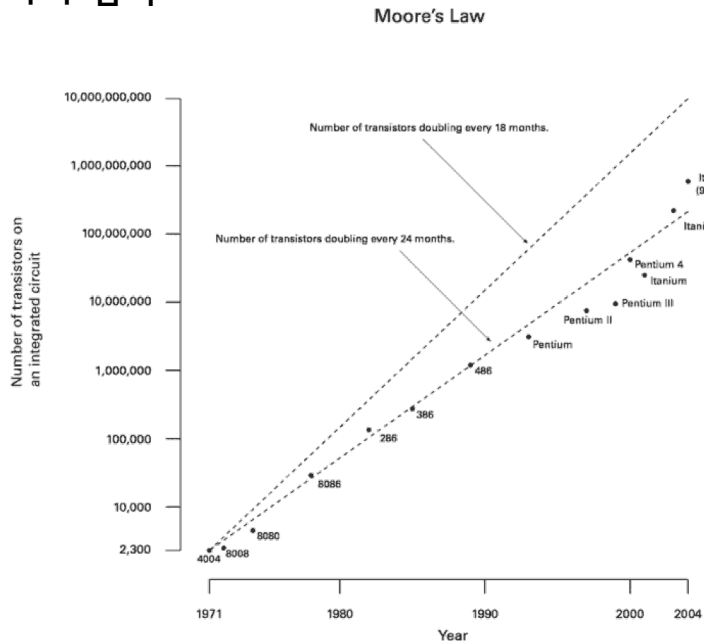


# 병렬 프로그래밍

(주) 아임구루  
아이오 교육센터  
윤찬식

# 왜 병렬 프로그래밍은 중요한가?

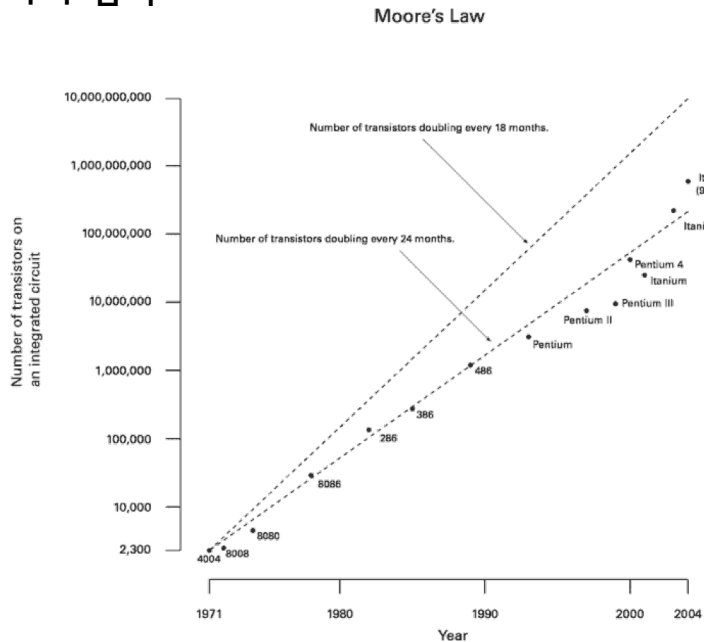
- 무어의 법칙



- 반도체의 집적회로 성능은 18개월마다 2배로 증가한다.
  - : 컴퓨터의 성능은 일정 시기마다 배가하며 기하 급수적으로 증가한다.

# 왜 병렬 프로그래밍은 중요한가?

- 무어의 법칙



Intel 8086 Processor (1978)	29,000	5MHz
Intel Core i7 (2008)	731,000,000	2,930MHz

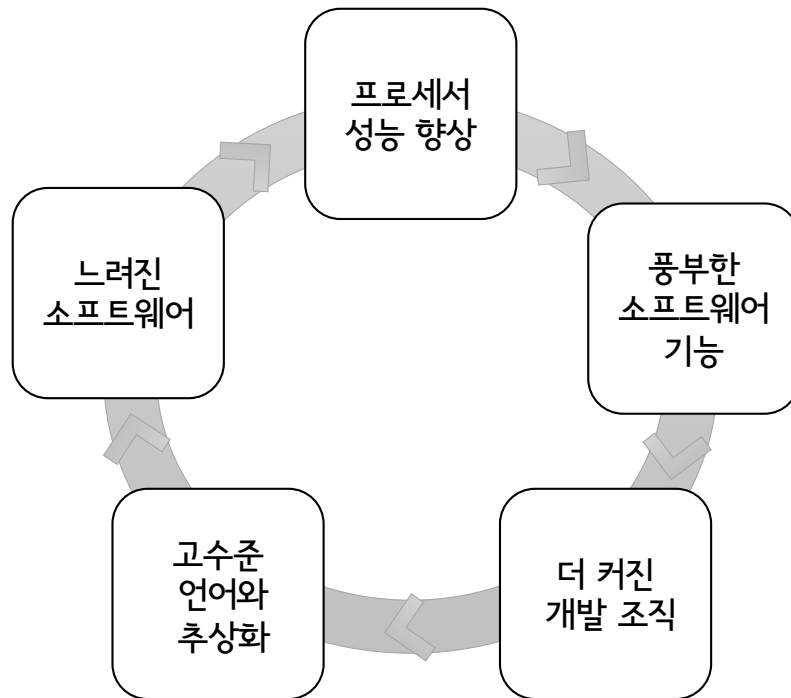
‘30년’동안

[반도체의 수] 는 25,206배

[클럭 속도] 는 586배

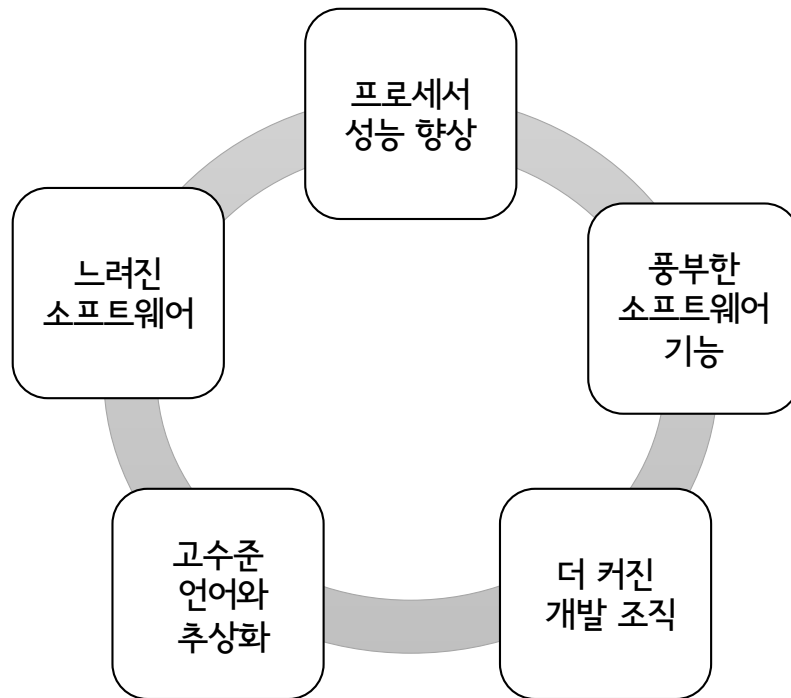
- 반도체의 집적회로 성능은 18개월마다 2배로 증가한다.
  - : 컴퓨터의 성능은 일정 시기마다 배가하며 기하 급수적으로 증가한다.

## 왜 병렬 프로그래밍은 중요한가?



- 2년마다 프로세서 성능이 향상함에 따라 기존의 프로그램을 수정하지 않아도 빨라진 컴퓨터의 혜택을 누릴 수가 있었다.

## 왜 병렬 프로그래밍은 중요한가?

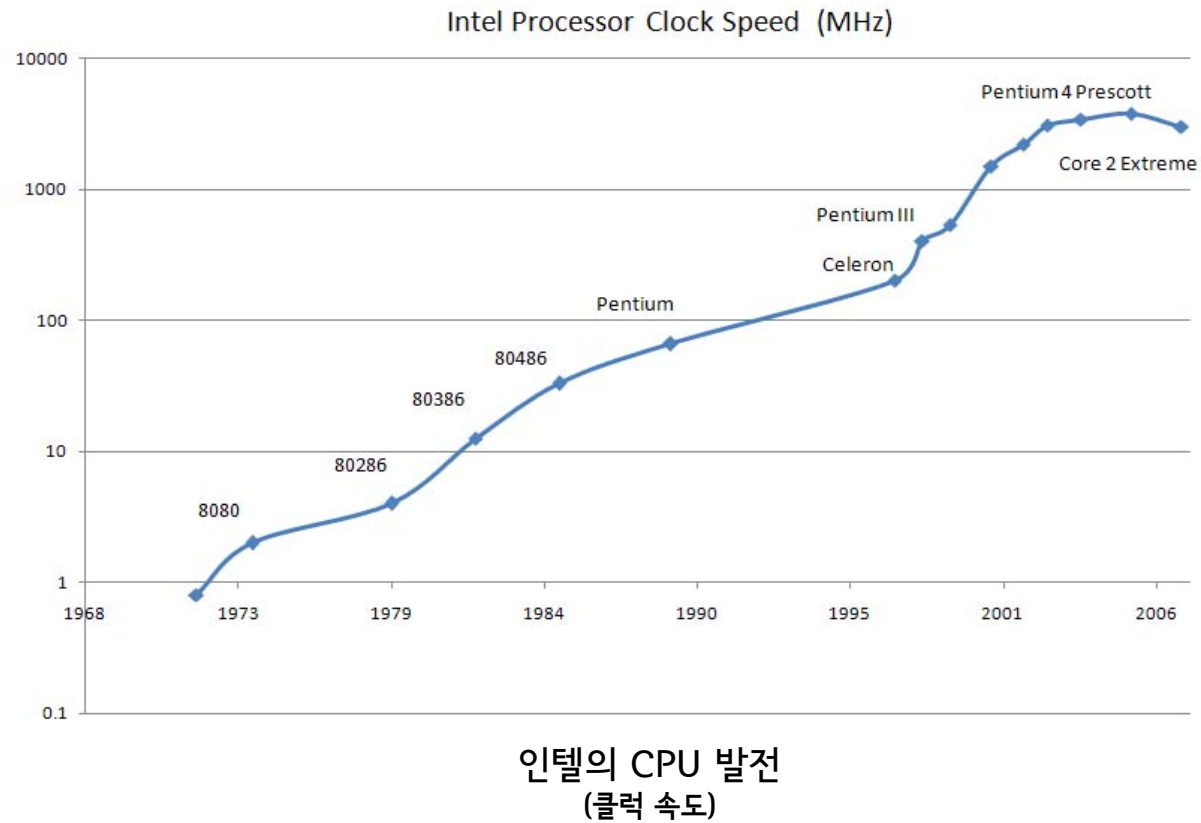


- 2년마다 프로세서 성능이 향상함에 따라 기존의 프로그램을 수정하지 않아도 빨라진 컴퓨터의 혜택을 누릴 수가 있었다.

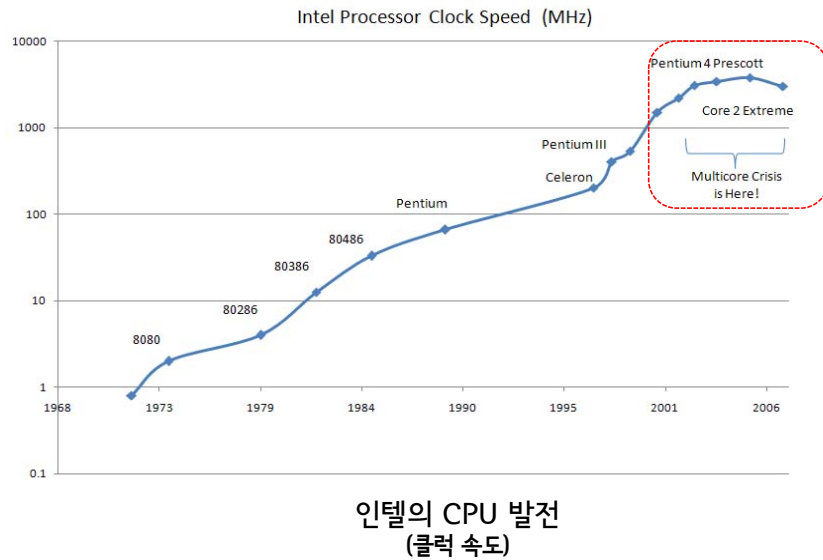
## 그러나

- 2004년 이후 이러한 추세가 느려지고 있다.
  - 프로세서가 싱글 스레드 성능 보다는 듀얼코어, 쿼드코어 같은 멀티 코어로 발전.

# 왜 병렬 프로그래밍은 중요한가?



# 왜 병렬 프로그래밍은 중요한가?



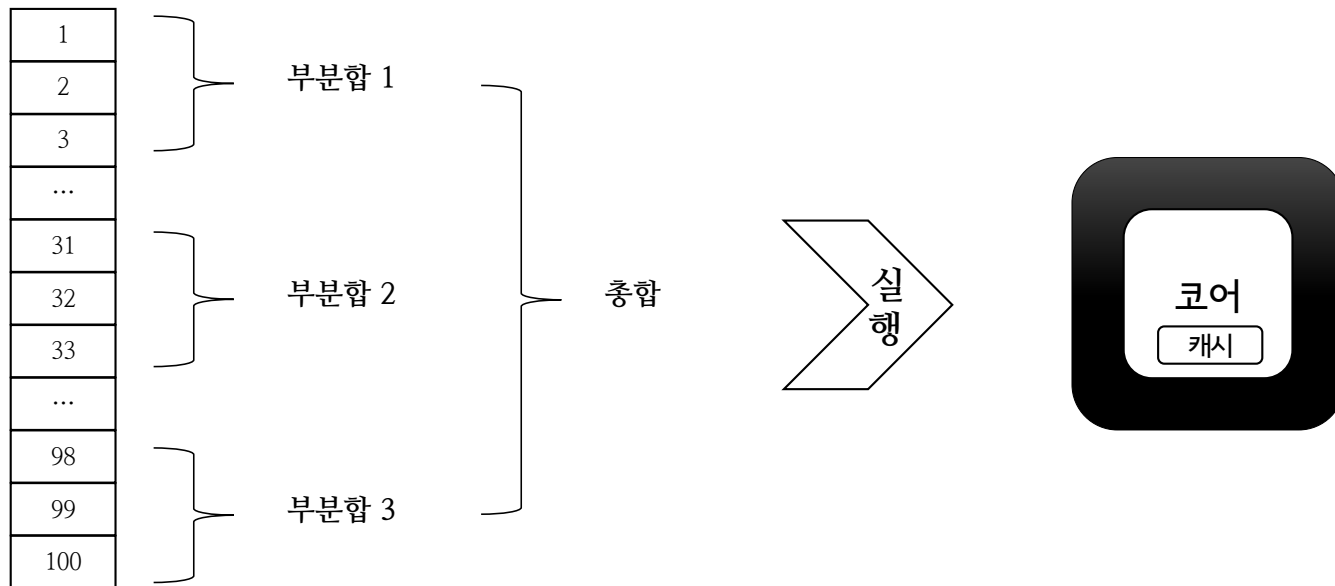
- 2004년 이후로 정체된 CPU 클럭 속도
  - 2006년 Core2 Duo : 2.1GHz ~ 2.67GHz
  - 2008년 Core i7 : 2.67Ghz ~ 2.93Hz
- 싱글 스레드의 성능 차이는 20%
- 2004년 이전에는 일년에 대략 40~50% 향상
- 2004년 이후에는 일년에 대략 9.5% 향상
- The free lunch is over.

## 그렇다면 병렬 프로그래밍이란?



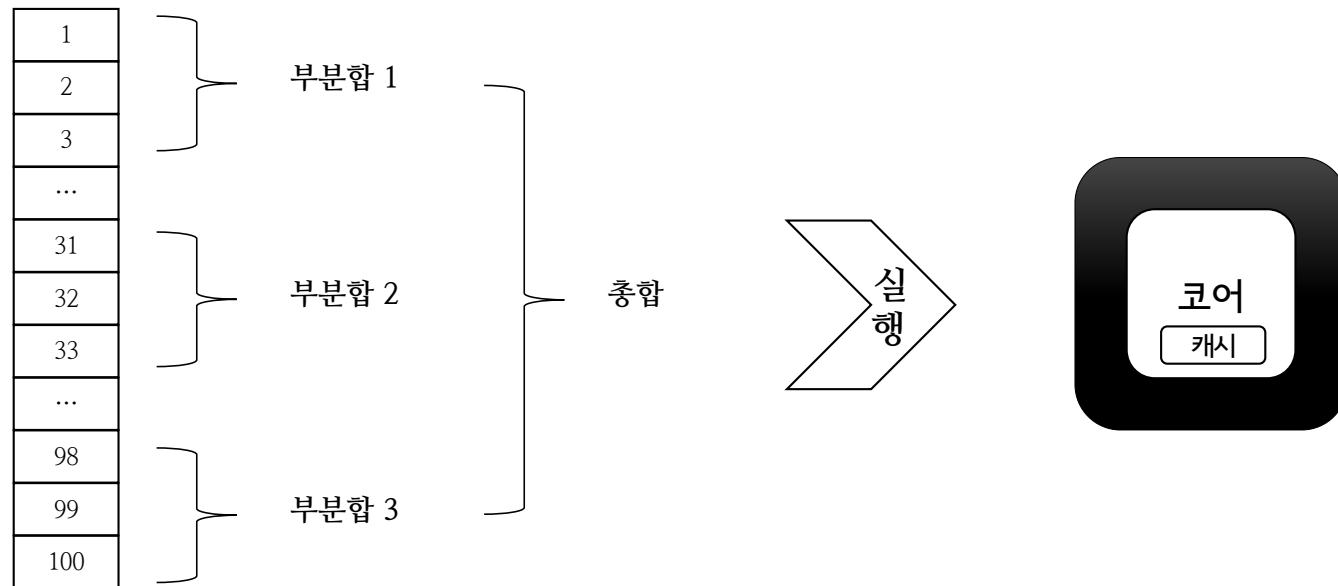


## 그렇다면 병렬 프로그래밍이란?



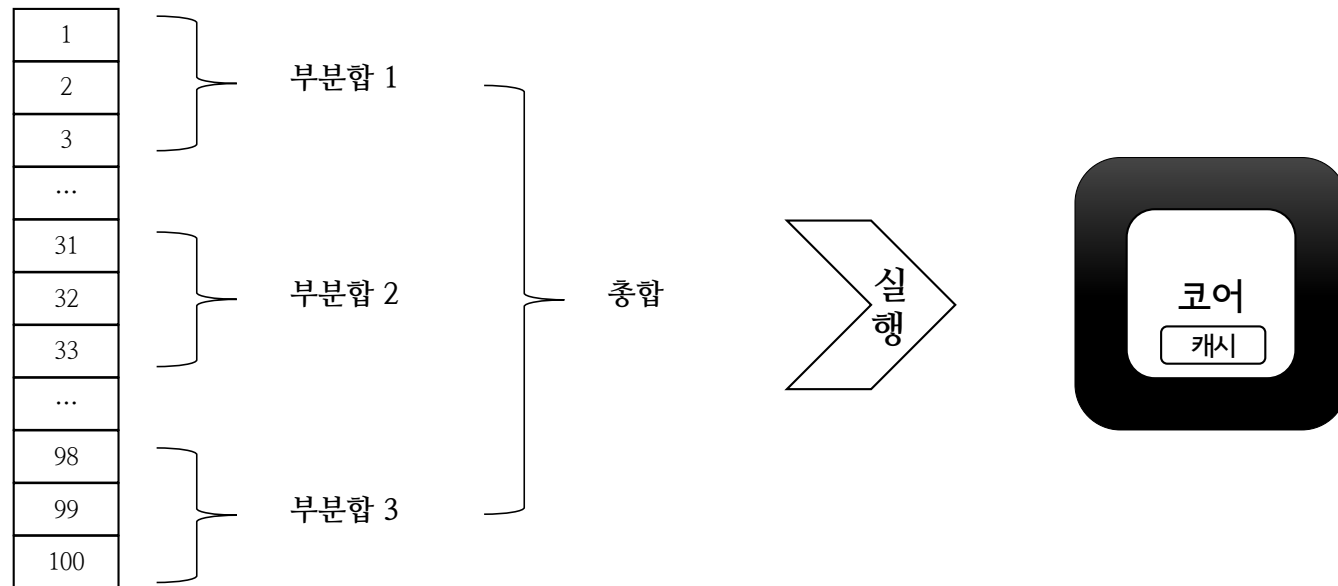
## 그렇다면 병렬 프로그래밍이란?

멀티스레드로 작성된 프로그램을 싱글 코어 프로세서에서 동작 시켰을 때, 병렬 프로그래밍이라 말할 수 있는가?



## 그렇다면 병렬 프로그래밍이란?

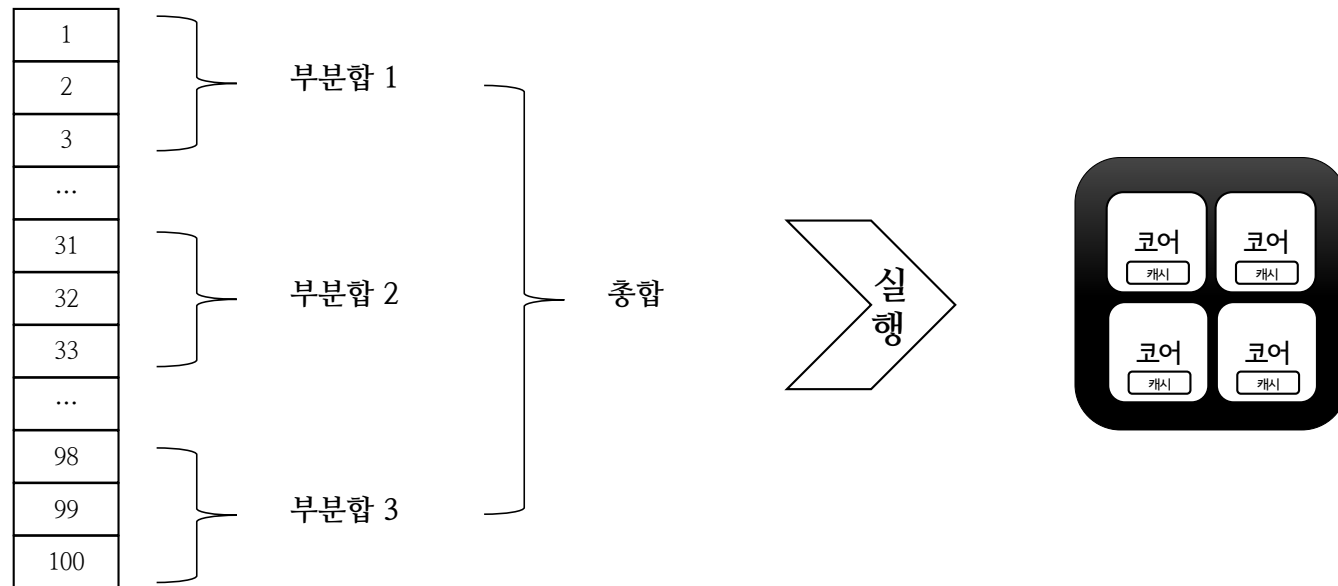
멀티스레드로 작성된 프로그램을 싱글 코어 프로세서에서 동작시켰을 때, 병렬 프로그래밍이라 말할 수 있는가?



Concurrency 는 프로그램의 성질이고, Parallel execution 은 기계의 성질이다.

## 그렇다면 병렬 프로그래밍이란?

병렬 프로그래밍이란 Concurrent 알고리즘으로 작성된 프로그램을 병렬 컴퓨터 에서 동작 시켰을 때 성립한다.



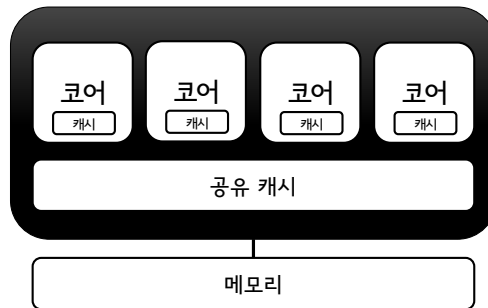
Concurrency 는 프로그램의 성질이고, Parallel execution 은 기계의 성질이다.

# 병렬 컴퓨터 구조

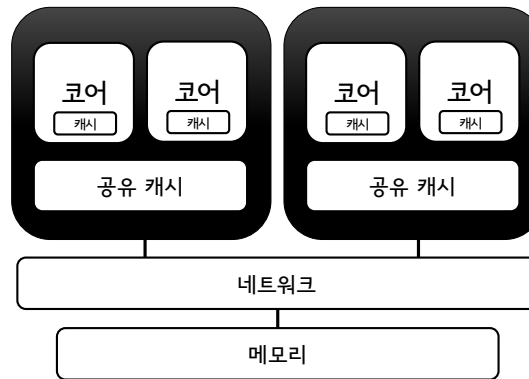
- MIMD(Multiple Instruction Multiple Data)
- 여러 명령어가 여러 데이터를 처리하는 것으로 현재 일반적인 멀티 프로세서 구조. 명령어와 데이터는 메모리에 존재한다.
- 메모리를 어떻게 여러 프로세서가 보느냐에 따라 세분화 된다.
  - 공유 메모리 구조
  - 분산 메모리 구조

## 병렬 컴퓨터 구조 - 공유 메모리 구조

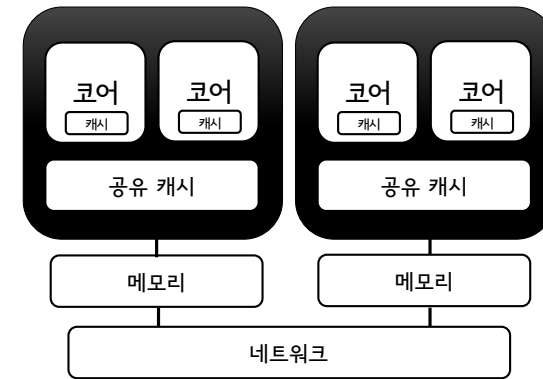
- 프로세서와 코어가 여러 개 있더라도 메모리 주소 공간이 모두 공유된다.



칩 멀티프로세서(CMP) 구조



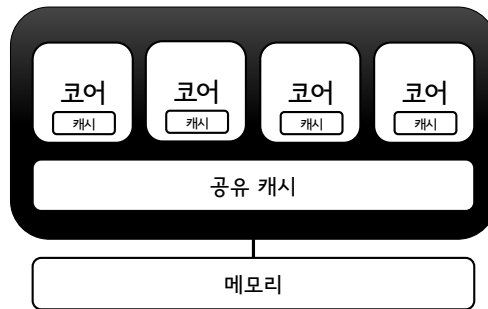
대칭형 멀티프로세서(SMP) 구조



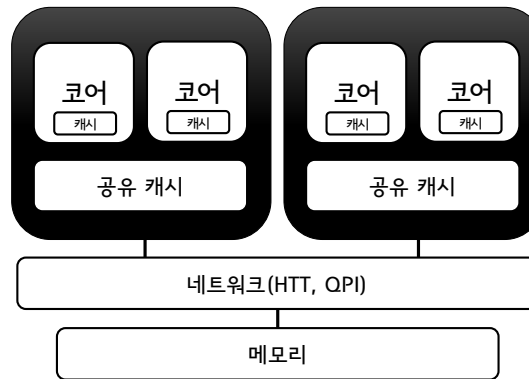
SMP-NUMA 공유메모리 구조

# 병렬 컴퓨터 구조 - 공유 메모리 구조

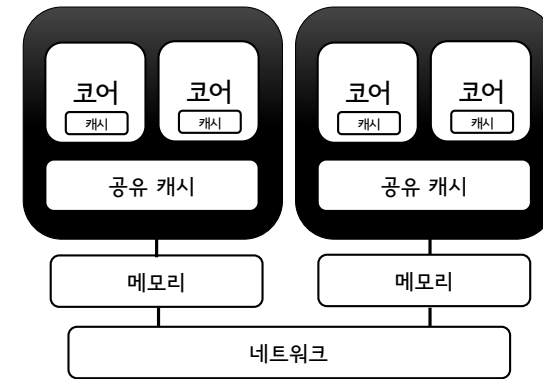
- 프로세서와 코어가 여러 개 있더라도 메모리 주소 공간이 모두 공유된다.



칩 멀티프로세서(CMP) 구조



대칭형 멀티프로세서(SMP) 구조



SMP-NUMA 공유메모리 구조

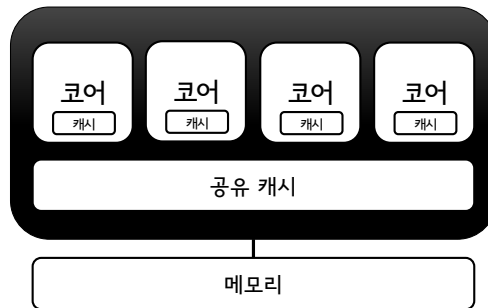
- 칩 하나에 여러 코어가 올라오기 때문에, 자명하게 메모리를 공유하는 것이 합당하다.
- 일반적인 병렬 컴퓨터 구조.
- 캐시를 공유할 수 있다는 장점이 있다.

- 서버 또는 워크스테이션에서 사용하는 멀티 프로세서 구조.
- 과거에는 하나의 프로세서가 하나의 코어였기 때문에 SMP 구조가 일반적이었다.

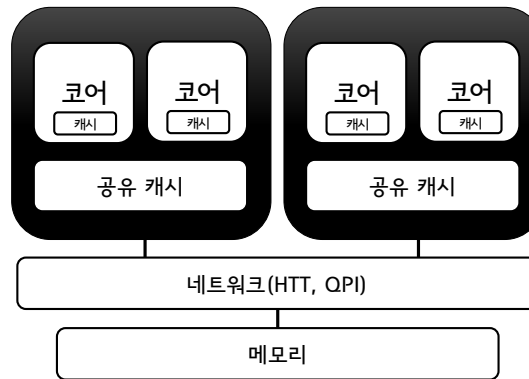
- 시스템 전체적으로는 메모리 주소 공간이 공유되지만, 물리적으로는 다른 곳에 있는 것을 허용한다.
- 한 프로세서와 가까이 있는 물리적인 메모리로의 접근이 다른 메모리보다 빠르다.

# 병렬 컴퓨터 구조 - 공유 메모리 구조

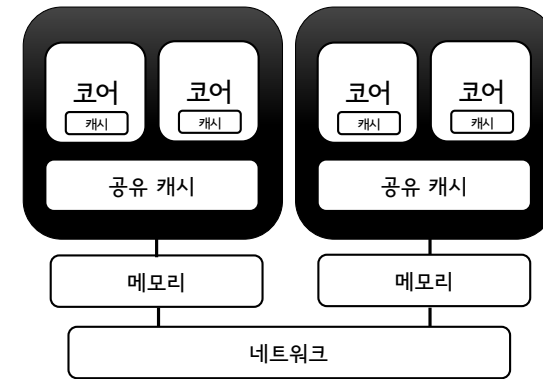
- 프로세서와 코어가 여러 개 있더라도 메모리 주소 공간이 모두 공유된다.



칩 멀티프로세서(CMP) 구조



대칭형 멀티프로세서(SMP) 구조



SMP-NUMA 공유메모리 구조

- 칩 하나에 여러 코어가 올라오기 때문에, 자명하게 메모리를 공유하는 것이 합당하다.
- 일반적인 병렬 컴퓨터 구조.
- 캐시를 공유할 수 있다는 장점이 있다.

- 서버 또는 워크스테이션에서 사용하는 멀티 프로세서 구조.
- 과거에는 하나의 프로세서가 하나의 코어였기 때문에 SMP 구조가 일반적이었다.

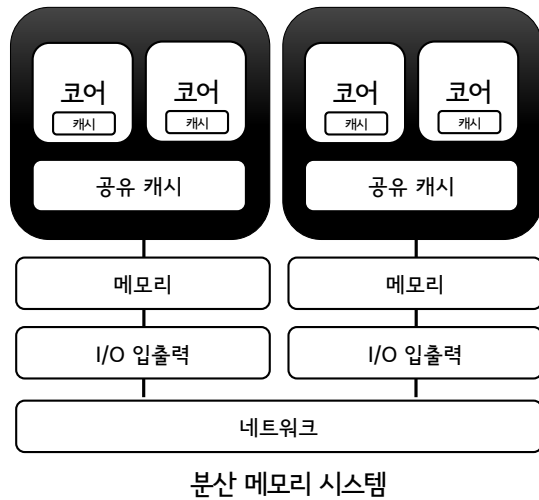
- 시스템 전체적으로는 메모리 주소 공간이 공유되지만, 물리적으로는 다른 곳에 있는 것을 허용한다.
- 한 프로세서와 가까이 있는 물리적인 메모리로의 접근이 다른 메모리보다 빠르다.

기존의 멀티 스레드 프로그래밍 방법론을 적용할 수 있다.  
하지만 캐시, 메모리, 멀티 코어 계층 구조에 따라 최적화가 필요하다.



## 병렬 컴퓨터 구조 - 분산 메모리 구조

- 독립적인 컴퓨터를 고속의 네트워크를 연결한 형태. - 클러스터(cluster), 그리드(grid)



- 각 컴퓨터는 자신의 메모리 내용만 볼 수 있으므로, 다른 컴퓨터(노드)의 내용을 보려면 명시적인 메모리 교환이 필요하다.
- 프로그래밍 방법론에도 큰 차이가 있다.
- 수천, 수만 개의 노드를 연결한 슈퍼컴퓨터가 이러한 구조를 따른다.

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(Amdahl's Law)
  - 컴퓨터 시스템의 일부를 개선할 때 전체적으로 얼마만큼의 최대 성능 향상이 있는지 계산.
  - 프로그램 전체를 **두 배** 빠르게 수행되도록 작성한다면 실제 실행도 두 배 빨라진다.

$$\frac{1}{(1 - P) + \frac{P}{S}} = \frac{1}{(1 - 1) + \frac{1}{2}} = 2 \text{ (speed up)}$$

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(Amdahl's Law)
  - 컴퓨터 시스템의 일부를 개선할 때 전체적으로 얼마만큼의 최대 성능 향상이 있는지 계산.
  - 프로그램 전체를 **두 배** 빠르게 수행되도록 작성한다면 실제 실행도 두 배 빨라진다.

$$\frac{1}{(1 - P) + \frac{P}{S}} = \frac{1}{(1 - \textcolor{red}{1}) + \frac{\textcolor{red}{1}}{\textcolor{blue}{2}}} = 2 \text{ (speed up)}$$

- 하지만, 현실적으로 프로그램 전체의 성능을 개선하기는 어렵다. 부분적인 최적화만이 가능하다.
- 프로그램의 50%를 두 배 빠르게 수행되도록 작성한다면 실제 실행은 ?

$$= \frac{1}{(1 - \textcolor{red}{0.5}) + \frac{\textcolor{red}{0.5}}{\textcolor{blue}{2}}} = 1.33 \text{ (speed up)}$$

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(Amdahl's Law)

- 컴퓨터 시스템의 일부를 개선할 때 전체적으로 얼마만큼의 최대 성능 향상이 있는지 계산.
- 프로그램 전체를 **두 배** 빠르게 수행되도록 작성한다면 실제 실행도 두 배 빨라진다.

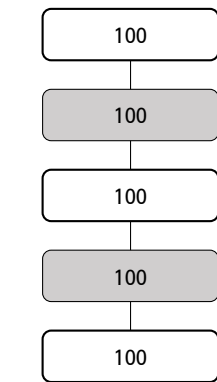
$$\frac{1}{(1 - P) + \frac{P}{S}} = \frac{1}{(1 - \textcolor{red}{1}) + \frac{\textcolor{red}{1}}{\textcolor{blue}{2}}} = 2 \text{ (speed up)}$$

- 하지만, 현실적으로 프로그램 전체의 성능을 개선하기는 어렵다. 부분적인 최적화만이 가능하다.
- 프로그램의 50%를 두 배 빠르게 수행되도록 작성한다면 실제 실행은 ?

$$= \frac{1}{(1 - \textcolor{red}{0.5}) + \frac{\textcolor{red}{0.5}}{\textcolor{blue}{2}}} = 1.33 \text{ (speed up)}$$

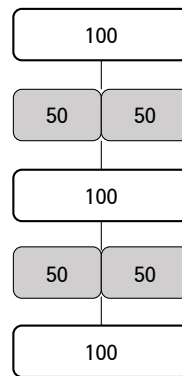
# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(Amdahl's Law)



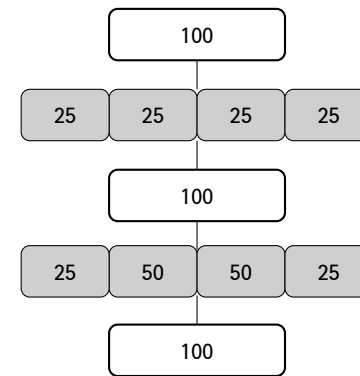
작업량 :	500
시간 :	500
속도증가 :	1배

병렬 처리를 사용하지 않는 원래 프로그램의 상태



작업량 :	500
시간 :	400
속도증가 :	1.25배

병렬 처리 부분을 추가하여 개선

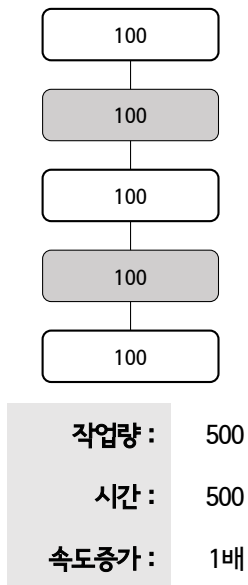


작업량 :	500
시간 :	350
속도증가 :	1.4배

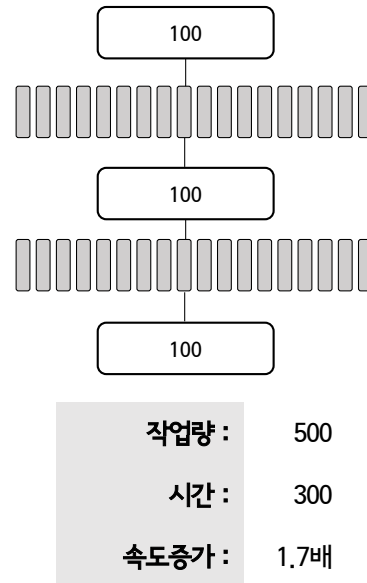
병렬 처리 부분을 추가하여 개선

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(Amdahl's Law)

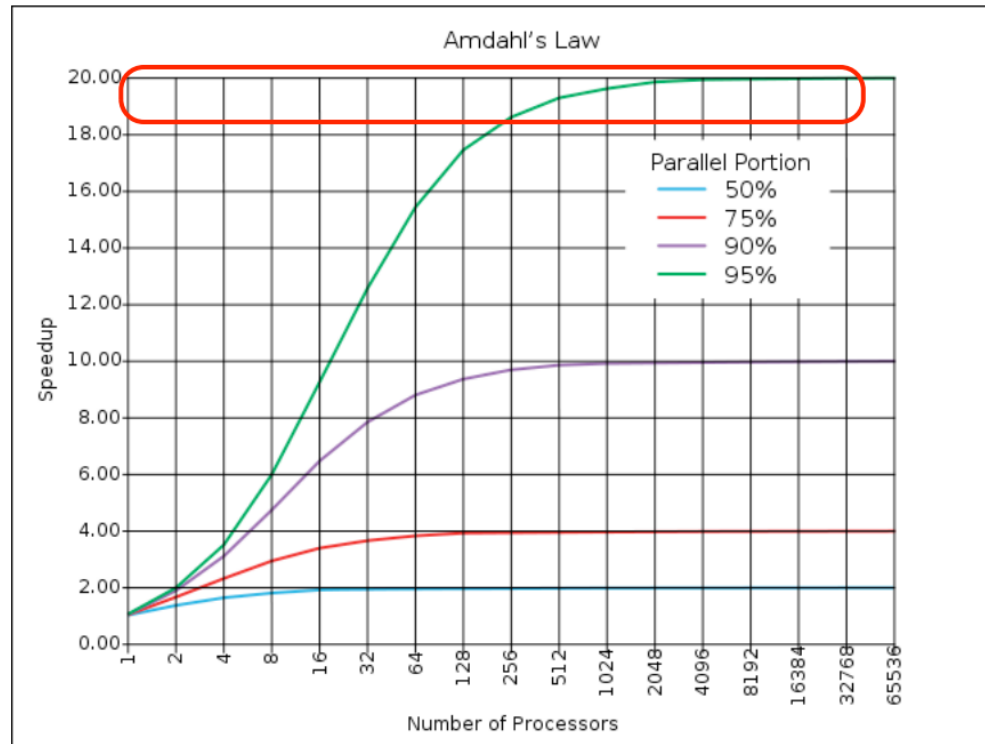


병렬 처리를 사용하지 않는 원래 프로그램의 상태



병렬 처리 부분을 추가하여 개선

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?



- 병렬처리 가능한 부분이 50%인 경우에는 전체 시스템의 성능 향상이 2배에서 수렴.
- 병렬처리 가능한 부분이 95%에 달하는 경우라도 20배 정도에서 성능향상이 수렴.
- 암달의 법칙에 따르면 아무리 코어를 많이 확충한다고 하더라도 시스템의 성능 향상에는 한계가 있다.
  - ‘암달의 저주’

# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 암달의 법칙(저주) 에 대한 구스타프손의 반박
  - “대규모의 반복적인 데이터들을 대상으로 하는 문제들은 효율적으로 병렬화 가능하다“
  - “문제를 충분히 작게 만들어서 현재 사용 가능한 프로세서가 그 문제를 실질적인 시간 안에 풀 수 있도록 하면, 나중에 더 많은 프로세서를 사용해서 더 큰 문제도 같은 시간 내에 해결할 수 있다“
- $n$ 을 문제의 크기라고 할 때 , 병렬화 되지 않은 부분을  $a(n)$ , 병렬화 된 부분을  $b(n)$  이라 할 때, 병렬 환경에서의 프로그램의 실행은 아래와 같이 기술된다.

$$a(n) + b(n) = 1$$

- 위 경우를 완전히 순차적인 환경에서 돌린다고 하면, 상대적인 실행 시간은 다음과 같다.  
( $p$ 는 병렬 환경에서 프로세서의 개수이다)

$$(a(n) + p \cdot b(n))$$

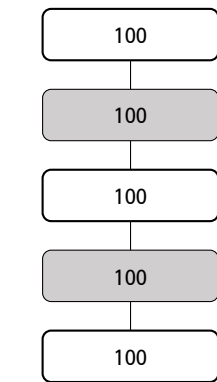
- 그러므로, 성능 개선치(speed up) 는 다음과 같다.

$$S = p - a(n) \cdot (1 - p)$$



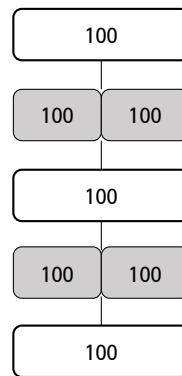
# 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 구스타프슨의 법칙(Gustafson's Law)



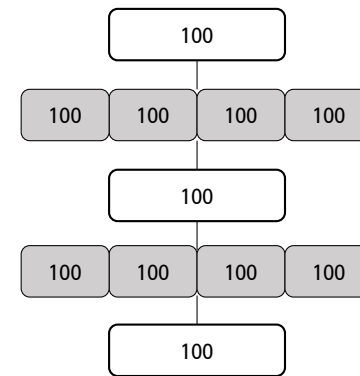
작업량 : 500  
시간 : 500  
속도증가 : 1배

병렬 처리를 사용하지 않는 원래 프로그램의 상태



작업량 : 700  
시간 : 500  
속도증가 : 1.4배

병렬 처리 부분을 추가하여 개선

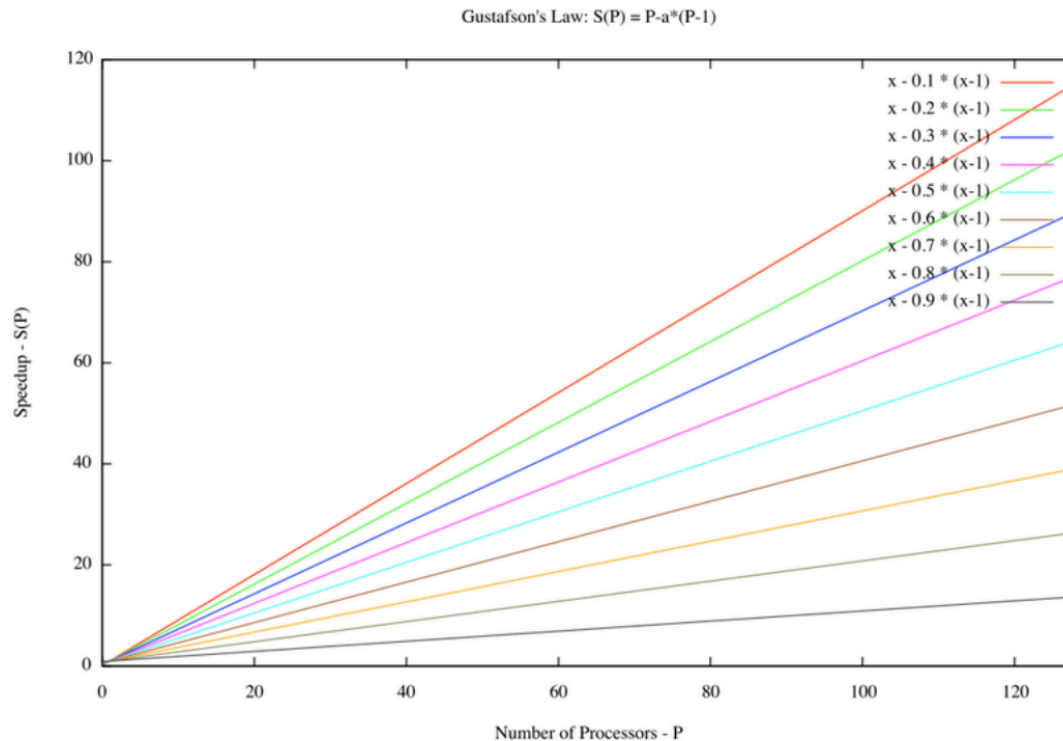


작업량 : 1100  
시간 : 500  
속도증가 : 2.2배

병렬 처리 부분을 추가하여 개선

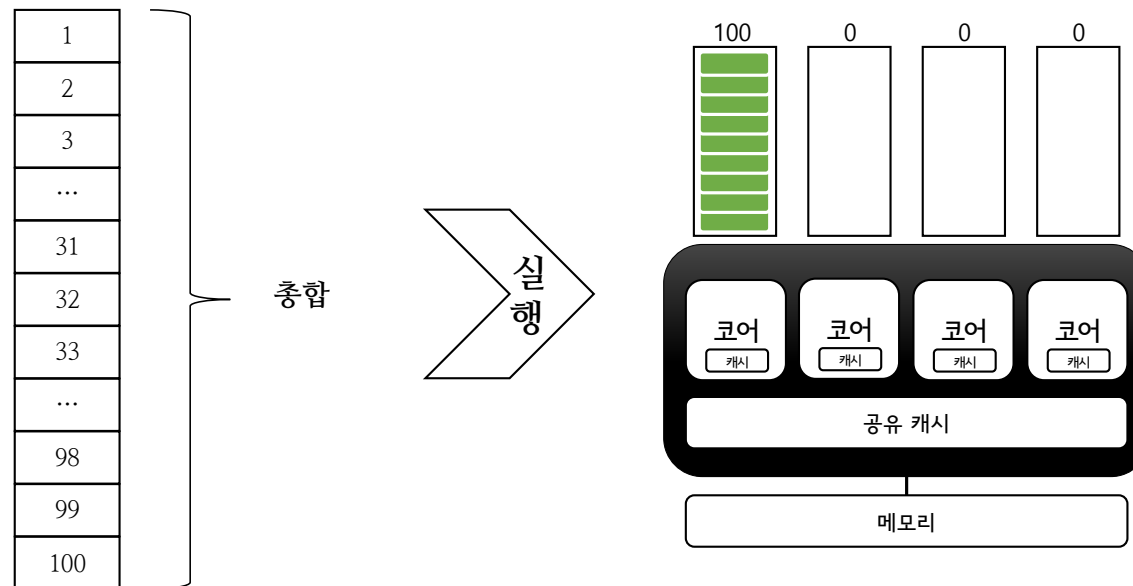
## 병렬 처리시 응용 프로그램은 얼마나 빨라지는가?

- 병렬화율이 10% 인 경우에도 프로세서의 개수가 증가하면 처리 속도가 향상되며, 90% 인 경우에는 거의 선형적인 속도 증가를 예측할 수 있다. 또한 암달의 예측과는 달리 거의 수렴하지 않는다.



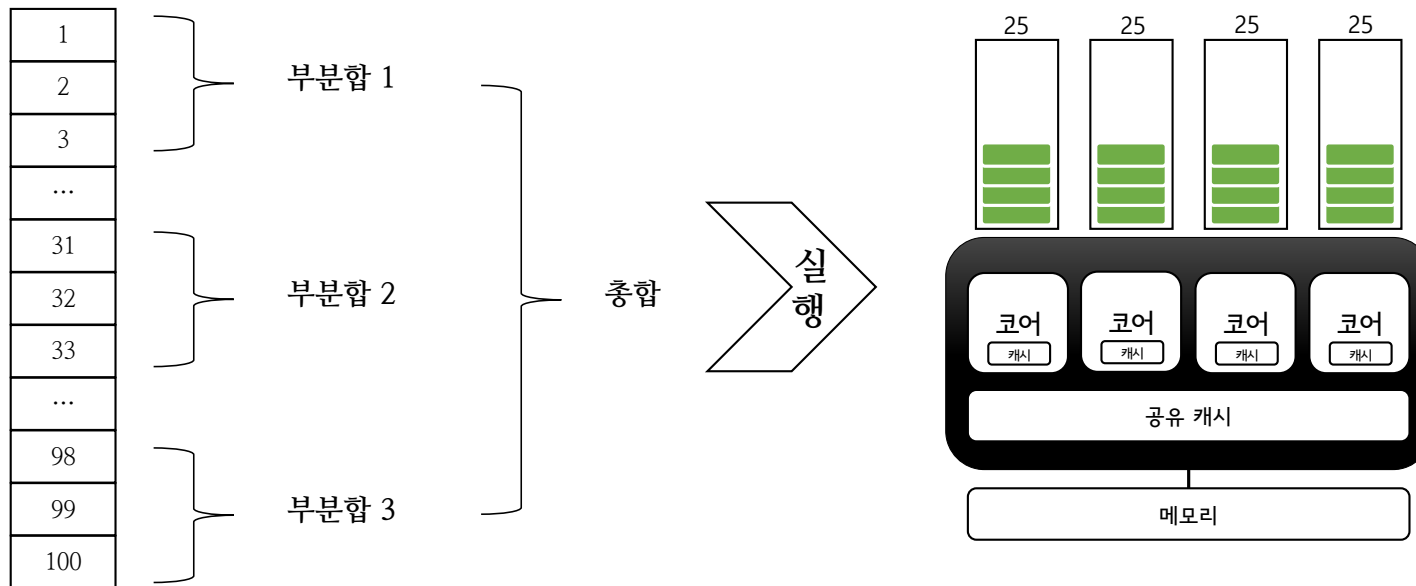
# 병렬 프로그래밍은 왜 어려운가?

- 싱글스레드 프로그램을 병렬 컴퓨터에서 동작 시킨다고 하더라도 성능 향상은 전혀 기대할 수 없다.



# 병렬 프로그래밍은 왜 어려운가?

- 멀티스레드로 작성하면 된다. 하지만 현실은?



# 병렬 프로그래밍은 왜 어려운가? - 현실

- Data Race Condition - 복수의 스레드에서 같은 공유 메모리에 '쓰기'

Single Thread

```
LONG sum = 0;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000; i++)
    {
        sum += 2;
    }

    return 0;
}
```

2 Threads

```
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000 / THREAD_NUM; i++)
    {
        sum += 2;
    }

    return 0;
}
```

```
sum = 100000000
Time : 0.111134 sec
계속하려면 아무 키나 누르십시오 . . .
```

```
sum = 51028082
Time : 0.174820 sec
계속하려면 아무 키나 누르십시오 .
```

# 병렬 프로그래밍은 왜 어려운가? - 현실

- Lock 의 도입.

Single Thread

```
LONG sum = 0;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000; i++)
    {
        sum += 2;
    }

    return 0;
}
```

```
sum = 100000000
Time : 0.111134 sec
계속하려면 아무 키나 누르십시오 . . .
```

2 Threads

```
CRITICAL_SECTION cs;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000 / THREAD_NUM; i++)
    {
        EnterCriticalSection(&cs);
        sum += 2;
        LeaveCriticalSection(&cs);
    }

    return 0;
}
```

```
sum = 100000000
Time : 7.702195 sec
계속하려면 아무 키나 누르십시오 .
```

# 병렬 프로그래밍은 왜 어려운가? - 현실

- Lock 의 도입. - 성능이 매우 나빠진다. : **77** 배의 성능 차이

Single Thread

```
LONG sum = 0;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000; i++)
    {
        sum += 2;
    }

    return 0;
}
```

4 Threads

```
CRITICAL_SECTION cs;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000 / THREAD_NUM; i++)
    {
        EnterCriticalSection(&cs);
        sum += 2;
        LeaveCriticalSection(&cs);
    }

    return 0;
}
```

```
sum = 100000000
Time : 0.111134 sec
계속하려면 아무 키나 누르십시오 . . .
```

```
sum = 100000000
Time : 7.702195 sec
계속하려면 아무 키나 누르십시오 .
```

# 병렬 프로그래밍은 왜 어려운가? - 현실

- Atomic Operation 의 도입 - 여전히 **7배**의 성능 차이.

Single Thread

```
LONG sum = 0;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000; i++)
    {
        sum += 2;
    }

    return 0;
}
```

```
sum = 100000000
Time : 0.111134 sec
계속하려면 아무 키나 누르십시오 . . .
```

4 Threads

```
CRITICAL_SECTION cs;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000 / THREAD_NUM; i++)
    {
        EnterCriticalSection(&cs);
        sum += 2;
        LeaveCriticalSection(&cs);
    }

    return 0;
}
```

```
sum = 100000000
Time : 1.044214 sec
계속하려면 아무 키나 누르십시오 .
```



# 병렬 프로그래밍은 왜 어려운가? - 현실

- 멀티 코어 프로세서에서 제대로 된 성능을 뽑아내기 위해서는 고려할 점이 많다. - 어렵다!!!.

Single Thread

```
LONG sum = 0;
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    for (int i = 0; i < 50000000; i++)
    {
        sum += 2;
    }

    return 0;
}
```

```
C:\Windows
sum = 100000000
Time : 0.111134 sec
계속하려면 아무 키나 누르십시오 . . .
```

4 Threads

```
DWORD WINAPI ThreadFunc(LPVOID arg)
{
    int lsum = 0;
    for (int i = 0; i < 50000000 / THREAD_NUM; i++)
    {
        lsum += 2;
    }

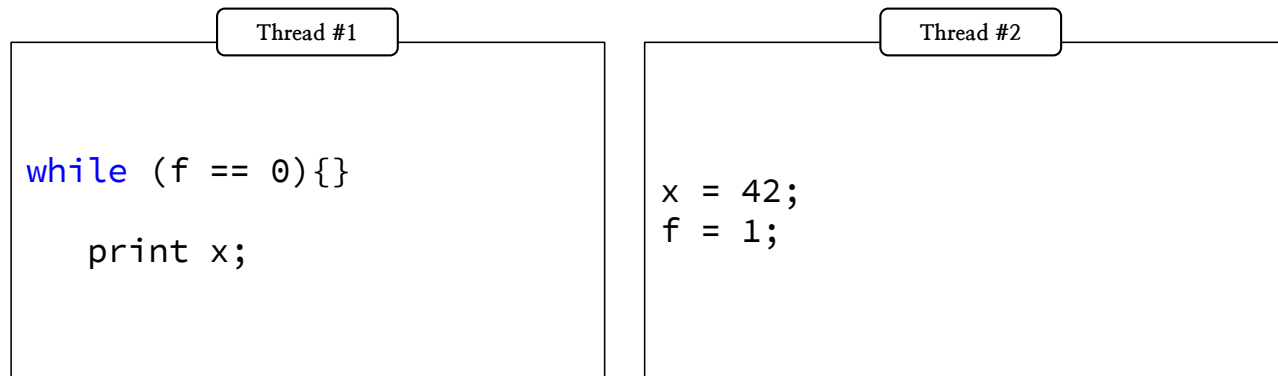
    EnterCriticalSection(&cs);
    sum += lsum;
    LeaveCriticalSection(&cs);

    return 0;
}
```

```
C:\Windows
sum = 100000000
Time : 0.028590 sec
계속하려면 아무 키나 누르십시오 .
```

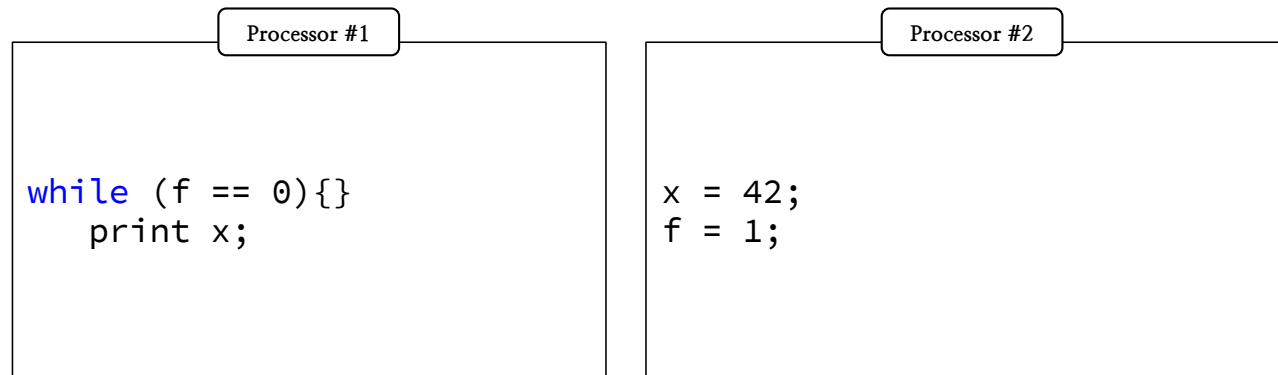
# 병렬 프로그래밍은 왜 어려운가? - 현실

- 다음 코드는 멀티 코어 상에서 문제가 발생할 수 있다.



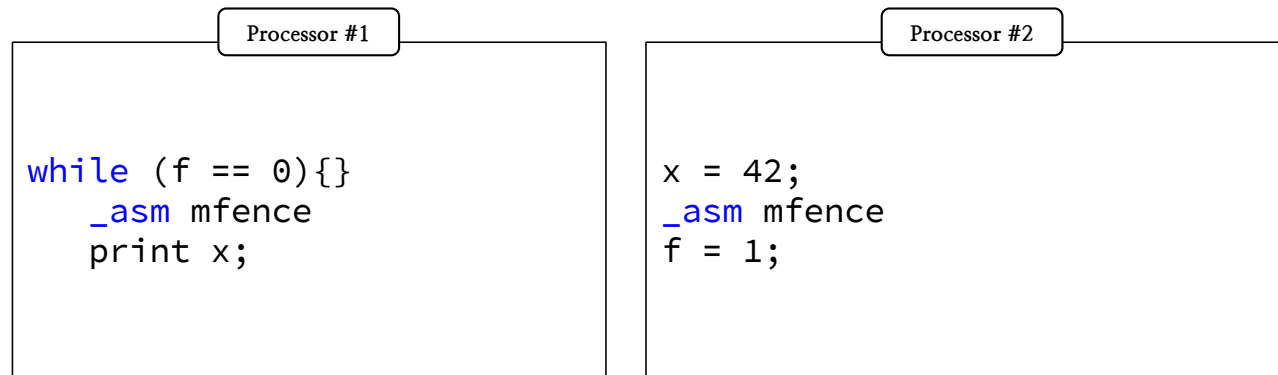
# 병렬 프로그래밍은 왜 어려운가? - 현실

- 다음 코드는 병렬 컴퓨터 상에서 문제가 발생할 수 있다.
  - 왜? 비순차적 명령어 처리(Out-of-order execution, OoOe) 때문
  - ; 고성능 CPU가 특정한 종류의 지연으로 인해 낭비될 수 있는 명령 사이클을 이용하는 패러다임.
  - ; 명령 실행 효율을 높이기 위해 순서에 따라 처리하지 않는 기법.



## 병렬 프로그래밍은 왜 어려운가? - 현실

- 비순차적 실행으로 인한 문제는 '메모리 배리어' 를 통해 해결 가능하다.
- 메모리 배리어 : 중앙 처리 장치나 컴파일러에게 특정 연산의 순서를 강제하도록 하는 기능이다.
- `_asm mfence`
- `__asm__ __volatile__("mfence" ::: "memory");`



# 병렬 프로그래밍은 왜 어려운가?

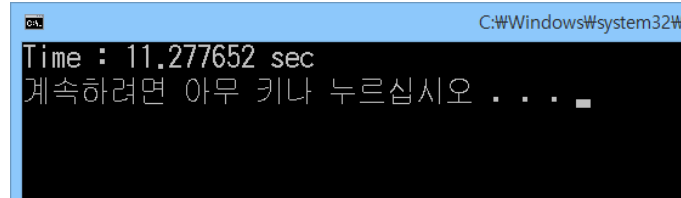
- 공유되는 자원이 아닌 데이터에 합산하는 형태의 멀티스레드 프로그램이다.
- 하지만 이 프로그램에는 **문제점** 이 존재한다.

2 Threads

```
volatile int data1;
volatile int data2;

DWORD WINAPI TestThread1(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 0);
    for (int i = 0; i < 1500000000; i++)
        data1 = data1 + i;
    return data1;
}

DWORD WINAPI TestThread2(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 2);
    for (int i = 0; i < 1500000000; i++)
        data2 = data2 + i;
    return data2;
}
```



C:\Windows\system32\cmd.exe

Time : 11.277652 sec  
계속하려면 아무 키나 누르십시오 . . .

# 병렬 프로그래밍은 왜 어려운가?

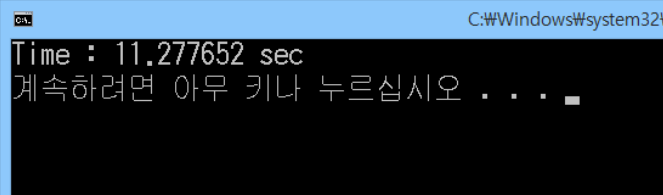
- 프로그램은 정확하게 동작하지만 멀티 코어 상에서는 ‘**성능상의 이슈**’가 발생한다.

2 Threads

```
volatile int data1;
volatile int data2;

DWORD WINAPI TestThread1(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 0);
    for (int i = 0; i < 1500000000; i++)
        data1 = data1 + i;
    return data1;
}

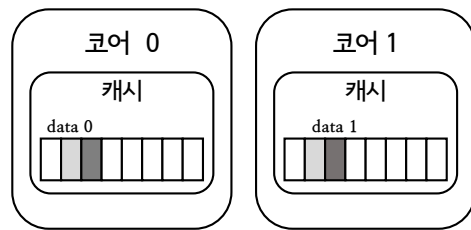
DWORD WINAPI TestThread2(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 2);
    for (int i = 0; i < 1500000000; i++)
        data2 = data2 + i;
    return data2;
}
```



C:\Windows\system32\cmd.exe

Time : 11.277652 sec  
계속하려면 아무 키나 누르십시오 . . .

# 병렬 프로그래밍은 왜 어려운가?



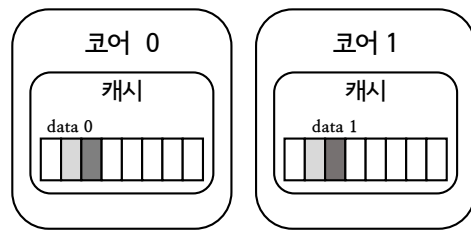
```
volatile int data1;
volatile int data2;

DWORD WINAPI TestThread1(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 0);
    for (int i = 0; i < 1500000000; i++)
        data1 = data1 + i;
    return data1;
}

DWORD WINAPI TestThread2(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 2);
    for (int i = 0; i < 1500000000; i++)
        data2 = data2 + i;
    return data2;
}
```

- 각 코어는 각자 전용 L1 캐시(private cache) 를 가진다.
- 캐시는 읽기/쓰기 의 단위가 4바이트 의 작은 단위가 아니라 64비트 같은 캐시 라인 크기 만큼 한꺼번에 처리된다.
- 연이어 선언된 전역 변수는 컴파일러가 인접한 주소에 두는 것이 일반적이다.
- 따라서 두 변수는 동일한 캐시 라인에 놓일 확률이 매우 높다.

# 병렬 프로그래밍은 왜 어려운가?



2 Threads

```
volatile int data1;
volatile int data2;

DWORD WINAPI TestThread1(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 0);
    for (int i = 0; i < 1500000000; i++)
        data1 = data1 + i;
    return data1;
}

DWORD WINAPI TestThread2(LPVOID arg)
{
    SetThreadAffinityMask(GetCurrentThread(), 1 << 2);
    for (int i = 0; i < 1500000000; i++)
        data2 = data2 + i;
    return data2;
}
```

- 멀티 코어에서 캐시는 사본이 있더라도, 모든 코어가 항상 최신 내용을 볼 수 있도록 처리한다.
- 멀티 코어 캐시 코히런시 프로토콜(MSI, MESI, MOSEI)  
“어떤 코어가 한 캐시 라인을 쓰려면 다른 코어가 가진 사본을 모두 무효화시키는 신호를 보내야만 한다.”
- 실제로 공유되는 데이터가 없음에도, 끊임없이 서로의 캐시 라인을 무효화하고 그에 따라 캐시 미스를 계속 유발하게 된다. ; **가짜 공유 문제**



# 병렬 프로그래밍은 왜 어려운가?

- Visual C++ : `__declspec(align(CACHE_LINE))`
- GCC : `__attribute__((aligned(CACHE_LINE)))`

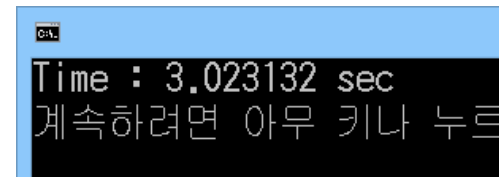
2 Threads

```
#define CACHE_LINE 64
volatile __declspec(align(CACHE_LINE)) int data1;
volatile __declspec(align(CACHE_LINE)) int data2;

DWORD WINAPI TestThread1(LPVOID arg)
{
    volatile struct {
        int data1;
        char padding[CACHE_LINE];
        int data2;
    } DATA;

    DWORD WINAPI TestThread1(LPVOID arg)
    {
        // ...
    }

    DWORD WINAPI TestThread2(LPVOID arg)
    {
        // ...
    }
}
```



Time : 3.023132 sec  
계속하려면 아무 키나 누르

## 병렬 프로그래밍은 왜 어려운가?

- 최신 프로세서는 L3 공유 캐시가 존재함으로 성능 손실이 이전보다 줄었다.
  - Core i7 : 25% 성능 차이.
  - Pentium D : 200% 이상 차이 날 수도 있다.
- 이러한 문제는 전역 변수 뿐 아니라 malloc, new 같은 동적 할당에서도 발생한다.
  - gcc, Visual C++ 기본 힙 할당자는 멀티 스레드 환경에 최적화 되어 있지 않다.
  - 연속해서 두 번 할당하면 인접한 주소를 반환하여 같은 캐시 라인에 놓일 확률이 있다.

```
int main()
{
    printf("memory 1: %d\n", malloc(4));
    printf("memory 2: %d\n", malloc(4));
}
```

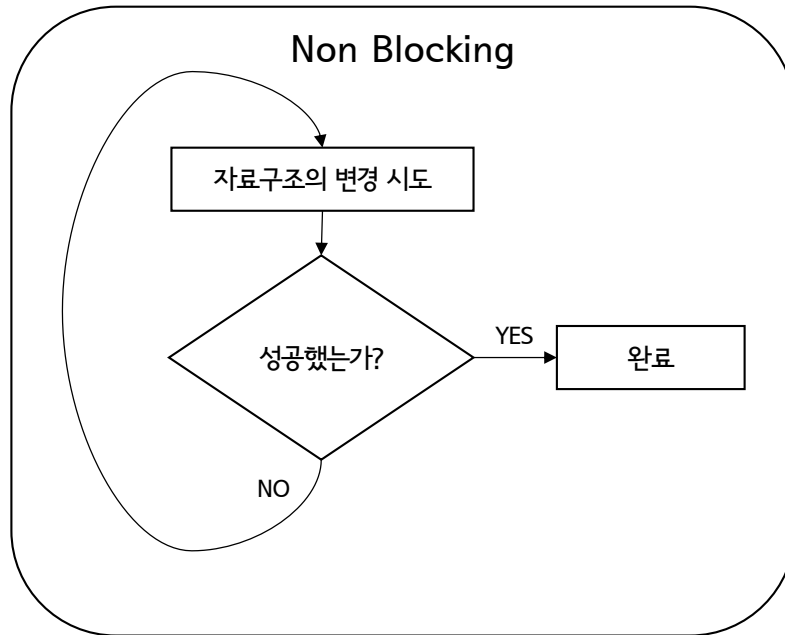
```
mbp:~ ourguide$ ./a.out
memory 1 : 140506797455648
memory 2 : 140506797455664 ] 16 byte
```

- 멀티 스레드 프로그래밍은 버그 없이 돌아가도록 하기도 어렵지만, 최적의 성능을 내면서 돌아가게 하기도 어렵다.

## 병렬 프로그래밍은 왜 어려운가?

- Lock 을 사용하여 문제를 해결하던 기존의 방법은 나쁜 성능을 야기한다.
  - 결국 Lock 을 최소화 하여야 한다.
- 하지만, 스택, 큐 등 여러 개의 스레드에서 접근해야 하는 공유 데이터에 대해서는 락이 빈번하게 호출 될 수 밖에 없다.
  - STL 등 많은 라이브러리에서 제공하는 컨테이너 및 컬렉션은 스레드에 안전하지 않다.
- Lock-free(무잠금) 알고리즘으로 설계된 자료 구조가 필요하다.

# Lock-free 알고리즘



- Non-Blocking 알고리즘
- 자료구조의 변경을 시도
  - 다른 스레드가 이미 변경을 했으면, 실패, 다시 시도.
- 변경 여부를 확인하고, 값을 셋팅하는 연산 자체는 원자적으로 수행되어야 함
  - CAS (Compare And Swap)
  - x86 : CMPXCHG
- ‘잠금’을 사용하는 것보다 복잡한 형태로 구현될 수 밖에 없다.
  - 성긴 동기화
  - 세밀한 동기화
  - 낙천적인 동기화
  - 게으른 동기화
  - 비멈춤 동기화

## 병렬 프로그래밍은 왜 어려운가? - 정리

- 의도하지 않은 결과가 나올 수가 있다.
  - 데이터 레이스
  - 원자성 위반
  - 순서 위반
  - 데드락
- 좋은 성능을 위해서는 고려해야 하는 것이 많다.
  - 프로세서 아키텍처
  - 캐시 성능
  - 무잠금 자료구조
- 기존의 싱글 스레드로 작성한 프로그램은 재작성 되어야만 한다.
  - 병렬화 가능한 부분을 고르는 것이 쉽지 않다.
  - 데이터 의존성을 고려해야 한다.

# Lock-free 알고리즘

- Lock-free 자료구조는 락이 가지는 확장성(scalability)의 한계를 극복하기 위해 고안되었다.
  - 멀티 스레드 프로그래밍을 병렬 컴퓨터상의 코어의 개수에 따라 성능이 증가할 것이라 기대하지만, 락을 사용한 다면 실제 성능은 더욱 나빠지는 경우가 일반적이다.
- 하지만 Lock-free 알고리즘은 뮤텝스 처럼 일반화 할 수 없다.
  - 자료구조 타입에 따라 다르게 설계되어야 한다.
  - 하드웨어의 제약이 있다.
    - 한번에 데이터를 atomic 하게 교환할 수 있는 양이 제한적

## 병렬 프로그래밍 미래 - Transactional memory

- Lock-free 의 대안으로 1993년 Herlihy 교수가 제안하였다.
- TM은 여러 단계의 작업을 '한번에' 처리하는 것처럼 보이게 한다.
- Q1 에서 값을 뽑아, Q2로 옮기는 작업을 원자적으로 하는 코드

```
void TransferValue(FIFO Q1, FIFO Q2) {  
    Q1.lock.acquire();  
    Q2.lock.acquire();  
    v = Q1.dequeue();  
    Q2.enqueue(v);  
    Q2.lock.release();  
    Q1.lock.release();  
}
```

## 병렬 프로그래밍 미래 - Transactional memory

- Lock-free 의 대안으로 1993년 Herlihy 교수가 제안하였다.
- TM은 여러 단계의 작업을 '한번에' 처리하는 것처럼 보이게 한다.
- Q1 에서 값을 뽑아, Q2로 옮기는 작업을 원자적으로 하는 코드

```
void TransferValue(FIFO Q1, FIFO Q2) {  
    Q1.lock.acquire();  
    Q2.lock.acquire();  
    v = Q1.dequeue();  
    Q2.enqueue(v);  
    Q2.lock.release();  
    Q1.lock.release();  
}
```

```
void TransferValue(FIFO Q1, FIFO Q2) {  
    atomic {  
        v = Q1.dequeue();  
        Q2.enqueue();  
    }  
}
```

Transactional memory



## 병렬 프로그래밍 미래 - Transactional memory

- 락으로 보호되는 코드는 스레드가 하나만 있다고 하더라도 항상 락을 잡고 놓는 과정이 필요하다.
  - 항상 '최악의 경우' 만을 가정.
- TM은 메모리 변화를 잠시 버퍼링하였다가, TM이 끝나는 부분에서 한방에 변화.
  - 추가적인 경쟁이 없다면 오버헤드가 거의 발생하지 않는다.

```
void TransferValue(FIFO Q1, FIFO Q2) {  
    Q1.lock.acquire();  
    Q2.lock.acquire();  
    v = Q1.dequeue();  
    Q2.enqueue(v);  
    Q2.lock.release();  
    Q1.lock.release();  
}
```

```
void TransferValue(FIFO Q1, FIFO Q2) {  
    atomic {  
        v = Q1.dequeue();  
        Q2.enqueue();  
    }  
}
```

Transactional memory

# 병렬 프로그래밍 미래 - Transactional memory

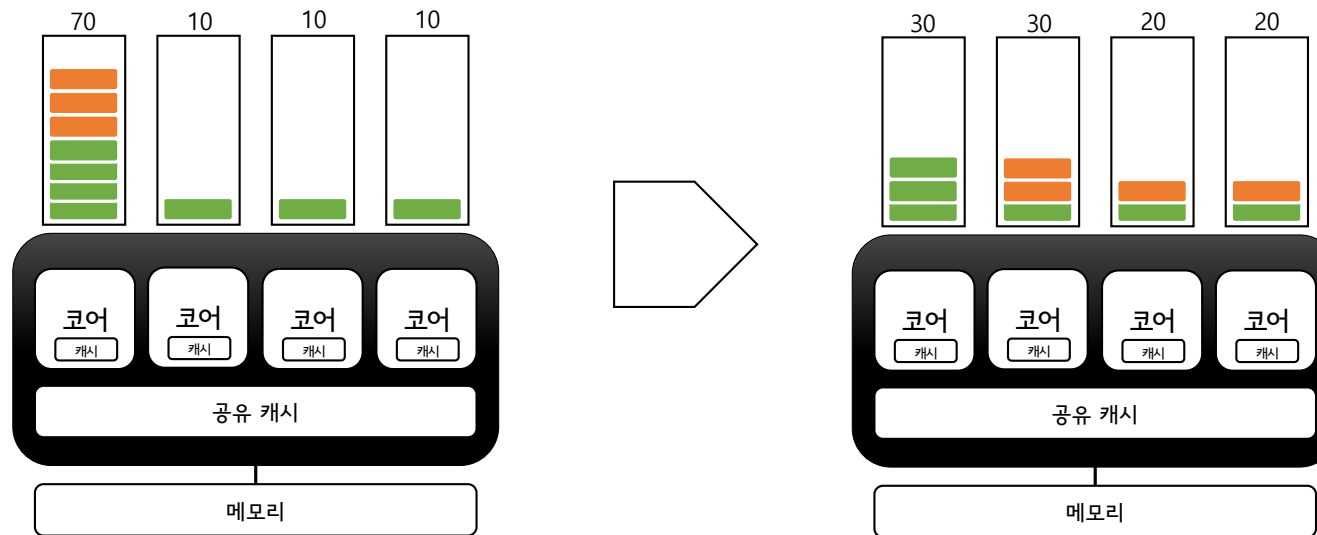
- TM 의 구현 방식
  - STM - Software Transactional Memory
  - HTM - Hardware Transactional Memory
- Intel TSX - Transactional Synchronization Extensions
- 현재 최신 CPU(하스웰) 에서 HTM을 활용 가능하다!!!!

## 병렬 프로그래밍 라이브러리

- Intel Threading Building Blocks - C++
  - Microsoft Parallel Pattern Language - C++
  - Microsoft Task Parallel Library - .Net
  - Java Parallel Package
  - Boost.Lockfree
- 
- **그나마**, 위의 라이브러리를 활용하면 병렬 프로그래밍을 좀 더 편하게 할 수 있다.
  - Lock-free 로 설계된 컬렉션 및 컨테이너를 제공한다.
  - Work stealing 이 구현되어 있다.

## 병렬 프로그래밍 라이브러리 - Work stealing

- 상대적으로 쉬는 스레드가 있으면 바쁘게 일하는 스레드의 일감을 훔쳐와 처리.



## 참고자료

- Concurrent and Parallel Are Not The Same : <http://www.linux-mag.com/id/7411/>
- 암달의 법칙, 구스타프손의 법칙 : [https://docs.google.com/file/d/0B89JlvOZYVauYzBiYzhjYTUtZDI3Ny00YzNlTg5MWUtYmZjYTQ2NjE0Zjkx/edit?hl=en\\_US](https://docs.google.com/file/d/0B89JlvOZYVauYzBiYzhjYTUtZDI3Ny00YzNlTg5MWUtYmZjYTQ2NjE0Zjkx/edit?hl=en_US)
- 프로그래머가 몰랐던 멀티 코어 CPU 이야기 (김민장, 한빛 미디어)
- 인텔 스레딩 빌딩 블록 (제임스 레인더스, 지앤선)
- 멀티 프로세서 프로그래밍 (모리스 힐리히 외 1명, 한빛 미디어)
  
- [인텔 Haswell에 추가되는 Transactional Synchronization Extensions](#)
- [락을 대체할 수 있는 트랜잭셔널 메모리](#)
  
- [http://ko.wikipedia.org/wiki/비순차적\\_명령어\\_처리](http://ko.wikipedia.org/wiki/비순차적_명령어_처리)
- [http://ko.wikipedia.org/wiki/메모리\\_배리어](http://ko.wikipedia.org/wiki/메모리_배리어)