하이퍼큐브 시스템에서 데이타 비대칭성을 고려한 향상된 병렬 결합 알고리즘

(An Advanced Parallel Join Algorithm for Managing Data Skew on Hypercube Systems)

원 영 선 † 홍 만 표 ††

(Young Sun Weon) (Man Pyo Hong)

요 약 본 논문에서는 하이퍼큐브 시스템에서 결합 연산을 효율적으로 처리할 수 있는 향상된 병렬 결합 알고리즘을 제안한다. 새로운 알고리즘은 릴레이션 R을 처리함에 있어 하이퍼큐브 구조에 적합한 방송 알고리즘을 사용함으로써 하이퍼큐브 구조에 최적인 병렬 결합 알고리즘을 보이게 된다. 또한 병렬화 성능의 최대 주안점인 부하균등 문제와 데이타 불균형으로 인한 과부하 문제를 완전히 해결하고 결집 효과의 특성을 수용함으로써 전체 성능이 향상된다. 새로운 알고리즘은 해쉬를 기반으로 하는 알고리즘에서 구현하기 어려운 non-equijoin 연산을 쉽게 구현할 수 있다는 장점을 가지며, 비용 모형을 통해 분석한 결과 기존의 병렬 결합 알고리즘들에 비해 보다 나은 성능을 나타냄을 확인한다.

키워드 : 하이퍼큐브, 병렬 결합, 부하균등, 데이타 불균형

Abstract In this paper, we propose advanced parallel join algorithm to efficiently process join operation on hypercube systems. This algorithm uses a broadcasting method in processing relation R which is compatible with hypercube structure. Hence, we can present optimized parallel join algorithm for that hypercube structure. The proposed algorithm has a complete solution of two essential problems - load balancing problem and data skew problem - in parallelization of join operation. In order to solve these problems, we made good use of the characteristics of clustering effect in the algorithm. As a result of this, performance is improved on the whole system than existing algorithms. Moreover, new algorithm has an advantage that can implement non-equijoin operation easily which is difficult to be implemented in hash based algorithm. Finally, according to the cost model analysis, this algorithm showed better performance than existing parallel join algorithms.

Key words : hypercube, parallel join, load balancing, data skew

1. 서 론

최근 초대형 데이타베이스(VLDB) 시스템의 실용성이 대두됨에 따라 방대한 양의 데이타를 효율적으로 처리하기 위하여 빠른 응답시간을 요구하며, 병렬화를 통하여 성능 향상을 기대할 수 있는 다양한 형태의 새로운 알고리즘들이 개발되고 있다.

특히 결합(join) 연산은 다른 연산에 비해 빈번히 사용되고 수행시간이 비교적 긴 연산이므로 결합 연산의 처리 효율성은 데이타베이스 시스템의 전체 성능에 커다란 영향을 미친다. 그러므로 이러한 결합 연산의 비용을 최소화하기 위해 병렬 알고리즘에 대한 많은 연구가 제안되었다[1,2,3,4,5,6,7,8].

이러한 알고리즘들은 중첩 루프 결합(Nested-Loop Join), 정렬 병합 결합(Sort Merge Join), 해쉬 결합(Hash Join), 그리고 분산 결합(Distributive Join) 등으로 분류된다.

중첩 루프 결합은 각 처리기에 수평적으로 분포되어 있는 결합될 두 릴레이션 중 하나의 릴레이션을 파이프라인 방식으로 모든 처리기에 방송(broadcasting) 함으

† 비 회 원 : 충남대학교 정보통신공학부 BK 전임교수
ysweon@ece.cnu.ac.kr
†† 종신회원 : 아주대학교 정보 및 컴퓨터공학부 교수
mphong@madang.ajou.ac.kr
논문접수 : 2002년 2월 20일
심사완료 : 2002년 12월 7일

로써 각 처리기에서 독립적으로 병렬 결합을 수행한다. 따라서 하나의 릴레이션 만을 방송하기 때문에 통신비용을 절감할 수 있다는 특징이 있지만, 각 노드에서는 한 릴레이션의 모든 튜플들에 대한 결합 연산을 수행하기 때문에 결과적으로 불필요한 연산을 많이 하게 된다. 즉 빈번한 오류 결합(false join)으로 불필요한 CPU 비용을 지불하게 된다[9].

정렬 병합 결합은 각 처리기에 분포된 두 릴레이션을 각각 정렬하고 각 처리기로 재분산을 수행한 후, 해당 처리기에서 정렬된 릴레이션을 병합하면서 지역 결합을 수행한다. 이때 릴레이션을 각 처리기로 분산하기 위해 분산표를 사용한다. 이 방법은 릴레이션들을 처리하기 위해 많은 오버헤드를 갖는다는 단점은 있으나, 특정 처리기로의 자료 편중을 막을 수 있다는 큰 장점을 가지고 있다[10,11].

해쉬 결합은 각 처리기에 분포된 두 릴레이션의 결합 키 값을 해쉬 함수를 이용하여 동일한 특성을 갖는 튜플들 간에 서로 상이한 버켓(bucket)에 분포시킴으로써 각 처리기에서 독립적으로 결합을 수행한다. 따라서 결집 효과(clustering effect)로 인해 오류 결합을 줄일 수 있는 장점을 지닌 반면 해쉬 함수에 따른 자료 편재로 인한 재분산으로 통신비용이 증가한다는 문제를 갖고 있다. 그러나 여러 가지 병렬 결합 방식들 중 비교적 우수한 방식으로 인정되고 있다[5,10,11,12].

분산 결합은 각 처리기에 수평적으로 분포되어 있는 결합될 두 개의 릴레이션 중 하나의 릴레이션을 모든 처리기에 순서적으로 완전히 정렬시키고, 나머지 릴레이션은 처리기 간에는 정렬을 그리고 처리기 내에서는 정렬되지 않은 형태인 부분적인 정렬을 시켜, 각 처리기 내에서만 결합 연산이 이루어지도록 함으로써 병렬 수행을 가능하게 하는 결합 방식이다. 이때 완전한 정렬을 필요로 하는 릴레이션의 정렬 과정에서 특정 처리기로의 병합 과정과 나머지 릴레이션의 부분 정렬에 사용될 분산표를 생성하는 과정이 포함되게 된다. 따라서 이 방법은 정렬 병합 결합의 결점인 결합될 두 릴레이션들의 정렬에 소요되는 비용을 줄이는 특징을 가지고 있으나, 정렬과 병합 그리고 분산표의 생성과 분산에 따른 많은 오버헤드를 여전히 갖게 된다[1,4,6].

병렬 컴퓨터 시스템에서는 연산에 사용하는 처리기의 수가 많아질수록 통신비용의 비중이 보다 중요한 요소로 작용한다. 하이퍼큐브 구조는 응용 분야에서 요구하는 통신망 구조를 쉽게 제공할 수 있으며 비교적 짧은 지름(diameter), 대칭성(symmetry), 유연성(flexibility)과 같은 좋은 특성으로 인하여 병렬 시스템의 처리기 상호 연결 구조로서 널리 사용된다.

이와같은 특성을 갖는 하이퍼큐브 시스템에서의 하이브리드 해쉬 결합 알고리즘은 릴레이션을 분배하는 과정에서 특정 처리기에 자료가 편중되는 버켓 오버플로우가 발생하고 그로 인해 전체 성능을 크게 저하시키는 결과를 가져왔다. 따라서 오버플로우를 해결하기 위한 방법으로 재분산을 수행하거나, 또는 하이퍼큐브 시스템을 여러 개의 서브큐브로 분할하여 서브큐브 간에는 하이브리드 해쉬 결합 방식을, 서브큐브 내에서는 중첩 루프 결합 방식을 이용하는 알고리즘 등이 제시되었다[7].

또한 분산 결합 알고리즘의 장점인 부하균등 문제를 어느 정도 해결하면서 동시에 전체적인 통신시간을 줄일 수 있는 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘이 제안되었다[3,8]. 그러나 이 알고리즘은 전체 통신시간이 기존의 알고리즘 중 비교적 우수한 방식으로 알려져 있는 하이브리드 해쉬 결합 알고리즘과 거의 유사할 정도로 효과를 거두었으나, 부하균등 문제와 데이타 비대칭성 현상을 완전히 해결하지는 못하였다. 즉 특정 노드에 데이타가 편중되는 최악의 경우가 발생하면 데이타가 몰려 있는 특정노드의 처리가 완료될 때까지 다른 노드들은 대기하고 있어야 하므로 전체 성능을 크게 떨어뜨릴 수 있다는 결정적인 단점을 가지고 있다.

본 논문에서는 하이퍼큐브 구조로 연결된 다중 처리기 환경에서 부하균등 문제와 데이타 비대칭 현상으로 인한 과부하 문제를 완전히 해결하면서 동시에 하이퍼큐브 시스템에 적절한 방송(broadcasting) 알고리즘을 이용하여 전체적인 통신시간을 줄일 수 있는 효율적인 병렬 결합 알고리즘을 제안한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구로써 병렬 분산 결합 알고리즘과 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘에 대해 간단히 살펴본다. 본 논문에서 제시한 부하 균등과 데이타 비대칭성 현상을 완전히 해결한 새로운 병렬 결합 알고리즘은 3장에서 기술하고, 4장에서는 새로운 알고리즘의 성능 분석을 위한 분석 모형을 제시한다. 5장에서는 분석 모형을 통해 다양한 방법으로 성능을 평가하고, 기존의 알고리즘들과 성능을 비교한다. 마지막으로 6장에서는 본 논문에서 얻은 결과를 정리한다.

2. 병렬 결합 알고리즘

결합 연산의 효율적인 처리를 위한 많은 결합 알고리즘들이 제안되어 왔다. 그 중, 새롭게 제안하는 알고리즘과 관련된 병렬 분산 결합 알고리즘과 하이퍼 퀵 정

렬을 이용한 병렬 결합 알고리즘은 다음과 같은 특징을 가지고 있다. 결합 연산을 수행할 두 릴레이션 중 하나의 릴레이션은 완전히 정렬하고 다른 릴레이션은 부분적으로 정렬한다는 것을 기본으로 한다. 따라서 이러한 알고리즘들은 특정 처리기로의 자료 편중을 막을 수 있다는 정렬 병합 결합 알고리즘의 장점과, 결합 가능한 릴레이션을 한 처리기로 모이게 하는 결집 효과를 지님으로 인해 오류 결합을 줄일 수 있다는 해쉬 결합 알고리즘의 장점을 접목시킨 또 다른 형태의 결합 알고리즘들이다[1,3,4].

다음은 병렬 분산 알고리즘[4]과 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘[3]에 대해 살펴본다.

2.1 병렬 분산 결합 알고리즘

하이퍼큐브 시스템에서의 병렬 분산 결합 알고리즘은 각 노드에서 순차 분산 결합 알고리즘을 병렬로 수행함을 의미한다.

먼저 릴레이션 R을 하이퍼큐브 시스템의 노드 수로 분배하고 각 노드에서 병렬적으로 정렬을 수행한 후, 정렬된 릴레이션들을 특정 노드로 병합한다. 이 노드에서는 병합된 결과를 이용하여 분산표를 만들고, 작성된 분산표 전체와 각 노드에 해당되는 릴레이션 R의 서브 릴레이션들을 전송한다. 이때 분산표는 각 노드의 주소와 릴레이션 R의 서브 릴레이션인 R_i의 최대 키 값으로 이루어진다. 다음 릴레이션 S는 분산표의 내용에 의해 서브 릴레이션으로 분배한다. 이때 $i < j$인 S_i의 모든 결합 키 값은 S_i의 결합 키 값보다 작아야 한다. 그러나 S_i 내부의 튜플들은 결합 키 값에 의해 정렬되어 있을 필요가 없다. 마지막으로 각 노드에서 병렬적으로 결합 연산을 수행한다. 만약 서브 릴레이션 i노드에 할당된 R_i와 S_i가 메모리 내에 들어갈 수 없을 만큼 큰 경우에는, 노드 내에서의 결합을 순차 분산 결합 알고리즘으로 처리한다.

2.2 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘

분산 결합 알고리즘은 릴레이션을 처리하기 위해 많은 통신 시간이 소요된다. 특히 릴레이션 R을 정렬하여 특정 노드에 병합하고 그 결과를 이용하여 분산표를 만들고 각 노드들로 전송하는데 사용되는 시간은 많은 통신 시간을 필요로 한다. 따라서 릴레이션을 처리함에 있어 하이퍼큐브 시스템에 적절한 하이퍼 퀵 정렬[13]을 변형하여 사용하고, 이때 발생하는 중간값이 분산표의 기능을 수행하게 함으로써 릴레이션 R을 병합하는 과정과 분산표를 만들고 방송하는 단계를 충분히 줄일 수 있다.

이러한 개념으로 착안된 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘은 먼저 릴레이션 R을 하이퍼큐브 시스템의 각 노드로 균등하게 분산시키고 각 노드에서 결합 키 값에 의해 퀵 정렬을 수행한다. 정렬되어진 각 노드에서는 결합 키 값의 중간값을 기준값으로 설정하여 다른 노드들에게 그 기준값을 방송(all-to-all broadcasting)한다. 다음으로 각 노드에서는 그 기준값들의 중간값을 전체 기준값으로 설정한다. 이는 각 노드들에서 동시에 병렬적으로 수행된다. 각 노드에서는 전체 기준값과 비교하여 자신의 노드에 해당되지 않는 튜플들을 출력 버퍼를 통해 해당하는 차원에 대응되는 노드와 교환하고, 각 노드에 할당된 튜플들을 병합한다. 기준값의 설정에서부터 지금까지의 과정을 하이퍼큐브 시스템의 차원 수만큼 반복한다. 다음 릴레이션 S는 릴레이션 R의 기준값에 의해 서브 릴레이션으로 분배한다. 마지막으로 각 노드에서는 릴레이션 R과 S의 서브 릴레이션을 가지고 병렬적으로 결합 연산을 수행한다.

3. 새로운 하이퍼큐브 부하균등 병렬 결합 알고리즘

병렬 분산 결합 알고리즘으로부터 출발된 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘은 부하균등 문제를 어느 정도 해결하면서 동시에 하이브리드 해쉬 결합 알고리즘에서는 구현하기 어려운 non-equijoin을 수행할 수 있는 알고리즘으로 릴레이션 R의 처리비용을 최대한 줄이고자 하였다[3]. 그러나 이 알고리즘에서는 중간값의 중간값을 기준값으로 선택함으로써 가능한 부하균등 문제를 해결하려고 하였으나, 병렬 분산 결합 알고리즘에서와 같이 완전하게 부하 균등 문제를 해결하지는 못하였다.

또한 각 단계에서 선택된 기준값에 의해 매 단계마다 각 노드로 릴레이션 R을 분배함으로서 중간 단계에서 특정 노드에 데이타가 편중되는 데이타 비대칭성 현상이 발생할 수 있다. 이는 릴레이션을 처리하는 중간 단계에서 과부하를 야기함으로서 전체적인 시스템의 성능을 저하시키는 결정적 단점으로 작용한다.

따라서 릴레이션 R의 처리가 완료되었을 경우 부하균등을 이루어야 하는 문제와 함께 중간 단계에서 발생할 수 있는 데이타 비대칭성 문제를 완전히 해소할 수 있는 효율적인 병렬 결합 알고리즘을 제안한다.

새로운 알고리즘은 하이퍼큐브 시스템의 특성을 최대한 살려 그에 적합한 방송(broadcasting) 알고리즘을 이용하여 최상의 성능을 갖는 결합 알고리즘으로, 릴레이션 R의 효율적 처리를 통하여 완전한 부하균등을 이

루고 동시에 중간 단계에서의 데이타 비대칭 문제를 완전히 해결함으로써 전체적인 처리시간을 충분히 줄일 수 있다.

새로운 결합 알고리즘은 크게 릴레이션 R을 정렬하고 분산하는 과정, 릴레이션 R의 기준 키 값에 의해 릴레이션 S를 처리하는 과정, 그리고 두 릴레이션을 각각의 처리기에서 병렬적으로 결합하는 과정의 세 단계로 이루어진다.

3.1 릴레이션 R의 처리 단계

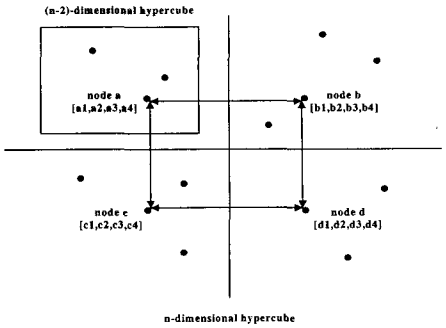
릴레이션 R의 튜플들을 정렬 분산하기 위한 방법은 다음과 같다. 먼저 릴레이션 R을 각 노드에 분산시키기 위한 모든 기준값을 결정한다. 다음 그 기준값에 의해 릴레이션 R을 분산시킨다. 이때 기준값에 의해 정렬 분산되는 중간 단계에서 특정 노드에 데이타가 편중되는 데이타 비대칭성 문제를 해결하기 위하여 모든 기준값을 미리 결정지어 놓고 분산 작업을 시작한다.

릴레이션 R의 정렬 분산 과정을 자세히 살펴보면, 릴레이션 R을 하이퍼큐브 시스템의 각 노드로 균등하게 분산시키고 각 노드에서 결합키 값에 의해 퀵 정렬을 수행한다. 다음, 정렬되어진 각 노드의 결합키 값들로부터 중간값을 기준값으로 선택하여 해당 노드와 교환한다. 부하균등을 이루는 전체 기준값을 결정하기 위해서는 각 노드에서 선택된 기준값을 모두 활용해야하며 이를 위해서 교환 단계를 필요로 한다.

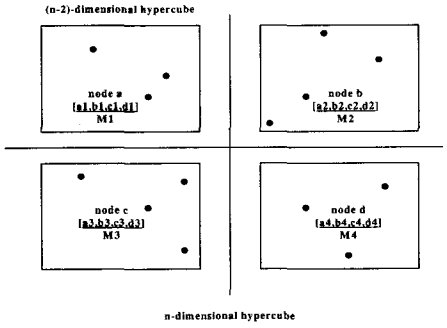
첫 번째 기준값을 선택하기 위해서는 전체 하이퍼큐브 시스템에서 처리되기 때문에 교환 단계를 필요로 하지 않으나, 두 번째 기준값의 선택에서는 n차원 하이퍼큐브 시스템을 (n-1)차원의 두 개의 서브큐브로 나누어서 각각 수행하므로 각 서브큐브에서 필요한 각 노드의 기준값 후보들을 해당 서브큐브로 이동시키기 위한 교환 작업을 필요로 한다. 다음 각 서브큐브에서는 교환된 기준값 후보들을 가지고 서브큐브 내에서 방송(all-to-all broadcasting)을 수행한다. 이때 각 서브큐브들은 전체 노드 수 만큼의 해당 기준값 후보들을 소유하게 되고 이 기준값 후보들을 정렬하여 전체적인 기준값을 결정한다. 다음 각 서브큐브에서는 결정된 전체 기준값을 하이퍼큐브 시스템 전체에게 복원시켜준다. 이때 각 서브큐브들은 서로 다른 해당 기준값을 가지게 되고 따라서 각각의 서로 다른 기준값을 가지고 있는 서브큐브들 간에 방송(all-to-all broadcasting)을 수행한다.

그림 1은 n차원 하이퍼큐브 시스템에서 새로운 알고리즘을 이용하여 릴레이션 R을 처리할 때, 3번째 기준값을 결정하는 과정을 나타내고 있다.

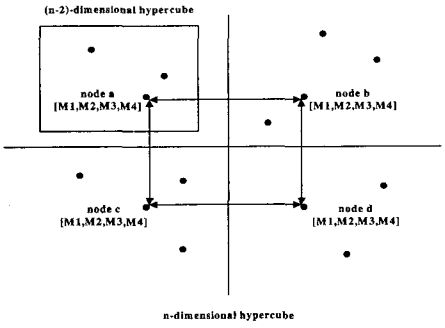
그림 1(a)는 각 노드에서의 기준값 후보를 선택한 결과를 보이고 있다. 노드 a의 경우는 노드내의 정렬된 4개의 서브블럭으로부터 각 서브블럭의 중간값인 $a1$, $a2$, $a3$ 그리고 $a4$를 기준값 후보로 선택한다. 이와 같이 선택된 기준값 후보들을 대응되는 노드와 교환한다. 교환된 결과는 (b)와 같으며 교환된 기준값 후보를 가지고 각 서브큐브내에서는 방송(all-to-all broadcasting)



(a) 기준값 후보들의 선택 및 교환



(b) 각 서브큐브에서 전체 기준값 결정



(c) 기준값 복원

그림 1 새로운 알고리즘을 이용한 3번째 기준값의 결정 과정

단계를 거쳐 전체 기준값을 결정한다. (b)에서 노드 a의 경우는 $M1$이 기준값으로 결정되었다. 다음 각 서브큐브에서 병렬적으로 수행되어 결정된 기준값을 모든 노드에 복원시켜 주어야 하며 그 결과는 (c)와 같다. 매 단계마다 서브큐브의 차원을 하나씩 줄여가면서 이와 같은 과정을 반복 수행한다.

이 과정이 수행되고 나면 각 노드는 (전체 노드의 수 -1) 만큼의 기준값을 가지게 되고, 그 기준값을 기준으로 하여 각 노드는 자신의 노드에 해당되지 않는 튜플들을 해당 출력 버퍼를 통해 해당 노드로 전송한다. 이때 각 노드에서 해당 노드로 전송될 튜플의 수는 각 단계마다 전체 튜플 수를 전체 노드 수로 나눈 만큼의 크기이며 이러한 단계를 전체 노드 수만큼 순차적으로 반복 수행한다. 동시에, 전송되어온 내부적으로 정렬된 서브블럭을 현재 소유하고 있는 정렬된 서브블럭과 병합을 수행한다. 통신 작업과 연산 작업은 동시에 이루어 질 수 있으므로 매 단계마다 함께 병행 수행한다. 이렇게 함으로써 각 노드에서 전송될 데이타의 크기는 (전체 릴레이션의 크기/노드수^2)이므로 오버플로우가 발생하는 노드는 없게 되고 동시에 특정 노드에 데이타가 편중되는 비대칭성으로 인한 과부하도 해결할 수 있다.

3.2 릴레이션 S의 처리 단계

릴레이션 S의 처리 과정은 먼저 릴레이션 R과 같이 릴레이션 S를 각 노드에 균등하게 분산시킨다. 분산되어 있는 모든 튜플에 대하여 기준값과 비교하여 자신의 노드에 해당되지 않는 튜플들을 출력 버퍼를 통해 해당하는 차원의 대응되는 노드와 교환한다. 이와 같은 과정을 하이퍼큐브 시스템의 차원 수 만큼 반복 수행한다.

3.3 릴레이션 R과 S의 결합 단계

릴레이션 R과 S의 처리 과정이 모두 끝나면 각 노드는 해당되는 튜플만을 지니게 되고, 이로써 각 노드에서는 병렬로 결합 연산을 수행한다.

결합 연산을 수행하기 위해 노드 내의 릴레이션 R이 메모리에 한꺼번에 들어올 수 있는 경우에는 릴레이션 S도 메모리 내로 들어올 수 있도록 페이지 단위로 분할되어야 한다. 그러나 지역 결합을 위하여 분할된 R과 S가 메모리로 들어 올 수 없는 경우에는 각 노드에서 순차 분산 결합 알고리즘을 이용하여 결합을 수행한다. 이들은 다시 재분할하여야 하는 과정이 필요하며 이를 위한 분산표는 만들어야 한다.

다음은 새로운 알고리즘을 기술한 가상 코드이다.

```
Each processor reads the local partition of R, page by page;
Each processor sort using sequential quick sort its local partition of R; /* in parallel */
For i=1 to n;
In each node, select 2^(i-1) median value as candidates of 2^(i-1) locally sorted blocks;
        Exchange with 2^(i-2) candidates (i-1) times;
In each n-(i-1)-dimensional hypercube, perform the all-to-all broadcasting of the exchanged candidates;
Each node performs quicksort on its candidate list;
Each node select the median value as the pivot;
Inter n-(i-1)-dimensional hypercube, perform the all-to-all broadcasting of pivot;
End_For

Use pivot to partition sorted values;
Send and receive corresponding list;
At the same time, merge lists into a single sorted list of values;

Each processor reads the local partition of S, page by page;
For every tuple t in the input buffer Do
    If tuple belongs to MyNode Then
      Insert t into S_location ;
    Else
      Store t in the corresponding output buffer;
    End_If
End_For

For i:=n-1 downto 0 Do          /* following two operations are executed in parallel */
      Send output buffer i to neighbor i;
      Receive tuples from neighbor i;
      For every tuple t in the input buffer Do
          If MyNode Then
            Insert t into S_location ;
        Else
        Store t in the corresponding output buffer;
        End_If
    End_For
End_For

If |R_i| ≤M-1 or |S_i| ≤M-1 Then
        Choose the smaller relation and load it into memory;
```

Do the join operation by searching the smaller relation for each tuple of the larger relation;

Else

$m := \lceil \frac{|R_i|}{M-1} \rceil$;

Divide the sorted relation Ri into R_{i0}, R_{i1}, ... , R_{im-1},

such that $|R_{i0}|=|R_i1|= \cdots =|R_{im-2}|=M-1$ *and* $|R_{im-1}| \leq M-1$;

For j:=0 to m-1 Do

Build the local distribution table using the maximum join attribute value R_{ij} *and j;*

End_For

Partition Si into S_{i0}, S_{i1}, ⋯, S_{im-1}, *using the local distribution table;*

For j:= 0 to m-1 Do

Join R_{ij} *and* S_{ij};

End_For

End_If

3.4 Non-equijoin 처리 방법

Equijoin 알고리즘과는 달리 릴레이션 S의 동일한 튜플이 여러 노드에 동시에 존재하는 non-equijoin의 수행 과정을 살펴보면 다음과 같다.

먼저 릴레이션 R의 처리는 equijoin의 경우와 동일하게 이루어진다. 즉 각 노드에서는 튜플들에 대해 퀵 정렬을 수행하고, 릴레이션 R을 전체적으로 완전히 정렬하고 균등하게 분산한다.

릴레이션 S는 동일한 튜플이 여러 노드에 동시에 존재하는 경우가 발생하기 때문에, 이런 경우에는 해당 튜플을 여러 노드로 전송해야 한다. 이는 다시 말해 기준값과 비교하여 튜플을 전달하는 과정에서 출력 버퍼를 통해 해당하는 차원의 대응되는 노드로 전송할 때 자신의 노드에 해당되는 튜플들을 복사한 후 다시 출력 버퍼로 옮겨 놓는 단순한 작업만을 필요로 한다.

릴레이션 R과 S의 처리 과정이 모두 끝나면, 각 노드는 해당되는 튜플만을 지니게 되고 이로써 각 노드에서는 병렬로 결합 연산을 수행한다.

4. 비용 모형

하이퍼큐브 시스템의 특성에 적합한 방송 알고리즘을 이용하여 데이타 비대칭 문제를 해결함과 동시에 부하 균등의 장점을 갖는 새로운 결합 알고리즘의 비용 성능 평가를 위해 분석적 비용 모형을 제시하며, 이는 다음과 같은 가정 하에서 이루어진다.

(1) 릴레이션 R의 크기는 릴레이션 S의 크기보다 작거나 같다.

(2) 각 처리기는 하나의 프로세서, 주기억장치, 그리고 디스크로 이루어져 있다.

새로운 결합 알고리즘에 대한 연산 비용은 크게 세 부분으로 나누어진다. 하나는 하이퍼큐브 시스템에 적절한 방송 알고리즘을 이용하여 릴레이션 R을 정렬하고 분산하는 비용이고, 다른 하나는 릴레이션 S의 분산 비용, 그리고 나머지 하나는 각 노드에서 릴레이션 R과 S를 결합하는 비용이다. 비용 모델을 위한 표기법은 표 1과 같다.

표 1 비용 모델을 위한 표기법

$\|\|R\|\|,\|\|S\|\|$	릴레이션 R과 S의 튜플 수
TP_R	릴레이션 R의 한 페이지내의 튜플 수
IO_{read}	디스크로부터 한 페이지를 읽기 위한 시간 (sec)
IO_{write}	디스크로 한 페이지를 쓰기 위한 시간 (sec)
comp	두 결합 키 값을 비교하는 시간 (sec)
move	주기억장치 내에서 한 튜플을 위한 시간(sec)
keyA	결합 키 값의 크기 (bytes)
PS	패킷의 최대 크기 (bytes)
comm	통신 비율 (bits/sec)
P_{ovhd}	패킷 오버헤드 시간 (sec)
TSR	릴레이션 R의 한 튜플의 크기 (bytes)
JS	릴레이션 R과 S의 튜플이 결합되어 질 비율
P	페이지의 크기 (bytes)
N	전체 노드 수
n	큐브의 차원 값, $\log_2 N$
TP_o	결과로 산출된 릴레이션의 한 페이지 내의 튜플 수
M+1	각 노드의 메모리의 크기 (pages)

4.1 릴레이션 R의 처리 비용

릴레이션 R을 처리하는 비용은 디스크로부터 릴레이션을 입력하는 비용과, 방송 알고리즘을 이용하여 기준값을 결정하고 그 결과에 의해 릴레이션을 완전히 정렬하고 균등하게 분산시키는데 소요되는 비용의 합이다.

4.1.1 디스크로부터의 릴레이션 입력 비용

각 노드의 디스크에 분산되어 있는 릴레이션 R의 튜플들을 노드 내에서 처리하기 위해 프로세서의 입력 버퍼를 통해 페이지 단위로 읽어 들인다. 이때 릴레이션 R의 튜플들은 각 노드에 균등하게 분포되어 있다고 가정한다.

$$R\_read = \lceil \frac{||R||}{N * TP_R} \rceil * IO_{read}$$

4.1.2 병렬 정렬과 균등 분산을 수행하는 비용

• 노드 내에서의 정렬 비용

각 노드에 분산된 릴레이션 R의 튜플들은 결합키 값에 따라 자신의 순서를 결정하기 위해 다른 튜플의 결합키 값과 비교되어야 하며 그 결과에 따라 자신의 위치로 이동되어야 한다. 즉, 각 노드에서는 퀵 정렬을 이용하여 튜플들을 정렬한다.

$$R\_sort = \frac{\|R\|}{N} * (comp + move) * \log \frac{\|R\|}{N}$$

• 기준값 후보 결정 비용

한 노드 내에서 기준값 후보를 결정하는데 소요되는 비용이다. 이미 노드내의 튜플들은 정렬되어 있으므로 각 노드의 정렬된 서브블럭에서 중간값 선택은 단지 한 번의 접근으로 가능하다. 따라서 그 비용은 상수값이다. 처음에는 각 노드내의 중간값을 기준값 후보로 선택하고, 다음에는 두개의 정렬된 서브블럭에서 각각의 중간값을 기준값 후보로 선택한다. 즉 i번째 단계에서는 2^{i-1}개의 정렬된 서브블럭으로부터 2^{i-1}개의 중간값을 기준값 후보로 선택하게 된다.

• 기준값 후보 교환 비용

부하균등을 이루는 전체 기준값을 결정하기 위해서는 각 노드에서 선택된 기준값 후보들을 모두 활용해야 한다. 즉, 하이퍼큐브 시스템 전체 노드 수 만큼의 기준값 후보 데이타로부터 전체 기준값을 결정하여야 한다. 따라서 각 노드에서 선택된 기준값 후보들을 활용하기 위하여 해당 노드와 기준값 후보들을 교환한다. 처음에는 하이퍼큐브 시스템의 차원이 n이므로 교환 단계를 필요로 하지 않는다. 그러나 다음 단계에서는 하이퍼큐브 시스템의 차원이 하나 감소하여 $(n-1)$차원 서브큐브에서 작업이 이루어지므로 하나의 기준값 후보를 한번 교환하는 단계를 필요로 한다. 즉, i번째 단계에서는 서브큐브의 차원이 $n-(i-1)$이므로 2^{i-2}개의 기준값 후보들을 $i-1$번 교환하는 단계가 필요하다. 따라서 이러한 과정을 하이퍼큐브 시스템의 차원 수 만큼 반복해야 한다.

$$R\_exchange_1 = \sum_{i=1}^{n-1} (move * \frac{2^{i-1} keyA}{TS_R} + 2^{i-1} keyA * \frac{8}{comm} + \lceil 2^{i-1} keyA * \frac{1}{PS} \rceil * P_{ovhd}) * i$$

• 기준값 후보들을 각 노드에 전송하는 비용

각 노드의 기준값 후보들을 해당 서브큐브에 전송한다. 이러한 전송은 처음에는 n번의 단계를 필요로 하고, 다음 단계에서는 $n-1$번의 단계를 필요로 한다. 즉, i번째 단계에서는 $n-(i-1)$번의 단계가 필요하다. 따라서 이러한 과정을 하이퍼큐브 시스템의 차원 수 만큼 반복해야 하므로 n번에 걸쳐 이루어진다.

$$R\_broad = \sum_{i=0}^{n-1} (move * \frac{2^i keyA}{TS_R} + 2^i keyA * \frac{8}{comm} + \lceil 2^i keyA * \frac{1}{PS} \rceil * P_{ovhd}) * (n-i)$$

• 기준값 후보들의 중간값을 전체 기준값으로 결정하는 비용

전체 노드 수 만큼의 기준값 후보들을 보유한 상태에서 각 노드에서는 기준값 후보들의 중간값을 전체 기준값으로 선택한다. 따라서 각 노드에서 모인 전체 노드 수 만큼의 기준값 후보들에 대해 퀵 정렬을 수행하여 중간값을 기준값으로 결정한다. 이와 같은 과정을 하이퍼큐브의 차원 수 만큼 반복해야 하므로 n번에 걸쳐 이루어진다.

$$R\_med = \sum_{i=0}^{n-1} N(comp + move) \log N$$

• 기준값 복원 비용

해당 서브큐브에서 결정된 전체 기준값을 전체 하이퍼큐브 시스템이 공유할 수 있도록 서브큐브의 차원에 맞게 복원시켜 주어야 한다. 이러한 복원 단계는 처음에는 각 노드의 기준값 교환이 없었으므로 복원이 필요하지 않다. 그러나 다음 단계에서는 서브 큐브의 차원이 $n-1$ 이므로 한개의 전체 기준값에 대해 한번의 복원을 필요로 한다. 그 다음 단계에서는 서브큐브의 차원이 $n-2$ 이므로 서브큐브 간에는 2차원이 형성되어 2차원의 서브큐브간의 방송(all-to-all broadcasting)을 통한 복원 작업이 이루어진다. 즉 i번째 단계에서는 서브큐브의 차원이 $n-(i-1)$ 이므로 서브큐브 간에는 $i-1$차원이 형성되어 $i-1$차원의 서브큐브간의 방송에 의한 복원 단계가 필요하다. 따라서 이러한 과정을 하이퍼큐브의 차원 수 만큼 반복한다.

$$R\_exchange_2 = \sum_{i=1}^{n-1} \sum_{j=1}^{i} (move * \frac{2^{j-1} keyA}{TS_R} + 2^{j-1} keyA * \frac{8}{comm} + \lceil 2^{j-1} keyA * \frac{1}{PS} \rceil * P_{ovhd})$$

• 기준값의 저장 비용

릴레이션 R을 분배하고 릴레이션 S를 처리하기 위해서는 기준값과 비교를 해야 하므로 각 노드에서는 해당 기준값을 노드내의 임의의 장소에 순차적으로 보관한다.

$$R\_store = \frac{keyA}{TS_R} * move * n$$

• 기준값과 튜플을 비교하여 해당 노드로 분배하는 비용

각 노드에서는 모든 튜플들을 기준값과 비교하여 자신의 노드에 해당되지 않는 튜플을 해당되는 노드의 출력 버퍼로 옮겨 놓는다. 이때 각 노드의 튜플들은 모두 정렬되어 있고 각 노드는 자신의 기준값을 알고 있으므로 하이퍼큐브의 노드 수 만큼 선형적으로 반복 수행한다. 이때 각 노드로 전송되어야 할 튜플의 수는 릴레이션 R의 전체 튜플을 전체 노드 수의 제곱으로 나눈 수이므로 중간 단계에서의 데이타 비대칭 문제에 따른 부하 불균형을 해결할 수 있다.

$$R\_comm = \frac{\|R\|}{N} * (comp + move) + \sum_{i=1}^{N-1}\left(\frac{\|R\|}{N^2} * TS_R * \frac{8}{comm} + \left(\lceil \frac{\|R\|}{N^2} * TS_R * \frac{1}{PS} \rceil\right) * P_{ovhd}\right)$$

• 교환된 튜플을 병합하는 비용

교환된 튜플을 병합하는데 소요되는 비용이다. 릴레이션 R의 분배가 종료되면 각 노드에서는 하이퍼큐브 시스템의 차원 수 만큼의 내부적으로 정렬된 서브블럭이 존재하게 된다. 따라서 이 서브블럭을 병합해야 한다. 그러나 하이퍼큐브 시스템의 차원 수 만큼 분배 과정을 반복하는 동안 시스템의 각 노드는 통신만 수행하고 연산은 유휴시간을 보내고 있다. 따라서 그 유휴시간을 줄이기 위해 통신과 연산 작업을 병행하여 수행한다. 이때 각 노드에 해당하는 서브블럭을 통신하는데 걸리는 시간이 서브블럭을 병합하는 시간에 비해 크므로 병합하는 비용은 통신비용에 포함되게 되고 따라서 별도의 병합 시간을 필요로 하지 않는다.

4.1.3 릴레이션 R을 처리하는 총 시간 비용

릴레이션 R을 처리하는데 필요한 비용은 다음과 같다.

$$Rcost = R\_read + R\_sort + R\_exchange_1 + R\_broad + R\_med + R\_exchange_2 + R\_store + R\_comm$$

4.2 릴레이션 S의 처리 비용

릴레이션 S를 처리하는 비용은 디스크로부터 릴레이션을 입력하는 비용과 기준값과의 비교를 통해 튜플들을 해당 노드로 전송하는 데 소요되는 비용의 합이다.

4.2.1 디스크로부터의 릴레이션 입력 비용

릴레이션 S의 튜플들이 N개의 노드에 균등하게 분포되어 있다는 가정하에 릴레이션 S의 튜플들을 노드 내에서 처리하기 위해 디스크로부터 프로세서의 입력 버퍼를 통해 페이지 단위로 읽어 들인다.

$$S\_read = \lceil \frac{\|S\|}{N * TP_S} \rceil * IO_{read}$$

4.2.2 기준값과 튜플을 비교하여 이웃한 노드의 출력 버퍼에 저장하는 비용

입력 버퍼 안의 모든 튜플들을 기준값과 비교하여 출력 버퍼에 옮겨 놓는다. 즉 출력 버퍼로의 이동을 위하여 튜플과 기준값과의 비교가 필요하고, 해당 출력 버퍼로 튜플들을 이동시켜야 한다. 이때 이동되는 튜플의 수는 노드 안에 있는 전체 튜플의 $\frac{1}{2}$정도이다. 이 과정은 하이퍼큐브의 차원 수 만큼 반복된다.

$$S\_middle = n * \frac{\|S\|}{N} * (comp + \frac{move}{2})$$

4.2.3 해당 노드의 튜플들을 결합을 위한 장소로 이동하는 비용

위의 단계의 마지막 수행 후 각 노드에는 자신의 노드에 적합한 튜플들만이 놓이게 된다. 그러한 튜플들을 결합을 위한 장소로 이동시켜야 한다.

$$S\_final = \frac{\|S\|}{N} * move$$

4.2.4 해당 노드로 튜플을 전송하는 비용

기준값과 비교된 튜플을 해당 노드로 전송한다. 이러한 과정을 하이퍼큐브의 차원 수 만큼 반복하므로 n번 수행한다.

$$S\_comm = n * \left(\frac{\|S\|}{2*N} * TS_s\right) * \frac{8}{comm} + n * \left(\lceil \frac{\|S\|}{2*N} * TS_s * \frac{1}{PS} \rceil\right) * P_{ovhd}$$

4.2.5 릴레이션 S를 처리하는 총 시간 비용

릴레이션 S를 처리하는데 필요한 비용은 다음과 같다.

$$Scost = S\_read + S\_middle + S\_final + S\_comm$$

4.3 릴레이션 R과 S의 결합 비용

주어진 기억 장소 내에서 결합이 이루어질 수 있도록 릴레이션의 분할이 필요하다. 각 노드에 균등하게 분배되어진 릴레이션의 크기가 기억장소보다 큰 경우에는 순차 분산 결합 알고리즘을 이용하여 처리한다. 기억 장소의 크기가 제한되어 있고 그 중 대부분을 릴레이션 R을 위한 공간으로 확보하였기 때문에 노드 i에서 릴레이션 S를 처리하기 위해서는 릴레이션 S의 서브 릴레이션인 S_i를 다시 분할하여야 한다.

4.3.1 릴레이션 S의 분할 비용

각 노드로 전달되어진 릴레이션 S는 메모리 안으로 들어올 수 있는 페이지의 크기로 우선 분할한다. 이를 위해 각 노드에서는 자신의 디스크에 존재하는 릴레이션 S를 읽어온 후 페이지로 분할하고 분할된 릴레이션을 다시 디스크에 저장하여야 한다.

$$join\_part\_S = \frac{\|S\|}{N * TP_R} * (IO_{read} + IO_{write})$$

4.3.2 릴레이션을 메모리로 읽어 오는 비용

분할이 끝난 후 결합을 실행하기 위해 릴레이션 R과 S를 메모리로 옮겨와야 한다. 릴레이션 R은 $M-1$개의 페이지씩 이동되며, 릴레이션 S는 한 페이지씩 이동된다.

$$join\_load\_RS = (\frac{\|R\|}{N * TP_R} + \frac{\|S\|}{N * TP_S}) * IO_{read}$$

4.3.3 릴레이션 R과 S를 결합하는 비용

메모리로 이동되어온 릴레이션 R과 S를 이용하여 결합을 실행한다. 이때 각 튜플이 결합될 확률은 JS이며, 결합된 결과는 다시 디스크에 저장된다.

$$join\_RS = \frac{\|R\|}{N} * \frac{\|S\|}{N} * (JS*N) * \frac{1}{TP_o} * IO_{write}$$

4.3.4 결합 비용

결합을 수행하는 데 필요한 비용은 다음과 같다.

$$join\_cost = join\_part\_S + join\_load\_RS + join\_RS$$

4.3.5 노드 내에서의 순차 분산 결합 처리 비용

만일 분할되어진 릴레이션의 크기가 메모리의 크기보다 큰 경우에는 이를 다시 분할하여 부분별로 결합을 실행한다. 이를 위해 노드 내의 릴레이션 R에 대한 분산표를 작성하고, 분산표에 의해 릴레이션 S를 분할하고 분할되어진 튜플들을 이용하여 결합을 수행한다.

• 분산표 내에 하나의 엔트리를 저장하는 시간

릴레이션을 분할하기 위해 분산표를 작성한다. 분산표에는 기준키 값만을 저장하고 이 키 값은 단순히 메모리 내에서의 이동시간만을 필요로 한다.

$$move' = move * \frac{IA}{TS_R}$$

• 노드에서의 분할 수(m)를 결정

메모리 내에 한번에 $M+1$개의 페이지가 들어올 수 있다. 즉, 한 페이지는 릴레이션 S가 들어올 공간이고, 다른 한 페이지에는 결합된 결과가 저장될 공간이다. 그러므로 릴레이션 R이 들어올 수 있는 공간은 $M+1$개의 페이지이다.

$$m = \lceil \frac{\|R\| / (N * TP_R)}{M-1} \rceil$$

• 노드 내의 분산표를 구성하는 시간

분산표 내의 원소의 개수가 m개이므로 분산표를 구성하는 시간은 다음과 같다.

$$SDJ\_table\_build = m * move'$$

• 분산표에 의해 릴레이션 S를 분할하는 비용

분산표에 따라 릴레이션 S를 구별하고, 릴레이션 S의 튜플 위치를 검색하기 위해서 이진검색을 이용한다. 그러므로 비교와 이동을 위한 비용이 필요하다.

$$SDJ\_part\_S = \frac{\|S\|}{N} * (comp * \log_2 m + move)$$

• 결합을 위한 비용

필요한 릴레이션 R과 S의 일부를 메모리로 가져와 결합을 수행한다. 이때 메모리에 한번에 들어올 수 있는 튜플의 수는 $(M-1)*TP_R(M-1)$이고, S_i의 각 튜플과의 비교 횟수는 $\log_2((M-1)*TP_R)$이다. 동일한 결합키 값을 갖는 튜플이 여러 개 있을 경우에는 $(M-1)*TP_R*(JS*N)+1$번의 비교가 필요하다. 또한 결합된 결과는 결과를 저장하는 페이지로 이동되어야 한다. 그러므로 기억장소 내에서 결합을 위한 비용은 SDJ_join과 같다.

$$SDJ\_join = \frac{\|S\|}{N} * (comp * \log_2((M-1) * TP_R) + comp * ((M-1) * TP_R * (JS*N) + 1)) + \frac{\|R\|}{N} * \frac{\|S\|}{N} * (JS*N) * move$$

• 오버플로우의 총 처리 시간 비용

이러한 과정을 위한 총 비용은 SDJ_cost이다.

$$SDJ\_cost = SDJ\_table\_build + SDJ\_part\_S + SDJ\_join$$

4.3.6 노드 내에서의 결합 연산을 처리하기 위한 총 시간 비용

결합을 수행하는 데 필요한 비용은 다음과 같다.

$$Jcost = join\_cost + SDJ\_cost$$

4.4 총 비용

새로운 알고리즘에서 릴레이션 R과 S의 결합 연산을 수행하는 총 시간 비용은 다음과 같다.

$$total = Rcost + Scost + Jcost$$

5. 성능 분석

새롭게 제안한 부하균등 병렬 결합 알고리즘에 대해, 기존의 알고리즘들 중 비교적 성능이 우수한 방식으로 알려진 하이브리드 해쉬 결합 알고리즘과 병렬 분산 결합 알고리즘 그리고 변형된 하이퍼 퀵 정렬을 이용한 병렬 결합 알고리즘과 성능을 비교하여 성능 향상 정도를 보이고, 하이퍼큐브 차원의 변화와 릴레이션의 크기에 따른 성능 변화를 알아본다.

성능 분석은 위에서의 비용 모형에 기반을 두고 하였으며, 분석 모델에 사용된 매개 변수의 값은 표 2와 같다. 이 매개 변수는 기존의 다른 연구에서 사용된 변수 값과 동일하며, 동등한 조건에서의 비교를 위해 동일한 매개 변수 값을 사용한다.

표 2 분석 모델을 위한 매개 변수의 값

P	4 K bytes
TS_R , TS_S	128 bytes
TP_R , TP_S, TP_O	32 tuples/page
PS	64 K bytes
$comm$	8 M bits/sec
$comp$	3 $usec$
$move$	20 $usec$
P_{ovhd}	5 msec
IO_{read} , IO_{write}	25 msec
$keyA$	13 bytes

그림 2는 하이퍼큐브 시스템에 적절한 방송 알고리즘을 이용하여 새롭게 제안한 병렬 결합 알고리즘(New)의 전체 처리 비용을 릴레이션 R과 S의 처리 비용으로 구분하여 보이고 있다.

그림 3에서는 새로 제안한 모델(New)의 성능을 기존의 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘(MHPJ), 병렬 분산 결합 알고리즘(PDJ) 그리고 큐브 하이브리드 해쉬 결합 알고리즘(CHHJ)과 비교하여 보았다.

동일한 상태에서의 성능 비교를 위하여 기존의 알고리즘들이 주기억장치의 크기를 무제한으로 가정하였으므로 새로운 알고리즘에서도 결합을 위한 일부 디스크 접근 시간과 결합된 결과를 디스크에 기록하는 시간을 고려하지 않았다. 그림 3에서 보이는 바와 같이 새로운 결합 알고리즘은 병렬 분산 결합 알고리즘보다 우수하고, 하이브리드 해쉬 결합 알고리즘과 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘의 결과와는 거의 유사하다.

그러나 하이브리드 해쉬 결합 알고리즘과 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘은 시스템의 성능을 좌우하는 부하균등 문제를 해결하지 못하는 결정적 단점을 가지고 있다. 또한 릴레이션 R을 처리함에 있어 중간 노드에서 데이타의 불균형으로 인한 과부하 현상이 발생할 수 있다. 그렇지만, 새로운 결합 알고리즘은 이러한 부하균등 문제 및 데이타 불균형 문제를 완전히 해결할 수 있는 중요한 장점을 가진다. 또한 분산 결합 알고리즘보다 전체 통신시간을 충분히 줄일 수 있으며, 하이브리드 해쉬 결합 알고리즘에서는 구현하기 힘든 non-equijoin을 쉽게 구현할 수 있다는 장점을 지니고 있다.

그림 4에서는 릴레이션 R에 대해 새로운 알고리즘과 기존의 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘 그리고 병렬 분산 결합 알고리즘의 릴레이션 R에 대한 성능의 차이를 보이고 있다.

그림 4에서 알 수 있듯이 병렬 분산 결합 알고리즘에서는 하이퍼큐브의 차원이 증가할수록 릴레이션 R의 튜플이 이동되어야 할 거리가 멀어짐으로 인하여 통신 비용이 점점 증가하고 동시에 분산표를 작성한 후 이를 각 노드에 전송하기 위한 비용이 증가한다. 또한 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘에서는 하이퍼큐브 차원이 증가할수록 각 노드에서 지니는 튜플의 수가 감소되어 노드 내에서의 처리 비용이 점점 감소된다. 이러한 차이는 릴레이션 R의 튜플의 개수가 증가함에 따라 더욱 큰 차이를 보이게 된다. 하지만 부하균등 문제가 대두된다. 그러나 새로 제안한 알고리즘에서는 하이퍼큐브 시스템에 적절한 방송 알고리즘을 이용함으로써 부하균등 문제를 완전히 해결할 수 있으며, 동시에 기준값 결정과 튜플 분산 작업을 완전히 분리시킴으로써 중간 노드에서 발생할 수 있는 데이타 불균형으로 인한 과부하 현상을 완전히 해소하였다.

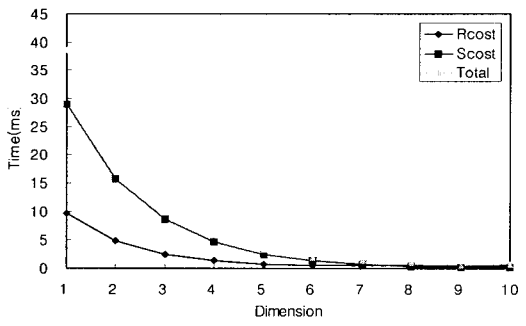


그림 2 새로운 알고리즘의 전체 처리 비용
($\|R\|=16\,K$, $\|S\|=64\,K$)

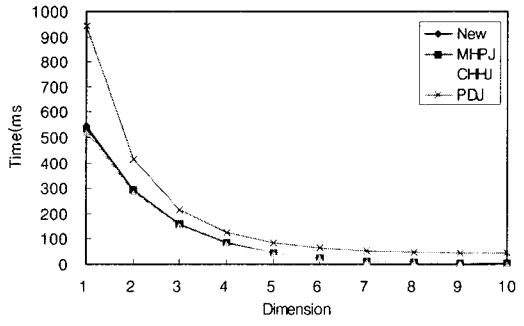


그림 3 기존의 알고리즘과의 비교
($\|R\|=128\,K$, $\|S\|=1024\,K$)

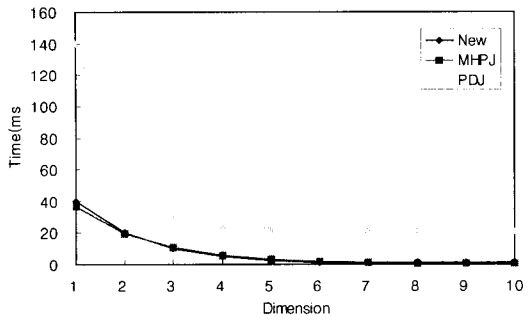


그림 4 릴레이션 R의 처리 비용의 비교($\|R\|=64K$)

표 3은 새로운 모델의 성능을 입증하기 위한 구체적 결과로써, 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘과 새로운 알고리즘의 릴레이션 R의 처리 비용 결과를 나타낸다.

새로운 알고리즘과 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘은 서로 근소한 차이를 보이고 있으나 하이퍼큐브 차원이 증가할수록 새로운 알고리즘이 약간 우수한 것을 볼 수 있다. 이는 변형된 하이퍼퀵 정렬을 이용하여 정렬을 수행할 경우 기준값 결정을 위해 하이퍼큐브 차원 수 만큼 반복해야 하지만, 새롭게 제안한 알고리즘은 하이퍼큐브 시스템에 적절한 방송 알고리즘을 이용함으로써 하이퍼큐브의 차원이 증가할수록 더 나은 결과를 보이게 된다. 그러나 어느 정도 차원이 증가하면 처리시간이 다시 증가하게 된다. 이는 각 노드에서 처리되는 튜플의 수가 너무 적기 때문이며, 각 노드에서 처리될 튜플의 수가 많아질수록 그 차원이 점점 증가하게 되므로 실제 데이타의 처리에서는 문제가 되지 않는다.

릴레이션 S를 처리하는 시간 비용은 기존의 알고리즘들과 비교하여 매우 미세한 차이를 보인다. 다시 말해 새로운 모델이 병렬 분산 결합 알고리즘 보다는 약간 우수하고, 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘과는 동일하다. 병렬 분산 결합 알고리즘의 경우에는 분산표를 검색한 후 각 튜플에 도달해야 할 목적지의 주소를 포함시켜 전달한다. 그로 인해 패킷의 크기가 커지므로 전송 비용이 증가하지만, 새로운 결합 알고리즘에서는 그러한 처리가 불필요하다. 단지 병렬 분산 결합 알고리즘에서의 분산표를 검색하는 비교 횟수보다 새로운 알고리즘들에서 각 튜플을 기준값과 비교하는 횟수가 더 많기 때문에 그에 대한 비용의 증가가 발생한다. 그렇지만 릴레이션 S를 처리하는 전체 비용 면에서는 약간 우수한 것을 알 수 있다.

제안한 알고리즘에서 릴레이션 R의 크기 변화에 따른 처리 비용을 살펴보면 그림 5과 같다. 노드의 수가 증가함에 따라 병렬성이 증가하고, 이로 인해 하이퍼큐브의 차원이 증가할수록 처리 비용이 감소된다. 그러나 어느 정도 차원이 증가하면 각 노드에서 처리되는 튜플의 수가 너무 적기 때문에 큰 효과를 얻을 수 없다.

그림 6에서는 릴레이션 S의 크기 변화에 따른 처리 비용을 비교하였다. 릴레이션 S의 크기가 증가함에 따라 전체 처리 비용이 증가되지만 노드의 수가 많아짐에 따라 병렬 처리의 효과가 뚜렷하게 나타나는 것을 알 수 있다. 이러한 효과는 릴레이션의 크기가 증가될수록 더욱 현저하게 나타난다.

릴레이션 S를 분배하는 과정에서 특정 노드에 데이타가 집중되는 문제는 발생될 수 있다. 이러한 경우에는

표 3 릴레이션 R의 처리 비용의 비교

Dimension	$\|R\|=64K$인 경우		$\|R\|=128K$인 경우		$\|R\|=512K$인 경우	
	New	MHPJ	New	MHPJ	New	MHPJ
1	39.920	36.776	81.343	74.596	337.419	306.735
2	20.170	19.804	41.069	40.122	170.232	164.640
3	**10.086**	10.623	**20.488**	21.486	**84.786**	87.969
4	**5.068**	5.691	**10.210**	11.474	**42.005**	46.828
5	**2.611**	3.062	**5.161**	6.130	**20.828**	24.863
6	**1.434**	1.675	**2.739**	3.297	**10.445**	13.192
7	1.048	0.974	**1.659**	1.811	**5.444**	7.022
8	1.032	0.625	1.332	1.064	**3.190**	3.782
9	1.419	0.466	1.548	0.696	**2.095**	2.097
10	2.525	0.409	2.588	0.529	**1.482**	1.561

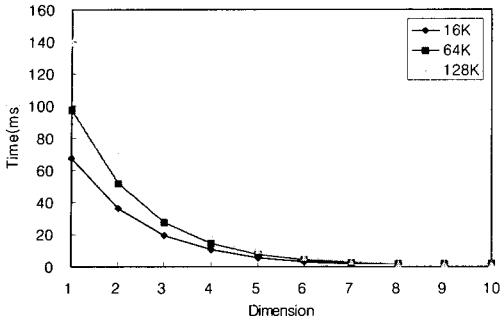


그림 5 릴레이션 R의 크기 변화에 따른 전체 처리 비용의 변화($\|S\| = 128K$)

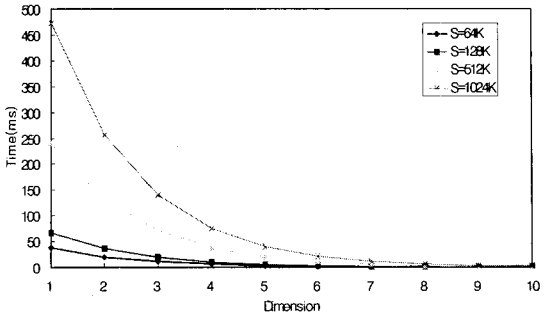


그림 6 릴레이션 S의 크기 변화에 따른 전체 처리 비용의 변화($\|R\| = 16K$)

병렬 분산 결합 알고리즘에서도 설명되어진 것처럼 인접한 노드들과의 재분산으로 해결될 수 있고, 이러한 재분산으로 어느 정도의 부하균등도 이룰 수 있다. 릴레이션 R과 S 중 어느 하나의 재분산이 이루어지면 다른 릴레이션도 마찬가지로 재분산이 필요하다. 그러나 이러한 처리를 위해 소모되는 비용은 전체 비용에 비해 미세하기 때문에 무시될 수 있다. 릴레이션 R을 이용하여 릴레이션 S를 분산하는 과정에서 극단적인 경우, 몇 개의 노드에 대부분의 튜플이 모여 있을 수 있는데, 이러한 경우에는 S를 먼저 재분산한 후 이 결과에 따라 R의 튜플들을 재분산 한다. 이처럼 극도로 편중된 경우에는 성능 향상이 크게 좋아지지 않는다. 그러나 대부분의 경우 튜플의 재분산은, 몇 개의 인접한 노드들 사이에서의 재분산으로 처리될 수 있다.

6. 결 론

하이퍼큐브 구조의 다중 처리기 환경에서 결합 연산을 효율적으로 처리하기 위한 새로운 병렬 결합 알고리즘을 제안하였다. 기존에 소개되어진 해쉬를 기반으로 하는 병렬 결합 알고리즘은 비교적 성능이 우수한 알고리즘으로 알려져 있으나, 자료가 한 곳으로 편중됨으로써 특정 노드에서 오버플로우가 발생되어 전체 성능을 저하시킨다. 본 논문에서는 부하균등이 잘 이루어지는 정렬 병합 결합 알고리즘의 특성을 이용하여 오버플로우 문제를 해결하였다.

병렬 분산 결합 알고리즘은 릴레이션 R을 정렬한 후 한 노드에 수집해야 하기 때문에 이를 위한 통신 비용이 증가되었고, 릴레이션 R을 한번에 보관할 수 있을 만큼의 기억 장소를 필요로 한다. 또한 분산표를 작성하여 각 노드에 전송해야하는 번거로움이 있었다.

이러한 단점을 해결하기 위한 방법으로 변형된 하이퍼퀵 정렬을 이용한 병렬 결합 알고리즘이 제안되었다. 이 알고리즘은 릴레이션 R의 처리를 위해 변형된 하이퍼퀵 정렬을 이용함으로써 전체적인 통신서간을 줄일 수 있었으나, 특정 노드에 자료가 편중되는 부하균등 문제를 완전히 해결하지 못하는 결정적 단점을 가지고 있다. 또한 기준값 결정 및 릴레이션의 분배를 병행하여 처리함으로써 중간 과정에서 데이타 불균형 현상이 발생할 수 있으며 이는 병렬 시스템의 특정 노드에 과부하를 야기시킴으로써 전체 성능을 크게 떨어뜨리는 결과를 초래하게 된다.

새롭게 제안한 알고리즘은 이러한 과정을 효율적으로 처리함으로써 부하균등 문제 및 데이타 불균형으로 인한 과부하 문제를 완전히 해결하였으며 동시에 전체적인 처리시간을 줄임으로써 전체 성능을 향상시켰다. 릴레이션 R을 처리하는 과정에서 하이퍼큐브 시스템에 적절한 방송(broadcasting) 알고리즘을 이용하여 튜플 전체를 한 곳으로 수집할 필요가 없으며, 튜플 전체를 보관하기 위한 기억 장소도 필요로 하지 않는다. 또한 기준값이 분산표의 기능을 수행함으로써 별도의 분산표를 작성하지 않아도 된다. 릴레이션 S를 처리하는 과정에서 튜플을 해당되는 노드로 전달하기 위해 목적지의 주소 값을 갖지 않고 이동하므로 통신 비용을 감소시켰다. 위와 같이 릴레이션 R과 S를 처리하는 과정에서의 비용 감소로 전체 통신 비용을 충분히 감소시켰다.

따라서 새롭게 제안한 알고리즘은 병렬화 성능의 최대 주안점인 부하균등 문제와 데이타 불균형으로 인한 과부하 문제를 해결하고 결집효과의 특성을 수용함으로써 기존의 알고리즘-정렬 병합 결합 알고리즘, 하이브리드 해쉬 결합 알고리즘, 분산 결합 알고리즘-의 장점만을 이용한 효율적인 알고리즘으로 전체 성능이 향상되었다.

참 고 문 헌

[1] Soon M. Chung, Arindam Chatterjee, "Performance Analysis of a Parallel Distributive Join Algorithm on the Intel Paragon", *International Conference on Parallel and Distributed Systems,* 1997.

[2] H.I.Choi, B.M.Im, M.H.Kim, Y.J.Lee, "An Efficient Parallel Join Algorithm Based on Hypercube Partitioning", *Proceedings of the 3rd Conference on Parallel and Distributed Information Systems,* pp50-57, 1994.

[3] S. Cho, Y. Weon, M. Hong, "A Parallel Join Algorithm Using Hyper Quick Sort", *Proceedings of the Ninth IASTED International Conference on Parallel and Distributed Computing and Systems,* USA, pp97-106, October, 1997.

[4] Soon M.Chung and Jaerheen Yang, "A Parallel Distributive Join Algorithm for Cube Connected Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 7(2), pp127-137, 1996.

[5] D.J. DeWitt, and R. Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proceedings of the 11th International Conference on Very Large Data Bases*, pp151-162, August, 1985.

[6] M.Negri and G.Pelagatti, "Distributive join : A new algorithm for joining relation", *ACM Transa tions on Database Systems*, 16(4), pp655-669, 1991.

[7] Edward R.Omiecinski, Eileen Tien Lin, "The Adaptive-Hash Join Algorithm for A Hypercube Multicomputer", *IEEE Transactions on Parallel and Distributed systems*, 3(3):334-349, May 1992.

[8] Youngsun Weon, Seokbong Cho, Kyuock Lee, Youngkwon Cha, Man Pyo Hong, "Performance Analysis of an Advanced Parallel Join Algorithm on Hypercube Systems", *Journal of KISS*, 26(6), 1999.

[9] Patrick Valduriez,Georges Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine", *ACM Transactions on Database Systems*, 9(1), pp133-161, March 1984.

[10] D.J.DeWitt, R.H.Katz, F.Olken, L.D. Shapiro, M.R.Stonebraker, D.Wood, "Implementation Techniques for Main memory database system", *Proceeding of SIGMOD Conf.*, pp1-8, June, 1984.

[11] Leonard D.Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems,* 11(3), pp.239-264, September 1986.

[12] Priti Mishra and Margaret H.Eich, "Join Processing in Relational Databases", *ACM Compu ing Surveys*, 24(1), pp.63-113, March 1992.

[13] Vipin Kumar, Introducing to parallel Computing design and analysis of parallel algorithms, *The Benjamin / Cummings Publishing Company Inc.*, 1994.

[14] Hui-I Hsiao, Ming-Syan Chen, Philip S. Yu, "Parallel Execution of Hash Joins in Parallel Databases", *IEEE Trans. Parallel and Distributed Systems*, 8(8), pp872-883, Aug. 1997.

[15] Donovan A.Schneider, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared- othing Multiprocessor Environment", *Proceeding of the 1989 SIGMOD Conference* ACM, pp.110-121, 1989.

원 영 선
1993년 경기대학교 전자계산학과 이학사. 1995년 경기대학교 전자계산학과 이학석사. 2001년 아주대학교 컴퓨터공학과 공학박사. 현재 충남대학교 정보통신공학부 BK21 전임교수. 관심분야는 병렬처리, 병렬알고리즘

홍 만 표
1981년 서울대학교 계산통계학과 이학사. 1983년 서울대학교 계산통계학과 이학석사. 1991년 서울대학교 계산통계학과 이학박사. 1983년~1985년 울산공과대학 전자계산학과 전임강사. 1985년~현재 아주대학교 정보 및 컴퓨터공학부 교수. 1993년~1994년 미네소타대학 전자공학과 교환교수. 관심분야는 병렬처리