인터넷 기반 클러스터 시스템 환경에서 부하공유 및 결함허용 알고리즘

최 인 복[†] · 이 재 동[††]

요 약

인터넷 기반의 클러스터 시스템 환경에서 알고리즘의 이식성을 높이기 위해서는 네트워크의 특성 및 노드의 이질성에 따른 부하 불균형, 그리고 네트워크나 노드의 결함과 같은 다양한 수행환경의 변화에도 효과적으로 적응할 수 있어야 한다. 본 논문에서 제안하는 Expanded-WF 알고리즘은 Weighted Factoring 알고리즘을 기반으로 부하공유를 위하여 적응할당정책과 개선된 고정 분할 단위 알고리즘을 적용하고 결함허용을 위하여 작업을 중복 수행하는 기법을 적용한다. 적응할당정책으로는 느린 종노드의 작업을 빠른 종노드가 대신 수행하는 기법을 적용하였고, 개선된 고정 분할 단위 알고리즘은 네트워크의 통신시간과 계산시간을 겹치게 하는 것이다. 두 개의 네트워크 환경으로 구성된 이기종의 클러스터 환경에서 PVM을 이용한 행렬의 곱셈 프로그램으로 실험한 결과, 본 논문에서 제안한 알고리즘이 NOW 환경에서 효율적인 Send, GSS, Weighted Factoring 알고리즘보다 각각 55%, 63%, 그리고 20% 효율적임을 보였으며, 또한 결함허용도 가능함을 보였다.

An Algorithm For Load-Sharing and Fault-Tolerance In Internet-Based Clustering Systems

In-Bok Choi[†] · Jae-Dong Lee[††]

ABSTRACT

Since there are various networks and heterogeneity of nodes in Internet, the existing load-sharing algorithms are hardly adapted for use in Internet-based clustering systems. Therefore, in Internet-based clustering systems, a load-sharing algorithm must consider various conditions such as heterogeneity of nodes, characteristics of a network and imbalance of load, and so on. This paper has proposed an expanded-WF algorithm which is based on a WF (Weighted Factoring) algorithm for load-sharing in Internet-based clustering systems. The proposed algorithm uses an adaptive granularity strategy for load-sharing and duplicate execution of partial job for fault-tolerance. For the simulation, the matrix multiplication using PVM is performed on the heterogeneous clustering environment which consists of two different networks. Compared to other algorithms such as Send, GSS and Weighted Factoring, the proposed algorithm results in an improvement of performance by 55%, 63% and 20%, respectively. Also, this paper shows that it can process the fault-tolerance.

키워드 : 클러스터 시스템(Cluster System), 부하공유(Load-Sharing), 결함허용(Fault-Tolerance), WF 알고리즘(Weighted Factoring Algorithm), 메시지 전달(Message Passing)

1. 서 론

고성능 컴퓨팅의 중요성이 증대되면서 1980년대 이후 슈퍼컴퓨터와 대용량 병렬 컴퓨터가 폭넓게 활용되어 왔다. 그러나, 고성능 컴퓨팅의 요구가 늘어나면서 이러한 시스템들의 높은 가격, 유지보수의 어려움 등 여러 가지 문제점이 드러나고 있다. 최근 마이크로 프로세서의 뛰어난 성능향상과 고속 네트워크의 보급으로 단일 컴퓨터들을 네트워크로 연결함으로써 하나의 병렬 컴퓨터처럼 작동하는 클러스터 컴퓨팅이 가능해졌다. 클러스터를 사용하는 이유는 무엇보다도 가격대 성능비가 우수하고 자체제작이 가능하여 문제 발생시 해결이 용이하기 때문이다[4, 7, 12, 17].

대부분의 클러스터 컴퓨터는 보다 나은 성능향상을 위해 Myrinet, SCI 또는 기가비트 이더넷과 같은 고속네트워크에 연결하거나, VIA나 MyrinetGM 등과 같이 사용자 수준

※ 이 연구는 2001학년도 단국대학교 대학 연구비의 지원으로 연구되었음.
† 준 회 원 : 단국대학교 대학원 컴퓨터과학및통계학과
†† 정 회 원 : 단국대학교 정보컴퓨터학부 교수
논문접수 : 2003년 4월 30일, 심사완료 : 2003년 6월 28일

의 특수한 프로토콜을 이용하고 있다. 하지만 이러한 방법들은 지역적으로 한정된 특별한 네트워크를 구성해야 하므로 확장성에 문제가 있을 뿐 아니라 추가적인 비용이 필요하다[1-3, 15].

인터넷이 발달하면서 대부분의 컴퓨터는 TCP/IP 프로토콜에 의해 네트워크에 연결되어 있으므로 인터넷에 연결되어 있는 컴퓨터들을 추가적인 네트워크의 구성이나 특수한 프로토콜 없이 클러스터를 구성하는 것이 가능해졌다. 하지만, 인터넷에는 여러 종류의 다양한 네트워크 환경들이 혼재되어 있고 이질적인 노드들로 구성되어 있으며, 네트워크의 단절 및 노드의 고장 등으로 인한 결함이 발생할 가능성이 많이 있다. 따라서, 인터넷 기반의 클러스터 환경에서는 기존의 클러스터 환경에서 고려하지 않았던 다양한 네트워크의 특성, 부하불균형, 노드의 이질성 및 노드의 결함과 같은 다양한 수행 환경의 변화에도 효과적으로 적응할 수 있도록 해야 한다[7, 17].

본 논문에서는 인터넷 기반의 클러스터 환경에서 Message Passing 방식의 고성능 클러스터 컴퓨팅 작업시 효율적인 부하공유와 결함허용이 가능한 Expanded-WF 알고리즘을 제안한다. Expanded-WF 알고리즘은 Weighted Factoring 알고리즘[16]을 기반으로 부하공유를 위하여 적응할당정책을 적용하는 동시에 개선된 고정 분할 단위 알고리즘을 적용하고 결함허용을 위하여 작업을 중복 수행한다. 적응할당정책은 상대적으로 느린 종노드의 작업을 빠른 종노드가 대신 수행하도록 하는 기법이며, 개선된 고정 분할 단위 알고리즘은 주노드에서 종노드로의 네트워크의 통신시간과 계산시간을 겹치게 하는 기법이다[2]. 본 논문에서 제안한 Expanded-WF 알고리즘이 인터넷 기반의 클러스터 환경에서 효율적임을 보이기 위해 두 개의 네트워크로 구성된 클러스터 환경을 구성하고, PVM을 이용하여 행렬의 곱셈을 계산하는 프로그램을 알고리즘별로 구현하여 성능을 측정한다[10, 13, 16].

본 논문은 다음과 같이 구성된다. 2장에서는 클러스터 환경에서의 부하공유를 위한 관련연구들을 소개하고, 3장에서는 본 논문에서 제안하는 Expanded-WF 알고리즘을 살펴본다. 4장에서는 인터넷 기반 클러스터 환경에서의 Expanded-WF 알고리즘의 성능을 실험을 통해 평가해 보고, 5장에서는 결론 및 향후 발전 방향을 언급하도록 한다.

2. 관련 연구

고성능 클러스터 시스템에서 스케줄링은 크게 시간분할(time sharing) 방법과 공간분할(space sharing) 방법으로 나눌 수 있다. 시간분할 방법은 여러 개의 프로그램들이 코스케줄링(coscheduling) 등을 통하여 전체 시스템을 공유하는 것이고, 공간분할 방법은 시스템을 여러 개로 분할시켜 각각의 분할된 영역에 하나의 프로그램을 수행하는 것이다. 많은 연구에서 공간분할 방법이 시간분할 방법보다 우수하다고 알려져 있다[5].

NOW(Network of Workstation) 환경에서 부하공유를 위한 알고리즘으로 Send 알고리즘과 GSS 알고리즘이 좋은 성능을 보인 것으로 Piotrowski와 Dandamudi의 연구결과에 나타났다[2, 11, 12, 19]. Flynn Hummel의 연구 결과에 의하면 이기종 클러스터 환경에서는 Weighted Factoring 알고리즘이 좋은 성능을 나타냈으며[18], 최근에는 이기종 분산 클러스터 환경에 적합한 Adaptive Weighted Factoring 알고리즘이 연구되었다[14].

Send 알고리즘은 주노드에서 응답이 빠른 종노드에게 스케줄큐에 있는 분할 단위만큼의 다음 작업을 먼저 할당하는 FCFS(First-Come-First-Serve)방식의 대표적인 비선점형 스케줄링 알고리즘이다[2, 11, 19].

GSS(Guided Self-Scheduling) 알고리즘은 전체 데이터 중에서 아직 남아있는 데이터의 크기에 대하여 일정비율의 데이터를 할당하여 데이터의 크기를 점점 줄여나간다. N개의 데이터와 P개의 종노드에 대하여 다음 종노드에게 스케줄링할 i번째 덩어리의 크기(G_i)는 식 (1)과 같이 결정된다[11, 13, 19].

$$G_i = \left\lceil (1 - \frac{1}{P})^i \frac{N}{P} \right\rceil \tag{1}$$

Weighted Factoring 알고리즘은 종노드의 계산 성능의 비율에 따라 가중치(weight)을 부여하고 그 가중치에 따라 반복적으로 데이터의 크기를 동적으로 줄여나간다. N개의 데이터와 P개의 종노드에 대하여 i번째 묶음에 있는 j번째 데이터 덩어리의 크기(F_{ij})는 식 (2)와 같이 결정된다[13, 18, 19].

$$\begin{aligned} F_{ij} &= \left\lceil (1-\frac{1}{x})^i \frac{N}{x} \frac{W_j}{\sum_{k=1}^{k=p} W_k} \right\rceil \\ &= \left\lceil (\frac{1}{2})^{i+1} N \frac{W_j}{\sum_{k=1}^{k=p} W_k} \right\rceil \end{aligned} \tag{2}$$

3. Expanded-WF 알고리즘의 설계

이 장에서는 인터넷 기반 클러스터 시스템 환경에서 부하공유 및 결함허용을 위한 Expanded-WF 알고리즘을 설

계한다.

3.1 전역 스케줄러 및 서브루틴 설계

Weighted Factoring 알고리즘에 적응할당정책을 적용하기 위해서는 각 종노드의 성능 및 작업할당에 대한 정보를 관리할 수 있는 스케줄러를 운영하는 것이 필요하며(전역 스케줄러), Message Passing 방식의 기본적인 명령인 send/receive 명령 수행에 대하여 추가적인 스케줄러 정보변경 작업이 필요하다.

N개의 데이터와 P개의 종노드에 대한 전역 스케줄러의 데이터 구조는 다음과 같다.

```
struct slave_node {
  int job[ ⌊ log2N ⌋ ];        // 할당된 작업
  int status[ ⌊ log2N ⌋ ];     // 수행상태(0 : 미수행,
                                  1 : 전송/수행중, 2 : 완료)
  float weight ;               // 가중치(종노드의 성능)
  int remain ;                 // 미수행중인 job의 크기
  int doing ;                  // 현재 전송/수행중인 job의 크기
} schedule[P] ;
```

이 전역 스케줄러에서 각 종노드에는 $\lfloor \log_2 N \rfloor$ 개의 작업이 할당된다. 각 작업의 상태는 status 변수에 의해 관리되며, weight은 종노드의 상대적인 성능을 나타낸다. 그리고 remain과 doing 변수는 적응할당정책을 적용하기 위해 수행해야할 작업량과 현재 수행하고 있는 작업량을 유지한다.

주노드가 종노드에 작업을 할당할 때에는 주로 send 함수를 이용하여 종노드에 작업을 전송한다. 이렇게 send 함수를 이용하여 각 종노드에 작업을 전송할 때, 주노드는 종노드의 상태를 관리하기 위하여 전역 스케줄러에 다음과 같은 추가적인 작업들을 수행한다.

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

SEND 서브루틴에서는 해당 job의 상태를 '전송/수행중'으로 설정하고(2행), '미수행'작업량을 줄이며(3행), 현재 '전송/수행중'인 작업량을 증가시킨다(4행).

종노드로부터 임의의 부분적인 결과 데이터를 전송받기 위해서는 receive 함수를 이용하게 되는데, 이 때에도 전역 스케줄러에 다음과 같은 추가적인 작업들을 수행하여 종노드의 상태를 관리한다.

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

RECEIVE 서브루틴에서는 종노드로부터 부분적인 결과를 전송 받으면(1행), 해당 작업에 대한 상태를 '완료'로 설정하고(2행), '전송/수행중'인 작업량을 감소시킨다(3행).

3.2 적응할당정책에 의한 부하공유 기법

Weighted Factoring 알고리즘은 작업 초기에 수행된 종노드의 성능평가에 의한 가중치만을 이용하여 부하를 조절하기 때문에 작업 도중에 발생하는 종노드의 변화에는 대처하기 어려운 단점을 가지고 있다. 이를 보완하기 위하여 실행시간에 종노드의 가중치를 동적으로 변화시키는 방법도 연구되고 있다[14].

일반적으로, 클러스터 시스템에서는 부하가 큰 노드에서 일부 작업을 부하가 작은 노드로 이동시키는 work stealing (작업 이주) 기법이 사용되고 있지만, work stealing 기법은 데이터 재분배를 위한 추가적인 통신 오버헤드와 이로 인한 또 다른 부하 불균형을 초래할 수 있는 단점을 가지고 있다. 이질적인 계산능력을 가진 NOW 환경에서의 연구에 의하면, work stealing 기법보다는 우선 순위를 낮추는 적응할당정책이 좋은 성능을 보였으며[5], Hummel은 가장 느린 몇 개의 종노드들을 인위적으로 제외시킴으로써 수행시간이 단축됨을 보였다[18].

이러한 연구들을 기반으로 볼 때, 작업도중 발생하는 종노드의 변화에 효율적으로 대처하기 위해서는 스케줄러 상의 작업을 모두 마친 종노드가 성능에 비해 가장 느리게 작업을 수행하는 종노드의 작업을 대신 처리하도록 함으로써 자연스럽게 느린 종노드의 작업량을 줄이거나 제외시킬 수 있도록 하는 적응할당정책을 적용하는 것이 바람직하다.

주노드는 임의의 종노드로부터 부분적인 결과 데이터를 전송 받았을 경우, 해당 종노드의 다음 작업을 선정해야 한다. 이 때, 주노드는 이러한 적응할당정책을 적용하기 위하여 해당 종노드의 다음 작업을 아래의 SELECT_JOB_for_LS와 같이 선정한다. 임의의 종노드 i로부터 schedule[j].[k]의 부분적인 결과 데이터를 전송 받았을 경우, 주노드는 종노드 i의 다음 작업을 선정하기 위하여 다음의 함수와 같이

수행한다.

```
Function SELECT_JOB_for_LS (schedule[j].job[k], i)
 • Input : schedule[j].job[k], received partial result data ;
           i, index of a arbitrary slave node
 • Output : schedule[m].job[n], next partial job for i^th
            slave node
1   if (schedule[i].remain != 0) {
2       m = i ;
3       n = k + 1 ;
4   }
5   else if (Exist schedule[0 … (P-1)].remain != 0) {
6       m = 0 ;
7       max_remain = schedule[0].remain × schedule[0].weight ;
8       for (index = 1 ; index < P ; index++) {
9           temp_remain = schedule[index].remain
                          × schedule[index].weight ;
10          if (max_remain < temp_remain) {
11              max_remain = temp_remain ;
12              m = index ;
13          }
14      }
15      for (index = ⌊ log2 N ⌋ -1 ; index > 0 ; index--) {
16          if (schedule[m].status == 0) {
17              n = index ;
18              break ;
19      }
20  }
21  return (schedule[m].job[n]) ;
```

해당 종노드의 스케줄러에 아직 작업이 남아 있는 경우(1행)에는 스케줄에 따라 차례로 작업을 수행한다(2~3행). 해당 종노드에 대한 작업을 모두 마쳤을 경우에는 다른 종노드의 작업상황을 검색한다. 다른 종노드에 아직 '미수행' 작업이 남아 있는 경우(5행)에는 성능(weight)에 비해 작업 수행이 가장 늦은 종노드를 선택하고(6~14행), 선택된 종노드의 아직 전송되지 않은 작업 중 가장 마지막 작업을 선택하여(15~19행) 대신 수행하도록 한다. 이렇게 함으로써 자연스럽게 성능에 비해 느린 종노드의 계산량을 줄이는 적응할당정책을 적용하게 된다.

3.3 개선된 고정 분할 단위 알고리즘에 의한 부하공유 기법

이 절에서는 인터넷 기반 이기종 환경에서 개선된 고정 분할 단위 알고리즘에 의한 부하공유 기법을 적용하여 수행시간의 성능을 향상시키는 FOR-LOAD-SHARING 알고리즘을 제안한다. FOR-LOAD-SHARING 알고리즘은 Send 알고리즘 및 개선된 고정 분할 단위 알고리즘(preSend 알고리즘)[2]과 같은 고정 분할 단위 알고리즘이다.

preSend 알고리즘에서 주노드는 종노드에게 분할 단위만큼 작업을 전송한 후 종노드로부터 결과를 수신할 때까지 휴지상태를 유지하는 것이 아니라, 적당한 시간이 지나면 종노드로부터의 결과 수신 여부와 관계없이 다음 작업을 미리 전송하도록 하여 종노드의 계산시간과 통신시간을 겹치게 함으로써 성능을 향상시켰다[2].

인터넷 환경은 다양한 네트워크와 이질적인 노드들로 구성되어 있어서 NOW 환경에서의 preSend 알고리즘과 같이 적당한 시간간격마다 작업을 전송하는 것은 어려운 문제이다. 따라서, 제안하는 알고리즘에서는 주노드가 각 종노드에 우선 두 개씩의 작업을 연속적으로 전송한 후, 결과를 전송한 종노드에 처리하고자하는 다음 작업을 전송한다. 각 종노드는 하나의 작업을 수행하여 결과를 주노드에 전송한 후, 다음 작업의 수신을 위해 휴지 상태로 가지 않고 즉시 다음 작업을 수행할 수 있다. 이러한 방법을 통하여 네트워크 통신시간과 계산시간을 겹치도록 함으로써 전체적인 수행시간을 줄일 수 있다.

이러한 방법에 기반을둔 FOR-LOAD-SHARING 알고리즘은 다음과 같다.

```
Algorithm FOR-LOAD-SHARING
 • Input : P, number of slave nodes ;
           schedule[P], a scheduled array for P slave nodes
 • Output : result, merged partial data which are received from
            slave nodes (e.g. array, a variable)

1   SEND (schedule[0 … (P-1)].job[0], 0 … (P-1)) ;
2   SEND (schedule[0 … (P-1)].job[1], 0 … (P-1)) ;
3   while (all partial results are not gathered by master node) {
4       RECEIVE (schedule[j].job[k], i) ;
5       schedule[m].job[n] = SELECT_JOB_for_LS
                             (schedule[j].job[k], i) ;
6       SEND (schedule[m].job[n], i) ;
7       MERGE (pratial result data of schedule[j].job[k]
                to the result) ;
8   }
```

주노드는 처음 하나의 작업을 각 종노드에 보낸 후(1행), 종노드들이 첫 번째 작업을 수행하는 동안 두 번째 작업을 보낸다(2행). 이후, 종노드에서는 하나의 작업이 완료되면 주노드로부터 새로운 작업을 기다릴 필요 없이 작업을 수행하는 동안 전송받은 다음 작업을 수행할 수 있으므로 계산시간과 네트워크 시간을 겹치도록 하여 전체 계산시간을 줄일 수 있다.

3.4 작업중복에 의한 결함허용 기법

NOW나 COW와 같은 클러스터 컴퓨팅 환경에서는 노드

들의 안정성을 보장하지 않는다[2, 12, 17, 19]. 기존의 부하 공유 알고리즘 또한 노드의 고장에 대한 고려를 하지 않았다. 그 이유는 NOW나 COW와 같은 기존의 클러스터 환경은 잘 정비된 지역적 네트워크 환경으로 구성하기 때문이다. 하지만, 인터넷 기반의 클러스터 환경에서는 네트워크의 문제로 인한 전송지연이나 연결단절이 있을 수 있고, 프로그램 수행도중 노드의 고장으로 인한 결함이 발생할 가능성이 높다.

따라서, 이러한 문제를 해결하기 위하여 여기에서는 중복된 작업할당 방법을 통하여 결함허용성을 제공하도록 한다. 할당된 작업을 모두 마친 종노드에 대하여 성능에 비해 '전송/수행중'인 작업이 많은 종노드의 작업을 중복수행하도록 함으로써 결함허용성을 제공할 수 있다. 임의의 종노드 i로부터 schedule[j].job[k]의 부분적인 결과 데이터를 전송 받았을 때, 주노드는 결함허용을 위하여 부분적으로 다음과 같이 i번째 종노드에 전송할 다음 작업을 아래의 서브 함수와 같이 선정한다.

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

1  if (schedule[i].remain == 0 && schedule[0 ··· (P-1)].remain
       == 0) {
2     m = 0 ;
3     max_doing = schedule[0].doing × schedule[0].weight ;
4     for (index = 1 ; index < P ; index++) {
5        temp_doing = schedule[index].doing
                        × schedule[index].weight ;
6        if (max_doing < temp_doing) {
7           max_doing = temp_doing ;
8           m = index ;
9        }
10    }
11    for (index = ⌊ log2 N ⌋ -1 ; index > 0 ; index--) {
12       if (schedule[m].status == 1) {
13          n = index ;
14          break ;
15       }
16    }
17 return (schedule[m].job[n]) ;
```

해당 종노드의 작업이 모두 전송되었고 다른 종노드에도 '미수행'작업이 없는 경우(1행)에는 성능(weight)에 비해 가장 많은 작업량을 남겨두고 있는 종노드를 선택한 후(2~10행), 스케줄러 상의 '전송/수행중'인 작업중 가장 마지막 작업을 선택한다(11~16행).

이와 같이 작업을 분배하는 경우에는 여러 종노드들이 하나의 작업을 공통으로 수행할 수 있다. 따라서, 주노드는 중복된 결과 중 가장 빠르게 도착한 결과만을 취하고 나머지는 무시한다. 중복된 작업 수행은 고장에 대한 결함을 허용할 수 있는 효율적인 방법 중의 하나이다[19]. 임의의 종노드에 고장이 발생하였을 경우, 주노드는 이 종노드로부터 어떠한 응답도 전송받을 수 없다. 이는 해당 프로그램 또는 시스템 차원에서의 추가적인 작업들을 수행해야 하는 불편함을 감수해야만 한다. 따라서, 중복수행이라는 간단한 메커니즘을 이용하여 결함을 허용하는 방법이 효과적임을 알 수 있다.

3.5 Expanded-WF 알고리즘

여기서는 앞에서 설명한 내용을 기반으로 확장된 Weighted Factoring(Expanded-WF) 알고리즘에 대하여 살펴보도록 한다.

Expanded-WF 알고리즘의 설계를 위한 기본적인 아이디어는 앞 절의 부하공유 기법(FOR-LOAD-SHARING)에 결함허용을 위한 서브 함수(SELECT_JOB_for_FT)를 적용하는 것이다. FOR-LOAD-SHARING 알고리즘은 상대적으로 느린 종노드의 작업을 빠른 종노드가 대신 수행하는 적응할당정책(SELECT_JOB_for_LS)에 네트워크시간과 계산시간을 겹치도록 하는 개선된 고정 분할 단위 알고리즘을 적용하며, 결함허용을 위하여 중복수행(SELECT_JOB_for_FT) 하도록 작업을 분배한다.

Expanded-WF 알고리즘에서 주노드는 부하공유와 결함허용을 위하여 임의의 종노드에 전송할 작업 선정을 다음의 3 단계에 걸쳐 수행한다.

① 1 단계 : 해당 종노드의 수행되지 않은 작업이남아있는 경우, 스케줄러에 남아 있는 해당 종노드의 다음 작업을 선정한다.

② 2 단계 : 해당 종노드의 '미수행'작업이 없고 다른 종노드에는 '미수행'작업이 남아있는 경우, 가중치(성능)에 비해 미수행된 작업의 비율이 가장 높은 종노드를 선택하고 그 종노드의 스케줄러상의 미수행 작업 중 가장 마지막 작업을 선정한다.

③ 3 단계 : 모든 종노드의 미수행 작업이 없는 경우, 가중치(성능)에 비해 수행되고 있는 작업의 비율이 가장 높은 종노드의 작업중인 가장 마지막 작업을 선정한다.

이러한 Expanded-WF 알고리즘은 다음과 같다.

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

1  EVALUATE (performance of slave nodes) ;
2  CREATE (scheduler by Weighted Factoring algorithm) ;
3  SEND (schedule [0 … (P-1)].job [0], 0 … (P-1)) ;
4  SEND (schedule [0 … (P-1)].job [1], 0 … (P-1)) ;
5  while (all partial results are not gathered by master node) {
6     RECEIVE (schedule [j].job [k], i) ;
7     if (schedule [i].remain != 0 or Exist
          schedule [0 … (P-1)].remain != 0) {
8         schedule [m].job [n] = SELECT_JOB_for_LS
          (schedule [j].job [k], i) ;
9     }
10    else {
11        schedule [m].job [n] = SELECT_JOB_for_FT
          (schedule [j].job [k], i) ;
12    }
13    SEND (schedule [m].job [n], i) ;
14    MERGE (pratial result data of schedule [j].job[k] to the
                result) ;
15 }
```

주노드는 초기 수행된 종노드의 성능평가에 의해 결정된 가중치를 이용하여 각 종노드마다 Weighted Factoring 알고리즘에 의한 스케줄러를 운영한다(1, 2행). 주노드는 처음 하나의 작업을 각 종노드에게 보낸 후(3행), 응답을 기다리지 않고 미리 두 번째 작업을 보냄으로써 개선된 고정 분할 단위 알고리즘을 적용한다(4행). SELECT_JOB_for_LS 서브함수를 이용하여 상대적으로 느린 종노드의 작업을 빠른 종노드가 수행하게 하는 적응할당정책을 적용하고, SELECT_JOB_for_FT 서브함수를 이용하여 작업을 중복 수행함으로써 결함허용을 제공한다.

3.6 Expanded-WF 알고리즘 분석

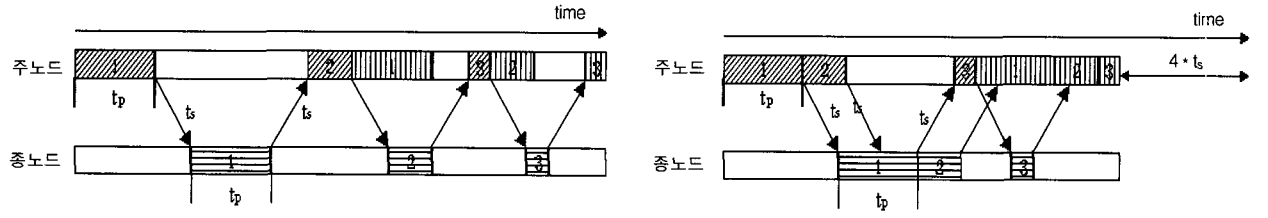
Expanded-WF 알고리즘에서는 총 수행시간을 줄이기 위해서 네트워크시간과 계산시간을 겹칠 수 있도록 다음의 작업을 미리 전송하도록 하였다. 이렇게 수행하였을 때 얻을 수 있는 총 수행시간의 이득은 주노드 및 종노드가 하나의 작업에 대해 계산을 수행하고 있는 시간에 대해 다른 작업이 전송되는 네트워크시간이다. (그림 1)은 하나의 작업에 대해 주노드와 종노드의 수행시간(t_p)이 같고 네트워크 전송시간(t_s)이 같을 때, 네트워크시간과 계산시간을 겹치지 않았을 경우와 겹치게 했을 경우의 총 수행시간 이득을 보여주는 예이다. (그림 1)(b) 경우와 같이 최대의 시간 이득을 위해서는 맨 처음 주노드에서 종노드로 전송되는 시간과 마지막 종노드에서 주노드로 전송된 시간을 제외한 모든 네트워크 전송시간이 계산시간과 겹쳐져야 한다.

따라서, N개의 데이터와 P개의 종노드에 대해 네트워크시간과 계산시간을 겹치게 함으로써 얻을 수 있는 총 수행시간의 최대 이득은 식 (3)과 같다.

$$t_{profit} = P * ((\lfloor \log_2 N \rfloor * 2t_s) - 2t_s) \qquad (3)$$

하나의 종노드에 대해 $\lfloor \log_2 N \rfloor$ 개의 작업이 할당되며, 각 작업은 주노드와 종노드 사이에서 2번씩 전송되게 된다. 여기에서 처음과 마지막의 전송시간을 제외하면 하나의 종노드에 대해 얻을 수 있는 최대 시간이득이 된다. 따라서, 이를 P개의 종노드에 대하여 적용하면, 식 (3)과 같은 결과를 얻을 수 있다.

Expanded-WF 알고리즘에서는 결함허용성을 위하여 마지막 작업들을 중복수행 한다. 중복된 작업량이 많아지면 종노드에 대한 효율성이 떨어지는 것으로 판단할 수 있다. 하지만, Expanded-WF 알고리즘에서는 다음 작업의 크기를 남아있는 작업크기의 절반으로 줄여나가므로 스케줄러의 마지막 작업은 전체 작업의 크기에 비해 상당히 작다. 대부분의 중복 수행된 작업들은 이러한 스케줄러의 마지막 부분에 해당하는 작업들이므로 중복 수행으로 인해 버려지는 데이터들은 전체 작업의 크기에 비해 매우 미비한 크기



(a) 네트워크시간과 계산시간을 겹치지 않았을 경우 (b) 네트워크시간과 계산시간을 겹쳤을 경우

▨ : send ▤ : make result ▥ : merge result.

(그림 1) 네트워크시간과 계산시간을 겹치게 함으로써의 시간이득

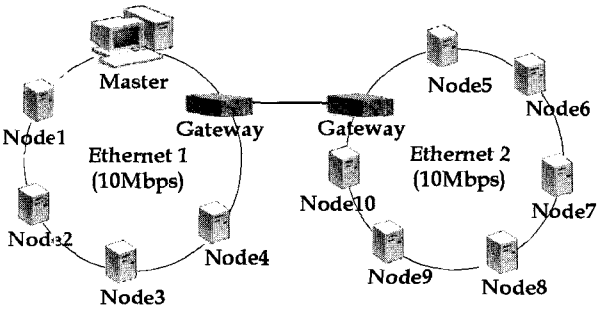
가 되기 때문에 이로 인한 소요시간은 거의 무시할 수 있다. 또한, Expanded-WF 알고리즘에서는 하나의 순간에 하나의 프로그램만을 수행하는 공간분할방식의 부하공유를 수행하므로 스케줄러에 작업이 남아있지 않은 종노드는 유휴(idle 한 종노드로 판단한다. 따라서, 오히려 성능이 좋은 종노드가 더 빨리 작업을 완료함으로써 현재의 작업에 대한 더욱 빠른 결과 응답시간을 얻을 수도 있으므로 중복된 작업을 수행하는 것이 오히려 전체적인 계산시간을 단축시킬 수도 있다.

4. 성능 평가

여기에서는 3장에서 제시한 Expanded-WF 알고리즘이 인터넷 기반 클러스터 환경에서 다른 알고리즘보다 효율적임을 보이기 위해 동일한 환경 하에서 Send 알고리즘, GSS 알고리즘, 그리고 Weighted Factoring 알고리즘과의 수행시간을 비교하였다. 수행시간의 비교를 위해 많은 응용 분야에서 이용되는 연산인 행렬 곱셈을 이용하였다. 두 개의 행렬을 곱셈하는 고성능 클러스터 실험 프로그램은 PVM3.4.4 라이브러리를 이용하여 작성하였다.

4.1 실험 환경

실험을 위하여 (그림 2)와 같이 총 11개의 컴퓨터를 이용하여 클러스터 환경을 구성하였으며, 이를 기반으로 프로그램을 수행하였다.



(그림 2) 실험 환경

하나의 주노드와 10개의 종노드로 구성하였으며, 인터넷 기반의 분산된 환경을 위하여 두 개의 네트워크에 분산시켜 배치하였다. 그리고 각 종노드의 가중치(weight)의 차이를 늘리기 위해 임의적으로 성능이 낮은 노드들을 주노드와는 다른 네트워크에 위치하도록 하였다. (그림 2)에서의 각 노드들의 이름은 <표 1>의 하드웨어/소프트웨어 구성표에 따른다. 이기종의 환경을 구성하기 위하여 각 노드의 성능을 다르게 설정하였으며, 운영체제도 Linux와 Solaris를 혼합하여 배치하였다.

주노드는 A. Piotrowski의 연구[11]에서 얻어진 결과에 따라 전체적인 계산성능을 향상시키기 위하여 비교적 낮은 CPU 성능의 PC를 선택하였으며, 주노드의 운영체제는 커널 및 네트워크성능에서 우수하다고 측정된 Linux를 선택하였다[5, 6].

〈표 1〉 하드웨어/소프트웨어 구성표

Node명	CPU	메모리	운영체제
Master	Pentium 3 450	320M	Linux(kernel 2.4)
Node 1	Pentium 3 733	128M	Linux(kernel 2.4)
Node 2	Pentium 3 733	128M	Linux(kernel 2.4)
Node 3	Pentium 3 450	128M	Linux(kernel 2.4)
Node 4	Pentium 3 300	128M	Solaris 8.0
Node 5	Pentium 3 300	128M	Solaris 8.0
Node 6	Pentium 3 450	128M	Linux(kernel 2.4)
Node 7	Pentium pro 133	64M	Linux(kernel 2.0)
Node 8	Pentium pro 133	32M	Linux(kernel 2.0)
Node 9	Pentium pro 133	32M	Linux(kernel 2.0)
Node 10	Pentium pro 133	16M	Linux(kernel 2.0)

4.2 실험 결과

4.1절에서 설명된 실험환경에서 각 알고리즘에 의하여 200×200, 300×300, 400×400, 500×500 크기의 행렬의 곱을 계산하는 프로그램을 수행하였다.

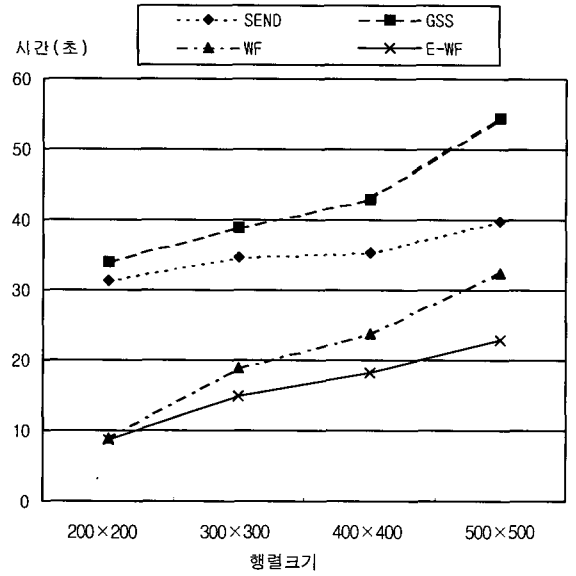
Weighted Factoring 알고리즘과 Expanded-WF 알고리즘의 수행시간 측정에서는 종노드의 성능평가와 스케줄러를 생성하는 시간을 제외하였다. 왜냐하면, 종노드의 성능평가에 의한 스케줄러의 생성은 프로그램 실행시 매번 필요한 것이 아니라, 클러스터를 구성할 처음에만 필요하기 때문이다. 따라서, 알고리즘의 총 수행시간은 실제로 종노드에 필요한 부분적인 데이터를 전송하면서 시작하여 종노드로부터 마지막으로 전송된 부분적인 결과 데이터가 주노드에 의해 병합된 시간까지 측정하였다.

〈표 2〉 데이터 크기별 알고리즘 수행시간

(단위 : 초)

알고리즘 / 행렬크기	Send	GSS	WF	E-WF
200×200	31.24	33.86	8.97	8.75
300×300	34.45	37.93	18.86	14.49
400×400	35.65	42.53	23.95	18.38
500×500	39.89	54.59	33.18	22.81

각 행렬의 크기별로 40회의 반복적인 프로그램 실행을 통하여 얻은 평균 실행시간(초)을 기록한 결과, <표 2>와 (그림 3)과 같은 결과를 얻을 수 있었다. 여기서 WF는 Wegithed Factoring 알고리즘을 의미하며, E-WF는 Expanded-WF 알고리즘을 말한다.



(그림 3) 데이터 크기별 알고리즘 수행시간

(그림 3)의 실험결과에 의하면, 데이터의 크기가 작을 때에는 비슷한 성능을 보이던 Weighted Factoring 알고리즘이 데이터의 크기가 커질수록 Expanded-WF 알고리즘보다 성능이 저하되는 현상을 보인다. 이는 초기에 평가된 종노드의 성능이 변화가 심한 인터넷 환경에서는 적합하지 못하기 때문이다. 따라서, 데이터의 크기가 커져 메시지 전송의 횟수가 많아질수록 종노드의 상태 변화에 동적으로 적응할 수 있도록 개선된 Expanded-WF 알고리즘이 높은 성능을 보인다.

행렬의 크기별 성능향상에 대한 평균 값으로 결과를 분석해 보면, Expanded-WF 알고리즘이 Send 알고리즘보다 55%, GSS 알고리즘보다 63%, WF 알고리즘보다 20%의 성능향상을 보임을 알 수 있다.

Expanded-WF 알고리즘은 적응할당정책과 개선된 고정 분할 단위 알고리즘을 동시에 적용하여 성능 향상을 도모하였다. 이러한 Expanded-WF 알고리즘의 성능향상에 대하여 적응할당정책이 주는 효과와 개선된 고정 분할 단위 알고리즘이 주는 효과를 각각 분리하여 알아보았다. 동일한 실험환경에서 Weighted Factoring 알고리즘에 각각의 기법을 분리하여 적용한 40회의 실험 결과는 다음의 <표 3>과 같다.

<표 3>의 결과를 <표 2>의 Weighted Factoring 알고리즘의 결과와 비교해 보면 평균적으로 적응할당정책만을 적용함으로써 약 5%의 성능향상을 보였고, 개선된 고정 분할 단위 알고리즘만을 적용함으로써 약 13%의 성능향상을 나타냈다. 이러한 결과로 볼 때, 인터넷 기반의 클리스터 환경에서는 노드의 이질성 보다는 네트워크를 통한 통신시간이 총 계산시간에 더욱 많은 영향을 미친다는 것을 알 수 있다.

〈표 3〉 부하공유 기법 적용에 따른 수행시간

(단위 : 초)

부하공유 기법 / 데이터크기	적응할당 정책만 적용	개선된 고정 분할 단위 알고리즘만 적용
200×200	8.92	8.80
300×300	17.96	16.39
400×400	22.64	20.54
500×500	30.17	25.98

<표 2>의 결과에서는 Expanded-WF 알고리즘의 성능향상이 Weighted Factoring 알고리즘보다 20%의 성능이 향상된 것으로 보였다. 따라서, 두 개의 부하공유기법을 각각 분리하여 적용하는 것보다 동시에 적용함으로써 더욱 좋은 성능향상을 가져올 수 있음을 알 수 있다.

Expanded-WF 알고리즘의 결함허용성을 측정하기 위하여 <표 3>에서의 최대 수행시간(54.59초)보다 큰 60초의 delay를 임의의 하나의 종노드에 적용하여 결함 허용성을 실험하였다. 여기에서 하나의 노드에 60초의 delay를 주는 것은 그 노드가 결함이 발생했다는 것을 의미한다. 각 행렬의 크기별로 40회의 반복적인 프로그램 실행을 통하여 얻은 평균 실행시간과 정상상태에서의 수행시간을 비교한 결과 <표 4>과 같은 결과를 얻을 수 있었다.

〈표 4〉 데이터 크기별 결함허용성 측정 수행시간

(단위 : 초)

종노드 상태 / 데이터크기	결함허용 (60초 delay 적용)	정상 상태
200×200	9.47	8.75
300×300	15.38	14.49
400×400	18.93	18.38
500×500	23.26	22.81

이 실험에서 Expanded-WF 알고리즘이 결함허용성을 제공하지 못한다고 가정한다면, 실험의 결과는 모두 60초 이상이 되어야 한다.

<표 4>의 실험결과에 의하면, 임의의 종노드에 결함이

발생하였음에도 불구하고 정상상태와 비교하여 5% 정도의 성능저하를 보이고는 있지만 정상상태와 비슷한 수행시간을 나타냄을 알 수 있다. 이로써 Expanded-WF 알고리즘이 결함허용에도 효과적으로 대처할 수 있음을 알 수 있다.

5. 결론 및 향후 과제

본 논문에서는 인터넷 기반 클러스터 환경에서 효과적인 부하공유 및 결함허용을 위하여 Expanded-WF 알고리즘을 제안하였다. Expanded-WF 알고리즘은 Weighted Factoring 알고리즘을 기반으로 효과적인 부하공유를 위하여 느린 종노드의 작업을 빠른 종노드가 대신 수행하는 적응할당정책을 적용하는 동시에 네트워크 통신시간과 계산시간을 겹치게 하였으며, 결함허용을 위하여 작업을 중복 수행하도록 하였다.

인터넷 기반 클러스터 환경에서의 Expanded-WF 알고리즘의 성능을 측정하기 위하여 두 개의 분산된 네트워크상에서 PC를 이용하여 이기종의 클러스터를 구축하고 PVM을 이용하여 200×200, 300×300, 400×400, 500×500의 크기에 해당하는 행렬의 곱셈 프로그램을 수행하였다. 실험의 결과를 행렬의 크기별 성능향상에 대한 평균 값으로 분석해 본 결과, 본 논문에서 제안한 Expanded-WF 알고리즘이 NOW 환경에서 효율적인 Send 알고리즘보다 55%, GSS 알고리즘보다 63%의 성능향상을 보였으며, Weighted Factoring 알고리즘보다도 20%의 성능향상을 보였다. 또한, 다른 알고리즘에서 제공할 수 없었던 안정적인 수행시간에 대한 결함허용성을 제공할 수 있음을 보였다.

인터넷 기반 클러스터 환경은 NOW 환경보다 더욱 동적이다. 이러한 환경에서 효율적인 부하공유를 위해서는 클러스터를 구성하는 노드들의 성능뿐만 아니라 네트워크의 성능까지 고려해야 하며, 결함허용을 위하여 노드의 고장 및 네트워크의 고장에 효과적으로 대처해야만 한다. 따라서, 향후에는 더욱 다양한 환경에서 적용이 가능한 부하공유기법에 대한 연구와 결함허용성을 제공하는 기존의 툴들과의 호환성 있는 결함허용기법에 대한 연구를 수행할 계획이다.

참 고 문 헌

[1] 강나영, 정상화, 장한국, "효율적인 정보 검색을 위한 VIA기반 PC 클러스터 시스템", 정보과학회논문지, Vol.29, No.10, 2002.

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

[3] 김선재, "VIA 기반의 병렬 라이브러리 구현 및 성능 평가", 서울대학교 공학석사학위논문, Dec., 1999.

[4] 김지형, 김동승, "저속 네트웍 PC 클러스터상에서 NOW-Sort의 성능향상", 정보과학회논문지, Vol.29, No10, 2002.

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

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

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

[8] 정훈진, 정진하, 최상방, "네트워크 기반 클러스터 시스템을 위한 적응형 동적 부하균등 방법", 정보과학회논문지, Vol.28, No.11, 2001.

[9] 한국과학기술원 컴퓨터구조연구실 NRL 프로젝트팀, "클러스터 시스템을 위한 SSI 지원 기술들에 관한 조사 보고서", Jan., 2001.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, "PVM : Parallel Virtual Machine-A User's Guide and Tutoraial for Networked Parallel," The MIT Press, 1994.

[11] 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.

[12] G. Pfister, "In Search of Clusters," 2nd Edtion, Prentice Hall, 1998.

[13] G. Shao, "Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources," PhD thesis, UCSD, June, 2001.

[14] I. Banicescu and V. Velusamy, "Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring," IPDPS 2001, IEEE Computer Society Press, San Francisco, 2001.

[15] IEEE Task Force on Cluster Computing(TFCC), http://www.ieeetfcc.org.

[16] Message Passing Interface Forum, "MPI : A Message-Passing Interface Standard," May, 1994.

[17] R. Buyya, "High Performance Cluster Computing," Prentice Hall, Vol.1, 1999.

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

[19] 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.

최 인 복

e-mail : pluto612@dku.edu

1999년 단국대학교 전자계산학과 학사

2002년 단국대학교 대학원 전자계산학과 석사

2002년~현재 단국대학교 대학원 컴퓨터과학및통계학과 박사과정

관심분야 : 분산 및 병렬처리, (모바일)인터넷 기술, 컴퓨터 네트워크

이 재 동

e-mail : letsdoit@dku.edu

1985년 인하대학교 전자계산학과 학사

1991년 Cleveland State Univ. Dept. of Computer & Information Science (M.S.)

1996년 Kent State Univ. Dept. of Computer Science (Ph.D.)

1987년~1988년 대우중공업 정보관리실 시스템 분석

1992년~1996년 Kent State University R.A & T.A

1996년~1997년 (주)두루넷 기술기획팀

1997년~현재 단국대학교 정보컴퓨터학부 교수

관심분야 : (Mobile) Internet Technologies/Applications, High Performance Computing (Clustering systems etc.), GIS Technologies and Applications, Many aspects of parallel/distributed processing