저속 네트웍 PC 클러스터상에서 NOW-Sort의 성능향상

(Enhanced NOW-Sort on a PC Cluster with a Low-Speed Network)

김 지 형 † 김 동 승 ††
(Jihyoung Kim) (Dongseung Kim)

요 약 병렬 외부정렬을 클러스터형 분산 컴퓨터에서 실행하는 경우에는 순수하게 주메모리에서 부분적인 정렬과 머지를 위해 실행되는 과정(순수 계산)뿐만 아니라 디스크로부터의 입출력 과정 및 각 노드들 간의 데이타 교환에 따르는 통신과정을 적절히 배치, 설계함이 필요하다. 그 주된 이유는 전체 수행시간이 순수 계산시간보다는 디스크 입출력에 소요되는 시간 및 통신의 소요시간의 비중이 크기 때문이다. 본 연구에서는 저속 네트웍 PC 클러스터를 계산도구로 하여 단위시간당 정렬 자료규모를 최대화함을 목표로 하여, 알고리즘적인 최적화를 통해서, 즉, 정렬 도중 통신과정에서 발생하는 지체시간을 최소화하여 전체적인 통신 성능을 높이고, 디스크 입출력 작업은 전송 규모와 횟수를 조절하여 계산과 통신작업등과의 중첩정도를 극대화시켜 외부정렬의 성능을 개선하였다. 실험 결과 새 알고리즘이 기존의 NOW-sort 알고리즘[1]에 비해서 동일한 PC 클러스터 환경에서 최대 45% 정도까지 실행시간을 단축시킬 수 있고, 확장성 면에 있어서도 더 우수한 것을 확인하였다.
키워드 : 정렬, I/O, 클러스터 컴퓨팅, 병렬 알고리즘

Abstract External sort on cluster computers requires not only fast internal sorting computation but also careful scheduling of disk input and output and interprocessor communication through networks. This is because the overall time for the execution is determined by reflecting the times for all the jobs involved, and the portion for interprocessor communication and disk I/O operations is significant. In this paper, we improve the sorting performance (sorting throughput) on a cluster of PCs with a low-speed network by developing a new algorithm that enables even distribution of load among processors, and optimizes the disk read and write operations with other computation/communication activities during the sort. Experimental results support the effectiveness of the algorithm. We observe the algorithm reduces the sort time by 45% compared to the previous NOW-sort[1], and provides more scalability in the expansion of the computing nodes of the cluster as well.
Key words : sort, cluster computing, parallel algorithm

1. 서 론

정렬은 데이타베이스, 과학기술연산 및 인터넷 서비스 등 다양한 컴퓨터 응용분야에서 사용되는 핵심적이면서도 필수적인 연산인 동시에 컴퓨터의 연산성능을 포함한 I/O 성능을 측정하기 위한 주요한 수단으로서 사용되어왔다[2,3]. 정렬 중에서도 복수의 컴퓨터를 통합하여 작업을 분할 처리하는 병렬 및 분산 정렬방식은 다중/분산 컴퓨터에 의한 성능 개선을 목표로 연구 개발되어 왔다. 이 경우 정렬의 수행시간은 각 프로세서의 성능, 알고리즘, 통신 대역폭, 연산과 통신의 중첩정도 등에 의해서 좌우되며, 프로세서간의 통신과정이나 통신양식, 부하의 배치의 불균등 정도 등에 따라서도 크게 달라진다[1,4,5,6].

데이타 규모가 주메모리 용량을 초과하지 않는 범위내에서 자료를 정렬하는 경우(내부정렬: internal sort인

· 본 연구는 학술진흥 재단의 선도연구(E0087, 1999년)의 연구비 지원에 의해 수행되었음.
† 비 회 원 : 고려대학교 메카트로닉스 협동과정
jhkim@pumpkinnet.co.kr
†† 종신회원 : 고려대학교 전기공학과 교수
dkim@classic.korea.ac.kr
논문접수 : 2002년 4월 9일
심사완료 : 2002년 8월 12일

경우), 머지(merge) 정렬, bitonic 정렬, odd-even *transposition* 정렬, 샘플정렬 [2,4,5,7] 등 다양한 기법의 병렬/분산 정렬 알고리즘이 고안되었다. 이 경우 연산시간은 최대부하를 갖는 노드에 의해 좌우되고, 특히 분산메모리 구조의 컴퓨터에서는 순수 계산시간(key의 비교연산에 소요되는 시간) 보다 프로세서 상호간의 자료교환에 드는 interprocessor 통신시간이 총 실행시간에 차지하는 비중이 높아 각 노드간의 부하 균등화 실현과 더불어 통신시간을 줄이는 것이 주 연구대상으로 되어 왔다[1,8,9,10].

한편 주메모리보다 자료량이 많아 디스크에서 한번에 모든 데이타를 메모리로 읽어들여 처리하지 못하는 외부정렬(external sort)의 경우에는 디스크에 있는 데이타를 적절한 크기로 분할하여 메모리에 로드하고 처리하여 다시 디스크에 기록함을 반복하게 된다. 이때 주메모리에 로드된 데이타를 정렬하는 데 드는 시간에 비해 그것을 파일에서 읽고 다시 디스크에 기록하는데 소요되는 시간이 훨씬 커서, 메모리 내에서 정렬하는 알고리즘의 속도보다 자료가 최종시까지 디스크에서 몇 번 읽혔다가 다시 디스크에 기록되는가가 중요한 성능지수 역할을 한다. 아울러 대량의 자료가 원래 위치에서 최종 노드의 디스크에 기록될 때 까지 여러번 이동하게 되어 네크웍을 통한 총 데이타 전송량 및 그 전송회수를 줄이는 것이 전체 소요시간을 단축하는데 관건이 된다[9,11].

지금까지 병렬/분산 외부정렬을 위한 다양한 방식이 제시되었다. Parallel Binary Merge Sort[12]는 분산 컴퓨터에서 머지 정렬(mergesort)을 실행하는 경우로서, 각 프로세서를 2진 트리의 leaf 노드로 대치시켜 각 leaf 노드에서 지역적인 정렬을 통해 각 노드의 데이타를 정리하고, 인접한 leaf 노드 2개씩 짝을지어 머지하여 2배의 크기로 머지결과를 얻고, 다시 대응되는 짝을 머지하여 4배 크기로 확대하고, 이런 방식을 반복하여 정렬결과를 얻는 구현방법이다. 그러나 P 노드가 존재할 때 적어도 log P 단계의 머지과정이 반복되어야 하는데, 매 단계마다 이동되는 데이타량이 전체 정렬대상의 절반정도가 되고, 또 데이타 규모가 주메모리를 초과하기 때문에 매 단계마다 데이타를 파일에서 읽어내고 머지한 후 다시 저장하지 않으면 안된다. 더욱이 단계가 계속될수록 머지에 참여할 노드수가 절반씩 줄어드는 것은 병렬/분산 실행효율을 급격히 떨어져 실행시간이 매우 커져 실용성이 없다.

Heapsort를 외부정렬에 적용한 external heapsort[13] 방식도 외부정렬의 고속화를 위하여 고안되었으나 이 역시 heap를 갱신하는 과정에서 자료가 M개의 동일규모의 파일(페이지)로 나누어져 저장되어있을 때, heap 재조정시마다 각 페이지 파일을 log M번 접근하게 되어 고속화에는 적합한 알고리즘이 되지 못한다.

클러스터의 확장성을 활용해 대규모 데이타 (수십 GigaByets ~ TeraBytes)정렬을 실행하는 것중 현재 가장 주목을 받고 널리 사용되고 있는 알고리즘이 NOW-Sort[9]이다. NOW-Sort는 고속연결망(Myrinet)으로 구성된 workstation 클러스터[8]에서 실행되도록 개발된 방식인데, bubble sort와 radix 정렬 알고리즘을 이용하고 Glunix상에서 최적화된 통신 프리미티브(AM: active message[14]) 및 디스크 입출력 알고리즘을 사용하여 높은 성능을 얻는다. 이 방식은 실행과정에서 각 key당 디스크 접근회수가 2R, 2W (2 read accesses, 2 write accesses)가 되어 위에 소개된 것 중 접근회수가 제일 작다[11,15,16]. 자세한 알고리즘은 2절에서 다룬다.

SPsort[10]의 경우 NOW-Sort 방식을 488 노드(1957 프로세서 내장)의 IBM RS6000에 적용시켜 1Terabyte의 자료를 1,057초에 정렬한 기록을 세웠고, 전체 정렬이 2R, 2W의 디스크 I/O를 통해 실행되었고, 순수 계산시간은 이 디스크 입출력과정에 소요된 시간에 중첩되어 전체 소요시간이 디스크/네트웍 통신시간으로 결정되었음을 보고하였다.

본 논문에서는 NOW-Sort를 PC 클러스터에 구현한 직전 연구결과[1]를 기반으로, 열악한 네트웍 전송속도를 감안하여 클러스터 내의 노드간 통신의 스케쥴링을 재설계하고 통신 타이밍과 트래픽을 적절히 분산시켜 네트워크의 혼잡도(congestion), 통신지연을 줄이며, 계산, 통신작업, 디스크 입출력의 세 작업의 중첩을 극대화함으로서 전체 정렬 수행시간을 단축한 연구결과를 제시한다.

이후의 구성은 다음과 같다. 제2절에서는 NOW-Sort 정렬 알고리즘을 소개를 하고 제3절에서는 본 논문에서 적용한 최적화 기법에 대해서 자세히 소개한다. 제4절에서는 제시된 최적화 기법을 성능평가를 위한 실험환경 및 실험결과를 알아보고 이전 알고리즘과 비교, 분석해 본다. 제5절에서는 결론을 기술하고, 앞으로의 연구방향을 제시한다.

2. NOW-Sort 외부정렬

외부정렬(*external sort* 또는 *disk sort*)은 디스크 상에 존재하는 데이타를 메모리로 읽어 들여 내부정렬을 수행한 후 그 결과를 다시 디스크에 기록하는 방식의 정렬을 말하며, 전체 실행시간은 내부정렬의 실행시간과 디스크 입출력에 걸린 시간을 합하여 구한다. 병렬외부

정렬을 수행할 때 총 데이타의 크기가 시스템의 전체 주메모리 규모보다 작은 경우 전체 데이타에 대해 한 번의 읽기와 한 번의 쓰기작업만을 거치면 정렬을 마칠 수 있어 one-pass 방식[14]이라고 한다. 한편 정렬 대상 자료의 크기가 시스템의 전체 주메모리 용량을 초과하는 경우에는 전체 데이타를 한 번에 메모리에 올릴 수 있는 크기의 그룹으로 나눈 후 각 그룹을 하나씩 메모리에 올려서 정렬을 수행한 후 (선택한 알고리즘에 따라) 각 그룹들을 머지정렬하여 최종 정렬된 결과를 얻게된다. 머지과정은 다시 디스크 입출력을 여러번 반복하여 진행되기 때문에, 그룹으로 분할할 때 key의 크기에 따라 나누고, 각 분할된 그룹내의 데이타를 각각 정렬하면 머지를 피할 수 있게 된다. 이 경우는 최초 입력 데이타를 읽어 들이는 과정 및 최종 출력 데이타를 저장하는 과정 이외에도 중간 결과물을 디스크에 저장하였다가 다시 메모리로 로드하고 처리 저장하는 과정을 거치기 때문에 two-pass 방식[9]이라고 한다.

NOW-Sort는 외부정렬을 대규모 분산컴퓨터에 적용하기 위해 고안된 것으로 원래 워크스테이션 클러스터상에서 2단계로 나누어 실행되도록 설계되었다. 제1단계에서는 각 노드가 자신의 디스크에 있는 데이타를 분류해서 크기별로 담당할 노드에게 전송하는 과정이다. 우선 key를 하나씩 읽어 그 key의 상위 수 비트를 써서 bucket 정렬하여 최종적으로 저장될 노드를 정하고 (즉, P0가 상위 비트 값이 제일 작은 key들을 맡고, P1이 그 다음, PP-1 이 제일 큰 key를 맡는 형태로) 주메모리상에 임시 저장했다가 bucket 별로 적정규모 이상으로 key가 차면 그 내용을 네트웍을 통해 해당 노드로 전송한다. 수신측 노드에서는 자신의 데이타에 대한 분류작업과 병행해서, 외부 타 노드로부터 전송되어 온 key를 다시 부분적인 정렬과정을 거쳐 일정크기(주메모리 크기)가 초과하지 않도록 잘라서 디스크에 기록하게 된다. 1단계에서 모든 노드가 자신이 맡은 자료의 분류와 전송, 그리고 저장을 마치면 2단계 과정이 시작된다. 제2단계에서는 각 노드가 자신의 디스크에 크기별로 분할되어 저장된 파일을 차례로 읽어들여 지역정렬을 수행하여 다시 저장하는 과정을 반복하고 마지막 파일에 대한 처리가 끝나면 정렬이 종료된다. 이 알고리즘에서는 각 key가 디스크 입출력면에서 볼때, 1단계의 분류과정에서 1번 Read, 최종위치의 노드에 도착하여 1번 Write, 2단계에서 주메모리에 로드될때 1번 Read, 끝으로 정렬되어 디스크에 저장될 때 1번 Write가 되어 key 하나당 2R, 2W (2 read accesses, 2 write accesses) 만큼의 디스크 접근이 실행되는데 이는 현재까지 알려진 외부정렬 알고리즘 가운데 최소치이다 [11,15,16].

본 논문은 NOW-Sort를 PC 클러스터에 구현한 이전연구[1]의 외부정렬 알고리즘을 수정하여 열악한 네트웍 조건하에 성능을 높이도록 시도된 연구 결과이다. 사용된 알고리즘은 전체 데이타를 노드별로 분할할 때 각 노드가 키의 값의 범위(구간)에 따라 겹치지 않게 (disjoint) 나누는 방식으로, 각 노드는 데이타를 자신의 디스크로부터 읽어 들여 그 키 값에 따라 (균일 분포의 경우, P개 노드를 사용할 때 정수형 키인 경우 최상위 log_2P bits를 그 key를 맡을 노드의 ID(PID)로 정한다. 한편 일반적인 데이타 분산을 고려하면 sampling을 통해 key값의 분할경계치를 정한 후 각 노드에게 broadcast 하여 사용하게 된다), 노드별로 분할 전송하고, 분할이 끝난 후 각 노드는 할당된 데이타들을 내부적으로 정렬하여 최종 정렬 결과를 얻는다. 이때 각 노드의 데이타가 주메모리 용량을 초과하여 데이타를 디스크에 임시로 저장하게 되고 이후 데이타의 분배가 완료되면 bucket의 순서에 따라 읽어 들여 내부정렬을 수행한 후 그 결과를 최종 파일에 순서대로 기록하여 하나로 만들게 된다. 따라서 one-pass방식에 비해서 디스크 입출력 시간이 길어지게 되어 이부분의 최적화가 전체 정렬의 성능향상에 더 큰 비중을 차지한다.

3. NOW-Sort 알고리즘의 최적화

3.1 이전 연구의 문제점

클러스터 시스템에서 각 노드를 연결하는 네트워크는 Myrinet이나 Gigabit Ethernet과 같은 고속(Giga bps) 장비들이 있지만 고가이기 때문에 대부분의 PC 클러스터에서는 저가의 100Mbps Fast-Ethernet으로 네트웍을 구성한다. 이 경우 네트워크 대역폭은 상용 병렬 컴퓨터에서 사용되는 전용 연결망에 비해 매우 작기 때문에 노드간 통신에서 동기화 반복정도나 지연은 전체성능에 막대한 악영향을 주게 되어 알고리즘 설계, 구현시 각별한 주의가 필요하다.

연구 대상의 외부 정렬에서는 디스크에서 읽어 들인 최초의 데이타를 키의 영역에 따라 담당할 노드로 전송하는 과정에서 통신 지연 문제가 생긴다. 각 노드는 일정량의 데이타를 디스크로부터 읽어 들여 각 레코드에 대해서 목적지 노드를 판별한 후, 레코드를 해당 송신 버퍼로 옮겨 저장한다. 이 과정은 일정한 규모의 데이타를 읽어 들일 때까지 반복되며 그 후에는 송신 버퍼에 저장된 내용들을 다른 노드로 전송한다. 이때 각 노드가 주고받게 될 데이타의 양을 사전에 서로 알려주는

all-to-all 통신과정[1]을 거친 후 또 한번의 all-to-all 통신과정으로 실제 데이타를 교환한다. 문제는 all-to-all 통신방식에 의해 전체 노드의 통신이 지연된다는 점이다. 매번 all-to-all 통신을 할 때마다 데이타 전송량의 차이나 시스템 진행속도의 차이 등에 따라 가장 느린 노드가 전송을 완료할 때까지 다른 노드들은 기다려야 하기 때문에 barrier synchronization을 실행한 것과 같은 지연이 발생하게 된다. (실제 실험 결과, 16노드에서 총 16GB(노드 당 1GB)의 데이타에 대해 정렬을 수행하는 과정에서 노드 당 1GB의 key를 교환하는 데는 약 175초가 소요되는 반면, 노드 당 약 4KB 정도밖에 안되는 송신버퍼내의 데이타 크기 관련 정보를 교환하는데 약33초가 소요되는 것으로 봐서, 통신 시 순수하게 데이타의 전송에 소요되는 시간 이외에도 동기화에 따른 오버헤드가 많은 비중을 차지하고 있음을 알 수 있다.) 이러한 동기화가 디스크 데이타를 읽어 각 노드에게 재분배할 때마다 두 번씩 나타나고, 이러한 분배과정이 데이타 재배치 완료까지 여러 번 반복되기 때문에 오버헤드(통신지연)가 누적되게 된다. 두번째는 노드 간 데이타 송수신 작업이 불필요하게 동기화되어 네트워크 부하(트래픽)가 특정 시간대에 집중해서 발생하게 되는 점이다. 다른 노드들에게 자신이 얼마만큼의 데이타를 보내주게 될 것인지를 all-to-all 통신을 통해서 알려주는 과정은 전체 노드에서 동일한 시점에 종료되게 되어, 그 직후에 진행되는 데이타 송수신 과정이 저절로 동기화되는 결과가 된다. 따라서 매번 데이타 송수신이 재개될 때마다 네트워크의 트래픽이 급격히 증가했다 감소하는 형태로 나타나게 되고, 네트워크 대역폭이 작은 PC 클러스터의 경우에는 한꺼번에 몰리는 트래픽을 제때 처리하지 못함으로서 추가적인 전송지연이 발생하는 것이다. 끝으로, 데이타 송수신시 개별 통신작업들 간의 중첩은 잘 이루어지지만 통신과 다른 작업 간의 중첩이 부족해서 통신수행 중에는 CPU나 디스크와 같은 다른 자원이 놀게(idling) 된다는 점이다.

3.2 개선된 통신 방식

새로운 방식에서는 항상 송신 버퍼에 일정한 양의 데이타가 저장되었을 때 송신을 시작한다. 또한 각 노드는 non-blocking 송수신 작업들의 완료시점에 상관없이 계속 작업을 수행해 나갈 수 있도록 하였다. 구체적으로 살펴보면, 우선 모든 노드들은 자신의 작업을 수행하면서 동시에 다른 프로세스가 보내주는 데이타를 받을 수 있도록 non-blocking 수신 작업을 개시(initiate)한다. 그 다음 디스크로부터 일정량씩 데이타를 읽어 들여 목적지 노드에 따라 해당되는 송신 버퍼에 저장하고 버퍼내의 데이타가 일정량이 되면 non-blocking 방식으로 해당 노드로 데이타를 전송해주게 된다. 한편 non-blocking 수신 작업에서는 읽어 들인 데이타에 대한 처리가 끝날 때마다 수신 완료된 작업이 있는지를 확인하여 해당 데이타를 처리한 후 다시 새로운 non-blocking 수신 작업을 개시한다. 그 후 각 노드는 모든 데이타를 처리할 때까지 위 과정을 반복하게 된다.

새로운 방식에서는 데이타의 전송 시점이 송신버퍼에 저장된 데이타량이 일정 크기가 되었을 때가 되어, 한 노드에서 볼 때 외부의 각 노드로의 데이타 전송 시점이 서로 달라질 뿐만 아니라, 노드 간에도 전송시점이 서로 달라진다. 확률적으로 볼 때 전송이 시작되는 시점은 특정 시각에 편중됨을 피하게 되고, 발생하는 트래픽의 형태도 시간에 따라 밀집되어 나타날 가능성이 작아진다. 또 일정한 크기로 데이타를 전송함으로써 송신 데이타의 크기에 대한 정보 교환이 필요 없어졌고, 데이타 전송의 완료 여부와 상관없이 계속적으로 작업이 진행될 수 있도록 함으로서 동기화부분들이 사라졌다. 마지막으로 네트워크 외의 다른 자원을 사용하는 여러 작업도 통신과 병렬적으로 수행되도록 하고, non-blocking 통신이 끝나는 시점에 의존적인 부분을 최소화하여 통신이 끝날 때까지 대기하는 시간을 단축시켰다. 그 결과 통신과 그 외의 작업 간의 중첩 비율이 높아지고 전체 실행시간 중 통신에서의 대기시간의 비중이 대폭 줄어들게 되었다.

3.3 디스크 입출력 방식의 개선

디스크 입출력은 외부정렬에서 가장 수행시간을 많이 차지하는 부분으로 인식되었으나, 실험 대상 PC 클러스터에서는 4바이트의 unsigned integer의 256MB의 데이타를 기수정렬하는 데 걸리는 시간은 약 38초였던 반면에 256MB의 데이타를 디스크에서 읽거나 쓰는데 걸리는 시간은 각각 약 10초정도에 불과하였다. 이 결과는 입력 데이타의 형태 및 내부정렬 알고리즘의 최적화 정도의 차이에 의해서 나타난 것으로 보이지만[2], 본 실험

1) All-to-all 통신이란 어떤 그룹 내에 속한 모든 노드들이 각각 자신 이외의 나머지 노드들과 데이타를 주고받는 통신방식을 의미한다. 통신의 종료시점은 모두가 교환을 마쳤을 때가 되고 따라서 가장 느린 노드에 의해 통신 소요시간이 좌우된다.

2) 이러한 결과는 실험 조건에 따라서 달라질 수 있다. 일반적으로 정렬 실험시 사용되는 데이타의 형태는 10바이트 키를 포함한 100바이트의 레코드로 구성되는데 정렬시 실제 사용되는 키의 양은 전체 데이타의 크기의 1/10 정도밖에 안되므로 전체가 키로만 구성된 4바이트 unsigned integer를 정렬하는 경우에 비해 그 실행시간이 많이 줄어들 것이다. 또 알고리즘의 최적화 정도도 여기에 큰 영향을 미칠 수 있다.

환경 하에서는 디스크 성능이 우수하여 디스크 입출력 시간 자체를 줄이기보다는 계산 및 통신작업과의 중첩을 통해서 디스크 입출력에 걸리는 시간을 숨길 수 있도록 하는데 중점을 두고자 한다.

One-pass 방식의 경우에는 데이타의 규모가 작기 때문에 디스크 캐쉬로 인한 작업의 시간단축 효과를 많이 볼 수 있어 디스크로의 쓰기 작업에 드는 시간은 상당히 작다. 따라서 디스크 작업을 중첩시키는 것은 큰 개선효과를 얻기 어렵다고 판단되어 최초의 데이타 읽기 과정에 대해서만 개선된 방식을 적용하였다. Two-pass 방식의 경우에는 전체 데이타에 대한 두 번의 읽기 작업과 두 번의 쓰기 작업을 거치는데 데이타의 규모가 상당히 크고 메모리 사용량 및 실행 시간등도 많이 늘어나게 되어 두 번의 읽기와 쓰기 모두에 대해서 중첩을 시도하였다.

3.4 새 알고리즘

통신 및 디스크 입출력 방식을 개선한 외부정렬 알고리즘은 다음과 같이 사전준비단계, 데이타 분배단계, 지역정렬단계의 3단계로 실행된다.

사전준비단계(Phase 0)

Task 0. Sampling - 불균일 분포의 데이타 정렬시에는 우선 각 노드에서 표본을 추출하여 균등분배가 되도록 각 노드가 맡을 키의 영역을 정하고 또한 노드 내에서 크기별 분류, 저장을 위한 sub-bucket 파일간의 키 영역도 결정한다.

데이타 분배단계(Phase 1)

이 단계는 데이타를 차례로 읽어 들여 크기 별로 해당 노드에 재분배한 후 도착한 데이타를 sub-bucket 별로 분류하여 디스크에 파일로 기록하는 과정이다. CPU, 디스크, 네트웍 자원의 효율적인 사용을 위해서 다음 네 가지 작업이 동시에 실행된다.

Task 1A. Data Classification & Thread Control - 계산 및 다른 thread들의 제어를 담당하는 주 thread로서, 별도의 read thread에게 자신의 디스크 상에 가지고 있는 입력 데이타 파일에 대한 읽기 명령을 내리고 데이타가 준비되면 이를 목적지별로 분리하여 해당 송신 버퍼에 저장한다. 송신버퍼에 저장된 데이타의 양이 일정규모에 도달하면 MPI의 non-blocking 송신 작업을 통해 데이타를 전송한다. 또한 non-blocking으로 수행중인 수신 작업들의 상태를 계속 체크하여 수신이 완료된 작업에 대해서는 수신된 데이타를 각각의 sub- bucket 파일 별로 주메모리의 쓰기 버퍼에 저장하고, 새로운 수신 작업을 개시하여 데이타 수신이 계속되도록 한다. 주메모리 상의 쓰기 버퍼가 다 차게 되면 별도의 write thread에게 파일 쓰기 요청을 보내어 디스크에 저장하도록 한다.

Task 1B. Data Distribution - 주 task에서 MPI의 non-blocking 함수를 호출하여 이루어지는 작업으로서 각 노드 간에 데이타를 주고받는 역할을 하게 된다.

Task 1C. File Read - 독립적으로 수행되는 thread인 read task는 주 thread로부터 파일 읽기 요청이 있게 되면 해당 부분을 읽어 들여 주 thread에게 전달한다. 이 때 읽어 들인 데이타를 전달받을 때까지 주 thread가 대기하는 시간을 최소화하기 위해 주 thread로부터의 읽기 명령의 전달은 하나의 읽기 작업이 끝난 직후 바로 다음에 사용될 데이타에 대한 읽기 작업이 개시되도록 스케쥴한다.

Task 1D. File Write - file read와 같이 독립적으로 수행되고 주 task의 요청에 따라 쓰기 버퍼 내의 데이타를 디스크에 기록하고 버퍼를 비워주는 역할을 한다.

그림 1은 이 단계에서 각 노드가 수행하는 작업을 보인 것으로, 편의상 한 노드를 file read 및 data send 작업을 수행하는 경우와 date receive 및 file write 작업을 수행하는 경우로 분리하여 표시하였다.

지역정렬단계(Phase 2)

이 단계는 디스크에서 최초로 sub-bucket file의 데이타를 읽어 들여 정렬한 후 다시 디스크에 기록하는 과정을 반복하는데, 시간을 단축하도록 다음의 세 작업이 병렬로 진행된다.

Task 2A. Local Radix Sort & Thread Control - 별도의 read thread에게 키의 영역에 따라 순차적으로 sub-bucket 파일을 읽어 들일 것을 요청한다. Read thread로부터 데이타를 전달받으면 기수정렬을 수행한 후, write thread에게 넘겨 디스크 쓰기를 진행토록 한다.

Task 2B. File Read - 독립적으로 수행되는 thread로서 주 thread로부터 읽기 요청이 들어오면 해당 sub-bucket 파일을 읽어 들여 주 task에게 전달한다. 이때 하나의 읽기 작업이 끝나면 바로 다음 sub-bucket 파일에 대한 읽기 요청이 전달 되도록 하여 읽어 들인 데이타에 대한 기수정렬이 수행되는 동안 다음 데이타에 대한 읽기 작업이 병행되도록 한다.

Task 2C. File Write - 역시 독립적으로 수행되는 thread로서 주 task로부터 쓰기 요청이 들어오면 해당 sub-bucket의 내용을 디스크상의 최종 출력 파일에 기록한다. 이때에도 항상 하나의 sub-bucket에 대한 기수정렬이 완료되었으면 기록 작업은 그 다음 데이타에 대한 기수정렬이 진행되는 동안 병렬적으로 이루어진다.

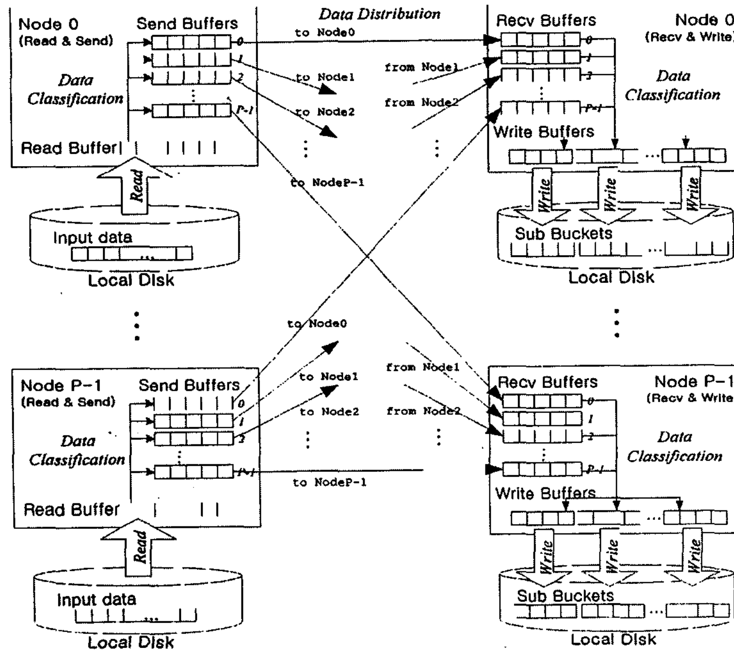

그림 1 데이타분배단계

3.5 실행 시간 예측

개선된 two-pass 알고리즘에 대한 실행 시간을 예측하기위해 우선 사용될 인자(parameter)들을 다음과 같이 정의한다.

S_n: 한 노드에 저장된 총 데이타의 크기

P: 정렬에 참여하는 노드(프로세서)수

B_r: 디스크로부터 데이타를 읽어 들일 때의 대역폭 (read bandwidth)

B_w: 디스크에 데이타를 기록할 때의 대역폭 (write bandwidth)

B_c: 노드 간 통신 채널의 양방향 통신 대역폭 (bidirectional communication bandwidth)

B_s: 한 노드가 단위시간당 기수정렬로 정렬할 수 있는 데이타의 크기

이상의 값들을 이용하여 two-pass방식의 전체 실행 시간을 구해보면 다음과 같다[17].

$$T_{total} = (aT_{read1} + bT_{comm} + cT_{write1} + \frac{1}{2}T_{comp}) + (dT_{read2} + eT_{write2} + T_{comp}) \quad (1)$$

여기서 T_{read1}, T_{write1}과 T_{read2}, T_{write2}는 각각 1단계 및 2단계에서 전체 파일의 읽기와 기록에 소요되는 총 시간을 의미하며, T_{comm}, T_{comp}는 각각 통신과 내부정렬(기수정렬)에 소요되는 총 시간을 뜻한다. 또 a에서 e까지는 작업의 중첩 정도에 따라 정해지는 상수이다.

Two-pass방식의 경우에는 첫 번째 단계에서 파일읽기, 데이타의 노드별 sub-bucket별 분류작업, 통신, 파일쓰기 작업이 존재하고 이들 작업은 모두 중첩되어 실행된다. 두 번째 단계에서는 파일읽기, 내부정렬, 파일쓰기 작업이 존재하고 이들 또한 모두 중첩이 되어 실행된다. 이중에서 데이타의 노드별, bucket별 분류에 드는 시간은 이 과정이 일종의 기수정렬 과정임을 생각해볼 때, 전체 4바이트에 대해서 4번의 기수정렬을 반복하는 최종정렬 단계에 비해서 노드별 분류시 한번씩, bucket별로 분류시 마다 한번씩의 기수정렬을 행하는 형식으로 진행된다. 따라서 제2단계에서의 내부정렬에서 소요되는 시간의 절반 정도가 들것으로 추산할 수 있다. 통신시간의 경우는 균일분포를 가정한다면 하나의 노드가 최초에 가진 데이타 중 다른 노드와 교환해야 될 분량은 전체 데이타 중의 $\frac{P-1}{P}$정도가 되며, 데이타의 송신과 수신이 모두 이루어져야 하는 상황을 고려해야 하기 때문에 통신 대역폭은 양방향(full-duplex) 통신시의 대역폭을 사용하였다. 각 부분에서 소요되는 시간을 대역폭 및 단위시간당 계산 속도를 이용하여 표시하면 다음과 같다.

$$\begin{aligned} T_{total} = & S_n\left(\frac{a}{B_r}+\frac{b(P-1)}{PB_c}+\frac{1}{2B_s}+\frac{c}{B_w}\right) \\ & +S_n\left(\frac{d}{B_r}+\frac{1}{B_s}+\frac{e}{B_w}\right) \\ = & S_n\left(\frac{a+d}{B_r}+\frac{b(P-1)}{PB_c}+\frac{3}{2B_s}+\frac{c+e}{B_w}\right) \end{aligned} \quad (2)$$

다른 작업과 중첩이 가능한 작업들에 대해서는 중첩 정도를 나타내는 인자로서 a에서 e까지를 사용하였는데, 이들은 각 작업의 중첩 정도에 따라 0부터 1사이의 값을 가지게 된다. worst case는 이 값들이 모두 1이 되는 경우라고 할 수 있고, best case는 각 작업 간에 중첩이 완벽히 이루어진 경우라 할 수 있는데 이 경우 중첩의 정도를 나타내는 인자들이 모두 0이기보다는 각 단계에서 동시에 진행되는 작업들 중 가장 느린 것들로 수행시간이 결정된다.

One-pass의 경우에는 two-pass의 경우에 비해 첫 번째 단계에서의 디스크 쓰기 작업과 두 번째 단계에서의 디스크 읽기 작업이 없어지게 되고, 두 번째 단계에서의 디스크 쓰기 작업은 다른 작업과 중첩되지 않으므로 이를 반영하여 구하면 된다.

4. 실험 및 성능평가

4.1 실험환경 및 시스템의 기본성능

실험은 MPI를 사용하여 C언어로 작성된 프로그램을 펜티엄III의 PC 클러스터 상에서 행하였다. 사용된 클러스터의 총 노드 수는 16대이고 각 노드에는 인텔 펜티엄 III 700MHz 프로세서, 512M의 RAM과 최대 전송률 80MB/sec의 9GB SCSI 하드디스크가 장착되어있고, Intel사의 100Mbps PCI Fast-Ethernet 카드가 사용되었다. MPI 라이브러리는 MPICH의 1.2.2와 LAM/MPI의 6.5.6을 이용해서 실험을 진행하였으나 실험의 진행이 빠르고 수월하였던 LAM/MPI의 결과만을 수록하였다. O/S로서는 Linux로서 Redhat 7.1(kernel version 2.4.2-2)을 사용하였다.

클러스터 시스템의 기본적인 성능에 대한 측정에서 계산 성능의 경우 초당 6.7~6.9MB 정도의 데이타에 대한 정렬이 가능함을 알 수 있었다. 통신 성능은 점대점(point-to-point) 통신의 경우, 전송 데이타의 양이 32KB가 넘어가면 단방향의 경우 약 9MB/sec, 양방향의 경우 약 8MB/sec 정도의 일정한 유효 대역폭을 보여 줌으로서 양방향 통신을 행하여도 단방향 통신에 비해 오버헤드는 그리 크지 않음을 알 수 있었다. 디스크 입출력 성능은 읽기는 약 27MB/sec, 쓰기는 약 25MB/sec 정도의 대역폭을 보여주었다.

본 실험에서 사용된 입력 데이타는 4바이트의 정수로 하고 균일 분포(uniform distribution)를 가지도록 하였다. 정해진 규모의 데이타를 디스크에 입출력하는 경우, 디스크 버퍼 용량을 증가시키면 디스크 입출력 접근회수가 감소하고 평균 전송량도 증가하나 과다하게 늘이면 유효 전송량은 다시 감소되기 때문에 실험적으로 최적의 버퍼용량을 찾아 사용하였다.

4.2 One-Pass 정렬 실험

실험은 노드 당 데이타의 크기가 16MB인 경우부터 256MB가 될 때까지 2, 4, 8, 16대의 노드를 사용해서 수행시간을 측정하였다.

그림 2, 3은 동일한 규모의 자료에 대한 정렬을 노드 수를 증가시키면서 두가지 알고리즘으로 실험한 결과로서, 총 256MB 크기의 데이타(키의 수는 64M개)를 정렬하는데 있어서 개선된 알고리즘이 기존의 알고리즘보다 모든 경우에 있어서 대략 20~30%정도의 시간 단축을 보이고 있고, 그 정도는 노드 수가 증가함에 따라 커지는 경향을 보여주고 있다. 이러한 성능 향상은 주로 통신 방식의 개선으로 인해서 얻어진 것으로 추정된다. 그림 3은 동일한 실험에서의 speedup을 log단위로 나타낸 것으로서 최대치는 16노드에서 기존 알고리즘은 약 10.2, 개선된 알고리즘은 약 10.9 정도가 된다.

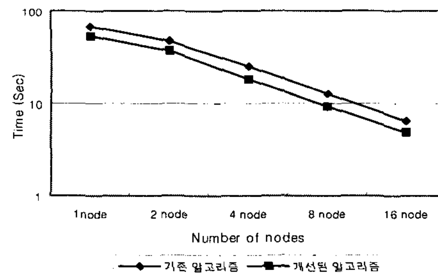

그림 2 노드 수 증가에 따른 실행시간 변화 (One-pass 방식, 256MB의 전체 데이타 크기)

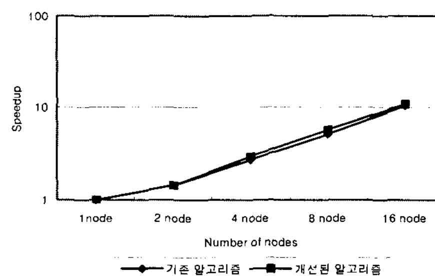

그림 3 노드 수 증가에 따른 Speedup 변화 (One-pass 방식, 256MB의 전체 데이타 크기)

그림 4에는 대표적인 실험 결과를 종합해 볼 때 실험 결과들을 표시하였는데, 새로운 알고리즘은 모든 경우에 있어서 기존 알고리즘보다 나은 성능을 보여주었고, 그 성능의 격차는 노드 수가 증가할수록, 데이타 크기가 커질수록 증가하는 양상을 보여 최대 약 40%정도의 실행 시간 단축을 이룩하였다. 또한 그림 4에서 보면 기존의 알고리즘은 노드 수를 2대에서 16대까지 증가시킴에 따라 그 실행시간이 최대 약 36%정도까지 증가하는데 비해, 새로운 알고리즘은 최대 약 6%정도가 증가에 그치는 것을 볼 때 노드 수를 증가시키면 통신의 오버헤드/지연 시간이 대폭 감소하므로 확장성도 상당히 향상된 것을 알 수 있다. 또한 그림5에서는 one-pass방식에서의 개선된 알고리즘의 예상 실행시간(best case, worst case)이 표시되어 있는데, best case 예상치의 115~130% 정도로 나타나 최대 성능에 근접한 결과를 보였다.

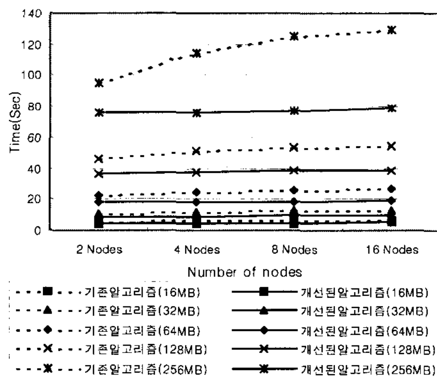

그림 4 데이타 크기 및 노드 수 증가에 따른 실행시간의 변화 (One-pass)

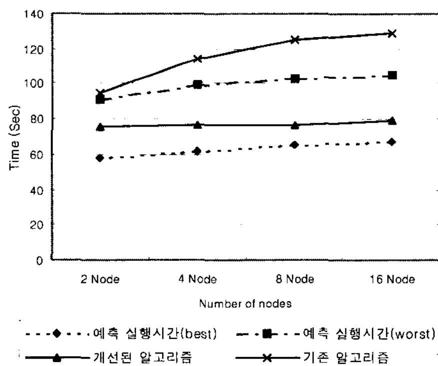

그림 5 데이타 크기 및 노드 수 증가에 따른 실행시간의 변화 (One-pass, 노드 당 256MB)

4.3 Two-Pass 정렬실험

Two-Pass방식에 대한 실험은 노드 당 데이타 크기가 256MB부터 2GB가 될 때까지 각각 2, 4, 8, 16대의 노드를 사용해서 수행시간을 측정하였다[3].

먼저 동일한 규모의 데이타를 정렬할 때 노드 수에 따른 성능실험의 결과를 그림 6, 7에 보였다. 그림 6를 보면 총 2GB 크기의 데이타(키의 수는 512M 개)를 정렬하는데 개선된 알고리즘이 기존의 알고리즘보다 25~35%정도의 시간 단축을 보여 주고 있다. 그림 7은 speedup을 나타내고 있는데 two-pass 방식의 경우에는 16노드일 경우 기존 알고리즘은 13.27, 개선된 알고리즘은 14.74 정도의 speedup을 보여 one-pass 방식보다 성능이 더 증가하였고, 노드 수 증가에 따른 개선 효과가 더 커짐으로써 one-pass 방식보다 확장성이 더 큰 것을 알 수 있다.

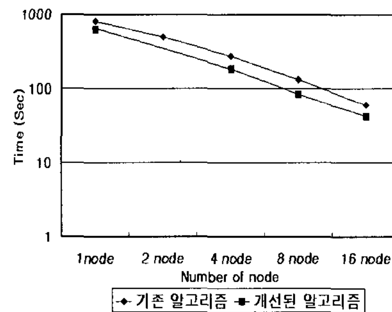

그림 6 노드 수 증가에 따른 실행시간 변화 (Two-pass 방식, 2GB의 전체 데이타 크기)

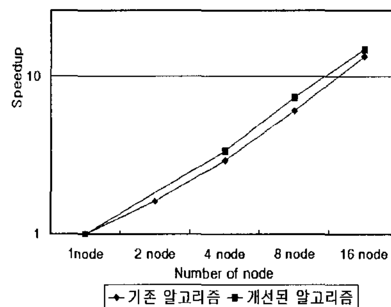

그림 7 노드 수 증가에 따른 Speedup 변화 (Two-pass 방식, 2GB의 전체 데이타 크기)

3) LAM/MPI를 사용할 경우에는 기존 알고리즘의 경우 노드 당 데이타의 크기가 2GB인 경우의 대부분, 개선된 알고리즘의 경우에는 노드 수가 2대일 때 LAM/MPI 자체의 문제로 인해서 결과를 얻을 수 없었다.

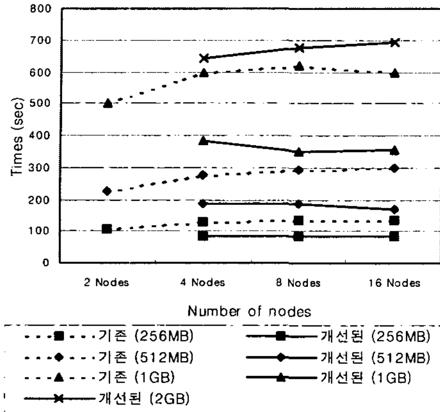

그림 8 데이타 크기 및 노드 수 증가에 따른 실행시간의 변화(Two-pass)

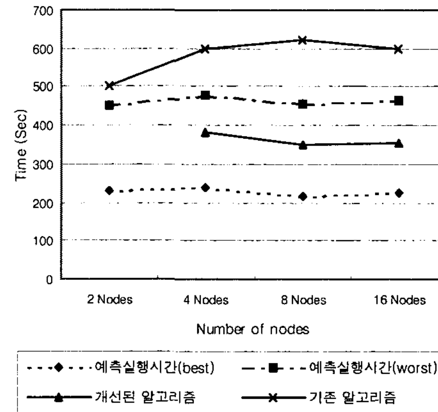

그림 9 데이타 크기 및 노드 수 증가에 따른 실행시간의 변화 (Two-pass, 노드 당 1GB)

그림 8에서는 노드 당 보유 데이타 크기를 고정시키고 노드 수를 증가시켰을 때 실행 시간의 변화를 데이타 규모에 따라 나타내었는데, 모든 경우에 있어서 새로운 알고리즘은 기존 알고리즘보다 나은 성능을 보여 주었고, 그 성능의 격차는 노드 수가 증가할수록, 데이타 크기가 커질수록 증가하는 것을 볼 수 있다. 전반적으로, two-pass 방식에서는 디스크 입출력방식에 대한 개선의 효과가 one-pass방식에 비해 더 크게 나타나서 새 알고리즘이 실행 시간을 최대 약 45% 단축시켰다. 그림 8에서 일부 경우에 노드 수가 증가해도 실행시간은 오히려 감소하는 현상이 나타난다. 이는 제1단계(phase1)에서 key를 노드 및 sub-bucket 별로 분류하는 과정에서 처리된 상위 비트의 수가 8을 넘어서게 되어, 제2단계에서 1바이트 단위로 총 4회를 수행하던 정렬 횟수가 3회로 줄어들어 그 소요시간도 3/4정도로 감소하였기 때문이다[4].

그림 9에는 two-pass방식에 대한 개선된 알고리즘의 실행시간과 3.4절에서 구한 예상 실행시간(best case, worst case)이 같이 표시되어 있는데, 완벽한 중첩을 가정하였던 best case의 130~160% 정도로 나타나 one-pass의 경우에 비해서는 저조한 결과를 보였다. 그 이유는 one-pass방식에 비해서 그 비중이 늘어난 디스크 입출력 작업에 대한 중첩 효과면에서 볼 때, 제2단계에서의 디스크 작업의 중첩은 거의 완벽하게 이루어지는 반면 제1단계에서의 디스크 입출력 작업의 중첩 비율은 매우 낮았기 때문이다. 제 1단계에서 중첩이 줄어든 이유는 계산, 통신 및 디스크 입출력 작업이 동시에 100% 수행되기에는 시스템 자원이 충분하지 못했기 때문으로 추정된다.

4.4 다른 연구결과와의 비교

개선된 알고리즘의 성능을 객관적으로 평가하기 위해서 Datamation[18]이나 Minute-sort[19]와 같은 benchmark에서 사용되는 입력 데이타와 동일한 형태의 입력 데이타에 대해서 정렬을 수행하여 현재 각 benchmark에서 가장 좋은 성능을 보이고 있는 결과와 비교하여 표 1에 정리하였다.

사용된 시스템의 사양을 비롯한 많은 부분이 서로 달라 공정한 판단이나 비교가 어렵긴 하지만, 하나의 노드에 대한 단위시간당 평균 정렬능력을 비교해 보면,

본 논문의 알고리즘: $(3.355 \times 1000)/(16 \times 40.06)$

$= 5.235[\text{ MB/ sec /node}]$

Datamation: $100/(32 \times 0.48)$

$= 6.510[\text{ MB/ sec /node}]$

Minutesort: $(21.8 \times 1000)/(64 \times 56.51)$

$= 6.028[\text{ MB/ sec /node}]$

로서 본 논문의 실험결과가 다른 연구결과에 비해서 다소 낮은 성능을 보이는 것을 알 수 있다. 이러한 결과의 주 원인은 하드웨어적인 성능 차이로서 추정된다. 초당 160MB/sec의 전송률을 가진 Myrinet을 노드 간 연결을 위한 네트워크로 사용한 Minutesort에서의 실험 시스템이나 Gigabit Ethernet을 사용한 Datamation의 경우에 비해서 초당 100Mbps의 Fast Ethernet을 네트워크로 사용한 PC 클러스터 시스템은 네트워크 성능에서

4) 노드 당 1GB의 데이타를 가진 경우를 예로 들면, 이전 알고리즘의 경우는 sub-bucket의 크기가 64MB정도가 되도록 구성되어 16 노드일 때 제1단계에서 처리된 비트의 양이 1바이트가 되는 반면, 개선된 알고리즘에서는 sub-bucket의 크기가 32MB정도가 되도록 하여 8노드일 때 처리된 비트의 양이 1바이트가 되었다.

표 1 본 논문의 연구결과와 다른 연구결과간의 성능비교

	본 논문의 알고리즘	Datamation	Minutesort(HPVM+NOW-Sort)
노드 수	16	32	64
데이타 크기	3.355GB(33.55 million records)	100MB(1 million records)	21.8GB(218 million records)
시간	40.06 초	0.48 초	56.51 초
방식	One-pass	One-pass	One-pass
H/W 사양	Intel Pentium3 700MHz × 1 512MB RAM IBM SCSI 9GB HDD × 1 (Interface B/W:80MB/sec) FastEthernet (100Mbps)	Intel Pentium3 550MHz × 2 896 MB RAM IBM SCSI 9GB HDD × 5 (Interface B/W:80MB/sec) GigabitEthernet (1Gbps)	**I/O node(32 nodes):** Intel Pentium3 300MHz × 2 384 MB RAM Maxtor IDE 20GB HDD × 4 (Used as RAID 0) IBM IDE 22GB HDD × 1 Myrinet (160 MB/sec) **Sorting node(32 nodes):** Intel Pentium3 400MHz × 2 1GB RAM Seagate SCSI 18GB HDD × 2 Myrinet (160 MB/sec)

약 10배 이상의 차이가 존재하고, CPU나 메모리에 있어서도 낮은 성능의 하드웨어를 쓰고 있어 열등한 결과를 가져온 것이다. 또 계산 부분에 있어서도 좀더 최적화가 필요한 것으로 생각되는데, minute sort에 사용된 NOW-Sort에서는 하나의 노드가 약 700MB정도의 데이타를 약 5초 내에 정렬하는데 비해서 본 알고리즘에서는 200MB정도의 데이타를 정렬하는데 약 12초 정도가 소요되어 CPU 속도차를 감안한다 해도 매우 큰 성능 상의 격차가 존재하기 때문이다.

전반적으로 볼 때 네트워크의 성능차이로 노드 간 통신에 걸리는 시간이 약 10배정도 증가하게 되고, 내부정렬 알고리즘의 최적화가 이루어지지 못한 것을 고려한다면, 비교 대상의 80~85%정도의 성능을 달성한 것은 본 논문에서 목표한 통신 및 디스크 입출력의 최적화가 성과를 거두었기 때문으로 판단된다.

5. 결 론

본 논문에서는 연산, 통신 및 디스크 입출력성능을 포함한 종합적인 컴퓨터의 성능측정의 지표 역할을 하는 외부정렬(NOW-Sort)을 고속화하고자 통신과 디스크 접근을 알고리즘적으로 최적화하는 방법을 제시하였다. 특히 PC 클러스터에서의 실행에 초점을 맞추어 네트워크와 관련된 부분의 성능을 향상시키는데 중점을 두었고, 디스크 입출력 방식의 개선에도 많은 노력을 기울였다. 실험을 통해서 알고리즘의 개선효과를 측정한 결과 one-pass 방식의 경우에는 최대 40%, two-pass방식에서는 최대 45%정도의 성능 향상이 있음을 확인하였다. 또한 새로운 알고리즘이 계산, 통신, 디스크 입출력이 모두 실행되는 제1단계에서는 통신과정의 개선효과는 뛰어나지만 디스크 입출력에서는 큰 변화가 없고, 계산과 디스크 입출력만이 존재하는 제2단계에서는 디스크 입출력 성능을 크게 개선시켰음을 확인하였다.

향후 연구로는 제1단계에서 디스크 입출력의 중첩이 제한되는 원인을 밝혀 성능 향상의 한계를 정확히 파악하고 최대한의 개선효과를 얻도록 하며, 주메모리 상에서의 내부정렬에서 소요되는 시간을 보다 단축하여 총 실행시간을 더욱 줄이도록 해야 할 것이다.

참 고 문 헌

[1] B. Ahn and D. Kim, "External sort on a cluster of PCs." *2000 Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, pp.1443-1448, Las Vegas, Nevada, USA, June 25-29, 2000.

[2] W. A. Martin, Sorting, ACM Computing Surveys, Vol. 3, No. 4, pp. 147-174, 1971.

[3] http://research.microsoft.com/barc/SortBenchmark, *Sort Benchmark Home Page*

[4] Y.C. Kim, M. Jeon, D. Kim, A. Sohn, "Communication-efficient bitonic sort on a distributed memory parallel computer." *Proc. Int'l Conference on Parallel and Distributed Systems* (ICPADS'2001), pp.165-170, Kyung-Ju, Korea, June 26-29, 2001.

[5] S-J Lee, M. Jeon, A. Sohn and D. Kim, "Partitioned Parallel Radix Sort," *Journal of Parallel and Distributed Computing*, Vol. 62, pp. 656-668, Academic Press, April 2002.

[6] A. Sohn, Y. Kodama. "Load balanced parallel radix sort." *Proc. the 1998 international conference on*

Supercomputing, pp 305 - 312, 1998.
[7] K.E. Batcher, "Sorting networks and their applications." *Proc. AFIPS Conference*, pp. 307-314, 1968.
[8] T.E. Anderson, D.E. Culler, and D.A. Patterson, "A Case for NOW(Networks of Workstations)." *IEEE Micro*, Feb. 1994.
[9] A.C. Arpaci-Desseau, R.H. Arpaci-Desseau, D.E. Culler, J.M. Hellerstein, and D.A. Patterson, "High-Performance Sorting on Networks of Workstations." *ACM SIGMOD '97*, Tucson, Arizona, May 1997.
[10] J. Wyllie, "SPsort: How to sort a terabyte quickly." *Technical Report,* IBM Almaden Lab., Feb. 1999, http://www.almaden.ibm.com/cs/gpfs-spsort.html
[11] L. Rivera, X. Zhang, A. Chien, "HPVM Minute-sort." *Sort Benchmark Home Page,* http://research.microsoft.com/barc/SortBenchmark/
[12] D. Taniar and J.W. Rahayu, "Sorting in parallel database systems." *Proc. High Performance Computing in the Asia Pacific Region, 2000: The Fourth Int'l Conf. and Exibition* Vol.2, pp. 830-835, 2000.
[13] L.M. Wegner, J.I. Teuhola, "The external heapsort" *IEEE Trans. Software Engineering,* Vol.15, No. 7, pp. 917-925, July 1989.
[14] C. Cerin, "An out-of-core sorting algorithm for clusters with processors at different speed." *Proc. 2002 Parallel and Distributed Processing Symp.,* April 15-18, Fort Lauderdale, FL, USA.
[15] A.C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein and David A. Patterson. "for the sorting record: experiences in tuning NOW-Sort." *Proc. the SIGMETRICS symposium on Parallel and distributed tools,* pp 124 - 133, 1998.
[16] F. Popovici, J. Bent, B. Forney, A.A. Dusseau, R.A. Dusseau, "Datamation 2001: A Sorting Odyssey." *Sort Benchmark Home Page,* http://research.microsoft.com/barc/SortBenchmark/
[17] 김지형, 통신과 디스크 입출력 최적화를 통한 병렬 외부정렬의 성능 향상, *석사학위논문, 고려대학교,* Jan. 2002.
[18] Anon et al., "A Measure of Transaction Processing Power." *Datamation*, V.31(7):112-118. also in *Readings in Database Systems*, M.J. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.
[19] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, D. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort." *ACM SIGMOD Record, Proceedings of the 1994 ACM SIGMOD international conference on Management of data, Volume 23 Issue 2,* 1994
[20] *LAM/MPI Parallel Computing,* http://www.lam-mpi.org
[21] *The Beowulf Project,* http://www.beowulf.org

김 지 형

1999년 고려대학교 전기공학과 학사. 1999년 ~ 2002년 고려대학교 메카트로닉스 협동과정 석사. 2000년 ~ 현재 펌킨넷코리아(주) 연구원. 관심분야는 클러스터 컴퓨팅, 인터넷, 네트워크, 운영체제

김 동 승

정보과학회논문지 : 시스템 및 이론
제 29 권 제 1 호