Pthread 라이브러리를 이용한 Linked List 병렬화 클래스 라이브러리의 설계 및 구현

김홍숙, 한동수
한국정보통신대학원

Design and Implementation of Parallelized Linked List Class Library using Pthread Library

Hong-Soog Kim, Dong-Soo Han
Information & Communications University

요 약

병렬 프로세서 시스템이 제공하는 하드웨어적인 장점을 이용하기 위해서는 병렬 프로그래밍을 통한 애플리케이션의 병렬화가 필요하다. 기존의 순차적 코드의 경우에 자동 병렬화 컴파일러 기법을 통하여 병렬 프로세서시스템이 제공하는 성능을 극대화하고 있다. 그러나 자동 병렬화는 과학 기술 계산용 코드와 같은 정형성을 지닌 코드에서는 유용하지만 비즈니스 응용에서 사용되는 동적인 자료구조를 사용하는 코드에서는 포인터에 의한 별명과 이에 따른 의존성 분석에 어려움으로 인해 많이 응용되고 있지는 못하다. 본 논문은 병렬 프로세서 시스템이 제공하는 기능을 이용하기 위한 한 방법으로 비즈니스 응용에서 많이 사용되는 동적인 자료 구조 중 linked list 클래스 라이브러리의 설계와 구현에 대하여 기술한다.

1. 서론

다수의 프로세서를 장착한 병렬 프로세서 시스템은 단일 프로세서가 가지는 성능상의 한계를 극복하기 위한 방안으로 제시되었다. 병렬 프로세서 시스템은 메모리 구조에 따라 Shared Memory 병렬 프로세서 시스템, Distributed Memory 병렬 프로세서 시스템과 Distributed shared memory 병렬 시스템 등이 있다[1].

병렬 프로세서 시스템에서 추구하는 공통적인 목적은 다수의 프로세서를 사용하여 단일 프로세서 시스템보다 나은 성능을 내는 것이다. 주어진 문제를 병렬 프로세서 시스템을 사용하여 속도 향상을 하기 위해서는 독립적으로 실행할 수 있는 병렬화 부분을 찾아서 각각의 병렬 작업을 프로세서에 분산함으로서 속도를 향상시킨다. 자동 병렬화 컴파일러는 데이터 흐름 분석, 제어 흐름 분석 등의 컴파일러 기법을 통하여 기존의 순차적 프로그램에서 병렬화 가능한 부분을 찾아내어 병렬화 가능 코드 부분에 병렬 처리를 위한 코드를 추가함으로서 병렬 프로세서가 제공하는 하드웨어 성능의 활용을 극대화하려는 노력의 일환이다[2]. 자동 병렬화 컴파일러는 주로 과학기술 계산용 코드의 병렬화에서 많은 성과를 거두고 있으나, 비즈니스용 애플리케이션의 자동 병렬화에는 여러가지 어려움을 가진다. 비즈니스 애플리케이션은 많은 부분 레코드와 같은 동적 자료 구조를 처리하여야 하는데, 이때 사용되는 포인터에 의한 별명(alias)로 인해 과학기술 계산용 코드에서 적용되었던 데이터 흐름 분석, 제어 흐름분석이 그대로 적용하기 어려우므로, 새로운 해석 기법에 대한 연구가 요구된다. 또한 자동 병렬화 컴파일러를 통한 병렬 프로세서 시스템의 이용은 이미 작성된 코드의 병렬화를 주목적으로 하고 있으며, 아직까지는 그 적용 범위가 과학 기술용 코드에 제한되고 있다. 그러므로 병렬 프로세서를 위한 비즈니스 애플리케이션을 작성하기 위해서는 프로그래머가 직접 병렬화 가능 부분을 MPI, PVM또는Pthread와 같은 라이브러리를 이용하여 작성하여야 한다. 그러나 이러한 병렬 프로그래밍은 기존의 순차적 프로그래밍에 비하여 복잡하고 오류가 발생하기 쉬우며 오류 발생시에 디버깅이 용이하지 않다는 한계를 가지고 있다.

본 논문에서는 이에 대한 대안으로 비즈니스 애플리케이션에서 많이 사용되는 linked list의 병렬처리를 위한 라이브러리를 제안한다. Linked list 병렬화 라이브러리는 기본적으로 단일 주소 공간을 지원하는 shared memory또는 distributed shared memory병렬 프로세서 시스템을 목표로 하고 있다. 용이한 사용 및 재 사용성을 위하여 linked list의 기본 연산인 Insert, Append, Delete, Search에 대한 기본 연산을 C++ 클래스화하여 제공한다. 또한 병렬 프로세서 시스템에서 제공하는 프로세서를 최대한 이용하기 위하여 Pthread를 이용한 병렬 연산을 구현한다[3,4]. Pthread를 사용한 병렬 연산처리 부분을 private method로 하여 사용자에게 감추고 대신에 private parallel method를 위한 interface method를 public으로 하여 공개함으로서 사용자가 내부적인 병렬화 메커니즘에 대한 지식이 없이도 해당 interface method만을 사용하여도 내부적으로 병렬화 처리가 가능하도록 하였다.

2. Linked List Operations

Linked list에 대한 기본적인 연산은 크게 리스트에 새로운 레코드를 추가하는 Insert, 리스트에서 특정 레코드를 삭제하는 Delete와 주어진 조건을 만족하는 특정 레코드를 찾는 Search연산이 있다. Linked list를 배열을 이용하여 구현하는 경우에, 특정 위치에 있는 레코드의 Search연산은 O(1)에 가능하나, Insert, Delete연산및 특정 조건을 만족하는 레코드를 탐색하는 연산은 O(n)의 시간이 소요된다. 반면, 레코드 리스트를 linked list 형태로 구현하는 경우에는 포인터를 위한 추가적인 메모리가 소요되는 단점을 가지기는 하지만 Insert, Delete연산이 O(1)시간에 가능하다, 그러나 특정 위치의 레코드를 찾는 Search연산, 특정 조건을 만족하는 레코드를 찾는 Search연산에는 O(n)의 시간이 소요된다. 본 논문에서 제안하는 Pthread를 이용한 linked list 라이브러리는 Search연산을 속도를 향상시키기 위하여 linked list를 p개의 균등한 분할로 나누고 각각의 분할을 p개의 쓰레드를 통하여 병렬 탐색하는 기법을 사용한다. 병렬 탐색을 위한 균등한 분할을 임의의 시간에서 유지하기 위해서는 Insert, Delete연산에 따라 linked list의 길이에 변화가 있을 때 적절하게 재분할을 계산한다. 분할의 유지에 따른 추가적인 자료구조는 p+1개의 포인터와 p+1개의 원소를 가지는 정수 배열과 추가적인 변수를 사용한다.

2.1 병렬 탐색을 위한 자료구조

[1] 본 연구는 한국 학술 진흥 재단 신진 교수 과제사업 (과제번호: 1998-003-E00447)에 의해 지원 받았음

병렬 탐색이 가능하기 위해서는 Insert, Delete연산시에 linked list의 길이에 변화가 발생하여 균등한 분할을 깨지는 경우에 재분할을 통하여 균등한 분할을 재구성하여야 한다. Linked list 클래스에서는 병렬 탐색을 위한 pthread의 생성 갯수를 concurrenc_level이라는 정수 값으로 유지하고, 이 숫자만큼의 균등 분할을 유지한다. 이를 위하여 각 분할의 시작 및 끝 위치를 저장하는 concurrency_level+1개의 분할 포인터 배열과 각 분할의 시작위치가 전체 linked list에서 몇 번째 레코드인지에 대한 정보를 유지하는 concurrency_level+1개의 위치 배열을 유지한다. 위치 배열은 특정 위치에 대한 레코드 삽입 삭제 시 전체 linked list를 탐색하지 않고 삽입 또는 삭제될 레코드가 위치하는 분할을 결정하는데 사용될 뿐 아니라 해당 분할의 크기를 계산하는데 사용한다.

[그림1]은 concurrency_level이 8인 경우이고 linked list의 레코드 수가 63개 일 때의 분할 포인터 배열의 포인터 정보를 도식화한 것이다.

2.1삽입, 삭제연산에 따른 재분할 정책

레코드 삽입 연산의 경우 [그림2]해당 분할이후의 분할 포인터를 후방으로 전진하여 마지막 분할을 제외한 모든 분할이 균등한 크기를 가지도록 유지한다. 그러나 마지막 분할의 크기가 다른 분할의 크기에 비해 커진다면 전체적인 병렬 검색의 속도는 마지막 분할의 크기에 의존하게 되므로, 적절한 재분할 정책이 필요하다. 본 연구에서는 7가지 재분할 정책을 사용하였다. 구현된 재분할 정책은 크게 미시적 재분할과 거시적 재분할로 구분된다. 미시적 재분할은 재분할의 기준을 concurrency_level을 기준으로 하여 마지막 분할의 초과 크기가 concurrenc_level이상이 되었을 때 재분할을 한다. 따라서 거의 모든 시점에서 균일한 크기의 분할이 유지된다는 장점을 가지지만 빈번한 재분할을 발생시킨다는 단점을 가진다. 반면 거시적 재분할법은 현재 분할의 크기를 기준으로 하여 마지막 분할이 균등 분할의 크기의 1/4, 2/4, 3/4또는 1배 이상의 초과 크기를 가지는 경우에 분할을 한다. 미시적 분할의 경우에는 linked list의 크기에 관계없이 concurrency_level에 따라 재분할을 하므로 분할의 크기가 1씩 증가하는 반면에 거시적 재분할은 현재 분할 크기에 따라 재분할하므로 분할의 크기의 증가 폭이 현재 linked list의 크기에 비례하여 증가하므로 재분할의 횟수는 적어지지만 마지막 분할이 비균등해지는 상황이 좀 더 빈번하게 발생한다.

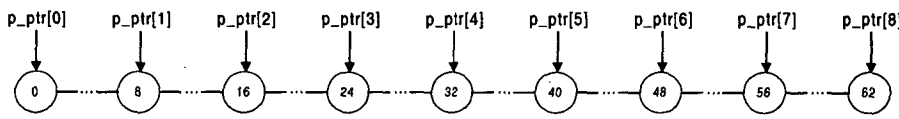
삭제 연산의 경우 linked list의 크기를 1만큼 줄이는 연산이다. 삭제의 경우에는 [그림3]와 같이 삭제되는 레코드가 속한 분할 이후의 분할 포인터를 전방 전진시킴으로서 마지막 분할을 제외한 다른 분할의 크기를 균등하게 유지한다.삭제 연산의 경우에는 다음과 같은 몇 가지 경우에는 특별히 처리하여야 한다.

균등 분할의 크기가 0인 경우는 linked list의 크기가 주어진 concurrency_level보다 적은 경우로서 이 경우에는 0th record delete, last record delete, other case로 구분이 된다. Concurrency_level은 8이고 균등 분할의 크기가 0인 경우 첫번째 레코드(0th record)삭제 시에는 7개의 분할 포인터를 이동하여야 한다. 마지막 레코드의 삭제의 경우에는 마지막 분할의 끝을 나타내는 p_ptr[8]을 후방향 전진시키고, 그 외의 경우에는 삭제되는 레코드 전후의 레코드의 포인터만을 변경하는 것으로 분할 정보는 유지된다. 균등분할의 크기가 1이고 마지막 분할의 크기가 1인 경우에 레코드 삭제는 균등 분할의 크기 가 0으로의 재분할이 요구된다.어느 경우의 레코드 삭제도 공통적으로 마지막 분할의 끝을 나타내는 포인터를 제외하고는 첫번째 레코드를 가리키도록 변경하고, 균등분할의 크기를 0로 변경함으로써 재분할을 하게 된다.

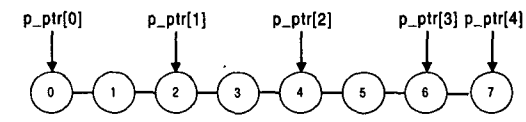
균등 분할의 크기가 2이상이고 마지막 분할의 크기가 1인 경우에는 마지막 분할의 크기가 1이므로 레코드 삭제는 기본적으로 분할 크기가 1만큼 줄어드는 재분할을 요구한다. 재분할은 삭제하고자 하는 레코드가 속한 분할 이전의 분할 포인터와 이후의 분할 포인터의 후방향 전진량에 있어서 차이가 있다. 이전 분할 i의 포인터는 i만큼 후방향 전진를 하고 이후 분할 k의 포인터는 k-1만큼의 후방향 전진를 한다. 이때 삭제하고자 하는 레코드 전후의 포인터는 마지막의 그림과 같이 서로 변경된 상태이다.

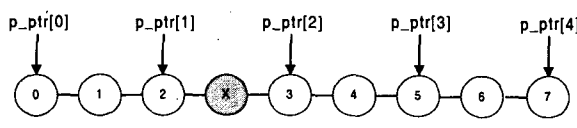
2.2 병렬탐색연산

병렬 탐색연산은 분할의 수와 탐색 시에 동일한 개수의 pthread를 생성하여 부분 탐색을 병렬적으로 수행한다. 생성된 thread는 각자의 분할 영역을 탐색하고 주어진 키를 찾은 경우에는 자신을 제외한 다른 threads를 cancel하고 탐색된 키를 포함하는 레코드에 대한 포인터를 리턴한다. pthread의 경우 C 인터페이스만을 제공하므로 C++의 메소드를 thread화하기 위해서는 friend로 선언된 dummy C Linkage인터페이스를 정의하고 그 안에서 다시 thread화하고자 하는 메소드를 호출하는 방법으로 C++클래스 메소드를 병렬화하여야 한다 [5,6,7].

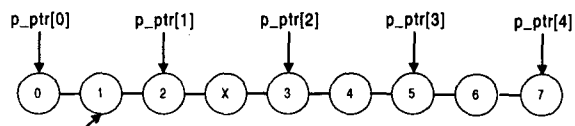


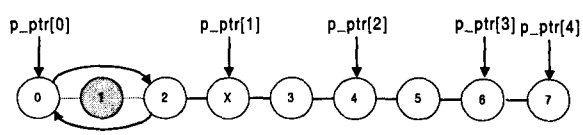
[그림1] 균등 분할을 위한 자료구조





[그림2] 삽입에 따른 균등 분할 포인터의 조정





[그림3] 삭제에 따른 균등 분할 포인터의 조정

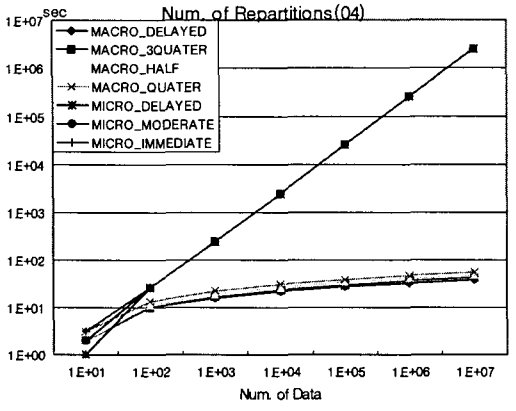
3. 실험을 통한 성능 평가

본 논문에서 제안한 균등분할 유지방안과 pthread를 통한 병렬 연산의 성능을 측정하기 위하여 다음과 같은 실험을 하였다. 실험환경은 [표 1]과 같다.

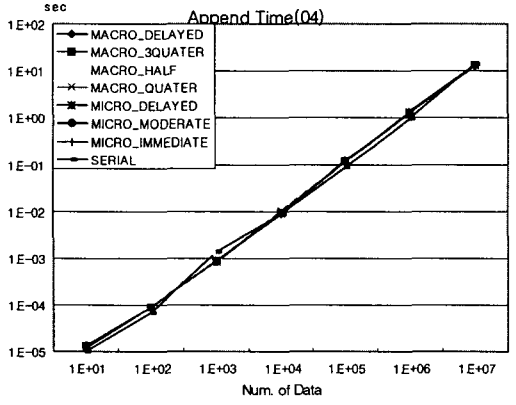
[표 1] 구현 및 실험 환경

Hardware	1Gbyte Main Memory Pentium III Xeon 500MHz 4 Way Processors
OS	SunOS 5.6 Generic i86pc
Compiler	GNU project C and C++ Compiler (v2.7)

먼저 각각의 재분할 정책에 따른 재분할 횟수와 재분할에 소요되는 시간과의 관계를 도출하기위한 실험으로 Append 연산의 수를 $10\sim10^7$번, concurrency_level을 2에서 128까지 변화시키면서 측정하였다. concurrency_level이 4인 경우의 실험 결과가 [그림 4]와 그림[5]이다. 그림에서 보는 바와 같이 빈번한 재분할을 하는 미시적인 재분할정책의 경우 삽입되는 레코드수에 비례하여 재분할 횟수가 증가하고는 있지만 재분할에 따른 Append연산 수행 시간은 거의 비슷하다. 이러한 결과를 통해 재분할에 따른 비용이 레코드를 삽입하는 연산에 비하여 큰 오버헤드가 아님을 알 수 있다.

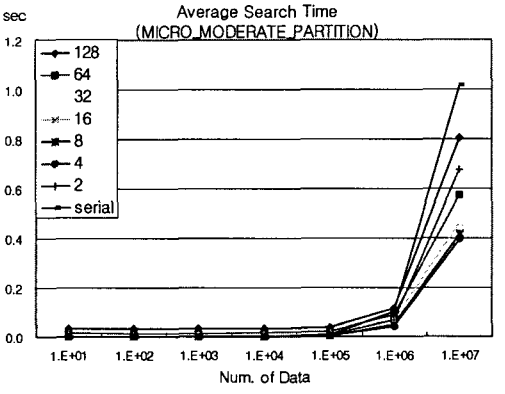


[그림 4]레코드 추가에 따른 재분할 횟수



[그림 5]레코드 추가에 따른 실행 시간

[그림 6]은 미시적 재분할 정책에 따른 concurrency_level과 탐색시간과의 관계를 보여주고 있다. concurrency_level이 4인 경우 가장 우수한 성능을 보이고 있다. 이 경우에는 기존의 순차적인 탐색에 비하여 2.6배 정도의 수행속도 향상이 있었다. 이는 시스템내의 프로세서의 수와 일치한다. 시스템내의 프로세서의 수의 2-3배까지의 concurrency_level내에서도 속도 향상이 이루어지지만 그 이상의 concurrency_level에서는 속도 향상이 미진함을 볼 수 있었다. 이는 thread생성에 따른 오버헤드가 분할된 리스트를 병렬적으로 수행하는데 따른 속도상의 잇점을 상쇄시키고 있는 것으로 분석이 된다. 그러므로 thread생성에 따른 오버헤드를 줄이기 위하여 SPMD 방식과 유사한 방식으로 초기에 thread를 한번만 생성하고 계속 유지하는 방안을 통하여 성능을 더욱 향상시킬 수 있을 것으로 기대된다.



[그림 6]병렬성 정도에 따른 평균 search시간

4. 결론 및 토의

본 논문에서는 병렬 프로세서 시스템이 제공하는 하드웨어적인 성능의 이용을 극대화하기 위한 방법으로 병렬화 linked list의 연산을 병렬화한 클래스 라이브러리를 설계하고 구현하였다. 실험결과 균등분할 정책의 오버헤드는 Insert,Append와 Delete 연산의 비용에 비해 작았다. 미시적인 균등분할정책이 빈번한 재분할에도 불구하고 거시적인 균등분할과 수행 속도면에서는 큰 차이가 없으므로 미시적인 균등분할정책이 Search연산에 따른 속도향상을 고려한다면 더 유리한 것으로 분석이 된다.

또한 병렬 탐색 시에 thread를 매번 생성하는 오버헤드로 인해 concurrency_level이 시스템상의 프로세서의 수를 초과시에는 성능향상의 정도가 떨어지는 것으로 분석이 되었다. 그러므로 병렬 탐색의 병렬성의 정도를 높이기 위해서는 thread를 초기에 생성하고 계속해서 재활용하는 방법으로 병렬 탐색을 구현하는 것이 바람직하다.

향후 연구과제로는 실험 결과의 분석에 따른 문제점을 보완하기 위하여 SPMD방식의 thread생성과 이에 따른 동기화 구현이 있다. 또한 트리와 같은 동적인 자료구조도 DB상의 인덱스등에서 빈번하게 사용이 되므로 트리 자료구조와 같은 동적자료구조에 대한 기본 연산을 thread를 사용하여 병렬화하는 연구를 수행할 예정이다.

참고문헌

[1] Jhon L. Hennessy et. al, "Computer Architecture :A Quantitative Approach", Morgan Kaufmann Publishing Inc. 1996

[2] 한동수외, 최종연구보고서 :병렬 Fortran 컴파일러 개발에 관한 연구, 한국전자통신연구원, 1998

[3] David R. Butenhof, "Programming with POSIX Threads", Addison Wesley, 1997

[4] Multithreaded Programming Guide, Sun Micro Systems,1997

[5] Personal mail in comp.programming.threads newsgroup

[6] Scott Meyers, "Effective C++", Addison Wesley, 1992

[7] Magaret A. Ellis an Bjarne Stroustrup, "The Annotated C++ Reference Manual", Addison-Wesley,1990