다중처리 시스템에서 향상된 부하분산에 관한 연구

김 중 민, 유 재 욱, 박 인 갑
건국대학교 전자정보통신공학과
전화 : 02-450-3495 / 핸드폰 : 017-255-8787

A Study on Adbanced Load Balancing for Multiprocessor System

Joong-Min Kim, Jae-Wook Yu, In-Kap Park
Dept. of Electronics Engineering, Kon-Kuk University
E-mail : krdio@kkucc.konkuk.ac.kr

Abstract

In this paper, an advanced load balancing algorithm in nth order Hypercube distributed system has been proposed. The new algorithm uses centralized load-balancing to avoid blocking phenomenon and processor thrashing, and shows the results which makes loads to approach average value of loads. The new algorithm is compared with several other algorithm and it shows a merit in cost function value.

Ⅰ. 서론

점차적으로 고성능을 필요로 하는 응용분야의 증대로 고성능 시스템이 필요하게 되었다. 필요한 고성능을 얻기 위해 하나의 프로세서의 성능을 높이는 방법과 다수개의 프로세서를 연결하여 구현하는 분산 멀티프로세서 시스템(distributed multiprocessor system)을 사용할 수 있다. 그러나 하나의 프로세서만을 이용하여 고성능을 얻고자 할 경우 물리적인 한계로 인해 프로세서의 성능 향상에 한계가 있다. 그러므로 프로세서들을 병렬로 구성한 분산 시스템을 사용하게 되었다. 이상적인 분산 시스템은 각각 개별적인 부하들을 처리하므로 시스템의 성능은 프로세서의 개수에 따라 선형적으로 증가한다. 그러나 실제로는 실행 중 발생하는 부하 불균형 현상 등의 이유로 시스템의 규모가 커지는 만큼의 성능향상이 이루어지지 않는다.

본 논문은 다른 구조에 비하여 구조적인 적응성이 매우 높고, 노드와 노드사이의 통신로의 개수가 많으며, 이동거리가 비교적 짧은 n 차원의 하이퍼큐브(hypercube) 구조의 분산 시스템에서 새로운 부하분산 알고리즘을 제안하고 제안된 알고리즘의 성능 평가를 위하여 컴퓨터 모의 실험을 수행하였다.

Ⅱ. 부하분산 정의 및 방법

2.1 부하분산의 정의

부하분산은 통신부담을 예측하기 어려운 환경이나 동적인 실행 환경하에서 프로세서들에 할당된 부하의 양을 균일하게 만들어 전체적인 실행시간을 단축하는 방법이다. 그림 1은 부하분산 시스템을 설명한 것으로 한 개 이상의 프로세서로 구성되어 있는 분산 시스템에서 각 프로세서로 부하가 할당되어 실행대기 상태에 있을 때, 부하가 특정의 프로세서에 과다하게 주어졌을 경우 이러한 프로세서의 부하 중 일부를 부하가 적은 프로세서로 이동하여 실행함으로써 시스템 전체의 처리효율을 높이고 부하의 실행시간을 단축하는데 그 목적이 있다.

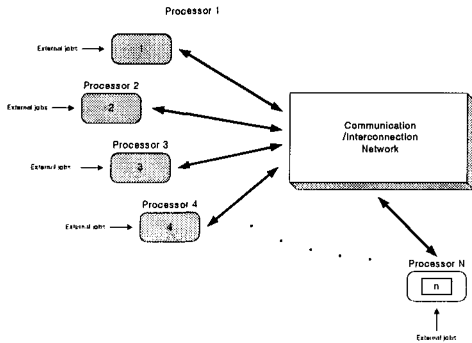

그림 3. 부하분산 시스템

2.2 부하분산의 방법

(1) 정적 부하분산

정적인 부하 이동은 시간의 경과에 따른 각 프로세서의 부하의 상태에 관계없이 영구적으로 특정의 프로세서를 그 부하분산 영역상의 서비스 프로세서로 설정하여 놓는 방법이다.

(2) 동적 부하분산

동적인 부하 이동은 시간의 경과에 따라서 부하분산 영역내의 프로세서의 역할이 부하의 상태 변화에 따라서 과부하 상태에 있을 때는 클라이언트로 작용하고 저부하 상태에 있을 때는 서버로 작용한다. 일반적으로 분산 시스템의 부하분산은 동적 부하분산을 말한다.

2.3 프로세서간의 연결 방법

각 노드간 기하학적 배치나 노드간의 연결을 topology라 한다. 각각의 노드는 하나 또는 여러 개의 프로세서로 구성되고 배치에는 고리(ring), 격자(mesh), 토러스(torus), 펫 트리(fat-tree), 하이퍼큐브(hypercube) 등의 방법들이 있다.

이 중 하이퍼큐브(hypercube)는 $N(=2^n)$ 개의 노드(Node : 분산 시스템에서는 프로세서에 해당)로써 n 차원의 이진 큐브를 형성하므로 binary n-cube 라고도 말한다. 하이퍼큐브에는 여러 가지 특성이 있다. 각 프로세서는 자신의 로컬 메모리를 가지고 있으며, n 차원의 하이퍼큐브일 경우 n 개의 다른 프로세서와 직접 연결되는 통신로를 갖는다. 하이퍼큐브의 차원이 3 이상일 경우에는 tree, grid, ring, torus 등 임의의 연결 구조를 갖는 시스템으로 노드 및 링크의 대응에 의한 전환이 가능하다. 따라서 하이퍼큐브는 다른 구조에 비하여 구조적인 적응성이 매우 높다.

Ⅲ. 개선된 부하분산 알고리즘

3.1 기존 알고리즘의 문제점 및 해결방안

(1) 기존의 알고리즘은 과부하된 노드의 부하량을 저부하 노드에게 모두 전송하거나 전혀 전송하지 않는 방법을 택하였다.
→ 과부하된 부하를 모두 전송하지 못하더라도 프로세서 trashing 현상이 발생하지 않는 범위 내에서 최대의 과부하 양을 전송한다.

(2) 최대 부하량을 가진 노드의 부하의 양이 전체 부하의 실행에 필요한 비용함수에 가장 큰 영향을 미친다. 부하분산이 모두 행하여지지 못할 경우(부하를 전송할 노드를 찾지 못하는 경우) 비용함수가 크게 증가하게 된다.
→ 부하량 sorting후에 부하의 최대값을 찾아서 최대값 노드의 부하만을 전송해 나간다. 그러므로 전체 노드간의 부하 량의 차이를 줄여감으로 전체 노드의 부하의 값이 평균값에 가깝도록 한다.

(3) 한 과부하 노드의 부하들은 정해진 경로의 저부하 노드에게만 과부하 노드의 부하를 전송한다.
→ 매 iteration(반복)마다 아직 저부하 상태인 노드 가운데 최단 거리인 곳으로 부하 이동하므로 동일 과부하 노드의 부하는 다수개의 저부하 노드로 분배되어 전송될 수 있다.

3.2 비용 함수

분산 시스템의 비용 함수 H는 식 3-2의 부하의 실행에 필요한 비용과 식 3-3의 데이터의 통신에 드는 비용의 합인 식 3-1로 표현할 수 있다. 표현될 수 있는데, 시스템의 성능을 향상시키기 위해서는 이 에너지, 즉 비용 함수를 최소화하도록 부하분산을 시켜야 한다.

$$H_{total} = H_{execute} + \tau H_{comm} \qquad \text{식 3-1}$$

$$H_{excute} = Max[W_1, W_2, \cdot\cdot\cdot, W_i] \qquad \text{식 3-2}$$

$$H_{comm} = \sum_i W_{i_{comm}} \qquad \text{식 3-3}$$

여기서 W_i은 i 번 노드의 부하량을, $W_{i_{comm}}$은 i 번 노드의 이동되어야 할 부하량을 의미한다. $H_{execute}$는 각각의 프로세서들이 같은 부하를 가질 때 최소화되며 H_{comm}은 통신시간이 작을 때 최대화된다. 하이퍼큐브의 차원이 n, 노드의 개수가 $N(=2^n)$이라면, 부하분산 이후의 비용 함수를 구하면 다음과 같다.

① $H_{execute}$는 노드의 부하량 W_i에서 저부하 노드로의 이동되는 부하량을 제외하거나 다른 노드로부터의

부하이동을 합한 것 중 최대 값이 되어 식 3-4로 표현 될 수 있다.

$$H_{execute} = Max[W_i - \sum_j M_{ij} \text{ for } i = 0, 1, \cdots, n] \quad \text{식 3-4}$$

② 노드마다의 거리가 다르므로 그 거리를 i 노드로부터 j 노드까지의 거리를 D_{ij}라고 하면 H_{comm}은 다음과 같다.

$$H_{comm} = \sum_i \sum_j M_{ij} \times D_{ij} \quad \text{식 3-5}$$

여기서 M_{ij}는 i node로부터 j 노드로의 이동된 부하량을 의미한다.

3.3 제안하는 개선된 알고리즘

초기의 부하량을 W_i이라고 하고 반복되는 iteration에 따라 계산되는 nW_i라고 하자. 부하분산을 취하기 전에는 $nW_i = W_i$이며 $N = 2^n$이다.

$$L_i = f(T_L - W_i) \quad \text{식 3-6}$$

$$Order_i = \begin{cases} 1, & W_i = Max[W_1, W_2, W_3, \cdots W_n] \\ 0, & otherwise \end{cases} \quad \text{식 3-7}$$

$$\text{if } \sum_i g(W_i - T_H) > 0$$
$$\text{and } \sum_i Order_i \le \sum_i Li \cdot g(T_H - W_i) \quad \text{식 3-8}$$

$$\begin{aligned} then \quad & W_i = W_i - Order_i, \\ & W_j = W_j + 1 \\ & \{\text{if } Min(D_{ij} \ in \ f(T_H - W_j) \cdot L_j = 1)\}, \\ & (i, j = 1, 2, \cdots, N) \end{aligned}$$

,where

$$L_i = g(T_H - W_i) \cdot f(T_L - W_i) \quad \text{식 3-9}$$

$$f(x) = \begin{cases} 1, & x \ge 0 \\ 0, & x < 0 \end{cases} \quad \text{식 3-10}$$

$$g(x) = \begin{cases} x, & x \ge 0 \\ 0, & x < 0 \end{cases} \quad \text{식 3-11}$$

식 3-6은 초기의 부하량이 저부하인지 아닌지를 결정하는 것으로 L_i는 저부하 상태이면 1, 그렇지 않으면 0의 값을 가진다. 식 3-7은 부하량의 Sorting 후에 최대의 부하량을 가진 노드의 $Order_i$는 1, 그렇지 않으면 0을 만드는 식이다. 식 3-8은 부하의 이동을 나타내는 식으로 아직 이동될 과부하량이 남아 있고, 최고 부하량을 가진 노드의 개수보다 저부하 노드의 부하이동 허용량이 크거나 같은 경우에 최고의 부하량을 가진 노드에서 최단 거리의 노드로 1개씩 부하를 이동하는 식이다.

알고리즘의 종료조건은 식 3-8을 만족하지 않을 때이며, 종료 전까지는 식 3-7과 식 3-8을 계속하여 반복한다.

3.4 개선된 알고리즘의 처리순서

(1) 모든 노드의 부하량을 계산한다.
(2) 최대 부하량을 가진 노드를 찾는다.
(3) 최대 과부하 노드의 부하중 1개를 최단거리에 위치한 저부하 노드로 전송한다. 상태는 T_H, T_L 경계로 과부하, 중부하, 저부하의 3가지 상태로 나누고 부하의 불규칙한 이동을 방지하기 위하여 과부하 노드에서 저부하 노드로의 이동만 허용한다.
(4) 만일 전체의 과부하량이 저부하량 보다 크다면 부하분산을 끝마치고 작다면 (1)부터 다시 반복한다.

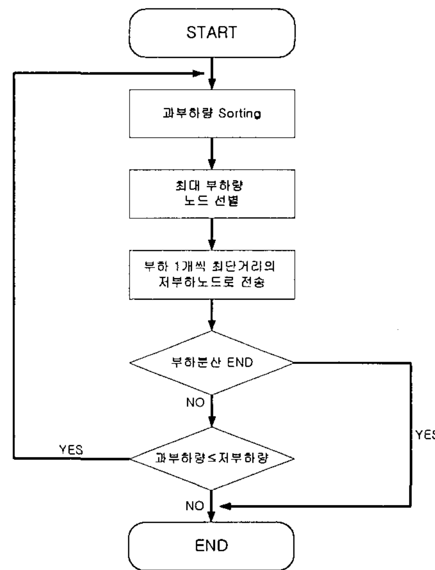

그림 2. 개선된 알고리즘 Flow Chart

3.5 개선된 알고리즘의 부하분산 실험 결과

제안된 부하분산 알고리즘은 하이퍼큐브의 차원 n이 2, 3, 4 일 때의 부하분산의 예제들을 가지고 모의실험 되었다. 노드의 숫자는 각 프로세서의 로컬 큐에 실행대기중인 부하의 수이며 각 노드의 주소는 "[ ]"안에 표시되었다. 각 노드의 부하상태는 T_H와 T_L에 따라 과부하(11이상), 중부하(6~10), 저부하(5이하) 3단계를 색으로 구별하여 나타내었다. 다음 결과는 차원 n=4일 때의 결과이다.

4차원 부하분산 예제 그림을 보면 과부하인 프로세서들이 processor trashing이 발생하지 않는 범위에서 저부하 프로세서로 과부하된 프로세서를 이동시켰다.

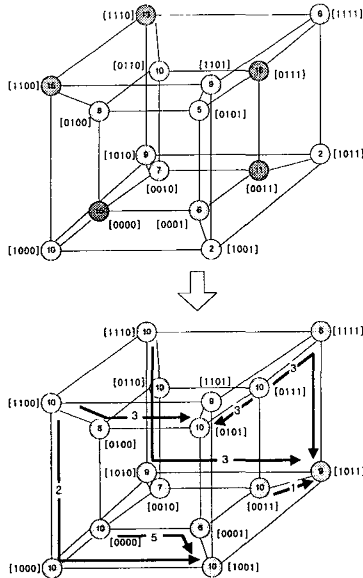

그림 3. 4차원 부하분산 예제(a)

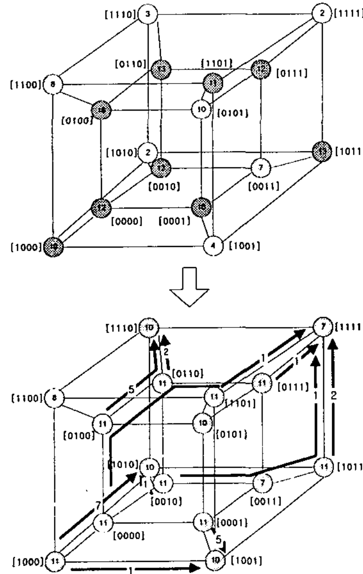

그림 4. 4차원 부하분산 예제(b)

제안된 부하분산 알고리즘과 다른 알고리즘에 의한 부하분산 결과를 보면 2차원의 경우에는 각 알고리즘의 부하분산 결과가 큰 차이를 보이지 않았지만 3, 4차원의 경우에는 차이를 보인다. 표 1은 다른 알고리즘들과 제안된 알고리즘에 의한 부하 분산 결과의 비교이다.

표 1. 4차원 부하분산 알고리즘 비교

그림	Cost Function	Gradient Method	Preferred List Method	제안 알고리즘
3	$H_{execute}$	10	15	10
	H_{comm}	39	34	36
	H_{total}	13.9	18.4	13.6
4	$H_{execute}$	19	19	11
	H_{comm}	41	41	28
	H_{total}	23.1	23.1	13.8

Ⅳ. 결 론

본 논문은 하이퍼큐브 구조의 병렬 프로세서 시스템에서 적합한 부하분산 알고리즘을 제안하였다. 제안된 알고리즘은 과부하 노드에서 인접노드에게만 부하를 전송하여 인접하지 않은 노드는 저부하이지만 부하를 받지 못하여 발생했던 blocking phenomenon을 억제하였으며 저부하 노드가 다수의 과부하 노드에서 너무 많은 부하를 받아 오히려 processor trashing 현상을 유발하는 문제점을 방지하기 위하여 한 프로세서에 모든 프로세서의 정보를 모아 실행하는 중앙집중형 알고리즘의 장점을 응용하였다. 또한 기존의 집중형 알고리즘에서는 최대 과부하노드가 가장 가까이에 있는 저부하 노드로 부하를 전송시켜 나머지 과부하 노드들은 전송하지 못하는 문제가 발생하였다. 하지만 제안된 알고리즘은 부하량 sorting후에 부하의 최대 값을 찾아서 최대 값 노드의 부하만을 저부하 노드로 전송하는 단계를 반복함으로써 전체 노드의 부하의 값이 평균값에 가깝도록 하였다.

제안된 알고리즘은 컴퓨터 모의실험을 통하여 기존의 부하분산 알고리즘과 제안한 알고리즘을 비교함으로써 부하분산의 효율성 증가를 검증하였으며 기존 알고리즘에서 발생했던 여러 문제점들을 개선하였다.

참 고 문 헌

[1] H. Kameda, J. Li, C. Kim, and Y. Zhang, "Optimal Load Balancing in Distributed Computer Systems." London: Springer-Verilog, 1996.

[2] Kang G. Shin and Yi-Chieh Chang, "A Coordinated Location Policy for Load Sharing in Hypercube-Connected Multicomputers," IEEE Trans. Computers, vol. 44, no. 5, pp. 669-682, May 1995.

[3] Sanjay R. Deshpande, "Scaleability of a Binary Tree on a Hypercube," University of Texas at Austin, Technical Report-86-01, January 1986.