

인텔 패러럴 스튜디오 2011 속으로

# 패러럴 어드바이저의 적합성 분석 (Suitability analysis)

인텔이 최근 발표한 병렬 프로그래밍을 위한 패러럴 스튜디오(Parallel Studio) 2011에는 새로운 패러럴 어드바이저(Parallel Advisor)가 추가되었다. 이름에서 말하듯 패러럴 어드바이저는 기존의 C/C++로 작성된 직렬 프로그램을 병렬화하는 데 있어 여러 도움을 주는 기능을 제공한다. 3회에 걸쳐 패러럴 어드바이저의 자세한 기능과 원리, 그리고 이와 관련된 기초적인 병렬 프로그래밍 지식에 대해 알아본다.

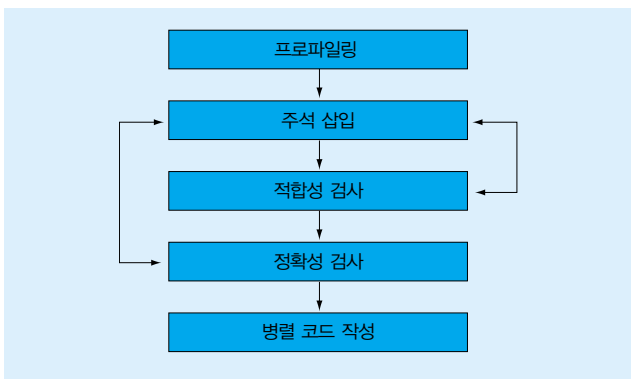
**연재순서**

- 1회 | 2010.11 | 병렬화를 이끄는 패러럴 어드바이저
- 2회 | 2010.12 | 패러럴 어드바이저의 적합성 분석(Suitability analysis)
- 3회 | 2011. 1 | 패러럴 어드바이저의 정확성 분석(Correctness analysis)



김민장 art.oriented@gmail.com, http://minjang.egloos.com | 미국 Georgia Tech 전산학 박사과정 학생으로 병렬 프로그래밍을 돕는 프로그램 분석 방법에 대해 연구하고 있다. 이번 연재에서 소개하는 패러럴 어드바이저의 기능에 관련된 연구를 인텔에서 했다. C/C++와 병렬 프로그래밍에 관심이 많다.

패러럴 스튜디오 2011에서 새롭게 선보인 패러럴 어드바이저는 직렬 프로그램 중 '어디를', 또 그 부분을 '어떻게' 병렬화할 것인가에 대해 도움을 주는 도구다.



〈그림 1〉 패러럴 어드바이저의 워크플로우(workflow)

패러럴 어드바이저는 〈그림 1〉처럼 병렬화하고자 하는 직렬 코드에 대해 (1) 프로파일링, (2) 주석 삽입, (3) 적합성 검사, (4) 정확성 검사를 수행한다. 이 과정에서 (3), (4)의 결과에 따라 주석을 고치고 다시 (3), (4)를 반복할 수 있다. 이번 컬럼에서는 패러럴 어드바이저가 지원하는 주석(annotation)과 적합성 검사(suitability analysis)에 대해 알아본다. 적합성 검사는 프로그래머가 삽입한 주석을 기반으로 프로그램을 분석해서 병렬화한 후 예상 성능 향상치를 구한다.

**적합성 검사는 왜 필요한가?**

지난 컬럼에서 알아본 서베이가 프로파일링으로 실행 비중이 큰

코드 영역, 특히 루프나 함수를 찾았지만 이 부분이 바로 병렬화로 이득을 얻는 것은 아니다. 이상적으로 프로그래머가 원하는 것은 프로세서 개수를 늘리는 만큼 병렬 프로그램의 수행 시간이 비례해 단축되는 것이다. 그러나 아무리 병렬화가 되더라도 프로그램의 정확성을 위해 직렬로 실행되어야 하는 부분이 있으면 병렬화의 효과는 저해된다. 또, 아무리 그러한 부분이 적거나 없더라도 각 스레드가 병렬로 처리하는 일의 양에 불균형이 발생한다면 역시 좋은 성능 향상을 기대할 수 없다. 이런 이유로 실제로 얻는 병렬 프로그램의 성능 향상은 이상치보다 낮을 때가 많다. 문제는 이러한 실제 성능 향상치를 미리 병렬화하기 전에 알아볼 수 있는 방법이 현재까지는 거의 없었다는 것이다. 기껏 힘들게 병렬화했는데 실제 얻어진 성능 향상이 미미하다면 시간 낭비다. 따라서 프로그래머가 실제로 병렬화하기 전에 어떤 방법으로 병렬 프로그램의 성능 기대치를 알 수 있다면 아주 좋을 것이다. 적합성 검사는 바로 이것을 위한 것이다. 정확성 검사는 실제 프로그램을 병렬화하지 않고 간단한 주석 삽입과 짧은 시간 내의 분석으로, 비록 아주 정확하지는 않더라도 비교적 정확한 수준으로 병렬 프로그램의 성능 기대치를 프로그래머에게 알려준다. 적합성 검사는 보통 프로그램의 원래 수행 시간의 2~3배 이내와 별로 크지 않은 메모리 사용량으로 분석을 마칠 수 있다.

또 다른 이유로는 적합성 검사 뒤에 이어지는 정확성 검사는 일반적으로 아주 긴 분석 시간과 상당히 많은 메모리가 필요하다. 실제 프로그램 수행 시간의 100배 이상 느려지는 것이 일반적이고 프로그램에 따라 수 GB가 넘는 메모리가 요구된다. 따라서 이런 비용이 상대적으로 큰 정확성 수행을 하기 이전에 가볍

고 빠르게 이 코드의 예측 성능을 가능한다면, 정확성 분석에 대한 낭비도 줄일 수 있을 것이다.

### 병렬화 성능 향상의 척도 - 스피드업

병렬화 후 프로그램의 성능 향상은 일반적으로 스피드업(speedup)이라는 기준으로 표현한다. 적합성 검사에서도 주어진 프로그램, 혹은 일부 코드 영역의 병렬화 후 스피드업을 예측한다. 스피드업은 직렬 프로그램의 수행 시간을 병렬화 후 얻은 수행 시간으로 나눈 값이다. 이상적일 때, 듀얼코어라면 스피드업은 2, 쿼드코어라면 스피드업이 4가 될 것이다. 즉, 듀얼코어와 쿼드코어에서 싱글코어의 2배, 4배 빠른 수행 시간을 얻을 수 있다. 그러나 앞서 설명한 이유, 작업 불균형과 직렬 부분으로 수행되어야 하는 부분의 존재로 이런 숫자는 얻기가 쉽지 않다. 그렇지만 행렬 곱셈과 같이 어떠한 동기화도 필요 없는 프로그램은 이런 이상적인 스피드업도 얻을 수 있다. 이런 부류의 프로그램을 처치 곤란한 병렬성(embarrassingly parallel)을 가진 프로그램이라 말한다. 여기에 속하는 응용도 무척 많다. 맵리듀스(MapReduce) 같은 방법론으로 처리가 가능한 작업도 병렬 처리 작업 자체는 서로 의존성이 없는 처치 곤란한 병렬성이 있어야 한다. 그런데 병렬화 후, 코어의 개수보다 더 높은 스피드업도 얻을 수 있다. 대표적으로 행렬 곱셈이 이룰 수 있는데, 이것은 캐시 적중률이 병렬화 코드에서 더 높아질 수 있기 때문에 가능한 것이다. 이런 컴퓨터 구조적인 요소를 제외하면 물리 프로세서가 N개라면 스피드업의 상한은 N이다.

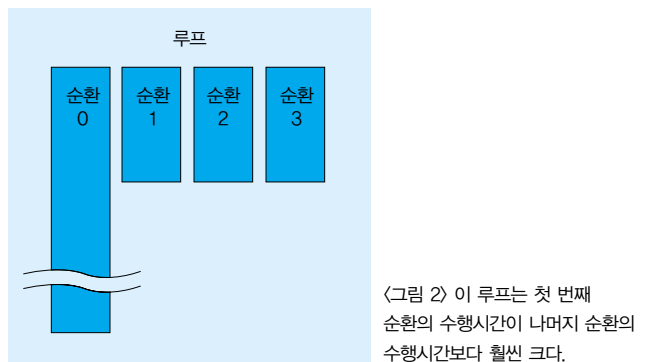
### 실제 프로그램의 스피드업 예측

적합성 검사의 목적은 주어진 직렬 프로그램의 예상 스피드업을 측정하는 것이다. 적합성 검사 이전에는 어떻게 스피드업을 예측했는지와 그 한계점을 알아본다. 적합성 검사는 기존 방법의 한계를 극복하기 위해 새로운 측면에서 시도하는 스피드업 예측 방법이다. 어떤 프로그램의 스피드업을 예측하는 방법은 먼저 해석적(analytical) 혹은 정적인(static) 방법이 있다. 아마 가장 간단하면서 오래된 방법으로는 바로 암달의 법칙(Amdahl's law)이 있다. 전산학을 전공하신 분이라면 반드시 알고 있을 내용이다. 어떤 프로그램이 있고 이를 병렬화하려는데 전체 수행 시간 중 P 비율만이 병렬화될 수 있다고 하자. 그럴 때, N개의 프로세서에서 기대할 수 있는 스피드업은 암달의 법칙으로 다음과 같이 표현된다.

$$\frac{1}{(1-P) + \frac{P}{N}}$$

만약 병렬화 가능한 부분의 비율만 안다면 이상적인 스피드업을 구할 수 있다. 예를 들어, 60%의 코드가 완전히 병렬화 가능하다면 듀얼코어에서는  $1/(0.4+0.6/2)=1.4$ 배, 쿼드코어에서는  $1/(0.4+0.6/4)=1.8$ 배 빠르게 처리가 가능하다는 것을 암달의 법칙은 말해준다. 이 간단한 암달의 법칙은 병렬화할 때, 결국 병렬화되지 않는 부분이 성능 향상을 제한한다는 간단해 보이지만 상당히 중요한 전산학의 기본 원리를 일깨워 준다. 물론 암달의 법칙에 대한 반론도 있다. 구스타프슨(Gustafson)은 1988년 그의 논문에서 암달의 법칙이 고정된 일만을 가정하고 있음을 지적했다. 그는 병렬화 문제에서 프로세서 개수가 늘 때 문제의 크기도 같이 늘린다면 더 많은 병렬성을 확보할 수 있음을 주장했다. 그렇다하더라도 암달의 법칙을 부정하는 것은 아니고 다른 시각에서 병렬화 문제를 바라본 것이다.

그런데 이 암달의 법칙이 실제 프로그램의 스피드업을 구하기에는 사실상 부족하다. 암달의 법칙은 매우 이상적인 가정을 암묵적으로 한다. 첫째, 프로그램은 완벽히 직렬로 수행하거나 아니면 완벽히 병렬로 수행되어야 한다. 둘째, 병렬로 수행되는 부분이 있다면 그 부분은 완벽히 같은 형태의 일이다. 셋째, 프로그램에는 임계구역(critical section)과 같은 어떠한 동기화 객체도 없다. 이러한 조건으로 사실상 실제 프로그램의 스피드업 예측에는 사용하기가 어렵다. 단적인 예로 <그림 2>와 같은 프로그램을 생각해 보자.



<그림 2>의 프로그램에서 어떤 루프는 첫 번째 순환이 다른 순환보다 훨씬 길다. 만약 이 루프를 병렬화한 후의 스피드업을 예측하고 싶다면 어떻게 해야 할까? 아쉽게도 이미 암달의 법칙의 두 번째 가정을 위반하므로 사실 쓸 수가 없다. 굳이 쓴다고 하더라도 앞선 수식의 P 값, 전체 수행 시간 중 병렬화 가능한 부분의 비율을 구하는 것이 직관적이지 않다. 프로그래머는 아마 이 루프 전체의 수행 시간을 이 비율로 산정할 것이다. 그것이 프로그래머의 의도였기 때문이다. 그렇다면 암달의 법칙은 단순히 이상적인 스피드업만 구해준다. 그러나 실제 수행은 그림과 같이 작

업 불균형으로 현저히 낮은 스피드업이 얻어진다. 이러한 이유로 암달의 법칙은 조금만 가정이 벗어나는 현실적인 프로그램에서 사용하기가 어렵다.

또 이런 해석적인 방법이 간과하는 사실은 병렬 프로그램의 실제 병렬 수행 과정이다. 구체적으로 말하면 병렬화된 코드가 어떻게 스케줄링되는가의 문제다. <그림 2>에서 4개의 순환이 듀얼 코어에서 실행된다고 생각해 보자. 이 경우, 비록 순환 0이 아무리 길어도 순환 1부터 순환 3을 다른 코어에서 모두 실행시킨다면 그나마 작업 불균형을 줄일 수 있다. 그러나 단순히 순환 0 과 순환 1을 첫 번째 코어에, 순환 2와 순환 3을 두 번째 코어에 할당한다면 스피드업은 더욱 악화될 것이다. 이러한 점 역시 암달의 법칙과 같은 해석적인 모델로는 담기가 아주 어렵다.

앞서 말했지만 암달의 법칙은 임계구역을 가정하지 않는다. 최근 발표된 논문에서 이어만(Eyerman)과 엑하우트(Eeckhout)은 암달의 법칙에 비록 이상적이지만 확률 기반의 임계구역을 모델링해 스피드업을 도출하는 수식을 구했다. 그러나 그들의 모델 역시 실제 프로그램의 스피드업 예측을 목표로 한 것보다는 임계구역이 실제 병렬 프로그램에서 어떤 의미를 가지는지 알아보기 위함이었다. 암달의 법칙이나 확장된 모델은 실제 프로그램의 스피드업을 예측하기에는 많이 부족하다. 적합성 분석은 이 약점을 보완하기 위한 새로운 방식의 스피드업 예측 기법이다.

### 적합성 검사의 기본 작동 원리

적합성 검사는 기존의 해석적인 방식과 사뭇 다른 측면에서 이 문제를 접근한다. 정적인 방법이 아니라 '동적인' 방식으로 스피드업을 예측한다. 즉, 실제 프로그램을 수행하면서 얻은 정보를 분석해 이 프로그램의 예상 스피드업을 얻는다. 따라서 여기에도 일종의 프로파일링이 들어간다. 적합성 검사에서 독특한 것은 이 프로파일링 정보를 기초로 프로그램을 병렬화하지 않았는데도 마치 병렬화한 것처럼 에뮬레이션(emulation)하는 것이다. 이 에뮬레이션으로 비로소 스피드업 예측이 가능하다.

적합성 검사를 하기 위한 기본 조건에 대해 알아보자. 적합성 검사는 반드시 직렬 프로그램에 대해서만 작동이 가능하다. 이미 프로그램이 멀티스레드로 작성되었다 치더라도 적어도 적합성 검사를 수행할 코드 영역은 반드시 직렬 코드여야 한다. 즉, 주스레드 외에 다른 스레드와의 어떠한 상호 작용도 없어야 한다. 이 조건은 사실상 제약 조건이라 볼 수는 없다. 이미 있는 직렬 프로그램을 병렬화할 때는 자명하게 충족되는 조건이다. 그리고 처음부터 새로 만드는 병렬 프로그램이라 하더라도 일반적으로 직렬 프로그램을 먼저 만들고 병렬화하므로 이 기본 전제 조건은 매우 합리적이라고 볼 수 있다. 적합성 검사의 개괄적인 작동 단

계는 다음과 같다.

**1 주식 삽입** : 곧 뒤에서 자세히 알아보겠지만 프로그래머가 일단 어떤 코드 영역이 병렬화되고, 또 임계구역으로 보호받을지 주식(C/C++ 매크로 형태)으로 코드에 기입한다. 참고로 이러한 코드에 추가적인 기술을 더하는 것을 일반적으로 어노테이션(annotation) 기법이라 한다. 여기서 하나 의문이 든다면, 병렬화하고자 하는 부분은 지난 컬럼에 소개한 서버이 기능으로 알 수 있지만 어떻게 임계구역을 알 수 있을까라는 것이다. 어떤 코드를 임계구역으로 설정해야 하는지는 정확성 검사에서 추론할 수 있다(정확히 말하면, 정확성 분석은 임계구역을 설정하는 데 힌트를 준다. 아직까지 구체적으로 이렇게 저렇게 하라는 도움을 주지는 못한다). 따라서 최초에는 임계구역에 관련된 주식은 아예 넣지 않거나 프로그래머의 기초적인 관찰이나 지식에 의존한다. 그리고 이어지는 정확성 분석에서 보다 이 임계구역을 면밀히 수정하고 다시 적합성 검사를 수행할 수 있다. 어차피 정확성 분석에는 오차가 있을 수 있으므로 임계구역과 관련해 매우 엄밀한 주식 삽입이 반드시 필요한 것이 아니다.

**2 프로파일링 단계** : 주석을 삽입한 코드를 재컴파일하고 대표적인 입력 값을 설정하고 적합성 분석 시작 버튼을 누르면 먼저 이 프로그램을 실제로 수행하면서 프로파일링 정보를 얻는다. 여기서 얻는 프로파일링 정보는 각 주식 사이의 소요 시간을 구한다. 예를 들어, 병렬화될 루프의 각 순환의 길이를 모두 측정한다. 임계구역은 그 구간의 소요 시간을 측정한다.

**3 분석 단계** : 프로파일링이 끝나면 실제 적합성 검사가 이뤄지는데 앞서 소개했듯이 핵심은 병렬 실행의 에뮬레이션에 있다. 앞서 얻은 프로파일링 정보를 가지고 2~32 코어에서의 예측 병렬 수행 시간을 에뮬레이션해서 그 결과를 프로그래머에게 보여 준다. 단, 여기서 에뮬레이션하는 병렬 컴퓨터는 매우 이상적인 것으로 캐시, 메모리 그리고 복잡한 운영체제의 스레드 스케줄링까지 반영되는 것은 아니다.

### 적합성 검사 실제 사용 - 주식(annotation) 넣기

적합성 검사와 정확성 검사는 프로그래머가 직접 소스 코드에 특별한 매크로를 기입해야 그 부분이 분석된다. 패러럴 어드바이저가 지원하는 주식에 대해 알아보자. 패러럴 어드바이저의 주석을 쓰려면 일단 비주얼 스튜디오에서 'Add New Item'의 Parallel Advisor 2011 항목에서 'advisor-annotate.h' 헤더 파일을 추가해야 한다. 적합성 검사에서 사용 가능한 주석을 알아

보자. 그 전에 이 주석이 직렬 코드 중 병렬 실행 부분과 뮤텍스로 보호 받아야 하는 코드 영역을 표시함을 기억하자. 주석에는 사이트(site), 태스크(task) 개념이 있다.

**1 ANNOTATE\_SITE\_BEGIN/ANNOTATE\_SITE\_END** : 사이트의 시작과 끝을 알려주는 주석이다. 사이트는 패러럴 어드바이저에서 하나 또는 여러 개의 패러럴 태스크(parallel task)를 가지는 코드 영역으로 정의된다. 쉽게 설명하면 병렬 작업이 실행될 수 있는 공간이다. 그런데 이 사이트를 벗어나려면 반드시 그 안의 모든 태스크의 실행이 종료되어야 한다. 스레드 조인(join) 혹은 배리어(barrier) 작업이다.

**2 ANNOTATE\_TASK\_BEGIN/ANNOTATE\_TASK\_END** : 태스크의 시작과 끝을 알려주는 주석이다. 태스크는 사이트 내에서 다른 태스크와 병렬로 실행되는 코드 영역으로 정의된다.

**3 ANNOTATE\_LOCK\_ACQUIRE/ANNOTATE\_LOCK\_RELEASE** : 임계구역의 시작과 끝을 알려주는 주석이다.

사이트와 태스크의 개념을 보다 정확히 이해하기 위해 간단한 코드 예를 살펴보자. 어떤 루프를 병렬로 시작하고 싶다면 일단 루프의 시작과 끝을 사이트로 정의한다. 그리고 루프 내의 순환을 태스크로 설정하면 OpenMP, TBB 같은 라이브러리로 루프를 병렬화하고자 하는 의도를 표현할 수 있다. <리스트 1>은 지난 회에도 사용한 N-queen 계산 프로그램의 예다. 이 코드는 패러럴 어드바이저를 설치하면 예제 프로그램으로 볼 수 있다.

<리스트 1> nqueen 프로그램의 주요 계산 루틴인 solve 함수

```
void solve() {
    int * queens = new int[size];
    for(int i=0; i<size; i++)
        setQueen(queens, 0, i);
}
```

지난 컬럼에서는 solve() 함수가 가장 많은 소요 시간을 보임을 알았다. 따라서 이 루프를 병렬화해 보자. 이 루프를 병렬화한다는 의도는 <리스트 2>처럼 주석을 삽입함으로써 기술할 수 있다.

앞서 설명했듯이 사이트는 병렬로 실행되는 태스크의 시작과 끝을 알려주므로 루프의 시작과 끝에 넣는다. 태스크는 병렬로 실행되는 코드 영역으로 루프를 병렬화할 때는 루프 순환이 각각의 스레드에서 병렬로 작동하므로 루프 순환의 시작과 끝을 태스크로 감싸면 된다. 괄호 안의 인자는 단순히 이 주석을 구분하는

지시자에 불과하므로 어떠한 문자열을 주어도 괜찮다. 이 코드를 실제로 병렬화한다고 한다면 <리스트 3>처럼 될 수 있다.

<리스트 2> solve의 for 함수를 병렬화하겠다는 주석

```
void solve() {
    int * queens = new int[size];
    ANNOTATE_SITE_BEGIN(solve);
    for(int i=0; i<size; i++) {
        ANNOTATE_TASK_BEGIN(setQueen);
        setQueen(queens, 0, i);
        ANNOTATE_TASK_END(setQueen);
    }
    ANNOTATE_SITE_END(solve);
}
```

<리스트 3> solve 함수를 OpenMP로 병렬화

```
void solve() {
    int * queens = new int[size];
    #pragma omp parallel for schedule(dynamic, 1)
    for(int i=0; i<size; i++)
        setQueen(queens, 0, i);
}
```

OpenMP의 문법을 모르는 독자는 다소 낯설겠지만 의미는 간단하다. 아래 for 루프를 병렬화하라는 것이고 루프 순환을 분배 및 스케줄링하는 방법을 schedule(dynamic, 1)에 기술했다. 현재 버전의 적합성 검사는 OpenMP의 schedule(dynamic, 1)이나 TBB의 work-stealing과 유사한 방법으로 스피드업을 예측하기에 직접적으로 적어 보았다. 여기서 숫자 1은 청크(chunk) 크기다. 이 (dynamic, 1) 스케줄의 의미는 먼저 루프 순환을 하나씩 나눈 뒤에(만약 숫자 8이라면 루프 순환을 8개씩 나눔) 그 덩어리(chunk)를 스레드에게 동적으로 할당하는 것을 뜻한다. 동적이라 함은 스레드와 일감의 관계가 실제 실행 시 결정되는 것이다. 어떤 스레드든지 자신의 작업을 마치면 아직 일감이 있는지 보고 있다면 바로 수행하는 구조가 OpenMP의 동적 스케줄링 방식이다. 반면, 정적 스케줄링은 실제 루프 순환의 실행 시간과 상관없이 정적으로 스레드 순환을 스레드에게 미리 할당해주는 것이다. 일반적으로 동적 스케줄링이 더 나은 스피드업을 보여주나 동적 스케줄링은 오버헤드가 있으므로 항상 정적 스케줄링보다 더 나은 것은 아니다.

병렬 루프가 종료되려면 병렬로 처리되는 루프 순환이 모두 종료되어야 한다(예외적으로 OpenMP의 nowait 키워드를 쓰면 이 조건이 완화된다). 그래서 parallel for의 끝 부분에는 임묵적인 배리어(barrier) 동기화 작업이 있다. 이것은 앞서 소개한 사이트의 개념과 동일하다. 임계구역을 설정하는 것은 그 부분만 알아낸다면 단순히 주석만 넣으면 될 것이다. 다음 컬럼에서 정확성

분석으로 임계구역을 추론해 넣는 부분에 대해 알아볼 것이다.

### 적합성 검사의 실제 사용 - 프로파일링 및 분석하기

주석 작업을 마쳤으면 재컴파일하고 그냥 적합성 검사 시작 버튼만 누르면 분석이 자동으로 이뤄진다. 여기에는 앞서 소개한 프로파일링과 에뮬레이션이 차례대로 실행된다.



〈그림 3〉 적합성 검사 결과의 예

최종 적합성 검사의 결과 화면은 꽤 직관적이다. 프로세서 개수에 따른 예상 스피드업이 그래프로 나타난다. 그리고 구체적으로 사이트와 태스크의 수행 시간 통계치도 나온다. 이 예에서는 입력 값으로 15를 줬다. 그렇게 되면 〈리스트 2〉의 루프가 15번 순환하게 된다. 결과에서 보듯이 사이트는 한번 불렀고, 태스크는 15번이 불렀음을 확인할 수 있다. 그리고 각 사이트/태스크 인스턴스의 시간 값이 나온다. 이 값을 기반으로 에뮬레이션하고 스피드업을 예측한다. 여기서 보면 비록 임계구역이 하나도 없고 거의 대부분의 시간을 solve 함수가 보내지만 듀얼 코어에서의 예상 스피드업은 2보다 작은 1.9가 나온다. 결과를 다시 주목하면 15개의 태스크가 10% 정도의 편차를 보인다. 즉, 작업 간의 불균형이 어느 정도 있어 이상적인 스피드업에는 도달하지 못했다.

그런데 결과 화면을 보면 병렬화 자체에 대한 각종 오버헤드를 줄이는 것에 대한 설명이 있다. 이 간단한 예에는 이런 오버헤드가 대두되지 않았지만, 복잡한 프로그램, 특히 임계구역이 빈번히 일어나거나 최상위(top-level) 루프가 아닌 내부의(inner) 루프가 병렬화된다면, 작업간의 불균형이나 임계구역에 따른 스피드업 감소 외에, 병렬화 자체에 대한 오버헤드가 심각해질 수 있다. 오버헤드에 대해 각각 하나씩 설명한다.

먼저 사이트 오버헤드는 말 그대로 사이트를 열고 닫는 비용을 말한다. 사이트를 여는 것은 실행될 스레드를 만들고(fork) 닫는 것은 모든 스레드가 종료할 때까지 기다리는 것(join)이다. 우리

가 살펴보는 예에서는 오직 한 번의 병렬 루프만 있으므로 포크와 조인이 각각 한 번씩만 있고 이 오버헤드는 거의 없다. 그런데 실제 OpenMP, Cilk 같은 현대 병렬 프로그래밍 라이브러리는 병렬 루프가 시작될 때 스레드를 만들고, 종료될 때 스레드를 파괴하지는 않는다. 미리 스레드를 만들어놓고 있으므로(parking) 실제 스레드 생성/파괴 비용에 해당하는 사이트 오버헤드는 일반적으로 적다. 태스크 오버헤드는 병렬로 처리되는 일감, 즉 태스크를 실제 스레드에 할당하는 비용을 가리킨다. 그리고 이 병렬 태스크가 종료되었을 때, 스레드로부터 이 작업을 종료하는 것도 포함된다. 사이트 오버헤드가 물리적인 스레드 생성 및 종료 시간이라면 태스크는 이 스레드에 직접 일감을 할당하고 회수하는 오버헤드라고 할 수 있다. 사이트 오버헤드나 태스크 오버헤드가 특히 중요해지는 것은 지나치게 많은 사이트가 열고 닫히거나, 아니면 너무 짧은 태스크가 빈번히 만들어질 때다. 이럴 때 병렬화로 얻는 이득이 이 오버헤드로 상쇄될 수 있다.

#### 〈리스트 4〉 내부 루프가 병렬화되었을 때

```
void matrixMultiply(float A[ ][ ], float B[ ][ ], float C[ ][ ])
{
    for (int i = 0; i < N; i++)
        #pragma omp parallel for schedule(static, N/nthreads)
            for (int j = 0; j < N; j++)
                for (int k = 0; k < N; k++)
                    C[i][j] += A[i][k]*B[k][j];
}
```

〈리스트 4〉는 간단한 행렬 곱셈 프로그램인데 일반적인 구현은 for 루프를 3개 중첩해 구현한다. 이 코드는 최상위 루프 또는 두 번째 루프에서 병렬화가 가능하다. 최상위 루프가 병렬화가 가능하지만 두 번째 루프에서 병렬화하는 것은 일반적으로 매우 비효율적이다. 패러렐 어드바이저의 용어를 빌자면 바로 사이트 및 태스크 오버헤드가 크게 증가하기 때문이다. N이 아주 크다면 태스크 오버헤드가 어느 정도 감소하겠지만 작은 숫자라면 하나의 for-j 순환에서 일어나는 계산량이 적으므로 태스크 오버헤드를 충분히 상쇄시킬 만큼 되지 않을 것이다. 물론 최상위 루프에서 병렬화가 가능하지 않을 때는 어쩔 수 없이 내부 루프의 병렬화를 생각해야 한다. 그 때는 반드시 이런 병렬화 자체의 오버헤드를 잘 고려해야 한다.

락 오버헤드는 락(뮤텍스, 세마포어, 크리티컬 섹션) 자체를 만들고 열고 놓는데 드는 비용을 가리킨다. 이 비용은 각 운영체제마다 다를 것이다. 예를 들어, 윈도우에서는 일반적인 임계구역은 유저 객체인 CRITICAL\_SECTION 객체를 이용하는 것이 커널 객체인 Win32의 Mutex, Semaphore를 쓰는 것보다 더 비용이 적다. 락 경쟁(contention)은 다른 스레드가 락을 쥐고 있을

때 그것을 기다리는 데 들어가는 비용이다. 종합적으로 이 락에 관련된 비용을 줄이려면, 최대한 불필요한 락은 만들지 않으며, 가능하다면 락을 잡는 범위를 줄여 병행성을 높여야 할 것이다. 락프리(lock-free) 자료구조 또한 사용 목적에 적합하다면 적절한 선택이다.

마지막으로 앞서 스케줄링 기법을 설명하면서 언급한 청킹(chunking)은 태스크 오버헤드와 관련이 있다. 청킹을 켜고 루프를 병렬화하면, 여러 루프 순환을 뭉치로 하나의 논리적 병렬 작업으로 간주하고 스레드에 할당한다. <리스트 4>에서는 OpenMP의 스케줄링 구문에 루프 순환 횟수를 스레드 개수만큼 나누는 것을 청크 값으로 했다. 그렇다면 N/nthreads만큼의 루프 순환이 덩어리로 각각의 스레드에 할당된다. 청크 크기가 커지면 태스크 개수가 줄어 스케줄링 오버헤드가 감소한다. 반면, 청크 크기가 1이라면 매 루프 순환이 각각 다른 스레드에게 할당하는 작업이 일어나므로, 특히 동적 스케줄링일 때 오버헤드는 커진다. 청킹은 일반적으로 좋다. 그러나 루프 순환마다 작업 불균형이 극단적으로 심할 때는 낮은 청크 크기가 오히려 더 좋을 수 있다. 청킹을 가능케 하려면 경우에 따라 프로그램의 구조를 바꿔야 할 때도 있다.

마지막으로 스레딩 모델이라는 항목이 있는데 기본 값으로 인텔의 TBB로 병렬화되었을 때의 스피드업 예상치를 보여준다. 이것은 OpenMP, TBB, Cilk 혹은 Win32/Pthreads로 병렬화했을 때 병렬화 자체의 오버헤드가 다르기 때문에 그것을 감안한 것이다.

### 적합성 분석의 한계점

적합성 분석은 비록 기존의 해석적인 방법보다는 훨씬 강력하지만 한계점도 분명 있다. 적합성 분석의 핵심은 병렬 실행의 에뮬레이션에 있다. 그런데 이 에뮬레이션이 여러 이상적인 가정을 하고 있어 실제 아주 복잡한 프로그램을 분석할 때는 오차가 클 수도 있다. 가정을 열거하면 다음과 같다: (1) 캐시, 메모리, 하이퍼스레딩 같은 컴퓨터 구조적인 요소는 고려되지 않음, (2) 락은 현재로서는 하나만 지원됨, (3) 세마포어나 조건 변수 같은 임계구역 외의 동기화 객체는 고려되지 않음, (4) 병렬화 라이브러리에 대한 오버헤드 값이 고정되어 있음, (5) 현대 운영체제의 선점형 및 라운드로빈 형식의 스레드 스케줄링은 고려되지 않음(FCFS, 선입 선처리 및 비선점형 방식에 가까운 스케줄링 방식으로 에뮬레이션).

### 적합성 분석의 정확도

적합성 분석의 정확도는 필자가 별도로 연구한 자료가 있으나

아직까지 공식적으로 출판되지 않아 이번 연재에서는 공개하지 못함을 아쉽게 생각한다. 일단 비교적 간단한 예(중첩되지 않은 병렬화 루프 + 불균형한 작업 분포 + 하나의 임계구역)는 상당히 높은 정확도를 보인다. 물론 캐시나 메모리가 고려되지 않았으므로 병렬화될 때 메모리/캐시 행동 패턴이 직렬 프로그램의 그것과 크게 다르지 않다는 추가 가정이 필요하다. 그러나 중첩된 병렬화 루프(nested parallelism)에서는 현재로서는 에뮬레이션의 몇몇 이상적인 가정으로 다소 오차가 있는 편이다. 그러나 다음 버전에서는 개선될 것이다. 적합성 분석은 프로파일링을 주석 내에서 소요된 '시간' 값을 기준으로 쓴다. 정확하게는 Win32의 QueryPerformanceCounter를 쓰는데 지나치게 짧은 주석 내 시간은 오차를 크게 할 수 있다.

지면 관계상 적합성 분석의 실제 구현 알고리즘을 설명하지는 못했지만, 현재 적합성 분석은 압축 과정이 여러 단계에서 적용되어 있다. 압축이 필요한 이유는 루프 순환이 아주 많거나 중첩된 루프일 경우에는 프로파일링 과정에서 아주 많은 데이터가 생성될 수 있기 때문이다. 그래서 수십 GB나 되는 메모리를 소비할 수도 있어 압축해야만 한다. 현재는 일종의 손실 압축을 하고 있다. 이 압축으로 인한 오차는 필자의 연구 결과로는 아주 크지 않은 것으로 일단 밝혀졌지만, 잠재적인 정확도 저해의 요인이 될 수 있다. 추후에 더 나은 기법으로 이 압축의 빈도를 최소한을 줄일 수 있을 것이다.

### 요약

적합성 분석은 직렬 프로그램의 예상 스피드업을 병렬화하기 전에 미리 알려주는 동적 프로그램 분석 기법이다. 프로그래머가 병렬로 실행될 부분과 임계구역 영역을 주석으로 기입한다. 그러면 패러럴 어드바이저는 이 주석 사이의 시간을 재는 프로파일링을 수행한다. 이 정보를 기반으로 가상의 병렬 컴퓨터에서 마치 이 코드가 병렬화되어 수행되는 것을 에뮬레이션한다. 여기서 얻어진 수행 시간 값으로 스피드업을 여러 조건에 따라 프로그래머에게 보여준다. 그러나 메모리, 캐시, 현대 운영체제와 같은 복잡한 것까지는 모델링하지 않으므로 실제 프로그램에서는 다소 오차가 나올 수 있다. +

#### 참고자료

1. J. L. Gustafson. Reevaluating Amdahl's law. Commun. ACM, 31, May 1988.
2. S. Eyerer and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In ISCA, 2010.
3. OpenMP for construct. <http://msdn.microsoft.com/en-us/library/b5b5b6eb.aspx>.