POSIX스레드에 의한 재귀적 알고리즘의 병렬화에서 병렬성 제어 방안

이 형 봉† · 백 청 호††

요 약

처리기를 여러 개 장착한 다중처리기 시스템의 근본 목적은 적은 비용으로 많은 성능 향상을 얻자는 데에 있다. 그러나 다중처리기 시스템을 충분히 활용하기 위해서는 병렬처리를 지원하는 특별한 언어를 사용하거나 병렬성을 탐색하는 도구의 도움을 얻어야 하는 경우가 대부분이다. 일반적으로 알고리즘에서 병렬화에 적합한 대표적인 영역으로 루프와 재귀호출 등이 알려져 있다. 이 중 재귀호출은 특별한 도구나 언어의 지원 없이 개념적인 측면에서 비교적 쉽게 병렬화 시킬 수 있다. 그러나 재귀호출이 깊어지면 통제되지 않은 병렬성이 과도하게 높아져 오히려 수행불능 상태가 되고 만다. 본 논문은 POSIX스레드를 이용하여 재귀호출로 구성된 알고리즘을 병렬화 시키는 과정에서 병렬성을 제어하는 방안을 제시한다. 이를 위하여 유닉스 시스템에서 프로세스와 스레드의 개념을 정립하고, 제안된 병렬성 제어 방안을 퀵 정렬에 적용한 결과를 실증적으로 분석하여 그 효용성을 검증한다.

A Device of Parallelism Control in POSIX Based Parallelization of Recursive Algorithms

Hyung-Bong Lee† · Chung-Ho Baek††

ABSTRACT

One of the major purposes of multiprocessor system is to get a high efficiency in performance improvement. But in most cases, it is unavoidable to use some special programming languages or tools for full use of multiprocessor system. In general, loop and recursive call statements of algorithms are considered as typical parts for parallelization. Especially, recursive call statements are easy to parallelize conceptually without support of any special languages or tools. But it is difficult to control the degree of parallelism caused by high depth of recursive call leading to execution crash. This paper proposes a device to contol parallelism in the process of POSIX thread based parallelization of recursive algorithms. For this, we define the concept of thread and process in UNIX system, and analyze the results of experimental application of the device to quick sorting algorithm.

키워드 : 프로세스(process), 스레드(thread), 재귀적 알고리즘(recursive algorithm), 분할 · 정복 알고리즘(divide and conquer algorithm), 병렬성 제어(parallelism control)

1. 서 론

최근 UMA(Uniform Memory Access) 혹은 NUMA(Non-Uniform Memory Access) 구조 등의 다중처리기 시스템[1, 2]이 보편화되어 각급 학교는 물론 대부분의 기업에 널리 보급되고 있다. 이런 형태의 다중처리기 시스템은 밀결합된 분산 시스템의 일종[2, 3]으로 응용프로그램 개발자에게는 하부의 구조를 은폐시킴으로써 종래의 단일처리기 시스템과 동일한 환경으로 투명성을 제공한다. 그러나 이러한 투명성은 구 시스템과의 호환성 측면에서는 장점이 될 수도 있으나 세밀한 성능 관점에서 보면 단점으로 나타난다. 그 이유는 처리기가 여러 개인 시스템이라 하더라도, 하나의 프로그램을 분리해서 동시에 처리할 수 없기 때문이다. 예를 들어 처리기가 4개인 대칭형 다중처리기 시스템(SMP, Symmetric MultiProcessor)에 큰 프로그램 하나가 입력된 경우에, 하나의 처리기 만이 할당되어 수행하고, 나머지 3개의 처리기는 유휴 상태를 유지한다.

위와 같은 현상은 여러 가지 원인에 의하여 발생되는데, 우선 운영체제의 스케줄링 단위에 문제가 있다. 스케줄링은 처리기가 분담하여 독립적으로 처리할 수 있는 일의 단위를

† 종신회원 : 호남대학교 정보통신공학부 교수
†† 종신회원 : 강원대학교 전자계산학과 교수
논문접수 : 2001년 3월 15일, 심사완료 : 2002년 5월 14일

말하는데, 전통적인 UNIX는 프로세스 단위로 설계되어 있다[3, 4]. 그러나 최근에는 OSF/1에서부터 출발하여 MACH 등의 마이크로 커널을 도입하여 스레드 단위의 스케줄링을 하는 UNIX가 늘어나고 있다[5, 6]. 또 다른 중요한 요인은 프로그램 자체가 여러 처리기에 의해 동시에 수행될 수 있는 병렬 프로그램으로 구성되어 있지 못하다는데 있다. 이를 해결하기 위하여 병렬 프로그래밍에 대한 연구가 오래도록 진행되어 왔으나 아직까지 보편화 되지 못하고 있는 것이 현실이다. 이는 병렬성을 세밀하게 고려한 프로그램 작성이 까다로울 뿐 아니라, 이를 지원하는 도구 또한 완전하지 못하기 때문이다.

그러나 최근 객체지향 프로그래밍, 윈도우 프로그래밍, 네트워크 프로그래밍, 화상처리 등 비교적 큰 처리단위의 병렬 프로그래밍 기법[2]이 요구되는 응용분야가 확산되면서 그에 대한 해결책으로 스레드를 사용한 프로그램의 병렬화가 실용적이고 현실적인 대안으로 제시되고 있다. 특히 분할정복(divide and conquer) 형태의 재귀적 알고리즘은 가장 좋은 병렬처리 대상 중의 하나이다. 그러나 재귀적 알고리즘의 재귀호출을 단순히 스레드로 대응시킬 경우 데이터의 상태에 따라 지나치게 많은 스레드가 생성되어 오히려 부작용이 더 커질 수 있기 때문에, 여기에는 적절한 스레드 개수를 유지할 수 있는 병렬성 제어가 요구된다. 따라서 본 논문에서는 UNIX환경을 중심으로 스레드에 대해 보다 폭 넓은 이해를 도모하고, 이를 바탕으로 POSIX스레드에 의한 재귀적 알고리즘의 병렬화에 있어서 병렬성 제어 방안을 제시한 후, 그 효용성을 퀵 정렬 알고리즘을 통해서 검증한다. 이를 위하여 2장에서 스레드에 대한 전반적인 고찰 및 스레드 관련 연구 분야를 살펴보고, 3장에서 재귀적 알고리즘의 병렬화 및 병렬성 제어방안을 모색하며, 4장에서는 그 결과를 퀵 정렬 알고리즘에 실증적으로 적용하여 그 결과를 분석한다. 마지막으로 5장의 결론으로 본 논문을 맺는다.

2. 스레드에 대한 고찰

2.1 프로세스와 스레드의 비교

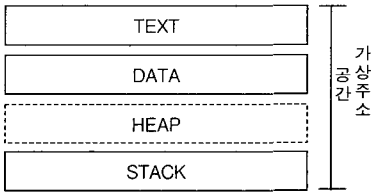
프로세스와 스레드 개념은 관심영역에 따라 응용프로그램 혹은 운영체제 수준에서 접근해 볼 수 있다. 그러나 병렬처리 관점에서는 운영체제의 일부로서 프로세스를 정의한 후, 스레드를 이해하는 것이 보다 효율적이다.

2.1.1 프로세스

운영체제 관점에서 본 프로세스는 원래 1960년대 Multics 설계시 도입된 개념[3]으로 흔히 말하는 타스크(task)의 의미를 가진다. 그것의 가장 물리적인 정의는 "운영체제 내에 PCB(Process Control Block)가 할당됨으로써 의미가 부여되는 개체"라고 말할 수 있다. 운영체제는 모든 자원 관리(할당 및 회수) 및 보안 관리를 PCB에 근간을 두고 수행한다. 즉, 운영체제가 필요로하는 모든 정보가 PCB에 저장되어 유지·관리 된다. PCB의 내용 중 본 논문과 결부되어 중요한 의미를 가지는 항목들은 다음과 같다.

- 메모리

특정 프로세스가 차지하는 메모리는 해당 프로세스의 몸체에 해당되는 부분으로 (그림 1)과 같이 크게 실행문, 데이터, 힙, 그리고 스택 부분으로 구성된다. 실행문 영역은 기계명령어 들의 집합으로 텍스트(text)라 부른다. 그 이유는 프로세스가 살아있는 동안 내용이나 크기가 변경되지 않기 때문이다. 데이터 영역은 프로그램에서 전역적 혹은 고정적(static)으로 정의된 변수들이 차지하는 곳으로 프로세스가 활동하면서 보관한 값을 항구적으로 유지한다. 힙 영역은 프로세스가 수행하면서 동적으로 할당받은 메모리 영역인데 다시 반납하지 않는 한 데이터 영역과 동일한 성격을 가진다. 스택 부분은 프로세스가 활동하면서 서브함수를 호출할 때 문맥(제어정보, 레지스터 값 등), 매개변수, 지역 변수들의 저장을 위해 동적으로 확장·축소를 거듭하고, 처리기에 독립적으로 할당된 스택포인터 혹은 프레임포인터에 의하여 관리된다. 위의 모든 메모리 영역은 가상주소공간에 위치하고, 주소변환테이블에 의해 물리메모리로 연결된다.



(그림 1) 프로세스의 메모리 이미지

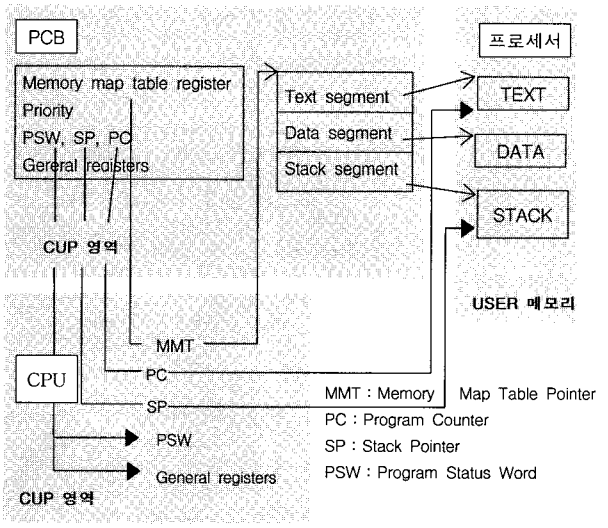
- 문맥

문맥은 프로세스의 현재 진행상태를 나타내는 일종의 스냅샷으로서 진행위치, 명령어 실행결과, 스택포인터, 기타 범용 레지스터 값 등의 집합을 말한다. 이들은 프로세스 진행도중 처리기를 잃었을 때 차후의 계속적인 진행을 위하여 문맥교환을 통하여 PCB에 저장된다

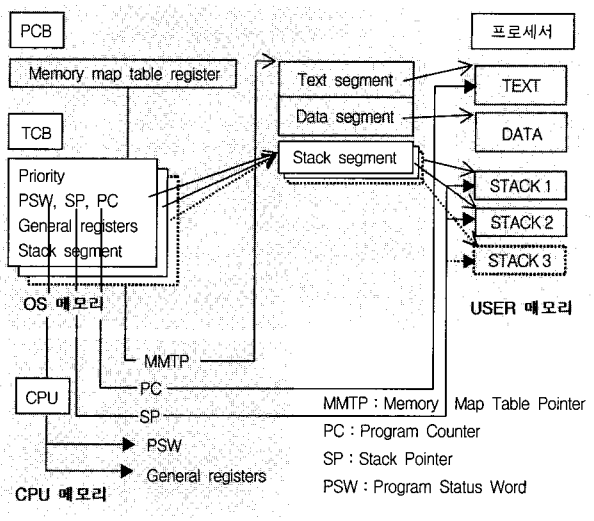
전통적인 운영체제는 기본적으로 위의 두 가지 항목을 바탕으로 처리기를 할당하기 때문에 현재 서비스를 받고 있는 프로세서는 또 다른 처리기의 서비스를 동시에 받을 수 없다. (그림 2)는 이와 같은 프로세스 기반 스케줄링 환경을 보여주고 있다.

2.1.2 스레드

스레드는 텍스트와 데이터 영역은 공유하지만 스택 부분은 별도로 할당되어 독립적으로 수행되는 단위를 일컫는다[7]. 여기서 스택영 역만을 분리하는 이유는 다음과 같다[2].



(그림 2) 프로세스 기반 스케줄링 환경



(그림 3) 스레드 기반 스케줄링 환경

우선 텍스트 영역은 실행 도중에 변경되지 않기 때문에 여러 수행 단위 즉 여러 처리기가 공유하더라도 메모리 접근을 위한 하드웨어 수준의 경쟁 외에는 문제가 없다. 데이터 영역의 공유는 두 수행 단위간 정보전달을 쉽게 하기 위한 배려로서 스레드를 사용하는 핵심 이유 중의 하나이다[7]. 즉 프로세스간 정보교환(IPC, Inter Process Communication)은 파이프, 공유메모리, 메시지 큐, 세마포 등 운영체제의 중계에 의지해야만 가능하므로 복잡하고 효율이 떨어진다. 그러나 스레드간에는 전역 변수자체가 모두 공유되므로 하나의 프로그램 공간 내에서 정보를 쉽게 교환할 수 있을 뿐 아니라, 데이터의 일부를 적절히 분담하여 처리할 수 있다는 또 다른 이점이 있다. 스택 영역은 각 스레드의 고유 데이터를 저장하는 장소로 사용되기 때문에 분리되어야 한다. 여기서 고유 데이터라 함은 스레드가 생성된 이후에 사용되는 모든 지역변수 및 관련 문맥을 말한다. 즉, 동일한 함수([8]에서는 코드블럭이라 함)를 여러 스레드가 수행할 때, 각 함수에 정의된 지역변수들은 모두 독립된 장소에 위치하므로 스레드간의 혼란을 막고 유일성을 유지할 수 있다.

위의 스레드는 다음과 같이 두 가지 형태로 구현된다.

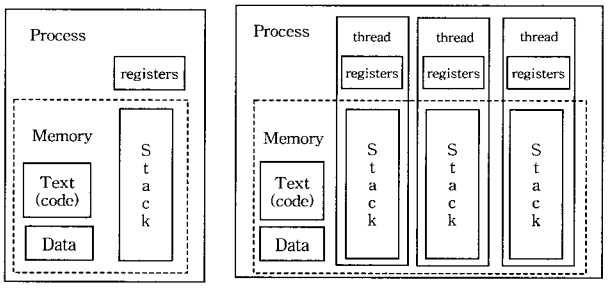
- 운영체제 수준의 스레드

이는 (그림 3)과 같이 PCB 하부에 존재하는 TCB(Thread Control Block) 단위의 스레드 기반 스케줄링에 의해 제공되는 스레드로서 OSF/1, NT와 같이 운영체제 중심에 MACH 등의 마이크로 커널을 필요로 한다. 스레드 기반 운영체제가 다중처리기 시스템에 탑재될 경우, 한 프로세스내에 존재하는 여러 개의 스레드 들이 병렬로 처리 될 수 있다.

- 라이브러리 수준의 스레드

프로세스 기반 스케줄링 환경에서는 스레드를 응용프로그램 수준에서 구현할 수 있다. 이는 전통적인 프로세스 기반 운영체제에서 스레드 환경을 임시적으로 제공하기 위해서 사용되는 기법으로 스레드의 병렬처리는 불가능하지만 인터리빙 형태의 동시처리는 가능하다. 예를 들어 IO부분과 계산 부분을 번갈아 처리함으로써 전체적인 업무처리 성능을 향상시킬 수 있다.

스레드를 보다 포괄적으로 분류해보면 단일 스레드와 다중 스레드로 나눌 수도 있는데, 위에서 제시한 스레드 개념에서 스택을 여러 개 관리하는 경우를 다중 스레드, 이러한 개념이 없는 경우를 단일 스레드라 한다((그림 4) 참조)[7].



(그림 4) 단일 스레드와 다중 스레드

2.2 스레드의 장점과 단점

스레드의 장점은 근본적으로 프로그램의 처리율, 반응시간, 계산속도 등의 성능향상에 있다. 단일 처리기 시스템에서는 IO등 느린 부분과 계산 위주의 빠른 부분을 번갈아 처리함으로써 전체적인 반응속도를 개선할 수 있고, 다중처리기 시스템에서는 다수의 스레드를 병렬처리 함으로써 성능을 크게 개선할 수 있다. 특히 GUI 에서의 사건처리, 분산 환경에서의 서비스 처리 등에서는 다중 스레드가 매우 효율적이다. 이 경우 성능상의 이점은 프로세스 기반 스케줄링 환경에서의 프로세스 생성(fork)에 의한 IPC방법에

대비되는데, 이는 스레드의 생성소멸에 대한 비용이 프로세스의 그것보다 저렴하다는 의미이다.

스레드의 단점으로는 우선 스레드간 동기화 문제로 인하여 입체적인 프로그래밍이 복잡하고 디버깅이 어려워 상당한 수준의 고급기술이 필요하다는 점이다. 또한 종래에 개발된 일반 라이브러리(non-reentrant library)의 사용에 제한을 받을 수 있는데, 이 경우 호출 전·후에 상호배제 잠금장치를 사용해야 한다. 또한 경쟁조건이 심하게 발생하여 스레드의 수행과정에 대한 추적이 어렵고, 교착상태 등의 문제가 자주 발생할 수 있으며, 우선순위가 낮은 스레드가 공유자원을 점유함으로써 높은 우선순위 스레드의 수행을 방해하는 우선순위 왜곡(priority inversion) 현상이 발생하기도 한다[7].

2.3 스레드 적용 모델

다중 스레드를 적용하는 프로그래밍 모델에는 (그림 5)와 같이 크게 3가지가 있다[4].

- Boss/Worker Model(Work/Queue Model)

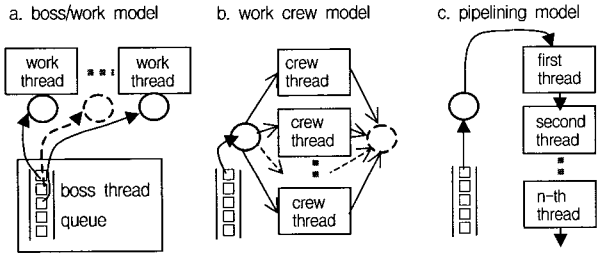
이 모델은 작업 큐를 boss가 관리하면서, 유휴 스레드에게 하나의 일거리 전체를 할당하는 형태인데, 통신 서비스 처리 등에 널리 사용된다.

- Work Crew Model

이 모델은 하나의 커다란 일거리를 여러 스레드가 분담하여(분할·정복) 처리하는 형태로서 본 논문에서 관심 있는 재귀적 알고리즘의 병렬처리 유형이 여기에 해당된다.

- Pipelining Model

이 모델은 한 스레드의 출력을 다른 스레드가 입력으로 받아 처리하는 과정의 연결 형태로서 계산과 인쇄를 전담하는 스레드를 사용하는 경우가 여기에 해당된다.



(그림 5) 스레드의 병렬처리 모델

위 모델들은 다중스레드 프로그래밍에 있어서 반드시 선택적이거나 상호변환이 가능하지는 않고, 처리 대상의 성격에 따라 특정 모델을 필수적으로 사용해야 할 때도 있다.

2.4 스레드 인터페이스의 표준화

앞에서 설명한 스레드 인터페이스는 시스템 및 공급자에 따라 각각 고유하게 구현될 수 있으나, 프로그램의 호환성을 위하여 POSIX(Portable Operating System Interface)1.c의 표준안이 pthread interface라는 명칭으로 제안되어 있고, 공급자들은 이를 표준화 기준으로 따르고 있다[15]. Pthread의 대표적인 몇 가지 인터페이스는 <표 1>과 같다[7].

<표 1> POSIX 스레드의 주요 인터페이스

인터페이스	설 명
pthread_create	스레드를 새로 생성하여 수행을 시작시킴
pthread_exit	스레드가 스스로 수행을 종료함
pthread_join	한 스레드가 다른 스레드의 종료를 기다림
pthread_detach	스레드가 차지하고 있던 모든 자원을 해제
pthread_kill	한 스레드가 다른 스레드를 강제로 종료함
pthraed_self	스레드가 자신의 고유 id를 확인함
pthread_attr_init	스레드 특성부여에 사용될 객체를 초기화
pthread_mutex_lock	동기화를 위한 상호배제 잠금을 실시함
pthread_mutex_unlock	상호배제 잠금을 해제함
pthread_cond_wait	한 스레드가 어떤 사건을 기다림
pthread_cond_signal	한 스레드가 사건을 발생시킴

2.5 스레드 관련 연구 분야

스레드에 관한 연구는 주로 스레드 자체를 구현하는 방법론과 응용 프로그램으로부터 스레드를 자동으로 생성하는 방안 등에 관하여 이루어지고 있다. [8]은 배열에 적용되는 루프에서 최적의 스레드를 생성하기 위한 배열의 지역화 방안에 대하여 논하고, [9]는 다중 스레드 기반의 병렬지원언어 환경인 DAVRID에서 재귀호출에 의한 스레드 생성시 병렬성을 효과적으로 제어할 수 있는 방안을 제시하고 있다. [10]은 루프가 있는 자바 프로그램의 원천코드로부터 병렬성을 인지하여, 이를 다중 스레드에 의해 재구성하는 방법론에 대하여 언급하고 있으며, [11]은 특히 벡터 연산을 위해 최적화된 스레드 구현 방안 MULVEC을 제안하고 있다. [12, 13]은 함수적 언어인 Id로 작성된 언어로부터 스레드를 효과적으로 유도해 내는 과정을 연구하고 있고, [14]는 앞 절에서 설명한 라이브러리 수준의 스레드를 실행 지원시스템이라 명명하고, 병렬 수행 지원 언어인 SR(Synchronizing Resources)의 실행지원 시스템을 POSIX 스레드를 이용하여 구현하는 방안을 제시하고 있다.

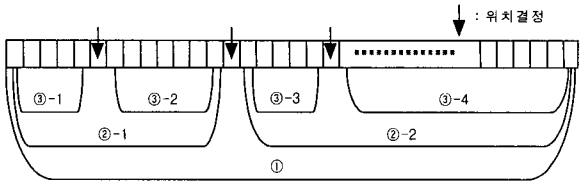
본 논문은 위의 연구들과 다르게 순수한 응용 프로그램 개발자 관점에서 재귀적 알고리즘을 실용적으로 POSIX스레드로 변환하여 병렬화 시키는 방법과, 이 과정에서 제기되는 병렬성 제어 방안에 관해서 논한다.

3. 재귀적 알고리즘의 병렬화 및 병렬성 제어

3.1 분할·정복 재귀함수 호출과 스레드의 대응

본 논문에서 다루고자 하는 재귀적 알고리즘의 범위는

분할·정복 형태의 재귀적 알고리즘이다. 이 형태는 알고리즘의 재귀호출 함수에서 다루는 데이터 영역이 매번 독립적으로 설정되는 특성을 가진다. 본 논문에서 참조하게 될 퀵 정렬[16]의 경우 (그림 6)과 같이 정렬함수에 주어지는 데이터 정렬범위가 차례로 양분되고, 분리된 부분은 결코 중복되지 않는다.



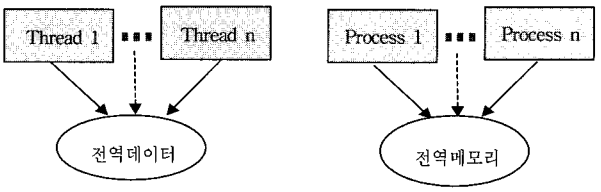
(그림 6) 재귀적 분할정복 알고리즘의 예

위의 예에서 데이터 배열은 전역변수에 저장되고, 처리범위 즉 왼쪽(left)과 오른쪽(right) 경계 값은 재귀함수에 매개변수, 즉 스택에 저장되어 전달되기 때문에, 앞에서 정의한 스레드의 정의에 따라, 주어진 각각의 영역처리를 위한 재귀함수 대신에 스레드를 생성하여 할당할 수 있다. (그림 6)의 처리 과정을 병렬처리하기 위하여 재귀 호출을 스레드로 변환하는 과정을 (그림 7)에 나타내었다.

A. 최초의 시작스레드가 데이터 전체(①)를 대상으로 처리를 시작한다.
B. 처리도중에 재귀적으로 처리할 대상(② -1, ② -2)을 발견한다.
C. 처리대상 중 한 부분을 처리할 스레드를 해당 범위를 지시하는 매개변수와 함께 생성하고, 자신은 나머지 대상을 재귀적으로 호출하여 처리한다.
D. 위에서 생성된 스레드는 모두 동일한 처리함수로 진입한다.
E. 재귀호출 혹은 새로운 스레드에서 호출된 처리함수 내에서 B~D 단계를 반복한다.
F. 함수가 종료하면 호출과정에 따라 재귀호출에서 복귀하거나 스레드가 종료된다.
G. 위의 B~F과정을 처리 대상이 더 이상 없을 때까지 반복한다

(그림 7) 재귀호출에 대한 스레드 대응 절차

(그림 7)의 절차에 따른 재귀적 알고리즘의 스레드에 의한 병렬화는 (그림 5)에서 전형적인 work crew model에 속하고, 이 절차에 따라 생성된 스레드 들은 전역 데이터에 대하여 모두 독립적으로 진행할 수 있으므로 높은 병렬처리 효과를 얻는다. 이와 같은 재귀호출에 대한 스레드 대응은, 전역 데이터를 공유메모리에 배치하여 프로세스로 대치할 수 있다. 이 경우 프로세스가 생성(fork)되면 메모리 이미지 전체가 복사되기 때문에 처리구간 전달을 위한 별도의 기법은 필요치 않다(그림 8 참조).



(그림 8) 스레드와 프로세스에 의한 병렬처리 개념

3.2 재귀적 알고리즘의 병렬화에서 병렬성 제어의 필요성 및 방안

(그림 7)에 나타낸 재귀함수 호출의 스레드 대응은 언뜻 보아 아주 간단해 보인다. 그러나 거기에는 다음과 같이 몇 가지 치명적인 문제점이 내재하고 있다.

- 스레드 개수의 한정

일반적으로 운영체제 내에 존재하는 (그림 3)의 TCB 개수가 한정되어 있으므로 스레드를 무한정 생성시킬 수 없다. 이는 데이터의 크기와 입력 상태에 따라 얼마만큼의 스레드가 필요할 것인가를 사전에 예측할 수 없기 때문에 더욱 심각하다.

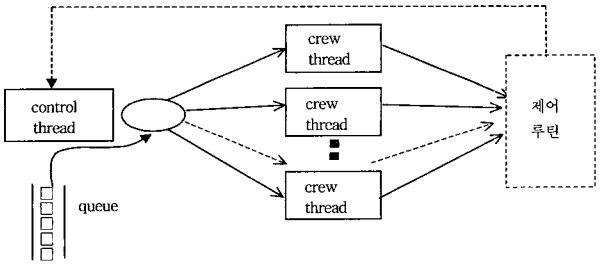
- 스레드 생성·소멸 및 스케줄링에 따른 부담 심화

스레드의 생성과 소멸, 처리기의 스케줄링 등은 모두 운영체제의 시스템 서비스에 의하여 이루어지는데 지나치게 많은 스레드의 활동은 전체적인 시스템의 효율을 크게 저하시킨다.

- 처리의 완료에 대한 보장 불가능

대부분의 스레드 기반 운영체제는 주 스레드(main thread), 즉 맨 처음 프로세스 생성과 함께 탄생한 시작 스레드가 종료하면 나머지 스레드들도 모두 강제 종료되도록 설계되어 있다. 그러나 (그림 7)의 방법에 따르면 주 스레드가 다른 스레드들이 모두 종료할 때까지 기다린다는 보장이 없다. 이는 데이터의 입력 상태에 따라 시작 스레드의 종료 시점이 달라지기 때문이다.

위에서 스레드 개수의 한정 문제는 스레드 생성의 성공 여부에 따라 스레드 혹은 재귀호출을 선택하면 간단하게 해결될 수 있다. 스레드의 잦은 생성·소멸에 의한 부담(overhead) 문제는 현재 생성되어 활동 중인 스레드 수를 전역적으로 관리하여, 임계 값을 초과할 경우 새로운 스레드 생성을 중단하고 재귀호출을 선택함으로써 해결할 수 있는데, 이것은 첫번째 문제점에 대한 해결책도 겸하고 있다. 마지막으로 가장 어려운 문제는 전체적인 처리의 완료를 인지할 수 있는 동기화 장치의 고안이다. 이 문제는 (그림 5)에서의 Boss/Worker와 Work Crew 모델을 조합한 Control/Work Crew 모델(그림 9)을 도입하여 해결할 수 있다.



(그림 9) 병렬성 제어를 위한 Control/Work Crew 모델

(그림 9)에서 제어(control) 스레드는 재귀적 알고리즘을 기동 시킨 후, 처리의 종료를 알리는 신호(사건)를 기다리고, 처리(crew) 스레드 들은 재귀호출에 의해서 생성된 후, 또 다른 처리 스레드를 생성할 필요성이 있거나 처리를 마치고 종료하고자 할 때 반드시 제어루틴을 사용한다. 제어루틴은 스레드 생성 요청에 대해서는 새로운 스레드 생성의 가능 여부를 결정하고, 종료 요청에 대해서는 그것이 마지막 스레드인가를 조사하여, 그럴 경우 처리종료 신호를 발생시킨다. (그림 10)에 위에서 제안된 새로운 모델에 따라 재귀호출을 스레드로 대응시키는 절차를 나타내었다.

여기서 제어루틴은 스레드 생성에 대한 정보를 전역적으로 관리하여 병렬성 제어역할을 담당한다. 이곳에는 현재 활동 중인 스레드 각각에 대한 id및 매개변수 영역, 전체 스레드 수를 저장하여 적정한 스레드 개수의 유지와 처리 전체의 종료를 인지할 수 있도록 한다. (그림 11)에는 (그림 10)의 단계 C에서 호출하는 제어루틴 알고리즘을 제시하였다. 임의의 처리 스레드에 의해서 Alloc_thread()가 호출되면 스레드 생성의 가능여부를 결정하고, 가능할 경우 Dispatch() 함수에서 시작하는 스레드를 생성해준 다음 자신은 성공적으로 복귀한다. 스레드 생성이 불가능하면 실패

A. 최초의 시작스레드가 데이터 전체를 대상으로 처리를 시작할 처리스레드를 생성한 후 자신은 종료 신호에 대기한다.
B. 처리스레드는 처리 도중 재귀적으로 처리할 대상들을 발견한다.
C. 제어루틴을 호출하여 처리대상 중 한 부분을 처리할 처리스레드의 생성을 요청한다. 처리스레드가 생성될 경우 자신은 나머지 처리대상을 재귀적으로 호출하여 처리한다. 처리스레드 생성이 거부되면 처리대상 모두를 하나씩 재귀적으로 호출하여 차례로 처리한다.
D. 위에서 생성된 스레드는 모두 동일한 처리함수로 진입한다.
E. 함수가 종료하면 호출과정에 따라 재귀 호출에서 복귀하거나 스레드가 종료된다.
F. 재귀적 혹은 새로운 스레드에 의해서 호출된 처리함수 내에서 B~E 단계를 반복한다.
G. 처리스레드가 종료할 때에는 제어루틴을 호출해서 자신이 마지막 스레드인지를 조사하고, 마지막인 경우 처리종료신호를 발생시킨다.
H. 종료신호가 발생하면 전체처리가 종료된다.

(그림 10) Control/Work Crew 모델에 따른 재귀적 알고리즘의 병렬화 및 병렬성 제어

사실과 함께 복귀한다. 생성된 스레드는 Dispatch()에서 퀵 정렬과 같은 처리함수를 호출한 후 복귀하면 스레드를 스스로 소멸시킨다. 이 때 자신이 마지막 스레드이면 종료 신호를 발생시켜서 제어 스레드에게 그 사실을 알린다.

```
table   Thread_table[MAX_THREAD] :
int     Nthread = 0 ;
mutex   semaphore = INIT_SEMA ;
external condition cond = INIT_COND ;
// 처리 스레드와 공유
Procedure Alloc_thread(paramater)
begin
   mutex_lock(semaphore) ;
   if (Nthread  MAX_THREAD) then
     begin
       mutex_unlock(semaphore) ;
       return FAIL ;
     end
   … allocate Thread_table[ ] index …
   Nthread++ ;
   mutex_unlock(semaphore)
   Thread_table[index].parameter = parameter[index ;
   create_thread(Thread_table[index].id, Dispatch,
                 Thread_table[index].parameter) ;
   return SUCCESS ;
end

Procedure Dispatch(parameter_index)
begin
   process_function(parameter of parameter_index) ;
   mutext_lock(semaphore) ;
   Nthread-- ;
   … release Thread_table[] of index ·of parameter_index …
   mutex_unlock(semaphore) ;
   if (Nthread = 0) then
     signal_thread(cond) ;
   exit_thread() ;   // thread termination
end
```

(그림 11) 스레드를 이용한 재귀적 알고리즘의 병렬화를 위한 병렬성 제어 알고리즘

4. 제안된 재귀적 알고리즘의 병렬화 및 병렬성 제어 방안의 효용성 검증

여기서는 POSIX 스레드 환경에서 앞에서 제안한 재귀적 알고리즘의 병렬화 및 병렬성 제어방안을 전형적인 분할정복 알고리즘인 퀵 정렬에 적용하여 그 효용성을 검증한다.

4.1 퀵 정렬, 제어루틴, POSIX 제어 스레드의 구현

(그림 12)에는 (그림 10)의 단계 A에서 언급한 퀵 정렬 전체를 관장할 제어 스레드를 C 언어로 구현한 모습을 나타내었고, (그림 13)에는 퀵 정렬의 기본 알고리즘[16]을 (그림 10)의 절차에 따라 재귀호출 부분을 제어루틴 호출로 대체시킨 모습(어두운 부분)을 나타내었다. (그림 14)에는 (그림 12)에 제시된 병렬성 제어 알고리즘을 <표 1>의 POSIX스레드를 적용하여 구현한 내용을 보였다. (그림 14)의 어두운 부분은 (그림 11)에 새롭게 추가한 부분인데, 이는 스레드

수의 유지에 있어서 한 두개 정도의 오차는 허용할 수 있으므로, 상호배제가 없는 상태에 사전 조사를 함으로써 상호배제로 인한 연산부담을 크게 줄이기 위한 것이다.

```
extern Item Data0[ ];  // 처리할 데이터
extern int  Ndata ;    // 데이터 크기

THREAD_POOL Th_pool[ ];    // 제어루틴 정보
int  Nthread = MAX_THREAD ; // 스레드 최대 수
int  Nalloc                 // 활동중인 스레드
pthread_mutex_t          Th_pool_mutex =
                    PTHREAD_MUTEX_INITIALIZER;
# define     Mklrp(l, r, p)  (((long)(l) <<  36) |
                              (long)(r) << 12 | (long)(p))
# define     Left(lrp)     (int)(((long)(lrp) >> 36))
# define     Right(lrp)    (int)(((long)(lrp) >> 12)
                              & 0x000000000fffffL)
# define     Thread(lrp)   (int)(((long)(lrp)  &
                                0x000000fffL))
pthread_cond_t  W_cond =
                    PTHREAD_COND_INITIALIZER ;
pthread_attr_t  T_attr ;        // 처리 스레드 속성
Sort_data()  // 제어 스레드
{
   pthread_attr_init(&T_attr) ;
   pthread_attr_setdetachstate(&T_attr,
                    PTHREAD_CREATE_DETACHED) ;
   pthread_cond_init(&W_cond, &W_attr) ;
   Allocate_thread(1, Ndata) ;   // 처리 스레드 시작

   pthread_cond_wait(&W_cond, &W_mutex) ;
}
```

(그림 12) 퀵 정렬의 병렬처리를 위해 POSIX 스레드로 구현한 제어 스레드

```
void * Quicksort(int left, int right)
{
   int      i, j ;
   Item     v, t ;

   if (left < right) {
      v = Data [ right ] ;
      i = left - 1 ;
      j = right ;
      for ( ; ; ) {
         while (Data [ ++i ] < v)
            ;
         while (Data [ --j ] > v)
            ;
         if (i >= j)
            break ;
         t = Data [ i ] ;  Data [ i ] = Data [ j ] ;  Data [ j ] = t ;
      }
      Data [ right ] = Data [ i ] ; Data [ i ] = v ;
      if (Allocate_thread(left, i-1) < 0)   // 제어루틴 호출
         Quicksort (left, i-1) ;              // 스레드 한계
      Quicksort (i+1, right) ;
   }
}
```

(그림 13) 스레드에 의한 병렬처리를 위해 변형된 퀵 정렬 알고리즘

```
Allocate_thread(int left, int right)
{
   int      i ;

   if (Nalloc >= Nproc)
      return(-1) ;
   pthread_mutex_lock(&Th_pool_mutex) ;
   if (Nalloc >= Nproc) {
      pthread_mutex_unlock(&Th_pool_mutex) ;
        return(-1) ;
   }
   for (i = 0 ;  i < Nproc; i++) {
      if (Th_pool[i].state != ST_BUSY)
           break ;
   }
   Th_pool[i].state = ST_BUSY ;
   Nalloc++ ;
   pthread_mutex_unlock(&Th_pool_mutex) ;
   pthread_create(&Th_pool[i].thread, &T_attr,
                  Dispatch, (void *)Mklrp(left, right, i)) ;
   return(i) ;
}

void Dispatch(void * lr)
{
   int      th ;
   Quicksort(Left(lr), Right(lr)) ;
   pthread_mutex_lock(&Th_pool_mutex) ;
     th = Thread(lr) ;
   Th_pool[th].state = ST_IDLE ;
     Nalloc-- ;
   pthread_mutex_unlock(&Th_pool_mutex) ;

   if (!Nalloc)      // 모든 처리 종료
      pthread_cond_signal(&W_cond) ;
   pthread_exit(NULL) ;
}
```

(그림 14) 퀵 정렬을 위한 POSIX 스레드 병렬성 제어 루틴

4.2 퀵 정렬에서의 병렬처리 및 병렬성 제어의 효용성 검증

4.2.1 시험환경

- 시스템 및 시험 데이터

(그림 12)~(그림 14)의 프로그램을 <표 2>의 조건하에서 실행시켰을 때의 성능측정 결과를 분석하였다.

<표 2> 퀵 정렬 병렬처리 시험환경

항 목		사 양
시스템 환경	시스템 모델	COMPAQ ES40
	CPU 모델	Alpha Ev67 667MHz
	CPU 개수	4
	메모리 크기	4G Byte
	버스 구조	SMP(UMA)
	운영체제	COMPAQ Tru64UNIX
시험 데이터	레코드 크기	1024(1K)
	레코드 개수	1K~10,000K
	정렬 키	정수(난수 발생)

● 성능지표

컴퓨터 시스템의 성능평가는 평가 목적에 따라 성능지표가 달라질 수 있는데, 반응시간(response time), 처리율(throughput), 시스템 이용율(utilization) 등이 그것들이다[17]. 병렬처리의 목적은 반응시간을 단축하는데 가장 큰 목적이 있으므로 본 논문에서는 반응시간 위주로 성능을 분석하고, 다만 병렬처리로 인한 부담을 참고하기 위하여 순수한 처리기 사용시간도 추가로 분석하였다.

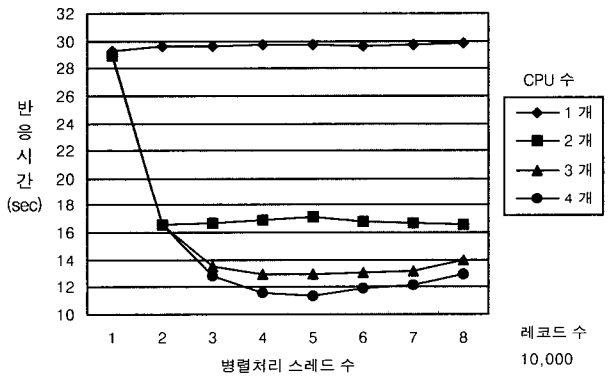
4.2.2 시험결과 분석

● 병렬처리의 효용성

병렬처리의 효용성은, 순차처리와 병렬처리의 개념을 <표 3> 과 같이 정의하여 각각을 상호 비교하였다. 즉 CPU와 스레드 모두 2개 이상인 경우만이 병렬처리 결과로 고려하고 나머지는 순차처리로 간주하였다.

〈표 3〉 순차처리와 병렬처리의 정의

CPU 분류 \ 스레드분류		스레드 개수	
		1 개	2개 이상
CPU 개수	1 개	순차처리	순차처리
	2개 이상	순차처리	병렬처리



(그림 15) (그림 12)~(그림 14)의 수행시간(1)

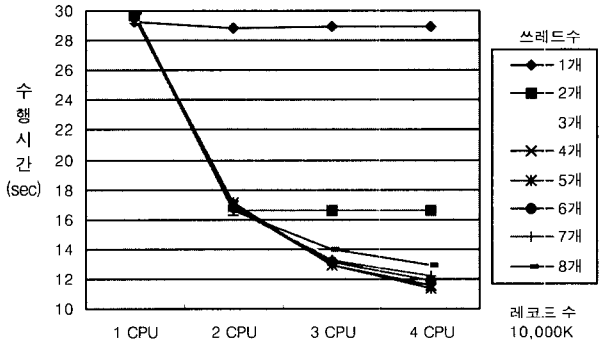
(그림 15), (그림 16)는 (그림 12)~(그림 14)의 프로그램을 <표 2>의 환경에서 CPU 수를 1에서 4개까지 늘려가며 수행시켰을 때의 성능측정 결과를 서로 다른 측면으로 보여주고 있는데, 그 결과는 아래와 같이 보편적으로 알려진 사실과 일치한다.

① POSIX스레드에 의한 병렬처리가 순차처리보다 우수하다.

② CPU 수 보다 적은 범위에서의 병렬처리 스레드 수의 증가는 성능개선에 도움을 준다. 그러나 CPU수가 많을수록 성능개선의 폭이 둔화된다

③ 병렬처리 스레드 수가 CPU 수보다 적을 경우 더 이상의 CPU 증가는 성능개선에 도움을 주지 못한다.

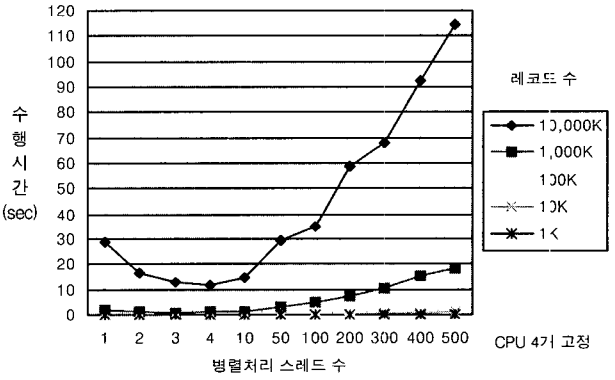
④ CPU 수보다 병렬처리 스레드 수가 많아지면 성능이 저하되는데, 특히 CPU수가 많을 수록 그 둔화 폭이 커진다.



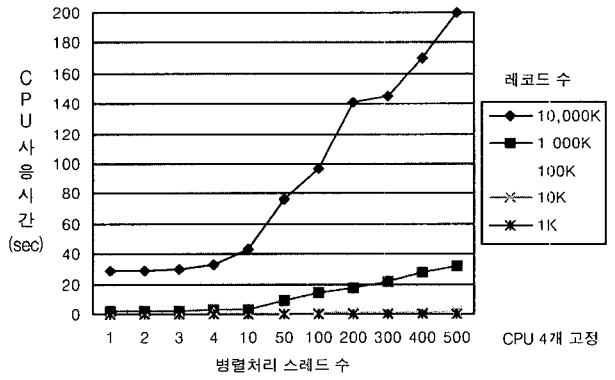
(그림 16) (그림 12)~(그림 14)의 수행시간(2)

● 병렬성 제어의 효용성

(그림 17)은 동일한 환경에서CPU 수를 4개로 고정했을 때 병렬처리 스레드 수의 증가에 따른 반응시간의 추이를 보여주고 있는데, 병렬성 제어를 하지 않아 너무 많은 스레드가 병렬처리를 시도할 경우 성능이 오히려 순차처리보다 저하되고 있음을 알 수 있다.



(그림 17) 병렬처리 스레드 수와 반응시간 관계

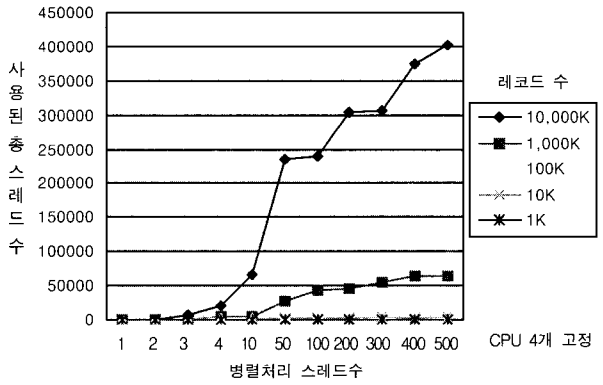


(그림 18) 병렬처리 스레드 수와 CPU사용시간 관계

(그림 18)은 병렬처리 스레드 수의 증가에 따라 내부적으

로 CPU가 실제로 소모한 시간의 추이를 보여주고 있는데, 병렬처리 스레드 수가 증가할수록 CPU 시간소모가 급격하게 증가하는 사실을 알 수 있다. 이는 스레드가 지나치게 많을 경우 스레드의 생성과 소멸, 문맥교환, 그리고 상호배제 연산에 그만큼 많은 시간이 필요하기 때문이다.

(그림 19)에는 처리기간 중 생성되었다가 소멸된 스레드의 총 수와 병렬처리 스레드 수와 상관관계를 나타내었다. 이 현상 또한 앞의 결과들과 일맥 상통하고 있음을 알 수 있다.



(그림 19) 병렬처리 스레드 수와 총 스레드 수 관계

이상의 분석과정은 제안된 병렬성 제어방안이 POSIX스레드로 구현한 재귀적 알고리즘의 병렬성을 효과적으로 제어 할 수 있고, 이를 이용한 병렬처리에 의해 다중처리기 시스템의 이점이 충분히 활용될 수 있음을 보여주고 있다.

5. 결 론

본 논문에서, 스레드에 의한 재귀적 알고리즘의 병렬화는 일차적으로 성능향상을 가져오지만, 지나친 병렬처리는 오히려 성능을 저하시키기 때문에 병렬성 제어가 필요하다는 점을 인식하고, POSIX 스레드 환경에서 병렬처리 스레드 수를 제어할 수 있는 장치를 제안하고, 그 효용성을 퀵 정렬을 통하여 실증적으로 검증하였다.

여기서 제안된 병렬성 제어 방안은 보편적으로 알려진 시스템의 최적 실행환경에 적응하기 위하여 재귀적 알고리즘의 병렬처리 스레드 개수를 임의로 제어할 수 있는 장치의 하나로서 4.2에서 보는 바와 같이 그 기능이 우수하다. 시스템 실행환경이 단순한 처리기 수 이외에 시스템의 현재 부하상태까지도 포함한다면 그 실행환경은 시시각각으로 변화한다고 보아야 하는데, 이 때 여기서 제안된 병렬성 제어 방안이 효율적으로 적용될 수 있을 것이다.

위의 병렬성 제어 방안은 병렬성 지원을 위한 특별한 언어나 도구의 지원 없이 일선의 응용 프로그램 개발자들이 실용적이고 현실적으로 병렬처리 프로그래밍에 적용할 수 있는 하나의 모델역할을 할 수 있다는 점에서 의의가 크다고 본다.

다만 다중 스레드 자체의 구조적 효율성은 공급 시스템에 따라 약간의 차이가 있을 것으로 보이기 때문에 이 부분에 대한 시스템 간 성능평가를 고려한 병렬성 제어가 된다면 POSIX스레드에 의한 알고리즘 병렬화는 더욱 유용하게 활용될 수 있을 것이다.

참 고 문 헌

[1] 모상만, 한우종, 윤석한, "Cache Coherence Protocols in NUMA Multiprocessors," 전자통신동향분석, 제13권 제5호, 1998.

[2] Kai Hwang, "Advanced Computer Architecture,"M cGraw-Hill, 1996.

[3] H. M. Deitel, "An Introduction to Operating Systems," Addison Wesley, 1983.

[4] 이형봉, "UNIX/LINUX 커널의 설계 및 구현", 홍릉과학출판사, 2001.

[5] O'Reilly & Associatiates, Inc., "Guide to OSF/1 : A technical Synopsis," O'Reilly & Associatiates, Inc., 1991.

[6] Jeffrey M. Denham, Paula Long, James A. Woodward, "DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation," Digital Technical Journal, Vol.6, No.3, 1994.

[7] Digital, "Guide to DECthreads," Digital, 1997.

[8] 양창모, 유원희, "다중스레드 코드 생성을 위한 배열 지역화", 정보처리논문지, 제3권 제6호, pp.1407-1416, 1996.

[9] 이대웅, 조태희, 한상영, "다중 스레드 구조 컴퓨터에서 프레임-토큰 기반 병렬성 제어", 정보과학회논문지(A), 제24권 제2권, pp.196-210, 1997.

[10] 황득영, 최영근, "자바 프로그래밍에서 병렬처리를 위한 중첩 루프 구조의 다중스레드 변환", 정보처리논문지, 제5권 제8호, pp.1997-2012, 1998.

[11] 윤성대, 정기동, "벡타 연산을 효율적으로 수행하기 위한 다중 스레드 구조", 정보처리논문지, 제2권 제6호, pp.974-984, 1995.

[12] 하상호, 김홍환, 한상영, "다중스레드 컴퓨터상에서 비정형성 함수 언어의 효과적인 루프 펼침 기법", 정보과학회논문지, 제22권 제6호, pp.946-956, 1995.

[13] 하상호, 한상영, 김홍환, 김수홍, "비정형성 함수 언어를 위한 향상된 스레드 형성 기법", 정보과학회논문지, 제21권 제12호, pp.2318-2329, 1994.

[14] 김영곤, 정영필, 박양수, 이명준, "POSIX쓰레드를 이용한 SR 실행지원시스템의 설계 및 구현", 정보처리논문지, 제4권 제4호, pp.1106-1120, 1997.

[15] John, S., Quarterman, Susanne Wilhelm, "UNIX, POSIX, and Open Systems," Addison Welsley, 1993.

[16] Donald E. Knuth, "Sorting and Searching," Addison Wesley, 1973.

[17] Domenico Ferrari, "Computer Performance Evaluation," PRENTICE-HALL, 1978.

이 형 봉

e-mail : hblee@honam.ac.kr

1984년 서울대학교 계산통계학과 졸업 (이학사)

1986년 서울대학교 계산통계학과 졸업 (이학석사)

1986년~1994년 LG전자 컴퓨터연구소

1994년~1999년 한국디지탈㈜

1997년~1999년 전자계산조직응용·정보통신기술사

1999년~현재 호남대학교 정보통신공학부 조교수

관심분야 : 프로그램 언어 및 보안, 운영체제, 멀티미디어 통신

백 청 호

e-mail : pch@kangwon.ac.kr

1968년 서울대학교 사범대학 수학 교육학과 졸업(학사)

1981년 성균관대학교 경영대학 졸업(석사)

1982년 경기대학교 졸업(박사)

1984년~현재 강원대학교 전자계산학과 교수

관심분야 : 수치해석, 이산수학, 질적데이터분석, 전산통계