POSIX Thread를 이용한 병렬 태스크 생성 메커니즘의 구현[1)]

윤봉식, 황선태
국민대학교 컴퓨터 학부

Implementation of Parallel Task Creation Mechanism in POSIX Thread

Bongsik Yoon, Suntae Hwang
School of Computer Science, Kookmin University

요 약

본 논문에서는 동적 병렬 태스크를 추출하는 메커니즘을 공유 메모리 구조상에서 구현하는데 있어서 최적화 기법에 대해서 소개한다. 최대의 호환성을 고려하여 POSIX thread와 C 언어를 사용하였는데 이에 따른 오버헤드를 여러 가지 구현 기법을 사용하여 줄이고자 노력하였다. 그리고 병렬 처리시 대부분의 계산은 각 처리기에서 순차 계산에 의해 이루어지므로 이 순차 계산 시에 발생하는 오버헤드를 줄이는 것을 최우선으로 하였다.

1. 서론

대부분의 병렬 처리 메커니즘들은 대규모의 병렬 컴퓨터를 가정하고 있어서 처리기(Processing Element) 수가 적을 때는 오히려 순차 연산보다도 병렬 처리한 것의 성능이 떨어지는 경우도 많다. 그런데 최근 하드웨어 기술의 발달과 가격인하로 인해 보다 고성능의 컴퓨터를 접할 수 있는 기회가 보다 많아 졌으며 2-4개의 프로세서를 장착한 다중 프로세서 머신도 쉽게 접할 수 있는 추세이다. 따라서 소규모의 병렬 처리에도 관심을 가질 필요가 있으며 이 경우 병렬 처리를 위한 오버헤드를 효과적으로 줄이지 못하면 단일 프로세서에서 순차 연산으로 처리한 것 보다 못한 결과를 내기가 쉽다[1].

컴퓨터의 보급이 폭 넓고 다양화되면서 소프트웨어의 호환성은 중요한 문제로 부각되는데 병렬 처리를 위한 하부 메커니즘 또한 마찬가지이다. 따라서 이 메커니즘을 구성할 때 가능하면 일반적인 운영체제에 널리 퍼진 컴파일러를 사용하는 것이 바람직한데 본 논문에서 소개할 메커니즘은 이런 호환성을 최대로 유지하기 위해서 POSIX thread library를 이용하며 이 메커니즘에서 처리될 함수는 C 언어로 작성됨을 가정한다. 이 함수는 유저가 직접 작성하여도 좋고 병렬 함수 언어와 같은 다른 언어에서 번역된 것이라 가정하여도 좋다. 이 때 사용되는 C 컴파일러는 GNU C 컴파일러라 가정한다. GNU C 컴파일러는 호환성을 걱정할 필요가 없을 정도로 널리 퍼져있고 쉽게 구할 수 있는 것이다.

2. 태스크 추출 메커니즘

병렬 프로그램의 계산이 진행되면서 병렬 처리 가능한 부분은 함수 단위로 태스크로 만들어져서 태스크 풀에 저장된다. 이 때 저장되는 정보는 그림 1에서 보는 바와 같이 실행할 함수의 포인터, 이 함수의 인자

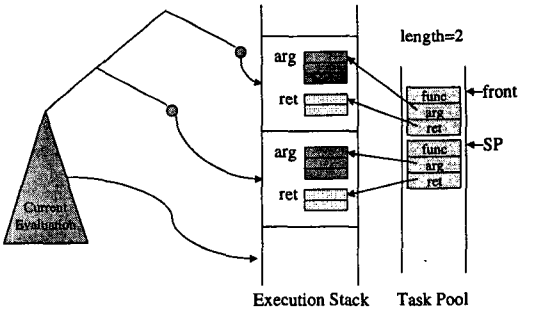

그림 1: Task의 생성

그리고 계산이 끝난 후 결과 값을 돌려줄 장소의 주소 등이다. 여기서 함수의 인자와 결과 값 반환 장소는 Automatic local variable로 Execution Stack에 잡힌다. 이는 heap에 할당하는 것보다 빠른데 태스크의 생성과 지역 계산에서의 처리가 스택처럼 이루어지면 가능한 일이다.

병렬화는 태스크를 수출함으로서 이루어지므로 병렬 처리를 하려는 함수는 태스크 버전을 따로 정의해야 한다. 즉 그림 2에서 처럼 nfib()과 똑같은 계산을 하지만 인자를 한 포인터로 전달받고 결과 값을 RETURN_VALUE()에 의해 지정된 반환 장소에 넣는 함수인 task_nfib()을 따로 정의해야한다.

각 태스크는 depth first 방식으로 생성되어 태스크 풀에 저장되며 현재의 지역 계산이 끝나면 가장 최근에 저장한 태스크부터 계산이 완료되었는지 체크한다. 이는 그림 2의 IS_EVALUATED()부분에 해당되는데 생성된 태스크의 계산 결과와 현재의 계산 흐름을 동기화[2] 하는 것이다. 해당 태스크가 이미 계산이 완료되어 있으면 반환된 값을 바

1) 본 연구는 정보통신부에서 지원하는 대학기초연구지원사업으로 수행되었음

```
typedef int nfib_arg_t;
int nfib(int);
void task_nfib(nfib_arg_t *arg, ret_t *ret);


int nfib(int n)
{
    if (n<2) return 1;
    else if (n<GRAIN) return 1 + nfib(n-2) + nfib(n-1);
    else {
        ret_t ret;
        nfib_arg_t new_arg = (nfib_arg_t)(n-1);
        PUT_TASK((fp_t)task_nfib,(arg_t *)&new_arg,&ret);
        return 1 + nfib(n-2) +
          (IS_EVALUATED(&ret) ? (int)ret.val : nfib(n-1));
    }
}

void task_nfib(nfib_arg_t *arg, ret_t *ret)
{
    int n = (int)*arg;
    int result;

    if (n<2) result = 1;
    else if (n<GRAIN) result = 1 + nfib(n-2) + nfib(n-1);
    else {
        ret_t new_ret;
        nfib_arg_t new_arg = (nfib_arg_t)(n-1);
        PUT_TASK((fp_t)task_nfib,(arg_t *)&new_arg,
                    &new_ret);
        result = 1 + nfib(n-2) +
          (IS_EVALUATED(&new_ret) ? (int)new_ret.val :
                    nfib(n-1));
    }
    RETURN_VALUE(ret, (generic_t)result);
}

MAIN()
{  . . .
  printf("nfib %d=%d\n",n,nfib(n));
}
```

그림 2: 병렬 nfib 함수

로 이용하고, 아무도 건드리지 않았으면 태스크 풀에서 이 태스크를 제거하고 현재의 계산 환경에서 해당 부분을 계산한다. 만일 태스크가 계산 중이면 현재의 계산 쓰레드는 block 되고 새로운 태스크를 얻어다가 이를 계산하는 새로운 쓰레드를 생성한다. 따라서 계산이 진행됨에 따라서 태스크 풀은 스택처럼 늘었다 줄었다하게 된다.

그리고 다른 프로세서가 계산할 새로운 태스크를 요구하면 지역 계산이 쓸데없이 중단되는 것을 막기 위해서 가장 오래된 태스크를 먼저 준다. 즉 태스크 풀은 지역 처리기가 볼 때는 스택이고 부하 균형 메커니즘이 볼 때는 큐인 것이다.

3. 최적화 기법

3.1. 지역 태스크 풀

병렬 처리 시에 대부분의 계산 시간 동안, 머신은 포화 상태에 있게되고 각 프로세서는 병렬 프로그램일지라도 순차로 계산하게 된다. 따라서 그림 2의 PUT_TASK() 같은 함수는 최대로 최적화 되어야 한다[2].

```
typedef struct {
    fp_t fp;
    arg_t *arg;
    ret_t *ret;
} task_t;

void *worker(void * id)
{
    task_t *a_task;
    INIT_POOL((int)id);
    . . . .
    a_task = GET_TASK();
    ((fp_t)a_task->fp)((arg_t *)a_task->arg,(ret_t *)a_task->ret);
}

void main()
{   . . . .

    INIT_POOL(0);
    . . . .

    for(id=1; id<N; id++) {
      pthread_create(&threads[id], &bounded, worker, (void *)id);
    }
    . . . .

    MAIN(); /* Evaluate the main stream */
}
```

그림 3: Run time system의 골격

태스크 풀을 중앙에 한 개만 유지할 경우 구조 자체가 복잡해지고 각 쓰레드 끼리 경쟁 상태에 있게 되므로 동기화를 해야하는 오버헤드가 발생하게된다. 따라서 그림 4와 같이 각 쓰레드 별로 태스크 풀을 따로 유지하기로 한다. 이렇게 되면 PUT_TASK()와 부하 균형을 위한 그림 3의 GET_TASK()간에만 동기화가 이루어지면 되며 GET_TASK()는 매우 드물게 발생하므로 서로 경쟁 상태에 있게되는 경우는 거의 없다.

각 쓰레드 별로 태스크 풀을 따로 갖게되면 분산 메모리 아키텍쳐 컴퓨터에서의 메커니즘과 별 차이가 없게되고 부하 균형 시에 메시지를 주고받아야 하는 상황이 발생할 수 도 있다. 하지만 모든 쓰레드들이 한 주소 공간에 있으므로 다른 쓰레드를 위한 태스크 풀도 자유롭게

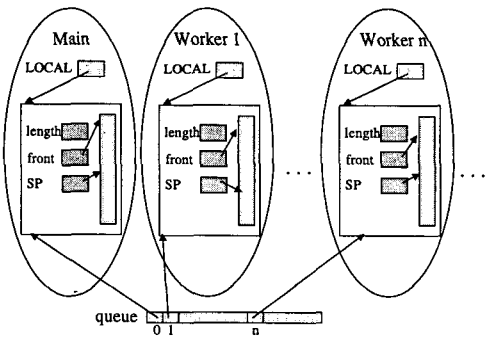

그림 4: Task Pool의 구조

2) 이는 태스크의 생성을 억제해서 이를 수도 있지만 본 논문에서는 다루지 않기로 한다.

액세스할 수 있다. 따라서 공전 상태의 쓰레드가 새로운 태스크를 찾아야 하는 상황이 되면 다른 쓰레드에게 메시지를 보내는 것이 아니라 자신이 직접 다른 쓰레드의 태스크 풀을 탐색하여 태스크를 찾아간다. 그림 4의 queue라는 전역 배열은 이런 방법의 부하 균형을 위한 것이다.

3.2. 태스크의 동기화

각 쓰레드의 태스크 풀은 지역 계산과 새로운 태스크를 찾는 공전 쓰레드 사이에서 공유하게 된다. 즉 PUT_TASK() 함수와 GET_TASK() 함수는 상호 배제 관계이어야 한다. 이를 위해서 POSIX thread에서 제공하는 mutex를 사용하여도 되지만 PUT_TASK() 나 GET_TASK()는 시간이 오래 걸리는 함수들이 아니므로 불필요한 문맥 교환보다는 busy waiting을 하는 것이 더 이롭다. 그리고 mutex에 관련되어 제공된 lock, unlock 함수를 호출하는 것도 그 횟수가 엄청나게 많기 때문에 호출 자체가 성능을 저하시키는 요인이 된다. 따라서 제공된 함수 대신에 해당 프로세서의 atomic instruction을 이용하여 SPIN LOCK을 만들어 사용하였다[3].

그 다음에 필요한 동기화는 반환 값 구조체에 있는 태스크의 상태를 액세스하기 위한 것이다. 관련된 함수들은 IS_EVALUATED, RETURN_VALUE, GET_TASK 등인데 태스크 풀에 태스크가 2개 이상일 때만 GET_TASK를 허용하면 GET_TASK는 이 동기화에서 제외시킬 수 있다. POSIX thread에는 쓰레드가 스스로 suspend되고 다른 쓰레드를 resume하는 방법이 없으므로 condition variable을 사용하여 값이 계산 중일 때는 condition wait하는 방법을 썼다. 여기서 condition wait로 block되기 전에 새로운 worker를 생성하여 최소한 처음의 N-way 병렬성을 유지하도록 한다. RETURN_VALUE는 값을 반환하면서 condition variable에 signal을 보내어 블록된 해당 쓰레드를 깨운다.

3.3. 불필요한 인자 전달의 방지

각 지역 계산은 지역 태스크 풀과 이에 관련된 정보들을 유지해야 한다. 즉 그림 2에서 PUT_TASK()에는 그림 3의 worker()에서 정의한 태스크 풀의 정보가 전달되어야 한다. 그런데 이를 전역 변수로 할 경우 동일 주소 공간에 그 instance가 1개뿐이어서 각 쓰레드가 서로 혼동하여 사용하려 하므로 잘못된 연산을 야기한다. 따라서 이 정보를 worker()의 지역 변수로 선언하고 이를 nfib과 task_nfib과 같은 모든 해당 함수에 추가의 인자로 넘겨주어야 하는데 그 횟수가 많으므로 오버헤드가 누적되어 비효율적이다.

```
register queue_t *LOCAL  asm("edi");
// register global variable on i386 machine
```

그림 5: GNU C의 register global variable

따라서 각 태스크 풀은 non-local variable이면서 각 thread가 각각의 instance를 가질 수 있는 클래스의 변수이어야 한다. 이를 위해서 POSIX thread library에서 thread specific data를 제공하지만 사용 빈도를 감안 할 때 매우 비효율 적이다. 따라서 다른 방법을 찾아야 하는데 GNU C compiler의 global register variable[4]이 이 목적에 매우 적절한 것으로 판명되었다. 즉 그림 5 처럼 전역 변수를 메모리가 아닌 레지스터에 정의함으로서 각 쓰레드 별로 instance를 갖게되는 것이다. 이는 그림 4에서 각 쓰레드가 LOCAL이라는 변수를 각각 갖는 것으로 나타나있다.

4. 결과

앞에서 설명한 태스크 생성 메커니즘을 2개의 CPU인 다중 프로세서 컴퓨터에서 POSIX thread를 가지고 구현하였다. 실험은 간단히 그림 2의 nfib 함수를 가지고 실시하였는데 nfib(40)을 실행한 결과들은 다음 표와 같다. 표안의 숫자들은 순수한 순차 프로그램을 1 CPU에서 실행한 시간을 1로 하였을 때의 상대적인 시간을 나타낸다.

	1 CPU	2 CPU	speed up	
			1CPU / 2 CPU	순수순차 / 2 CPU
순수한 순차 프로그램	1	-	-	-
최적화 이전의 병렬 프로그램	2.37	1.19 *	2	0.8
최적화 이후의 병렬 프로그램	1.71	0.86**	2	1.2

결과에서 보듯이 어떤 방법을 써도 2 CPU에서 1 CPU보다 2배의 speed up을 보인다. 하지만 * 표시한 데이터처럼 2 CPU에서도 1 CPU에서의 순수 순차 프로그램보다 못한 결과를 보이는 경우가 있다. 하지만 최적화를 하였을 때는 ** 표한 데이터처럼 2 CPU에서 1 CPU의 순수 순차 프로그램에 비해 약 1.2배의 speed up을 보인다.

5. 결론

병렬 처리 시 대부분의 계산 시간 동안 머신은 포화 상태에 있게 되고 각 프로세서는 병렬 프로그램이든 순차 프로그램이든 상관없이 순차 연산을 하게 된다. 따라서 이 순차 연산을 빠르게 수행하는 것은 전체 성능 향상에 크게 기여하는 것이다. 따라서 많은 프로세서를 사용하여 오버헤드를 극복하는 것보다는 오버헤드 자체를 줄이는 것이 더 중요한 요소가 되는 것이다. 결과에서 본 것과 같이 앞에서 설명한 최적화 기법은 성능 향상에 현저하게 기여한다. 즉 1 CPU에서 성능이 향상되었다는 것은 머신이 포화 상태 일 때의 성능이 향상되었다는 것이며 이는 전체 성능을 향상시키는 것이다. 본 논문에서는 태스크의 수를 줄여서 오버헤드를 극복하는 방안은 검토되지 않았는데, nfib 보다 메모리 액세스가 더 많은 실제 응용 프로그램에서는 어떤 결과가 나올지를 실험하는 것과 더불어 향후 과제로 남긴다.

참고 문헌

[1] S. Hwang, Lazy Decomposition: A novel technique to control parallel task granularity, *Proceedings of the IEEE Third International Conference on Algorithm And Architecture for Parallel Processing*, Dec 1997

[2] S. L. Peyton-Jones, Parallel Implementation of Functional Programming Languages, *The Computer Journal*, 32(2): 175--186, 1989

[3] Hank Dietz, Linux Parallel Processing HOWTO, Jan 1998

[4] R. M. Stallman, *Using and Porting GNU CC for version 2.7.2.1*, Jun 1996