06/03

Thread-Level Parallelism

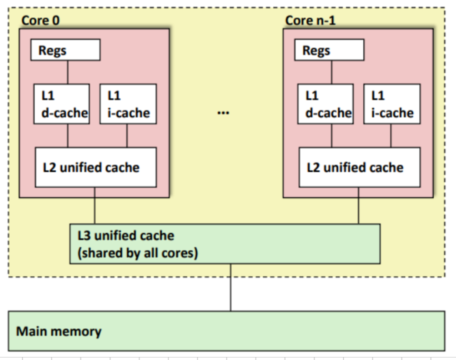
- Parallel Computing Hardware
  → Multicore : Multiple separate processors on single chip
  → Hyperthreading : Efficient execution of multiple threads on single core
- Consistency Models
  → What happens when multiple threads are reading & writing shared state

- Thread-Level Parallelism
  → Splitting program into independent tasks
  ex) parallel summation

Exploiting parallel execution

현재까지 우리는 I/O delay 때문에 thread를 써왔음.
그러나 Multi-core/Hyperthreaded CPU는 다른 기회 제공
↳ Thread 위에 work를 분배함으로써 실제로 parallel execution이 일어나게 할 수 있어. (자동으로 이런 분배 행위가 일어남)
ex) 다양한 APP
↳ Big Task를 Sub Task로 쪼개서 Threads에게 부여

Typical Multicore Processor



→ L1과 L2는 'Private Cache'로
각 Core에 대해 하나씩 존재
→ 반면 L3는 모든 Core가 공유

→ L1 Cache는 Data와 Instruction Cache로 구분
→ L2, L3 Cache는 Unified Cache

ex01) Parallel Summation

0부터 n−1 까지 더하기.

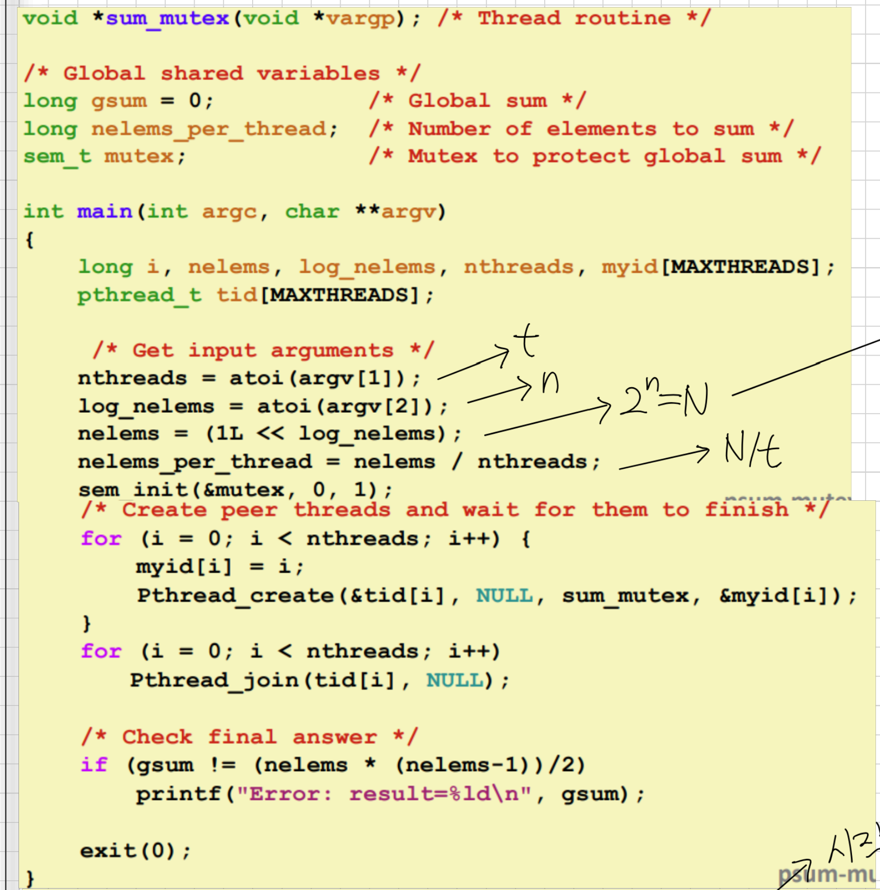
총 합은 뭐겠니 $\frac{n(0+n-1)}{2} = (n-1)n/2$

이것을 t 개의 range로 나눔. ⇒ 각 partition (range)에는 $\lfloor n/t \rfloor$ 만큼 수가 들어있어.
〃 은 thread가 각각 담당
+) 편의상 n이 t의 배수라고 하겠음.

+) 본 교재 machine
↳ 8 cores ⇒ 2x hyper threading, 마치 16개의 core가 있는 수준의 성능

First attempt : psum-mutex

: thread들이 각자의 부분합을 전역변수에 업데이트
+ semaphore를 통한 synchronization



```c
void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;              /* Global sum */
long nelems_per_thread;     /* Number of elements to sum */
sem_t mutex;                /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

    /* Get input arguments */
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

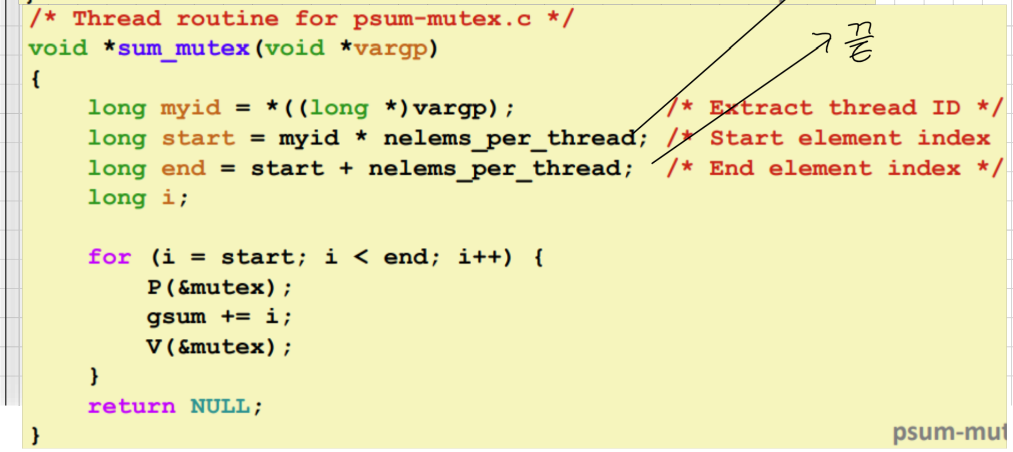
psum-mu

t
n
$2^n = N$
N/t

1L<<N
: 1을 왼쪽으로
n 비트만큼 이동

1L<<0 : 00001
2^0

1L<<1 : 00010
2^1
이런식.



```c
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

psum-mut

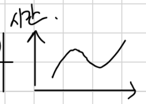
시작
끝

psum-mutex performance

8 cores , $N=2^{31}$

갈수록 느려짐 →

Threads (Cores)	1 (1)	2 (2)	4 (4)	8 (8)	16 (8)
psum-mutex (secs)	51	456	790	536	681

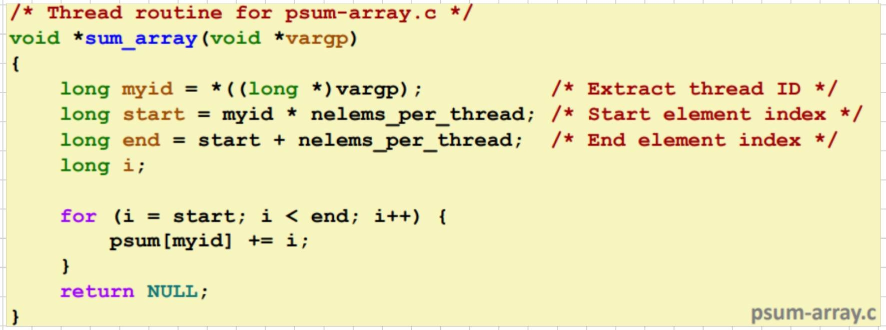


Thread가 늘어났는데 성능은 ↓ , 떠용용

사실 이속도도 굉장히 느린속도 .

Next Attempt : psum-array

: 각 thread들이 계산한 결과를 각각의 배열 index에 저장 (psum[i])



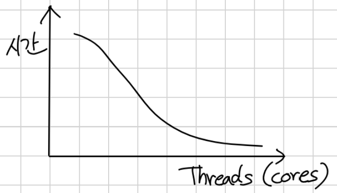
```c
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

psum-array.c

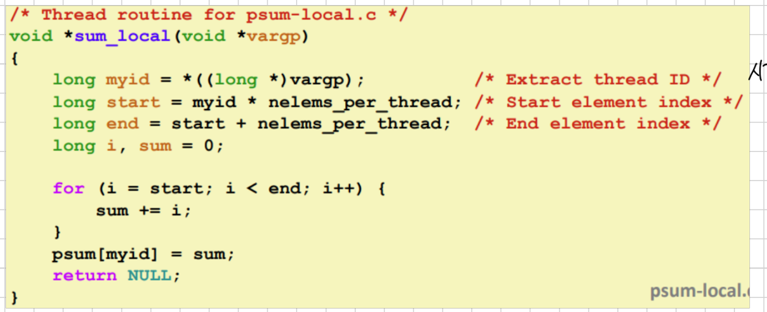
→ mutex synchronization 필요 X

psum-array performance



Next Attempt : psum-local

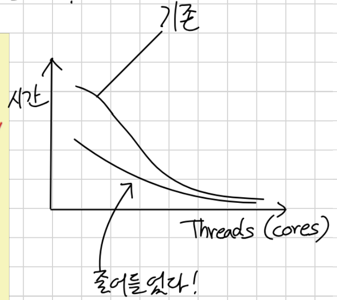
: thread routine 내에 지역변수 사용 ⇒ register를 쓰자!



```c
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

psum-local.c



이전에는 Indexing을 통한 array element에 대한 memory reference가 계속 일어났었는데, 만약 array가 너무 커서 Main memory 또는 cache가 아니라 Disk에 있었다면?
∴ 최종적인 sum 결과만 array에 넣기.

Characterizing Parallel Program Performance

p : process cores 개수 / T_k : k개 Core를 이용해 프로그램을 수행한 running time
(k 개의 Thread 라고 생각해도 됨)

Speedup $S_p = T_1 / T_p$

→ thread (core) 하나로 수행했을 때보다 Thread (Core) 여러 개로 수행하면 성능이 좋은 것
그래서 T_1 을 T_p로 나누어 얼마만큼 속도 향상이 이뤄졌는지 판단.

* Relative Speedup 언제? T_1 이 parallel version of code (1 core)
Absolute Speedup 언제? T_1 이 sequential version of code (1 core)
↓
Absolute speedup을 기준으로 판단 왜? parallelism의 이점을 파악하기 더 간단.

Efficiency $E_p = S_p / p = T_1 / (p T_p)$

→ Speed up을 p로 나눈 값. ☆(p는 core의 개수)
→ parallelism으로 인한 overhead 측정시 유용
→ 0~100 사이의 percentage로 측정

Performance of psum-local

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	8	8
Running time (T_p)	1.98	1.14	0.60	0.32	0.33
Speedup (S_p)	1	1.74	3.3	6.19	6
Efficiency (E_p)	100%	87%	82%	77%	75%

$1.74 \div 2 \times 100$ $3.3 \div 4 \times 100$ $6.19 \div 8 \times 100$ $6 \div 8 \times 100$

Memory Consistency

```
int a = 1;
int b = 100;
```

Thread1:
Wa a=2
Rb print(b)

Thread 2:
Wb b=200
Ra print(a)

결과가 non deterministic ⟶ memory consistency model에 의존

Thread consistency constraints
Wa — Rb
Wb — Ra

← sequential consistency
쓰레드 내부 순서만 지켜지면
쓰레드 끼리 아무렇게나 섞기 가능

↑ Overall effect consistent with each individual thread

↓ arbitrary interleaving

$$\begin{tikzcd}
 & R_b \arrow[r] & W_b \arrow[r] & R_a & 100,\ 2 \\
W_a \arrow[ur] \arrow[r] & W_b \arrow[r] \arrow[dr] & R_b \arrow[r] & R_a & 200,\ 2 \\
 & & R_a \arrow[r] & R_b & 2,\ 200 \\
 & R_a \arrow[r] & W_a \arrow[r] & R_b & 1,\ 200 \\
W_b \arrow[ur] \arrow[r] & W_a \arrow[r] \arrow[dr] & R_a \arrow[r] & R_b & 2,\ 200 \\
 & & R_b \arrow[r] & R_a & 200,\ 2
\end{tikzcd}$$

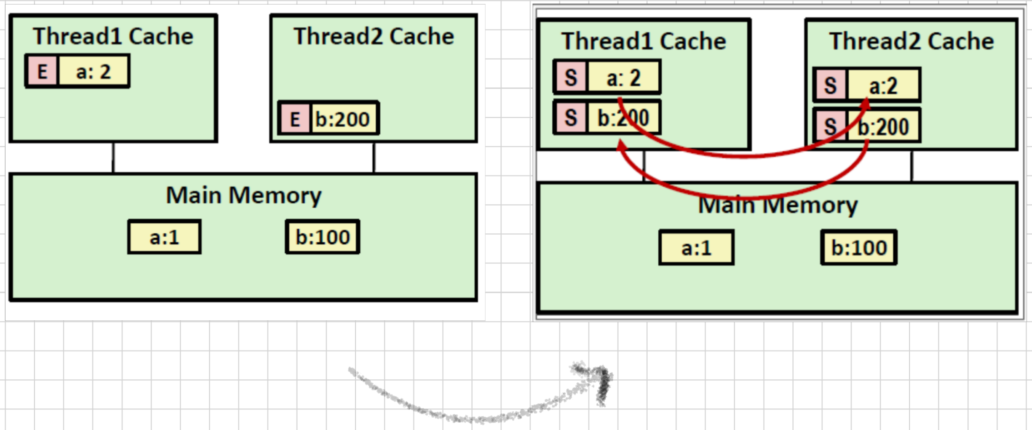
100, 1 / 1, 100 불가능

Non-Coherent Cache Scenario

write back cache는 값이 바뀌어도 DRAM에 다시 쓰지 ×

snoopy caches

Invalid	Cannot use value	: 이 값 못써! 다른곳에서 바뀌었어
Shared	Readable copy	: 읽기만 할수 있어! 다른 캐시들도 같은값 갖고 있어!
Exclusive	Writeable copy	: 나만 가지고 있어! 읽기, 쓰기 모두 가능


Thread1 Cache
E a: 2
Thread2 Cache
E b:200
Main Memory
a:1
b:100
Thread1 Cache
S a: 2
S b:200
Thread2 Cache
S a:2
S b:200
Main Memory
a:1
b:100