제20회 한국정보처리학회 추계학술발표대회 논문집 제10권 제2호 (2003. 11)

분산처리시스템에서의 효율적인 동적부하균등화 방법

김명규*, 채수환*
*한국항공대학교
e-mail:kimmk@mail.hangkong.ac.kr

Efficient Dynamic Load Balancing on Distributed Computer Systems

Myung-Kyu Kim*, Soo-Hoan Chae*
*Dept of Computer Engineering, Hankuk Aviation University

요 약

네트워크 시스템이 발달하면서 다양한 컴퓨터들을 연결하는 클러스터링 시스템 구축이 용이해졌다. 이러한 이기종 클러스터 환경을 구축함에 있어서 노드들간의 성능 불균형으로 인한 문제가 야기되는데 본 논문에서는 Message Passing 방식을 이용한 클러스터링을 구축함에 있어서 노드들의 자원의 정보를 이용하여 메모리의 과부하를 최대한 예방하여 작업을 메모리 여유가 있는 노드로 이주시킴으로써 시스템 안정성과 자원을 균등하게 사용할 수 있도록 제안하였다. 제안한 알고리즘을 구현하기 위해서 이기종 클러스터 환경에서 MPI를 이용하여 2차원 열에너지 전도 계산과 Matrix 곱셈 프로그램을 이용하여 제안한 알고리즘과 GSS, Send 알고리즘, Weighted Factoring알고리즘들과 상대 비교를 하였다.

1. 서론

단일 컴퓨터들의 뛰어난 성능 향상과 고속 네트워크의 보급으로 다양한 성능의 컴퓨터들을 네트워크상에서 TCP/IP로 연결하여 클러스터링을 구성하고 있다.

클러스터링 시스템은 네트워크로 연결되어 있으면서도 자체적으로 처리 능력을 갖는 프로세스들로서 높은 성능, 유용성 및 낮은 가격으로 확장 가능하다는 장점을 갖고 있어서 HPC(High Performance Computing) 서버나 대규모 병렬 컴퓨터들을 대체하고 있다. 이러한 클러스터링 시스템의 성능을 극대화하기 위해서는 성능에 가장 영향을 미치는 CPU, 메모리, 각종 I/O 장치의 자원을 효율적으로 이용하여 시스템의 전체 부하를 각 프로세서에 균등하게 유지시키고 처리율을 극대화하여 평균 반응 시간을 최소화시키는 작업의 스케줄링이 중요하다.

본 논문에서는 클러스터링 시스템에서 각 노드들이 가지고 있는 CPU 종류와 메모리 크기가 다른 이질의 시스템에서 노드들의 자원 성능을 고려하여 작업을 적절하게 할당할 때 메모리 부족으로 발생되는 페이지폴트 발생을 최대한 줄이고 각 노드들의 CPU의 유휴 시간을 최대한 줄여서 전체적인 성능을 향상과 더불어 시스템 안정화 시키는 작업부하 균등화 정책에 대해 제안하고자 한다[7][8].

4장에서 메모리 부하 정보를 재분배를 통해서 전체 노드들의 부하가 균등하기 할당하기 위해, CPU 성능 및 메모리 정보를 수집을 좀 더 효율적으로 하기 위해서 리눅스 어플리케이션의 Top과 같은 기능을 [그림 1]을 이용하여 정보를 수집하였다.

2. 관련 연구

프로세서를 효율적으로 활용하기 위해 작업을 분배하는 부하균등화 방법으로는 크게 정적부하균등화와 동적부하균등화 기법이 있다. 정적부하균등화는 작업의 예상 소요 시간을 가능한 경우에 각 노드에 작업을 미리 할당하는 방법이다. 이에 반해 동적부하균등화는 작업량을 예측하기가 어려울 때나 개별 작업의 수행 시간의 차이가 심한 경우에 유용한

```
#define DEF_PROCSTAT_BUFFER_SIZE (1 << 10)
int CPU_Usage(int s_Argc, char **s_Argv)
{
   const char *s_ProcStat = "/proc/stat";
   const char *s_CPUName = "cpu ";
   int s_Handle, s_Check, s_Count;
   char s_StatBuffer[ DEF_PROCSTAT_BUFFER_SIZE ];
   char *s_String;
   unsigned long s_CPUTick[2][5];
   float s_DiffTotal;
   memset(&s_CPUTick[0][0], 0, sizeof(s_CPUTick));
do
{
   s_Handle = open(s_ProcStat, O_RDONLY);
   if(s_Handle >= 0)
   {
    s_Check = read(s_Handle, &s_StatBuffer[0], sizeof(s_StatBuffer) - 1);
    close(s_Handle);
  if(s_Check > 0)
  {
   s_StatBuffer[s_Check] = '\0';
   s_String = strstr(&s_StatBuffer[0], s_CPUName);
   if(s_String)
    {
     s_Check = sscanf(s_String, "cpu %lu %lu %lu %lu", \
  &s_CPUTick[0][0], &s_CPUTick[0][1], &s_CPUTick[0][2], &s_CPUTick[0][3]);
    if(s_Check == 4)
    {
      for(s_Count = 0, s_CPUTick[0][4] = 0lu;
       s_Count <    4;s_Count++)s_CPUTick[0][4] += s_CPUTick[0][s_Count];
       s_DiffTotal = (float)(s_CPUTick[0][4] - s_CPUTick[1][4]);
    if(s_DiffTotal > 0.0)
    {
fprintf(stdout, "[CPU] User=%1.2f%%, Nice=%1.2f%%, System=%1.2f%%,
Idle=%1.2f%%\n", \
(float)( (float)((s_CPUTick[0][0] - s_CPUTick[1][0]) * 100lu) / s_DiffTotal ),\
(float)( (float)((s_CPUTick[0][1] - s_CPUTick[1][1]) * 100lu) / s_DiffTotal ),\
(float)( (float)((s_CPUTick[0][2] - s_CPUTick[1][2]) * 100lu) / s_DiffTotal ),\
(float)( (float)((s_CPUTick[0][3] - s_CPUTick[1][3]) * 100lu) / s_DiffTotal ));
memcpy(&s_CPUTick[1][0], &s_CPUTick[0][0], sizeof(s_CPUTick) >> 1); }
sleep(1);
}while(1);
return(0);
}

int meminfoc
(char *page, char **start, off_t off, int count, int *eof, void *data)
{
            struct sysinfo i;
            int len;
            unsigned int cached;
            cached =      atomic_read(&page_cache_size) -
atomic_read(&shmem_nrpages);
#define K(x) ((x) << (PAGE_SHIFT - 10))
#define B(x) ((x) << PAGE_SHIFT)
si_meminfo(&i);
            si_swapinfo(&i);
            len = sprintf(page, "        total:    used:    free:
              shared: buffers:  cached:\n"
                        "Mem:  %8lu %8lu %8lu %8lu %8lu %8u\n"
                        "Swap: %8lu %8lu %8lu\n",
                        B(i.totalram), B(i.totalram-i.freeram),
                B(i.freeram),
                        B(i.sharedram), B(i.bufferram),
                        B(cached), B(i.totalswap),
                        B(i.totalswap-i.freeswap), B(i.freeswap)
            len += sprintf(page+len,
                        "MemTotal:     %8lu kB\n"
                        "MemFree:      %8lu kB\n"
                        "MemShared:    %8lu kB\n"
                        "SwapTotal:    %8lu kB\n"
                        "SwapFree:     %8lu kB\n"
                        K(i.totalram),
                        K(i.freeram),
                        K(i.sharedram),
                        K(i.freeram-i.freehigh),
                        K(i.totalswap)
            return proc_calc_metrics(page, start, off, count, eof, len);
}
```

[그림 1] CPU 사용률 및 메모리 정보

방법으로 전체 작업의 일부분만을 초기에 배정하고 일찍 작업을 마친 노드에 추가로 작업을 할당하는 방법이다. 본 연구는 이질 환경 시스템이어서, 노드마다 수행시간이 많은 차이날 것이라 예상 되기 때문에 정적 부하 균등화 방법보다는 동적 부하 균등화 방법만을 고려하였다.

동적 부하 방법에서 사용하는 정책으로 고정, 가변, 적응 정책으로 나눌 수 있다.

2.1 고정 정책

고정 정책에 일반적으로 사용되고 있는 알고리즘은 Send 알고리즘이다. 예를 들어 Matrix의 내적 벡터 연산을 Send 알고리즘을 이용하여 계산할 때 먼저 각 슬래이브(slave) 노드에 순서적으로 첫 번째 Matrix와 두 번째 Matrix의 전체의 N열 중 일정한 열(t)을 보내서 계산한다. 계산을 마친 노드는 마스터(master) 노드에 결과값을 보내고 마스터 노드는 적절한 장소에 결과값을 취합한 뒤 다시 슬래이브 노드에 일정한 열을 보내는 것을 계산이 마칠 때까지 반복한다.

$$G_i = t \qquad (1)$$

2.2 가변 정책

가변 정책에 사용되고 있는 알고리즘은 대표적으로 GSS(Guide-Self Scheduling)이다[1].

GSS 알고리즘은 작업을 요청하는 프로세서에게 남아있는 작업량의 일정비율을 할당하는 스케줄링인데 연산량이 많은 작업들이 먼저 작업을 수행할 경우 일부 노드에 집중이 되어 나머지 노드들과 작업의 부하 불균형이 심하게 된다. 이를 방지하기 위해서 최대 한계치를 주는 방법을 적용하였다. 또한 반대로 작업이 진행 될수록 적은 양의 데이터를 연속적으로 받기 위해서 작업을 수시로 요청하여 오버헤드를 야기시키는 요인이 될 수 있으므로 노드에게 한계치 이하의 작업이 할당하지 않도록 하위 한계치(3열)를 두었다.

$$G_i = \left(1 - \frac{1}{p}\right)\frac{N}{P} \qquad (2)$$

2.3 적응 정책

Flynn Hummel의 WF(Weighted Factoring) 알고리즘이 있다. WF 알고리즘은 종노드의 계산 성능의 비율에 따라 가중치를 부여하고 그 가중치에 따라 반복적으로 데이터의 크기를 동적으로 줄여나간다. 전체 열의 크기가 N개인 Matrix연산에 대하여 i번째 묶음에 있는 j번째 데이터 덩어리의 크기를 F_{ij}라 고하면, F_{ij}는 다음과 같이 결정된다[2].

(3)

$$F_{ij} = \frac{1}{2} \quad {}^{i+1}\frac{N}{x} \frac{W_i}{\sum_{k=1}^{k=p} W_k}$$

3. 실험 환경

클러스터 시스템에서 일반적으로 멀티프로그램을 수행하기 때문에 대부분 라운드 로빈 방식을 선택하지만 본 실험에서는 고정 방식인 Send 알고리즘, 가변 정책인 GSS(Guided Self-Scheduling) 알고리즘과 적응 정책에서는 가중치 알고리즘을 CPU 중심의 가중치 방식에 메모리 크기를 고려한 제안된 알고리즘과 비교 분석하였다.

위의 실험을 위해서는 ULTRA SPARC60 워크스테이션 1대를 마스터 노드로 하고 6대의 슬래이브(slave) Linux PC를 100bps Ethernet으로 연결한 상태에서 MPI를 하여 간단한 NOW 환경을 만든 후 임의의 크기의 Matrix를 일정한 간격으로 20개 생성시켜서 알고리즘별로 Matrix 연산을 수행하였다.

이기종 시스템을 구성하기 위하여 6대의 PC는 구성은 <표 1>과 같이 구성하였다.

<표 1> 슬래이브 시스템 구성도

CPU 속도(MHz)	MFLOPS	Memory(MB)	Linux 버전
933	310	512	2.4.7-10
933	310	128	2.4.7-10
800	280	512	2.4.18-3
800	280	256	2.4.18-3
800	280	128	2.4.7-10
500	160	384	2.4.18-3

4. 연구 접근 방향

4.1 CPU 정책

CPU 자원의 작업을 균등하게 할당하기 위하여 노드의 CPU 처리 속도(FLOPS)를 기준으로 초기의 작업을 할당하고, 슬래이브 노드가 마스터 노드에게 메모리 위험 메시지를 보내기 전까지는 마스터 노드는 식(3)을 바탕으로 노드에 가중치를 CPU에 중심으로 가중치를 계속적으로 알파(α)를 조절하여 유휴 노드에 대한 부하를 조절하였다.

4.2 메모리 영향도

먼저 메모리가 시스템 성능에 어느 정도 영향을 미치는가를 알아보기 위해서 입력값으로 2차원 열 전도 문제를 Jacobi Iteration 방법으로 오차 범위를 0.0001로 주고 병렬 프로그램을 수행한 실행 시간을 <표 2>에 나타내었다.

<표 2> CPU 800 시스템하에서 메모리 영향도

(단위는 Second, ∞는 시스템 Out)

	128M (case Ⅰ)	256M (caseⅡ)	512M (caseⅢ)	비고
4000 x 4000 (113M)	4154	4129	4067	
5000 x 5000 (195M)	189360	9327	9322	case Ⅰ page-fault 발생
6000 x 6000 (239M)	∞	18776	18791	
8000 x 8000 (431M)	∞	265666	26066	case Ⅱ page-fault 발생
10000x10000 (524M)	∞	∞	47701	

<표 2>에서 볼 수 있듯이 페이지폴트(page fault)는 노드의 가용 공간보다 작업의 메모리 요구량이 클 때 메모리 과부하로 발생하여 최악의 경우 시스템이 정지되거나 이미 실행하고 있던 작업들의 실행시간에 막대한 지장을 발생시킨다.

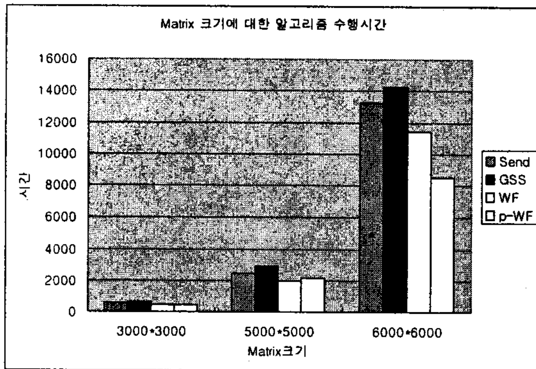
따라서 메모리의 한계점을 메모리의 전체 크기에서 시스템 파일이 사용하는 메모리 크기를 제외한 나머지 가용 메모리 크기를 얼마만큼 정하느냐에 따라 시스템의 성능 향상에 중요한 영향을 끼칠 수 있다

메모리 한계치를 정함에 있어서 작업의 노드에 할당하고 수행되어야 할 작업의 메모리 요구량을 알 수 있다고 발표되었지만[3], 실제로 작업의 메모리 요구량을 미리 예측하는 일은 상당히 어려운 일이다. 본 논문에서는 메모리 요구량을 미리 측정하는 어려움에 있어서 메모리의 한계치를 두어 노드에 입력길이를 줄여야 하는데, 메모리 한계치 크기를 논문 [6]에 의해 메모리 한계치와 메모리 크기가 같다고 가정하는 것은 짧은 기간동안 메모리에서 실행하고 있는 작업들이 일부가 끝나서 페이지 폴트가 끝날 것이라고 예상하는 것으로 페이지 폴트율의 비용보다 작업 이주의 비용이 저렴하다고 가정했기 때문이다. 이는 일반적인 시스템 작업 할당 구조가 파이프라인(pipeline)구조[4]일 경우이고 현재 실험한 구조와 같이 Matrix 계산과 같은 경우에는 논문 [5]에 의해 메모리 한계점에 따른 성능 비교 결과를 참고하여 메모리 한계점을 60%으로 정하였다.

4.3 제안한 알고리즘 및 결과치

Matrix 계산은 작업 도중에는 다른 노드와 데이터를

교환이 필요 없는 trivial parallelism[4]이므로 시스템 성능 측정에 많이 사용되는 slowdown 이나 페이지 폴트보다는 전체 수행 시간으로 시스템 성능을 측정하였다. 또한 메모리 요구량을 미리 알수가 없어서 각 슬레이브 노드가 마스터 노드에게 신호(signal)을 보내는 것을 [5]에 의해서 60%으로 정하고 한계치가 넘었을 때 마스터 노드에게 신호(signal)을 주어서 마스터 노드가 위험 수치를 넘는 노드에 대해서는 CPU중심이 아니라, 메모리 여유 공간을 중심으로 가중치 알파(α)를 주어서 노드에 할당하였다.



[그림 2] Matrix 크기에 대한 각 알고리즘 수행시간

[그림 2]에서 알수 있듯이 메모리 요구량이 작은 것에 WF가 앞서지만 메모리 요구량이 큰 환경에서는 CPU의 유휴 시간을 좀 더 많아지지만 페이지 폴트의 감소로 인해서 실제적인 계산 능력은 제안한 알고리즘이 뛰어나다는 것을 알 수 있다.

5. 결론

본 논문에서는 클러스터링 시스템에서 각 노드들이 가지고 있는 CPU 성능과 메모리 크기가 다른 시스템에서 각 노드의 자원 성능을 고려하여 작업을 적절하게 할당하는 작업부하 균등화 정책을 제안하였다. 노드에 메모리 크기를 미리 알 수 있다면 좀 더 효과적인 동적 부하 알고리즘을 해결할 수 있겠지만 CPU의 큐(queue)에 들어오는 작업의 크기를 알 수 없어서 빠른 처리와 페이지폴트를 최소화하여 시스템의 안정적인 면을 고려하였다.

제안한 알고리즘이 CPU와 메모리 성능의 차가 심한 이기종 클러스터 환경에서 효율적인 것으로 나타났다. 인터넷 기반의 분산 시스템은 이 제안 알고리즘을 적용한 보다 동적이고 시스템 결합에 치명적이기 노드의 결함 허용(fault tolerance)에 대한 연구가 좀 더 필요하다.

<Acknowledgement>

본 논문은 과학기술부. 한국과학재단 지정 경기도 지역협력연구센터(RRC)인 한국항공대학교 인터넷정보검색연구센터의 지원에 의한 것입니다.

참고문헌

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

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

[3] A.Barak and A.Braverman, "Memory ushering in a scalable computing cluster", Journal of Microprocessors and Microsystems, Vol.22, No.3-4, August 1998, pp.175-182.

[4] Sajjan G.Shiva "Pipelined and Parallel Computer Architectures", HarperCollinsCollege-Publishters, 1996

[5] 허말순, "클러스터 컴퓨팅 시스템에서 메모리 부하를 고려한 작업부하 균등화 정책", 한양대학교 석사학위논문, 2002

[6] X.Zhang, Y.Qu, and L.Xiao, "Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources," Proc. 20th Int'l Conf. Distributed Computing Systems, (ICDCS '2000), pp. d233-241, Apr.2000

[7] F.Douglis and J.Ousterhout, "Transparent Process Migration:Design Alternatives and the Sprite Implementation," Software Practice and Experience, vol.21,no.8,pp.757-785,1991

[8] X.Du and X.Zhang, "Coordinating Parallel Processes on Networks of Workstations," J.Parallel and Distributed Computing, vol.46,no.2, pp.125-135, 1997