의 결과를 살펴보고, 5 장에서는 결론 및 앞으로의 과제를 언급함으로써 논문을 마친다.

2. 관련연구

고성능 클러스터 시스템에서 스케줄링은 크게 시간분할(time sharing) 방식과 공간분할(space sharing) 방식으로 나눌 수 있다. 시간분할방식은 여러 개의 프로그램들이 코스케줄링(coscheduling)을 통하여 전체 시스템을 공유하는 것이고 공간분할 방식은 시스템을 여러 개로 분할시켜 각각의 분할된 영역에 하나의 프로그램을 수행하는 것이다. 부하공유란, 공간분할 방식의 스케줄링 방식에서 빠른 실행결과를 얻기 위한 방법을 말한다[2].

[3]의 연구에 의하면, NOW 환경에서 대체적으로 Send 알고리즘과 GSS 알고리즘이 좋은 성능을 보인 것으로 나타났다. 그리고, 이기종 클러스터 환경에서의 대표적인 부하공유 알고리즘으로는 Flynn Hummel 의 Weighted Factoring(줄여서, WF)알고리즘이 있다[4].

각 알고리즘들에서 다음 작업의 데이터 크기를 결정하기 위한 일반화된 공식은 다음의 [표 2.1]과 같이 나타낼 수 있다[8].

알고리즘	공식
Send	$S_i = n$
GSS	$G_i = \left\lceil (1 - \frac{1}{p})^i \frac{N}{p} \right\rceil$
WF	$F_{ij} = \left\lceil (1-\frac{1}{x})^i \frac{N}{x} \frac{W_j}{\sum_{k=1}^{k=p} W_k} \right\rceil$

[표 2.1] 알고리즘별 스케줄링된 데이터 크기 결정 공식

3. 결함허용 기법

이 장에서는 부하공유 기법에 결함허용성을 보장하기 위한 중복된 작업할당 기법에 대하여 논의한다.

2 장의 [표 2.1]의 공식에서 보이는 대표적인 부하공유 알고리즘들은 각 종노드들에게 작업을 분배하기 이전에 정해진 공식에 의하여 각 알고리즘별 데이터 크기에 대한 스케줄러를 만들 수 있다. 따라서, 일반적인 부하공유 알고리즘에 효율적인 작업의 분배를 위하여 다음과 같은 스케줄러를 작성할 수 있다. N 개의 데이터와 P 개의 종노드에 대한 스케줄러의 데이터 구조는 다음과 같다.

```
struct slave_node{
  int job[max_index];
  int status[max_index];//0:미수행,1:수행중,2:완료
  float weight; //가중치
  int doing; //현재 수행중인 job 의 크기
  int remain; //매수행중인 job 의 크기
} sched[P];
```

주노드에서 임의의 i 번째 종노드로부터 부분적인 결과 데이터 sched[j].job[k]를 전송 받았을 때, slave_node 구조체 내의 변수들은 메시지 전달방식 고성능 클러스터에서의 기본적인 명령인 send/receive 명령 수행 시 아래의 서브루틴과 같이 변화한다.

```
Procedure SEND (sched[j].job[k], i)
1  send (data of sched[j].job[k] to i^th node);
2  sched[j].status[k] = 1;
3  sched[j].remain -= size of sched[j].job[k];
4  sched[j].doing += size of sched[j].job[k];

Procedure RECEIVE (sched[j].job[k], i)
1  wait (arbitrary partial result data sched[j].job[k]
     from i^th node);
2  sched[j].status[k] = 2;
3   sched[j].doing -= size of sched[j].job[k];
4
```

이렇게 함으로써 스케줄러 상의 작업의 상태, 현재 수행중인 작업의 크기, 그리고 남겨진 작업의 크기 등을 관리하여 효율적인 작업분배를 할 수 있도록 한다.

결함허용을 위한 작업의 중복할당 방법은 해당 종노드의 스케줄러상의 작업을 모두 마친 종노드에 대하여 작업속도가 느린 종노드의 작업을 중복 수행하도록 하는 것이다. 주노드는 해당 작업을 모두 마친 종노드에게 아래와 같이 다음 작업을 부여한다.

①다른 종노드의 아직 미수행인 작업이 남아있는 경우, 종노드의 성능에 비해 미수행 작업량이 가장 많은 종노드의 작업을 대신 수행하도록 하거나,

②다른 종노드의 미수행 작업이 없는 경우, 종노드의 성능에 비해 현재 수행중인 작업량이 가장 많은 종노드의 작업을 중복수행 하도록 한다.

임의의 종노드 i 로부터 부분적인 결과 데이터 sched[j].job[k]를 전송 받았을 때, 주노드는 결함허용을 위하여 i 번째 종노드에 전송할 작업을 아래의 서브함수와 같이 선정한다.

```
Function   SELECT_JOB_for_FT(sched[j].job[k],i)
● Input : sched[j].job[k], received partial result data;
          i, index of a arbitrary slave node
● Output : sched[m].job[n], next partial job
          for i^th slave node

1   if(sched[i].remain != 0){
2       m=i;
```

```
3     n=k+1;
4   }
5   else if(Exist sched[0...(P-1).remain!=0]){
6     m=0;
7     max_remain=sched[0].remain*sched[0].weight;
8     for(index=0;index<P;index++)
9       temp_remain=sched[index].remain *
                    sched[index].weight;
10      if(max_remain < temp_remain){
11        max_remain=temp_remain;
12        m=index;
13      }
14    }
15    for(index=max_index; index>0; index--){
16      if(schedule[m].status[index] == 0){
17        n=index;
18        break;
19      }
20    }
21  }
22  else {
23    m=0;
24    max_doing=sched[0].doing*sched[0].weight;
25    for(index=0;index<P;index++)
26      temp_doing=sched[index].doing *
                    sched[index].weight;
27      if(max_doing < temp_doing){
28        max_doing=temp_doing;
29        m=index;
30      }
31    }
32    for(index=max_index; index>0; index--){
33      if(schedule[m].status[index] == 1){
34        n=index;
35        break;
36      }
37    }
38  }
39  return   schedule[m].job[j];
```

이와 같이 작업을 분배하는 경우에 전체 작업이 끝마쳐질 시점에서 주노드에는 하나의 작업에 대하여 여러 종노드들이 수행한 결과가 공통으로 도착할 수 있다. 따라서, 주노드는 중복된 결과 중 가장 빠르게 도착한 결과만 취하고 나머지는 무시한다. 중복된 작업 수행은 결함허용을 위한 하나의 중요한 방법이 될 수 있다. 임의의 종노드에 고장이 발생하였을 경우, 주노드는 해당 종노드로부터 어떠한 응답도 전달 받을 수 없다. 이는 해당 프로그램 또는 시스템 차원에서의 추가적인 작업들을 수행해야 하는 불편함을 감수해야만 한다. 따라서, 중복수행이라는 간단한 메커니즘을 이용하여 결함을 허용하는 방법이 효과적이다.

위의 결함허용을 위한 프로시저 및 서브함수를 이용한 부하공유 알고리즘은 다음과 같이 적용된다.

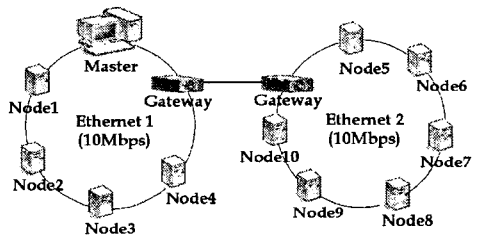
Algorithm TDS_for_FT

- Input : N, size of job; P, number of slave nodes
- Output : result, merged partial data are received from slave nodes(e.g. array, a variable)

```
1   CREATE(scheduler by Send/GSS/WF algorithm);
2   SEND(sched[0...(P-1)].job[0], 0...(P-1));
3   while(all partial results are not gathered by master
    node){
4     RECEIVE(sched[j].job[k], i);
5     sched[m].job[n] =
        SELECT_JOB_for_FT(sched[j].[k],i);
6     SEND(sched[m].job[n],i);
7     MERGE(partial result data of sched[j].job[k]
        to the result);
8   }
```

4. 실험 및 결과

분산 클러스터 환경에서 중복된 작업할당을 통하여 효과적으로 결함허용성을 제공하는지를 알아보기 위하여 [그림 4.1]과 같이 두 개의 네트워크로 분리된 환경을 구성하였다.



[그림 4.1] 실험환경

[그림 4.1]의 각 노드들의 하드웨어와 소프트웨어는 아래의 [표 4.1]과 같이 구성하였다.

노드명	CPU	메모리	운영체제
Master	P3 450	320 M	Linux(kernel 2.4)
Node1	P3 733	128 M	Linux(kernel 2.4)
Node2	P3 733	128 M	Linux(kernel 2.4)
Node3	P3 450	128 M	Linux(kernel 2.4)
Node4	P3 MMX 300	128 M	Solaris 8.0
Node5	P3 MMX 300	128 M	Solaris 8.0
Node6	P3 450	128M	Linux(kernel 2.4)
Node7	P-pro 133	64 M	Linux(kernel 2.0)
Node8	P-pro 133	32 M	Linux(kernel 2.0)
Node9	P-pro 133	32 M	Linux(kernel 2.0)
Node10	P-pro 133	16 M	Linux(kernel 2.0)

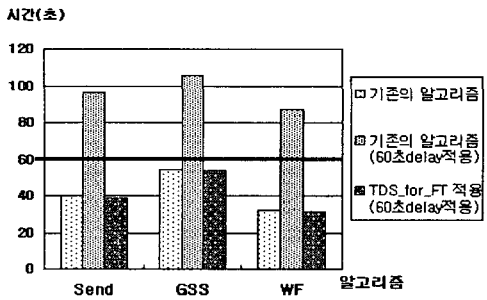
[표 4.1] 하드웨어/소프트웨어 구성표

이 장의 실험에서는 두개의 500*500 크기의 행렬을 곱셈할 수 있는 응용프로그램을 PVM3.4.4 라이브러리를 이용하여 실험하였다.

2 장의 관련연구에서 논의된 Send, GSS, WF 알고리즘을 이용하여 스케줄러를 작성하고(TDS_for_FT 알고리즘의 2 행), 각각의 알고리즘에 대한 수행시간을 측정하였다.

결함허용성을 테스트하기 위하여 임의의 하나의

노드에 결함을 가정하여 60 초의 delay 가 생기도록 조정한 결과를 측정하였다. 알고리즘별 20 회의 반복적인 프로그램의 실행을 통하여 얻은 평균 실행시간(초)를 기록한 결과, 아래의 [그림 4.2]와 같은 결과를 얻을 수 있었다. 참고로, Send 알고리즘에서의 고정된 분할 단위는 500 개의 index 들을 500 회 전송하도록 수행하였다.



[그림 4.2] 결함허용 실험결과(500*500 크기)

실험 결과에 의하면, 임의의 하나의 종노드에 60 초의 delay 를 적용하였을 때, 기존의 알고리즘들은 모두 평균 수행시간이 60 초 이상을 초과한 반면, TDS_for_FT 를 적용한 알고리즘들은 정상적인 상태에서의 수행시간과 비슷한 결과를 나타내었다. 이 실험에서 평균수행시간이 60 초 이상이 되었다는 것은 하나의 종노드에서 결함이 발생하였을 경우에는 정상적인 실행결과를 얻을 수 없음을 의미한다. 따라서, 임의의 종노드에 결함이 발생했을 경우에 기존의 알고리즘만을 이용한 경우에는 필요한 결과를 얻지 못한다는 결론을 얻을 수 있다. 결론적으로, 네트워크나 노드의 상태가 유동적인 분산 클러스터 환경에서는 본 논문에서 제시한 중복된 작업할당 기법을 통하여 결함허용성을 보장할 수 있음을 알 수 있다.

5. 결론

본 논문에서는 인터넷기반 분산 클러스터 환경에서 네트워크 및 노드의 결함에도 효과적으로 대처할 수 있도록 중복된 작업할당을 통한 결함허용 기법에 대하여 논의하였다.

인터넷기반의 분산클러스터 환경에서의 중복된 작업할당을 정책이 효과적임을 보이기 위하여 두 개의 분산된 네트워크상에서 PC 를 이용하여 분산 클러스터를 구축하고 PVM 을 이용하여 500*500 의 크기에 해당하는 행렬의 곱셈 프로그램을 수행하였다. 그 결과, 본 논문에서 설계한 TDS_for_FT 알고리즘을 적용한 부하공유 알고리즘들이 기존의 알고리즘에서는 할 수 없었던 결함허용성을 제공할 수 있음을 보였다.

분산 클러스터 환경은 NOW 와 같은 기존의 환경보다 더욱 동적이다. 이러한 환경에서는 네트워크의 지연이나 노드의 결함에 효과적으로 대처해야만 한다. 따라서, 향후에는 더욱 다양한 환경에서 효과적으로 부하공유 및 결함허용을 지원할 수 있는 알고리즘의 설계가 필요할 것이다.

참고문헌

[1] 구본근, "NOW 환경에서 개선된 고정 분할 단위 알고리즘", 정보처리학회논문지, Vol.8 No.2, 2001.

[2] 김진성, 심영철, "이질적 계산 능력을 가진 NOW 를 위한 공간 공유 스케쥴링 기법", 정보과학회논문지, Vol.27 No.7, 2000.

[3] 박윤용, 박정호, 임동선, "이종 분산 환경에서 UNIX 커널 성능 측정 방법에 관한 연구", 정보처리학회지, Vol.6 No.11, 1999.

[4] 유찬수, "리눅스 클러스터링", 정보처리학회지, Vol.18 No.2, 2000.

[6] A. Piotrowski and S. Dandamudi, "A Comparative Study of Load Sharing on Networks of Workstations", Proc. Int. Conf. Parallel and Distributed computing system, New Orleans, Oct. 1997.

[7] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-Sharing in Heterogeneous Systems via Weighted Factoring", SPAA, 1997.

[8] Yangsuk Kee and Soonhoi Ha, "A Robust Dynamic Load-Balancing Scheme for Data Parallel Application on Message Passing Architecture," PDPTA'98 (Internation Conf. on Parallel and Distributed Processing Techniques and Applications), pp. 974-980, Vol. II, 1998.