4주 : 2010년 3월 26일

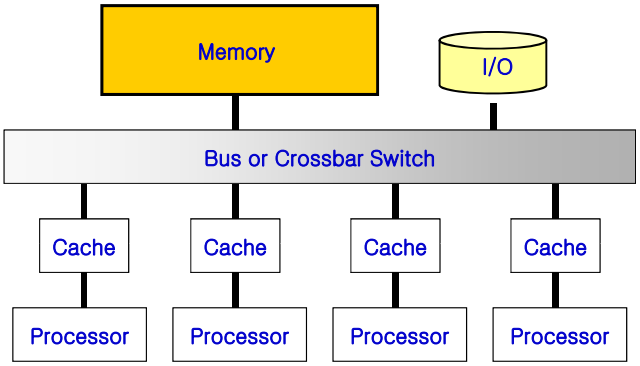
OpenMP 프로그래밍 기초

Hongsuk Yi

KISTI Supercomputing Center

OpenMP란 무엇인가?

- 공유메모리 환경에서 다중 스레드 병렬 프로그램 작성을 위한 응용프로그램 인터페이스(API)
  - Open specifications for Multi Processing



OpenMP의 역사

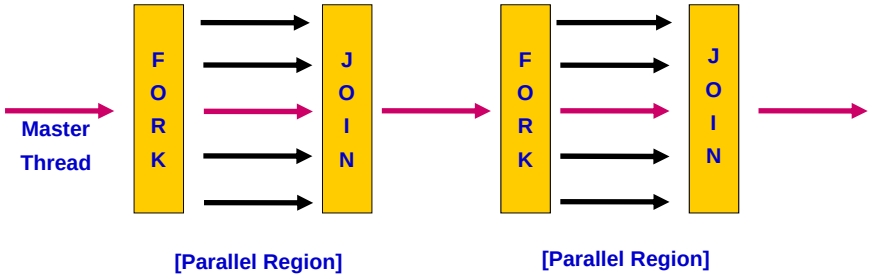
- OpenMP 역사
  - 1990년대 : 고성능 공유 메모리 시스템의 비약적 발전
    - 업체 고유의 지시어 집합 보유 → 표준화 필요
  - 1996년 openmp.org 설립
  - 1997년 OpenMP API 발표
    - C/C++ API 버전 2.0 : 2002년 3월
    - OpenMP 3.0 버전 : 2008년 10월
- OpenMP의 목표
  - 표준과 이식성
    - 간단하고 사용하기 쉬운 API 제공
    - 공유메모리 병렬 프로그래밍의 사실상 표준

OpenMP의 구성

- 컴파일러 지시어
  - 스레드 사이의 작업분담, 통신, 동기화를 담당
  - 좁은 의미의 OpenMP
    - C$OMP PARALLEL DO
- 실행시간 라이브러리
  - 병렬 매개변수(참여 스레드의 개수, 번호 등)의 설정과 조회
    - CALL omp_set_num_threads(128)
- 환경변수
  - 실행 시스템의 병렬 매개변수(스레드 개수 등)를 정의
    - export OMP_NUM_THREADS=8

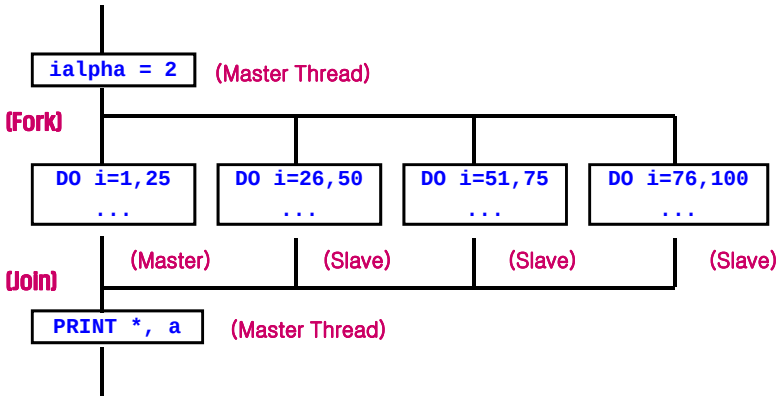
OpenMP 프로그래밍 모델

- Thread-Based Fork-Join 모델
- 컴파일러 지시어 기반 : 순차코드에 지시어 삽입
  - 컴파일러가 지시어를 참고하여 다중 스레드 코드 생성
  - OpenMP를 지원하는 컴파일러 필요
  - 명시적 병렬성 ➔ 동기화, 의존성 제거 등의 병렬화 작업 제어



OpenMP 프로그래밍 개요

순차 코드	병렬 코드
DO i = 1, 100 a(i) = a(i) + ialpha*b(i) ENDDO	!$OMP PARALLEL DO DO i = 1, 100 a(i) = a(i) + ialpha*b(i) ENDDO !$ OMP END PARALLEL DO



OpenMP 프로그래밍 개요

- OpenMP 지시어 및 사용법

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp critical
```

private/shared
default
reduction

OpenMP 프로그래밍 개요

- 실행시간 라이브러리
  - omp_set_num_threads(integer) : 스레드 개수 지정
  - omp_get_num_threads() : 생성된 스레드 개수 리턴
  - omp_get_thread_num() : 스레드 ID 리턴
  - omp_get_max_threads() : 사용 가능한 최대 스레드 개수 리턴
- 환경변수
  - OMP_NUM_THREADS : 사용 가능한 스레드 최대 개수
    - export OMP_NUM_THREADS=16 (ksh)
    - setenv OMP_NUM_THREADS 16 (csh)
- C/C++ : #include <omp.h>

OpenMP 프로그래밍 개요

- OpenMP의 전형적 사용방법 (루프의 병렬화)
  - 시간이 많이 걸리는 루프 찾음 (프로파일링)
  - 의존성, 데이터 유효범위 조사
  - 지시어 삽입으로 병렬화

OpenMP 프로그래밍 개요

- 데이터 유효범위 : private(tid)

C

```
/* hello_right */
#include <omp.h>
main(){
 int i, a, tid;
 #pragma omp parallel private(tid)
 {
   tid = omp_get_thread_num();
   for(i=0; i<10000; i++) a = i;
   printf ("I am %d, tid = %d ₩n",
          omp_get_thread_num(), tid);
 }
}
```

컴파일:
icc -o a.out -openmp a.c
실행:
export OMP_NUM_THREADS=4
./a.out

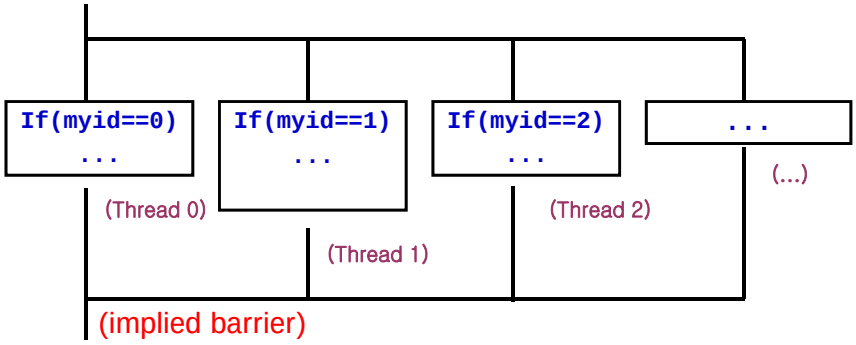
```
I am = 3, tid = 3
I am = 0, tid = 0
I am = 1, tid = 1
I am = 2, tid = 2
```

주요 지시어

- 병렬영역 지시어
  - parallel
- 작업분할 지시어
  - do/for
  - sections
  - single
- 결합된 병렬 작업분할 지시어
  - parallel do/for
  - parallel sections

병렬영역

- 생성 스레드 개수 설정
  - 환경변수(커맨드라인) : export OMP_NUM_THREADS = 32
  - 라이브러리 호출(코드) : CALL omp_set_num_threads(32)
- SPMD 스타일 : 병렬영역을 각 스레드에 복제해 실행
- FORK - JOIN : 병렬영역의 끝에서 동기화



병렬영역

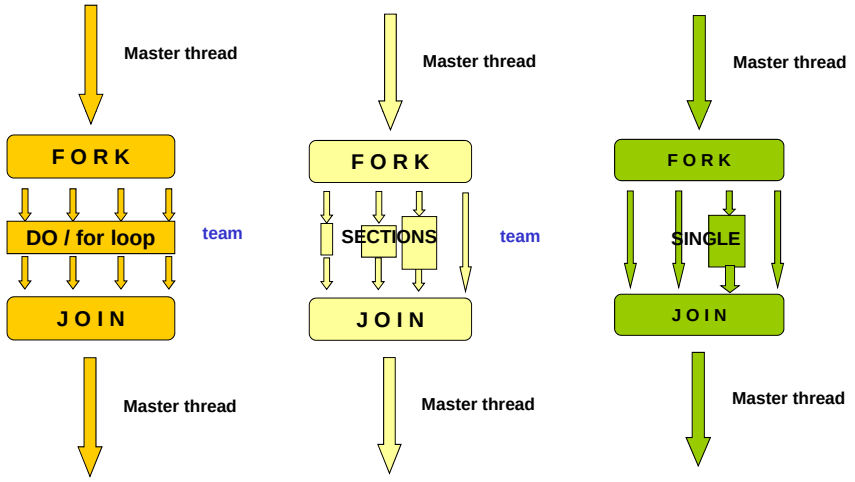
- 스레드 ID 사용 : 0~[# of threads - 1]
  - omp_get_thread_num()

```
#pragma omp parallel private(id,x)
{
    id = omp_get_thread_num();
    sum[id]=0.0;
    for (i=id; i< num_steps; i=i+NUM_THREADS){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}
for(i=0, pi=0.0;i<NUM_THREADS;i++)
        pi += sum[i] * step;
```

작업분할 구문

- 병렬영역 내부에 삽입하여 작업할당에 이용
- 새로운 스레드 생성 없이 기존 스레드에 관련작업 실행 분배
- 구문의 시작점 : 스레드 사이의 동기화 없음
  - 먼저 접근하는 스레드에 우선적 작업 할당
- 구문의 끝점 : 동기화(암시적 장벽)
  - 작업이 먼저 끝나도 다른 작업의 완료까지 대기
  - 대기 없이 다른 작업을 실행하려면 nowait clause 사용

작업분할 구문



do/for sections single

작업분할 구문 : for

- 바로 뒤에 오는 루프의 반복실행을 스레드에 분배
- 동기화 : 루프 끝에 암시적 장벽
  - 기다리지 않으려면 nowait clause 사용
- 사용 clause
  - private(var1, var2, …)
  - shared(var1, var2, …)
  - firstprivate(var1, var2, …)
  - lastprivate(var1, var2, …)
  - reduction(operator|intrinsic:var1, var2,…)
  - schedule(type [,chunk])
  - ordered
  - nowait

for: 예제

```
#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    sum[id] = 0.0;
  #pragma omp for private(x)
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}
for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
```

작업분할 구문 : sections

- 독립된 여러 개 작업을 각 스레드에 분산 실행
  - Functional decomposition
  - 동기화 : sections 구문 끝에 암시적 장벽
  - 기다리지 않으려면 nowait clause 사용

```
#pragma omp parallel private(id)
{
        id = omp_get_thread_num();
        sum[id]=0.0;
#pragma omp sections private(x)
{
#pragma omp section
        for (i=0; i< num_steps/2; i++){
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
#pragma omp section
        for (i= num_steps/2+1; i< num_steps; i++){
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
}
}
```

작업분할 구문 : single

- 병렬영역 내부에서 하나의 스레드만 이용해 작업 수행
- single 지시어에 가장 먼저 접근한 스레드에 작업 할당
- 동기화 : single 구문 끝에 암시적 장벽
- 오직 하나의 section을 가지는 sections 구문과 동일
- 병렬영역 내에서의 입/출력 수행에 주로 사용

결합된 병렬 작업분할 구문

- 병렬영역 안에 for 또는 sections 구문을 하나만 가지는 경우 nowait을 제외한 모든 clause 동일하게 사용 가능
- 동기화 : 병렬구문 마지막에 암시적 장벽 있음

```
#pragma omp parallel for private(id, x)
{
    id = omp_get_thread_num();
    sum[id] = 0.0;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}
for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
```

결합된 병렬 작업분할 구문

- Parallel for 지시어
  - 루프레벨 병렬화 실현 및가장 많이 사용되는 지시어
- Parallel sections 지시어

```
#pragma omp parallel sections private(id, x)
{
#pragma omp section
        id = omp_get_thread_num();
        sum[id]=0.0;
        for (i=0; i< num_steps/2; i++){
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
#pragma omp section
        id = omp_get_thread_num();
        sum[id]=0.0;
        for (i= num_steps/2+1; i< num_steps; i++){
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
}
```

동기화 지시어

- 상호배제 동기화
  - critical
  - atomic
- 이벤트 동기화
  - barriers
  - ordered
  - master

동기화 지시어 : critical

- 병렬영역 안에서 매 순간 오직 하나의 스레드에서 실행
  - 각 section들의 실행순서는 정해져 있지 않음
  - 하나의 section이 실행되면 다른 모든 section들은 대기
  - name이 없는 section들은 동일 section으로 간주

```
#pragma omp parallel private (id, x, sum)
{
        int id;
        id = omp_get_thread_num();
        sum=0.0;
        for (i=id; i< num_steps; i=i+NUM_THREADS)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        {
          pi += sum*step;
        }
}
```

동기화 지시어 : atomic

- 하나의 스칼라 변수를 경신하는 지정된 단일 문장(single statement)에 대해 여러 스레드가 접근하는 것을 방지
- mini-critical section의 역할

동기화 지시어 : barrier

- 모든 스레드들이 barrier에 도달할 때까지 대기
- 작업분할 구문 내에서 사용할 수 없음
- C에서는 structured block에 위치해야 함

```
#pragma omp barrier
```

동기화 지시어 : master

- master section을 마스터 스레드에서만 실행시킴
  - 다른 스레드들은 매스터 세션을건너뛰게 되고 암시적 장벽은 없음
  - 마스터 스레드에 의해 순차적으로 실행되어야 하는 입출력 작업을 위해 많이 사용

```
#pragma omp parallel \
shared(c, scale) private(j, myid){
 myid = omp_get_thread_num();
 #pragma omp master {
  printf("T: %d, Scale =", myid);
  scanf("%d", &scale);
 }
 #pragma omp barrier
 #pragma omp for
 for(j=0;j<N;j++)c(j)=scale*c(j);
}
```

동기화 지시어 : ordered

- ordered section 내부의 루프실행을 순차적으로 진행
  - 하나의 병렬루프에 하나만 허용
  - 병렬루프 지시어는 ordered clause를 가져야 함
  - ordered section을 실행하는 스레드는 매 순간 한 개

```
omp_set_num_threads(4);
#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
#pragma omp for private(i) ordered
  for(i=0;i<8;i++){
    ... compute ...
    #pragma omp ordered
    printf("T:%d,i=%d₩n",myid,i);
  }
}
```

threadprivate : C

```
#include <omp.h>
int x;
#pragma omp threadprivate(x)
main(){
 int tid;
 omp_set_num_threads(4);
 #pragma omp parallel private(tid)
 {
  tid = omp_get_thread_num();
  x = tid*10 + 1;
  printf("T:%d, inside 1st parallel region x=%d₩n",tid,x);
 }
 tid = omp_get_thread_num();
 printf("T:%d, outside parallel region x=%d₩n",tid,x);
 #pragma omp parallel private(tid)
 {
  tid = omp_get_thread_num();
  printf("T:%d, inside 2nd parallel region x=%d₩n",tid,x);
 }
}
```

기본 데이터 유효범위

- 유효범위가 지정되지 않은 변수는 기본적으로 공유됨
  - 서브루틴 내의 병렬영역에서 사용되는 자동 변수는 shared
  - 동적 할당 변수는 shared
- Default Private
  - 루프 인덱스
    - Fortran: 병렬영역의 정적 범위에서 순차 루프 인덱스도 private
    - C/C++: 정적 범위에서 순차루프 인덱스는 기본 shared!
  - 병렬영역에서 호출되는 서브루틴의 지역 변수 (but, save (static) 변수는 shared)
  - C/C++:
    - value parameter
    - 병렬영역의 정적 범위에서 선언된 자동 변수

private

- private(var1, var2, …)
  - 지정된 변수를 스레드끼리 공유하는 것 방지
  - private변수는 병렬영역 내에서만 정의됨
    - 병렬영역 밖에서 초기화 할 수 없음 (→ firstprivate)
    - 병렬영역이 끝나면서 사라짐 (→ lastprivate)
  - private 선언을 고려해야 하는 변수
    - 병렬영역 내에서 값을 할당 받는 변수

clause : shared, default

- shared(var1, var2, …)
  - 지정된 변수를 모든 스레드가 공유하도록 함
- default (private|shared|none)
  - private 또는 shared로 선언되지 않은 변수의 기본적인 유효범위 지정
  - parallel do(for) 구문 :
    - default 선언과 무관하게 루프 인덱스는 항상 private
  - default(none):
    - 모든 변수는 shared 또는 private으로 선언되어야 함
  - C : default(shared|none)

firstprivate and lastprivate

- firstprivate(var1, var2, …)
  - private 변수처럼 각 스레드에 개별적으로 변수 생성
  - 각 스레드 마다 순차영역에서 가져온 값으로 초기화
- lastprivate(var1, var2, …)
  - private 변수처럼 각 스레드에 개별적으로 변수 생성
  - 순차실행에서 마지막계산에 해당되는 값 즉, 마지막 반복실행의 값을 마스터 스레드에게 넘겨줌

reduction

- reduction(operator|intrinsic:var1, var2,…)
  - reduction 변수는 shared
  - 각 스레드에 복제돼 연산에 따라 다른 값으로 초기화 됨
  - 다중 스레드에서 병렬로 수행된 계산결과를 환산해 최종 결과를 마스터 스레드로 내 놓음

```
#pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
```

reduction

- Reduction Operators : C

Operator	Data Types	초기값
+	integer, floating point	0
*	integer, floating point	1
–	integer, floating point	0
&	integer	all bits on
|	integer	0
^	integer	0
&&	integer	1
||	integer	0

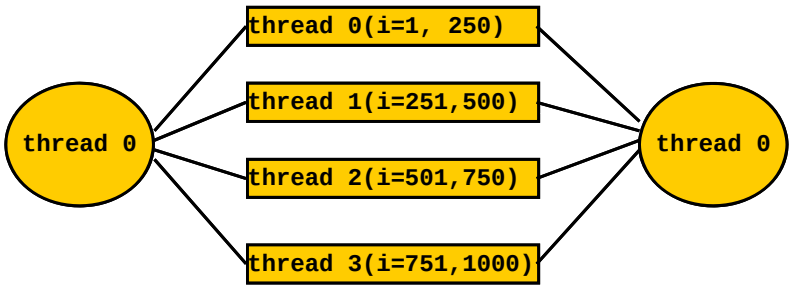
clause : schedule

- schedule(type [,chunk_size])
  - 루프의 schedule : 루프실행의 분배문제
  - 기본적인 schedule 정책 : 실행 회수 균등 분배
  - 작업의 균등 분배를 위해 schedule clause 사용
  - type (default는 implementation depend)
    - static
    - dynamic
    - guided
    - runtime

clause : schedule

- schedule (static)
  - 반복실행이 각 스레드에 균일하게 할당

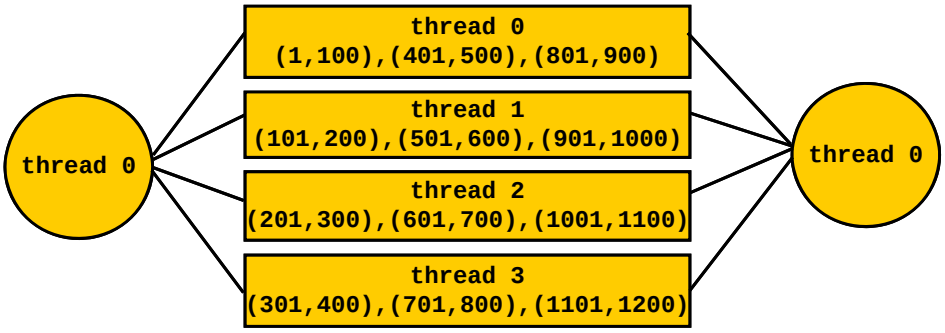
```
!$OMP DO shared(x) private(i) &
!$ schedule(static)
   DO i = 1, 1000
      x(i) = a
   ENDDO
```



clause : schedule

- schedule(static, chunk_size)
  - 총 반복실행 회수를 chunk_size로 나누어 chunk 생성
  - chunk들을 스레드에 라운드-로빈 방식으로 정적 할당

```
!$OMP DO shared(x) private(i) &
!$ schedule(static, 100)
   DO i = 1, 1200
      … work …
   ENDDO
```



clause : schedule

- schedule (dynamic, chunk_size)
  - 총 반복실행 회수를 chunk_size로 나누어 chunk 생성
  - chunk들을 스레드에 동적할당
    - 작업이 먼저 끝나는 스레드에 다음 chunk 할당
  - chunk_size가 없으면 디폴트 chunk_size = 1

clause : schedule

- schedule (guided, chunk_size)
  - dynamic scheduling
  - chunk 크기가 변한다.(N0:반복회수, P:스레드 개수)
    - Nn = MAX(Nn-1-size(chunkn-1),P*chunk_size) (n≥1)
    - size(chunkn) = CEILING(Nn/P)
  - 각 스레드가 DO구문에 도착하는 순서가 다를 때 유용

clause : schedule

- schedule(runtime)
  - 프로그램 실행 중에 환경변수 OMP_SCHEDULE 값을 참조
  - 재 컴파일 없이 다양한 스케줄링 방식 시도 가능

```
export OMP_SCHEDULE="static,1000"
export OMP_SCHEDULE="dynamic"
```

```
#pragma omp parallel
{
#pragma omp for schedule(dynamic,10)
  for(i=0; i<100; i++)
        printf(" I am %d, i = %d ₩n",
omp_get_thread_num(), i);
}
```

ordered

- ordered
  - 루프 내에 ordered 지시어가 나타날 것임을 지적
  - ordered 지시어는 ordered clause와 같이 사용
  - 병렬구문 안의 내용을 인덱스 순서대로 실행하게 함
  - 동기화 지시어 ordered 참조

copyin

- 마스터 스레드의 threadprivate 데이터에 다른 스레드들이 접근 가능하도록 함
  - 스레드들의 threadprivate 변수들을 마스터 스레드의 threadprivate 데이터로 초기화 하는데 사용

환경변수

- OMP_NUM_THREADS
  - 병렬영역에서 사용 가능한 최대 스레드 개수 지정
- OMP_SCHEDULE
  - schedule type이 runtime으로 지정된 루프들에게 scheduling 방식 지정
- OMP_DYNAMIC
  - 스레드 개수의 동적할당 여부 결정
  - TRUE : 병렬영역에서 실제로 사용되는 스레드 수를 최대 개수 범위 내에서 동적할당
- OMP_NESTED
  - nested 병렬성 지원여부 결정
  - OpenMP 표준은 지원, 그러나 대부분 업체는 지원하지 않음
  - 디폴트 : FALSE

실행시간 라이브러리 루틴

- 실행환경 루틴
  - omp_set_num_threads()
  - omp_get_num_threads()
  - omp_get_thread_num()
  - omp_get_max_threads()
  - omp_set_nested()
  - omp_set_dynamic()
  - omp_get_nested()
  - omp_get_dynamic()
  - omp_in_parallel()
  - omp_get_num_procs()
- 잠금(lock) 루틴

omp_set_num_threads

- void omp_set_num_threads(int)
  - 이어지는 병렬영역에서 사용할 스레드 개수 설정
  - 환경변수(OMP_NUM_THREADS) 설정에 우선함
  - 다음 호출이 있을 때까지 스레드 개수는 고정
  - 스레드 할당이 동적이면 사용 가능한 최대 스레드 개수를 나타냄
  - 병렬영역 안에서 호출할 수 없음

omp_get_num_threads

- int omp_get_num_threads(void)
  - 병렬영역 안에서 호출되어 생성된 스레드의 개수를 리턴
  - 순차영역에서 호출하면 1을 리턴

C

```
#include <omp.h>
num_threads = 16
omp_set_num_threads(num_threads);
#pragma omp parallel
{
  printf("# of threads = %d₩n",
        omp_get_num_threads());
}
```

omp_get_thread_num

- Fortran : INTEGER omp_get_thread_num()
- C : int omp_get_thread_num(void)
  - 병렬영역 안에서 생성된 스레드들의 ID를 리턴
  - 0 <= 스레드 ID <= omp_get_num_threads() - 1
  - 순차영역에서 호출하면 0(마스터 스레드)을 리턴

Fortran	C

INTEGER omp_get_thread_num
 	
!$OMP PARALLEL
  PRINT *, 'thread ID = ', &
  omp_get_thread_num()
!$OMP END PARALLEL
	#include <omp.h>
	
	#pragma omp parallel
	{
	  printf("thread ID = %d₩n",
	  omp_get_thread_num());
	}

omp_get_max_threads

- Fortran : INTEGER omp_get_max_threads()
- C : int omp_get_max_threads(void)
  - 병렬영역에서 사용 가능한 최대 스레드 개수 리턴
  - omp_set_dynamic(.TRUE. / 1)
    - omp_get_num_threads <= omp_get_max_threads()
  - omp_set_dynamic(.FALSE. / 0)
    - omp_get_num_threads = omp_get_max_threads()

Fortran	C

INTEGER omp_get_max_threads
!$OMP PARALLEL
  PRINT *, 'max threads = ', &
  omp_get_max_threads()
!$OMP END PARALLEL
	#include <omp.h>
	#pragma omp parallel
	{
	  printf("max threads = %d₩n",
	  omp_get_max_threads());
	}

omp_set_dynamic

- Fortran : CALL omp_set_dynamic(logical)
- C : void omp_set_dynamic(int)
  - 순차영역에서 호출되어 이어지는 병렬영역 들의 스레드 개수 할당을 동적으로 수행
  - 환경변수 OMP_DYNAMIC과 같은 역할
  - 환경변수 설정에 우선

omp_get_dynamic

- Fortran : LOGICAL omp_get_dynamic()
- C : int omp_get_dynamic(void)
  - 스레드 할당 방식 확인
  - 동적이면 .TRUE.(1) 아니면 .FALSE.(0) 리턴

Fortran	C

INTEGER omp_get_max_threads
LOGICAL omp_get_dynamic
PRINT *,'dynamic status =', &
omp_get_dynamic()
PRINT *,'serial:max threads =', &
omp_get_max_threads()
!$OMP PARALLEL
PRINT *,'parallel:max threads &
=', omp_get_max_threads()
!$OMP END PARALLEL
	#include <omp.h>
	printf("dynamic status = %d₩n",
	omp_get_dynamic());
	printf("serial : max threads = %d₩n",
	omp_get_max_threads());
	#pragma omp parallel
	{
	printf("parallel : max threads
	= %d₩n", omp_get_max_threads());
	}

omp_set_nested

- Fortran : CALL omp_set_nested(logical)
- C : void omp_set_nested(int)
  - nested 병렬성 지원 여부 결정
  - OpenMP 표준은 지원, 대부분 업체는 지원하지 않고 있음
  - 환경변수 OMP_NESTED와 같은 역할
  - 환경변수 설정에 우선
  - 디폴트 : FALSE

omp_get_nested

- Fortran : LOGICAL omp_get_nested()
- C : int omp_get_nested(void)
  - 호출되는 시점의 nested 병렬성 설정 여부 확인
  - 사용자가 nested 병렬성 지원을 설정해 주어도
  - 대부분 .FALSE. 또는 0이 리턴
-

Fortran	C

LOGICAL omp_get_nested
 	
CALL omp_set_nested(.TRUE.)
PRINT *, 'nested status = ', &
omp_get_nested()
	#include <omp.h>
	
	omp_set_nested(1);
	printf("nested status = %d₩n",
	omp_get_nested());

omp_in_parallel

- Fortran : LOGICAL omp_in_parallel()
- C : int omp_in_parallel(void)
  - 호출된 지점이 순차영역인지 병렬영역인지 확인
  - 병렬영역이면 .TRUE.(1) 순차영역이면 .FALSE.(0) 리턴

Fortran	C

LOGICAL omp_in_parallel
 	
PRINT *,'parallel region ?', &
omp_in_parallel()
!$OMP PARALLEL
PRINT *,'parallel region ?', &
omp_in_parallel()
!$OMP END PARALLEL
	#include <omp.h>
	
	printf("parallel region? = %d₩n",
	omp_in_parallel());
	#pragma omp parallel
	{
	printf("parallel region?
	= %d₩n", omp_in_parallel());
	}

omp_get_num_procs

- Fortran : INTEGER omp_get_num_procs()
- C : int omp_get_num_procs(void)
  - 프로그램에서 사용 가능한 물리적 프로세서의 총 개수 확인

Fortran	C

INTEGER omp_get_num_procs
PRINT *,'serial : &
# of processors=', & 	
omp_get_num_procs()
!$OMP PARALLEL
PRINT *,'parallel : &
# of processors=', & 	
omp_get_num_procs()
!$OMP END PARALLEL
	#include <omp.h>
	printf("serial : # of processors
	= %d₩n", omp_get_num_procs());
	#pragma omp parallel
	{
	printf("parallel : # of
	processors = %d₩n",
	omp_get_num_procs());
	}

OpenMP 병렬화

- 코드 일부를 병렬영역으로 지정 : parallel/end parallel
- 병렬영역은 다중 스레드에 복제되어 실행 : SPMD 스타일
- 작업분할 구문 또는 스레드 ID에 의한 작업할당
  - do/for (data parallelism)
  - sections (task parallelism)
  - 명시적인 작업 할당: 스레드 ID 사용
- 단일 루프 병렬화: parallel do/for
- 데이터와 지시어 유효범위, 데이터 의존성에 주의

병렬구문 사용의 주의점

- parallel do(for) 는 인덱스 변수에 의해 루프의 실행회수가 제어될 수 있는 루프에만 적용가능
  - 예) while, do-while 문에 적용 불가
- parallel do(for), parallel/end parallel 모두 시작점과 끝점이 하나인 코드 블록을 둘러 쌀 수 있음
  - 루프 실행이 완료되기 전에 exit (Fortran), goto (Fortran/C), break(C) 등을 이용하여 루프 밖으로 빠질 수 없음
  - 루프 내에서 STOP(Fortran), exit(C) 를 이용해 프로그램을 완전히 끝내는 것은 가능

BREAK!!

실습

: Matrix-Matrix Multiplication

순차코드

```
for (i=0; i<N; i++)
    {
      for (j=0; j<N; j++)
        {
          c[i][j]=0;
          for (k=0; k<N; k++)
            {
              c[i][j]=c[i][j]+a[i][k]*b[k][j];
            }
        }
    }
```


Inner loop

```
/* Compute matrix multiplication c=a*b */
  for (i=0; i<N; i++)
    {
      for (j=0; j<N; j++)
        {
          sum=0;
                    #pragma omp parallel for reduction(+:sum)
          for (k=0; k<N; k++)
            {
              sum=sum+a[i][k]*b[k][j];
            }
          c[i][j]=sum;
        }
    }
```

Outer loop

```
/* Compute matrix multiplication c=a*b */
      #pragma omp parallel for private(j,k)
  for (i=0; i<N; i++)
   {
     for (j=0; j<N; j++)
      {
        c[i][j]=0;
        for (k=0; k<N; k++)
         {
           c[i][j]=c[i][j]+a[i][k]*b[k][j];
         }
      }
   }
```


성능측정

- 계산시간
  - Dtime(&t1) : loop 시작 전
  - Dtime(&t2) : loop이 끝난 후
  - 연산수: 2* NRA*NRA*NRA
- 성능 측정
  - Gflops = 연산수/((t2-t1)*1000000000)

숙제

- 1개 Core에서
  - Matrix Size A, B, C
    - NRA=NCA=NRB=NCB=N
    - N을 256, 512, 1024, 2048,4096,8192로 값의 변화에 따른 성능
    - 즉, x축은 N; y축은 Gflops
- OpenMP 병렬 성능
  - NRA=NCA=NRB=NCB=4096로 고정하고
  - Threads를 1, 2, 4, 8, 12, 16로 했을 경우 성능은?
- 주의:
  - SGE를 통한 계산노드에서 실행할 것
  - 결과 값을 표를 만들고 2차원 그래프를 그릴 것

SGE(SUN Grid Engine)

Queue configuration

- 큐 구성

큐이름	Wall Clock Limit (시간)	작업 실행 노드	작업별 CPU수	Priority	SU Charge Rate	비 고
normal	48	Tachyon001-188	17-1536	Normal	1	
Long	168	Tachyon001-188	1-128	Low	1	Long running 작업
Strategry	TBD	Tachyon001-188	32-3008	High	1	Grand Challenge 작업
Special	12	Tachyon001-188	1537-3008	-	2	대규모 자원 전용 (사전예약)

- 사용자별 최대 **Running** 작업수 : **10**개 (작업 부하에 따라 수시로 조정될 수 있음)

- 큐 구성 정보 확인하기

```
$ showq
```

Job submission

- **Job Submission**

```
$  qsub job_script
```

- **Job script examples**

/applic/shell/job_examples/job_script에서 복사하여 사용

Job Script – Serial Program

- **Serial 프로그램(1CPU) 작업 스크립트 작성 예제(serial.sh)**

```
#!/bin/bash
#$ -V                          # 작업 제출 노드의 쉘 환경변수를 컴퓨팅 노드에도 적용 (default)
#$ -cwd                        # 현재 디렉터리를 작업 디렉터리로 사용
#$ -N serial_job               # Job Name, 명시하지 않으면 job_script 이름을 가져옴
#$ -q long                     # Queue name
#$ -R yes   # Resource Reservation
#$ -wd /work01/<user01>/serialtest # 작업 디렉터리를 설정. 현재 디렉토리(PWD)가
                                   # /work01/<user01>/serialtest가 아닌 경우 사용,
                                   # 그렇지 않으면 cwd로 충분함
#$ -l h_rt=01:00:00            # 작업경과시간(hh:mm:ss) (wall clock time), 누락 시 작업 강제 종료
#$ -M myEmailAddress           # 작업 관련 메일을 보낼 사용자 메일 주소
#$ -m e                        # 작업 종료 시에 메일을 보냄
serial.exe
```

Job Script – MPI Program(2)

```
#!/bin/bash
#$ -V                          # 작업 제출 노드의 쉘 환경변수를 컴퓨팅 노드에도 적용 (default)
#$ -cwd                        # 현재 디렉터리를 작업 디렉터리로 사용
#$ -N mvapich_job              # Job Name, 명시하지 않으면 job_script 이름을 가져옴
#$ -pe mpi_fu 32               # selec-bash-mpi에서 선택한 mvapich로 실행되며 각 노드의 가용 cpu를
                               # 모두 채워서(fu : fill_up) 총 32개의 MPI task가 실행됨.
#$ -q normal                   # 큐 이름(17개 이상의 CPU를 사용하는 경우에는 normal 큐를
                               # 16개 이하 CPU를 사용하는 경우 long 큐 사용)
#$ -R yes   # Resource Reservation
#$ -wd /work01/<user01>/mvapich # 작업 디렉터리를 설정. 현재 디렉토리(PWD)가
                                   # /work01/<user01>/mvapich가 아닌 경우 사용,
                                   # 그렇지 않으면 cwd로 충분함
#$ -l h_rt=01:00:00            # 작업 경과 시간 (hh:mm:ss) (wall clock time), 누락 시 강제 작업 종료
#$ -M myEmailAddress           # 작업 관련 메일을 보낼 사용자 메일 주소
#$ -m e                        # 작업 종료 시에 메일을 보냄
mpirun -machinefile $TMPDIR/machines -np $NSLOTS /work01/<user01>/mvapich/mpi.exe
```

Job Script – OpenMp Program

- **OpenMP 프로그램 작업 스크립트 작성 예제(openmp.sh)**

```
#!/bin/bash
#$ -V                          # 작업 제출 노드의 쉘 환경변수를 컴퓨팅 노드에도 적용 (default)
#$ -cwd                        # 현재 디렉터리를 작업 디렉터리로 사용
#$ -N openmp_job               # Job Name, 명시하지 않으면 job_script 이름을 가져옴
#$ -pe openmp 4                # OpenMP thread 수
#$ -q small                    # Queue name(OpenMP 작업은 small or long 큐 사용 가능)
#$ -R yes                      # Resource Reservation
#$ -wd /work02/<user01>/openmp     # 작업 디렉터리를 설정. 현재 디렉토리(PWD)가
                                   # /lustre1/<user01>/openmp가 아닌 경우 사용,
                                   # 그렇지 않으면 cwd로 충분함
#$ -l h_rt=01:00:00            # 작업경과시간 (hh:mm:ss) (wall clock time), 누락 시 작업 강제 종료
#$ -M myEmailAddress           # 작업 관련 메일을 보낼 사용자 메일 주소
#$ -m e                        # 작업 종료 시에 메일을 보냄
export OMP_NUM_THREADS=4
/work02/<user01>/omp.exe
```

Job monitoring (1/2)

1. 기본 작업 정보

```
$ qstat  (사용자 자신)
job-ID  prior   name       user         state    submit/start at       queue                  slots ja-task-ID
-----------------------------------------------------------------------------------
254   0.55500 work6      user1          r     04/02/2008 10:13:09    bmt.q@tachyon087          1
253   0.55500 work5      user1          r     04/01/2008 03:44:20    bmt.q@tachyon035          1
252   0.55500 work7      user1          r     04/01/2008 11:54:34    bmt.q@tachyon035          1

$ qstat –u '*' (모든 사용자)
```

2. 상세 작업 정보

```
$ qstat –f –u "*"
queuename                      qtype used/tot. load_avg arch          states
---------------------------------------------------------------
all.q@davinci02                BIP   1/4       0.14     lx24-amd64
    257 0.55500 sleep      root         r     04/01/2008 10:49:54     1
---------------------------------------------------------------
all.q@davinci03                BIP   1/4       0.13     lx24-amd64
    258 0.55500 sleep      root         r     04/01/2008 10:49:54     1
---------------------------------------------------------
```

Job monitoring (2/2)

1. **Pending 작업에 대한 상세 정보[Pending 이유] 출**

```
$ qstat –j
$ qstat –w v job_id
```

2. **qstat 옵션**

Option	Result
no option	명령을 실행한 사용자 job의 상세 list를 보여줌
-f / -F *[resource_attribute]*	Full output / * qstat –f | grep long Full output and show (selected) resources of queue(s)
-u *user_list*	명시한 user_id에 대한 상태를 보여줌. –u "*"는 전체 사용자의 상태를 보여줌. 주로 –f 옵션과 함께 쓰임.
-r	Job의 resource requirement를 display
-ext	Job의 Extended information을 display
-j <jobid>	Pending/running job에 대한 information을 보여줌
-t	Job의 subtask에 대한 추가 정보 display

Node monitoring

1. 노드 상태 모니터링

```
$ showhost

HOSTNAME        ARCH        NCPU(AVAIL/TOT)     LOAD  MEMTOT MEMUSE SWAPTO  SWAPUS
-----------------------------------------------------------------------------------------
tachyon001      lx24-amd64  0/16                16.03  31.4G  2.7G   0.0    0.0
tachyon002      lx24-amd64  0/16                16.00  31.4G  3.7G   0.0    0.0
tachyon003      lx24-amd64  0/16                16.04  31.4G  3.7G   0.0    0.0
tachyon004      lx24-amd64  0/16                16.03  31.4G  3.7G   0.0    0.0
tachyon005      lx24-amd64  1/16                15.53  31.4G  3.6G   0.0    0.0
tachyon006      lx24-amd64  0/16                16.00  31.4G  3.7G   0.0    0.0
tachyon007      lx24-amd64  0/16                16.01  31.4G  3.7G   0.0    0.0
tachyon008      lx24-amd64  2/16                14.03  31.4G  3.4G   0.0    0.0
tachyon009      lx24-amd64  0/16                16.00  31.4G  3.7G   0.0    0.0
tachyon010      lx24-amd64  0/16                16.03  31.4G  3.7G   0.0    0.0
tachyon011      lx24-amd64  0/16                16.03  31.4G  3.7G   0.0    0.0
tachyon012      lx24-amd64  0/16                16.02  31.4G  3.7G   0.0    0.0
tachyon013      lx24-amd64  0/16                16.03  31.4G  3.7G   0.0    0.0
```

Job control

1. 작업 삭제

```
$ qdel <jobid>          : 해당 <jobid>를 가지는 작업 삭제
$ qdel -u <username>  : <username>의 모든 작업 삭제
```

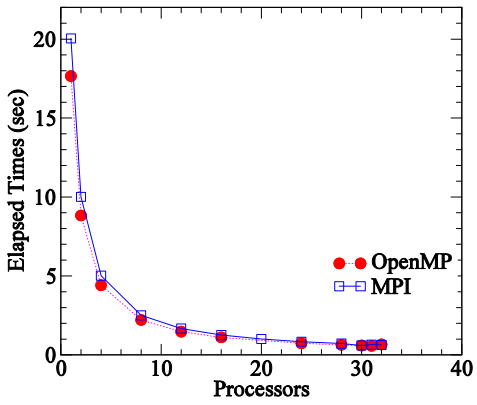
2. 작업 suspend/resume

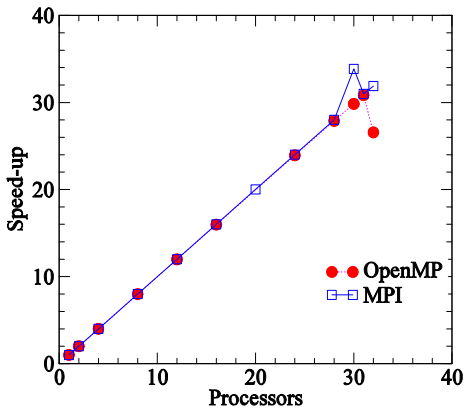
```
$ qmod -sj <jobid>        # suspend job
$ qmod –usj <jobid>       # unsuspend(resume) job
```

Amdahl's Law

- Amdahl's Law : 1967
  - The maximum speedup that can be achieved by that application is:

$$S_{max} = 1 / (1-p)$$





BREAK!!