Tasking Concept

- Master thread
  - generates tasks (decomposition)
  - ties each task to other thread (mapping)

- Execution ordering은 프로그램에 따라 다르다

OpenMP Tasks

supporting parallelization of irregular problem

(section은 할당되면 tied라서 Idle 발생할 수 있음!)

Irregular problems : exploratory decomposition, recursive algorithm, producer-consumer ...

- exploratory decomposition: unbounded loop

```
while(true){
    ...
    if(...) ~ break;
}
```

- recursive algorithm → quick sort with pivot
- producer-consumer → consumer는 producer가 일 안하면 consume 못함

task : work unit

- code to execute, data environment, ...

Task execution : thread에 어떻게 mapping 되느냐에 따라 다름

- tasks ⇔ threads
- task ⇔ thread
  - → no migration / stealing
  - → 해당 task를 first execute 한 thread에 tied 됨!

(untied로 task간에 work stealing 다 가능!!)

Usage

#pragma omp task [clause list]

* section은 thread에 한번 할당되면 그 thread가 끝까지 tied

* task는 tied 되는 이상 mid of task에서 thread 바뀔수 없음!

(Idling 최소화 + load-balancing)

- Conditional parallelization
  - if (scalar expression)
    - determines whether the construct creates a task
- Binding to threads
  - untied
- Data scoping
  - private (variable list)
    - specifies variables local to the child task
  - firstprivate (variable list)
    - similar to the private
    - private variables are initialized to value in parent task before the directive
  - shared (variable list)
    - specifies that variables are shared with the parent task
  - default (data handling specifier)
    - default data handling specifier may be shared or none

일반 parallel이랑 비슷~함 ...

OpenMP Tasks (Cont.)

task scheduling

tied : task에 tie된 thread 만이 execute 할 수 있다.

- task can only be suspended at suspend point ☆
  - task creation / finish / wait
  - barrier
- task가 barrier 에서도 안 끝나면, descendant 로만 넘겨줄 수 있음

✓ 권장...

untied : no scheduling restrictions

- ✓ can suspend at any point
- ✓ can switch to any task

} → locality / load balance 따라서 결정하면 됨

Example

```
#pragma omp parallel
{
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
    {
      printf("race ");
    }
    #pragma omp task
    {
      printf("car ");
    }
  }
}
```

single: one thread can access

task: task order는 arbitrary

→ A race car

→ A car race

} 둘 다 가능

- task 실행 순서는 arbitrary 이므로 ~~task~~ thread join 하려면 taskwait로 기다려야 함 ?

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n - 1);
#pragma omp task shared(y)
    y = fib(n - 2);
#pragma omp taskwait
    return x + y;
}

int main (int argc, char **argv)
{
    int n, result;
    n = atoi(argv[1]);
#pragma omp parallel
{
#pragma omp single
{
    result = fib(n);
}
}
    printf("fib(%d) = %d\n",
      n, result);
}
```

need shared for x and y;

suspend parent task until children finish ☆

create team of threads to execute tasks

only one thread performs the outermost call

threads N개 있으면

바로 매인 결과가. 내부에는 만들어져도 여기는 no-use

independent task! idle task의 map 될 수 있음

taskwait으로 task 다 queue에서 끝내기를 기다림

Performance Tuning

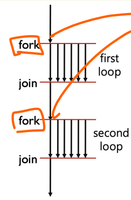
fork 오버 문제

fork with every parallelization — 최악

- Each parallel directive forks threads
- then joins thread after parallel block

```
#pragma omp parallel for
for (i=0; i<n; ++i) {
    …
}

#pragma omp parallel for
for (i=0; i<n; ++i) {
    …
}
```



fork이 항상 좋진 않음!

→ 컴파일러 context 발생으로 instruction ↑

parallelization with forked

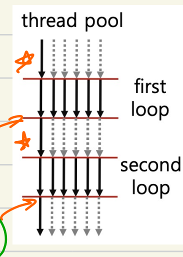
: eliminate unnecessary fork & join

```
#pragma omp parallel num_threads(n)
{
    #pragma omp for
    for (i=0; i<n; ++i) {
        …
    }
    #pragma omp for
    for (i=0; i<n; ++i) {
        …
    }
}
```

paralleli…

n개의 thread forked

n개로 해결!!



implicit barrier

implicit barrier

Performance Tuning (Cont.)

Parallelization at highest level

- for loop에 경우에는 outer-most loop로!

↓

(왜?)

- inner-loop에 parallel 걸면 fork-join 이 많이 생김 (implicit barrier … performance ↓↓)
- finer-grained task … clock 전부다 쓰면 불필요한 context switch 많아짐…

```
!$OMP PARALLEL
....
do j = 1, 20000
!$OMP DO
  do k = 1, 10000
  ...
  enddo !k
!$OMP END DO
enddo !j
...
!$OMP END PARALLEL
```

Slower

```
!$OMP PARALLEL
....
!$OMP DO
do k = 1, 10000
  do j = 1, 20000
  ...
  enddo !j
enddo !k
!$OMP END DO
...
!$OMP END PARALLEL
```

Faster

Merge Independent Loops

- 서로 independent loop는 같이 묶어도 좋다!

↓

(왜?)

- fork-join 줄일 수 있음
- implicit barrier로 인한 불필요한 synchronization 방지 가능

```
!$OMP PARALLEL
....
!$OMP DO
 statement 1
!$OMP END DO
!$OMP DO
 statement 2
!$OMP END DO
....
!$OMP END PARALLEL
```

Slower

```
!$OMP PARALLEL
....
!$OMP DO
 statement 1
 statement 2
!$OMP END DO
....
!$OMP END PARALLEL
```

Faster

Minimize Synchronization

- less Barrier / critical section / ordered
  - use named critical section … 모든 lock 하면, request 너무 많아지면서 ready thread들의 idle time 길어질 수 있음… (several lock 유지 / ready queue 고려…!?)
  - … to more fine-grained locking
- use nowait to eliminate redundant barrier (Asynchronization)
- use explicit flush carefully — evict cached value to memory
  - → Memory mapped I/O 에서는 memory 접근할 때 flush도 필요함. 하지만 reload 된다 …

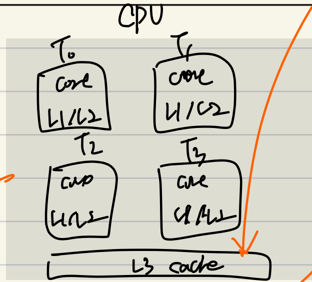
OpenMP Directives - Library-based Model

Directive 장점	· facilitate variety of thread-related tasks · frees programmer from - initializing thread attr. - setting up thread arg. - partitioning iteration spaces, ···
directive 단점	· data exchange is less apparent ★ ··· 실제로 이건 너무 imperative? → control 안해주니까? - data movement, false sharing, contention ↳ cache-line cross value에 대한 write 할 때 발생 (global variables with array) · less expressive than Pthreads

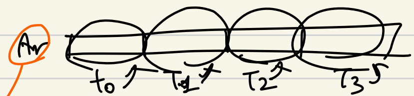
False sharing on same cache line

shared variable은 L3 cache에 위치함!?

Hardware 관점



each thread Access
global variable Arr[k]



adjacent data인 경우 발생!

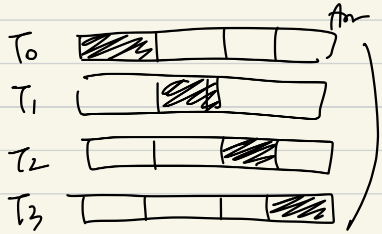
update on local copy
-> invalidate other copy
-> other should update copy to write

무한반복

stored at L3 cache

Cache line = Cache의 접근할때 64byte씩 전부 load / store

나머지



각 thread가 Arr 전부를 갖고있지만 자기 수행한 update (no data race)

data consistent 유지 안됨... need synchronize

각 thread(core) 끼리 computation 중간에 sync. 수행
clock cycle 소비↑ throughput↓

invalidate, Read, write

무한반복!?

Solution

- Padding ... 64byte padding 해서 독립된 cache line 갖도록
- Global variable 지양
- thread 개수↓ / task 크기↑ ... coarse하게 해서 false sharing 피하도록!!!

update 빈도(frequency)↓