ROVIN 2010 Robot Vision & Intelligence Lab.

고급프로그래밍

병렬프로그래밍

Professor Jeong, Mun-Ho

Robot Vision & Intelligence Laboratory
Kwangwoon University
(02-940-5625, mhjeong@kw.ac.kr)

Schedule

주차	주제		과제	퀴즈
1	과목소개	교과목 소개 (1), C++ 시작 (2)		
2	C++	C++ 프로그래밍의 기본 (3), **클래스와 객체 (4)**	1	1
3		객체생성과 사용 (5)	2	2
4		함수와 참조 (6, 3/26), 복사 생성자와 함수중복(7)	3	3
5		static, friend, 연산자중복 (8, 4/2), 연산자중복 상속(9)	4	4
6		상속 (10, 4/9), 가상함수와 추상클래스 (11)		5
7		템플릿과 STL (12, 4/16), 입출력(13)	5	
8		**중간고사**		
9		파일 입출력(14), 예외처리 및 C 사용(15)		6
10		람다식(16, 5/7) , 멀티스레딩(17, 5/9)	6	7
11		멀티스레딩(18, 5/14), 고급문법(19, 5/16)		8
12		고급문법 2(20, 5/21), 고급문법 3(21, 5/23)		
13	병렬프로그래밍	병렬프로그래밍(22, 5/28)	7	9
14		병렬프로그래밍		
15		**기말고사**		

오늘의 학습내용

- 병렬프로그래밍

병렬프로그래밍

순차프로그램의 속도향상

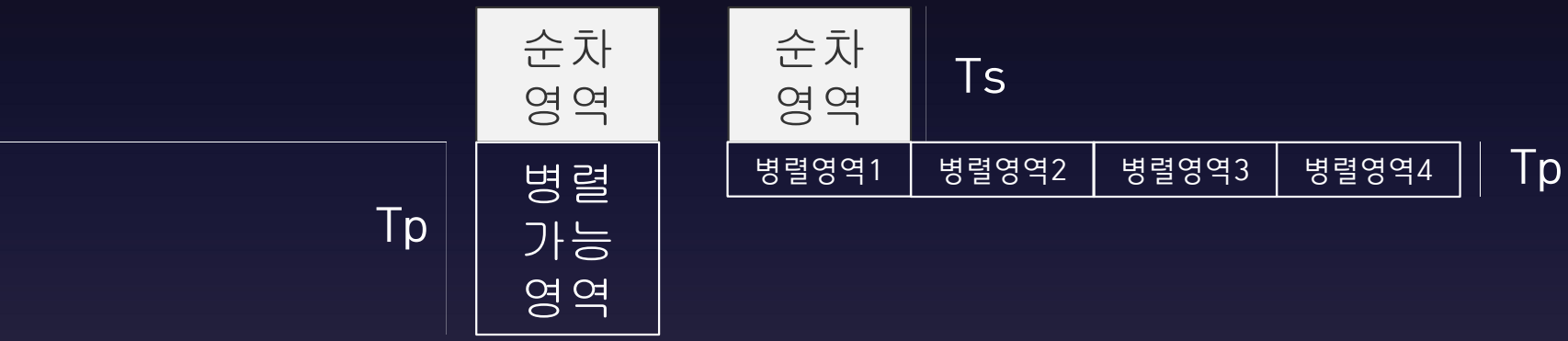
- CPU 클록 속도의 향상
  - 1Ghz CPU : 1초에 1기가 개의 명령어 수행
  - 90년대부터 2004년까지 클록 속도 경쟁
- 하드웨어 명령어 셋의 증가
  - R = a*b + c*d
  - R = muladd(a,b,c,d)
- CPU내부 캐시의 증가
  - 정보의 흐름: 하드웨어->메모리->캐시->레지스터
  - 메모리와 캐시는 10배 정도의 속도 차이가 남

병렬 프로그래밍의 배경

- 클록 속도 경쟁의 포기 : 전력사용 증가와 발열
  - CPU 동작 클록 2배를 위해 4배의 전력 상승
- 코어의 수를 늘리는 방향으로 전환
  - 멀티코어를 이용한 멀티스레딩
- SIMD(Single Instruction Stream-Multiple Data Stream)
  - 명령어 수준의 병렬화
  - 64bit 동시처리 (MMX, 1995), 128bit 동시처리(SSE, 2000년), 256bit 동시처리 (AVX, 2011년)

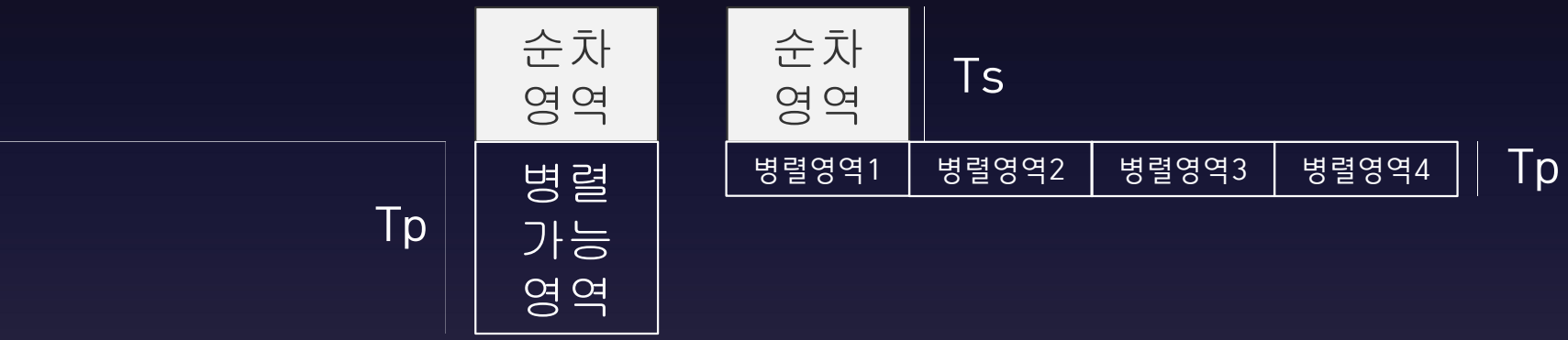
병렬화 효과

- 1967년 진 암달이 정의한 법칙



병렬화 효과

- 1967년 진 암달이 정의한 법칙



- 병렬화 전 : T1 = Ts + Tp
- 병렬화 후 : T2 = Ts + 0.25Tp
- 성능효과 = T1/T2 = 1.6

병렬프로그래밍의 난관

- 설계방법이 순차 프로그래밍과 다름
  - 사고전환의 어려움
- 병렬처리의 스레드 스케줄의 불확실성, 예전에 없던 버그 발생
- 유지 보수의 어려움
  - 병렬화 코드 추가로 가독성이 떨어짐
- 팀원의 병렬화에 따른 인식공유

OpenMP의 장점

- 기존 순차프로그램의 병렬화 변환이 쉬움
- 순차적 설계방식에서 자연스럽게 병렬화로 전환 가능
- 팀원 간의 공유 원활

OpenMP 준비

- 프로젝트 파일에 다음을 추가함

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += main.cpp
QMAKE_CXXFLAGS += -fopenmp
QMAKE_LFLAGS += -fopenmp
LIBS += -fopenmp
```

반복루프의 병렬처리

병렬화

```cpp
#include <iostream>
#include <math.h>
#include <chrono>

using namespace std;

int main()
{
    double dTime;
    chrono::system_clock::time_point tpStart, tpEnd;

    //초기화
    const  unsigned int nMAX = 100000000;
    float* fpData = new float[nMAX];


    for(int i=0; i<nMAX; i++)
        fpData[i] = i;

    tpStart = chrono::system_clock::now();


    for(int i=0; i<nMAX; i++)
        fpData[i] = sqrt(sqrt(sqrt(fpData[i])));


    //결과출력
    cout << "Data : " << fpData[0] << " "
         << fpData[1] << " " << fpData[2] << " "
         << fpData[3] << endl;

    delete[] fpData;

    tpEnd = chrono::system_clock::now();
    dTime = chrono::duration_cast<chrono::nanoseconds>(tpEnd -
            tpStart).count() / 1e6;

    cout << "Elapsed Time: " << dTime << " ms" << endl;

    return 0;
}
```

```cpp
#include <iostream>
#include <math.h>
#include <chrono>
#include <omp.h>

using namespace std;

int main()
{
    double dTime;
    chrono::system_clock::time_point tpStart, tpEnd;

    //초기화
    const  unsigned int nMAX = 100000000;
    float* fpData = new float[nMAX];

    for(int i=0; i<nMAX; i++)
        fpData[i] = i;

    tpStart = chrono::system_clock::now();

#pragma omp parallel
{

    for(int i=0; i<nMAX; i++)
        fpData[i] = sqrt(sqrt(sqrt(fpData[i])));
}

    //결과출력
    cout << "Data : " << fpData[0] << " "
         << fpData[1] << " " << fpData[2] << " "
         << fpData[3] << endl;

    delete[] fpData;

    tpEnd = chrono::system_clock::now();

    dTime = chrono::duration_cast<chrono::nanoseconds>(tpEnd-
            tpStart).count() / 1e6;
    cout << "Elapsed Time: " << dTime << " ms" << endl;

    return 0;
}
```

반복루프의 병렬처리

- 첫 번째 병렬화의 문제점
  - 코어 수만큼 자동으로 스레드 생성
  - 생성된 스레드에서 같은 계산을 수행
  - 작업분할이 필요함
- 수정된 병렬화
  - #pragma omp parallel : 스레드 생성
  - #pragma omp for : 생성된 스레드에 작업 자동 분배

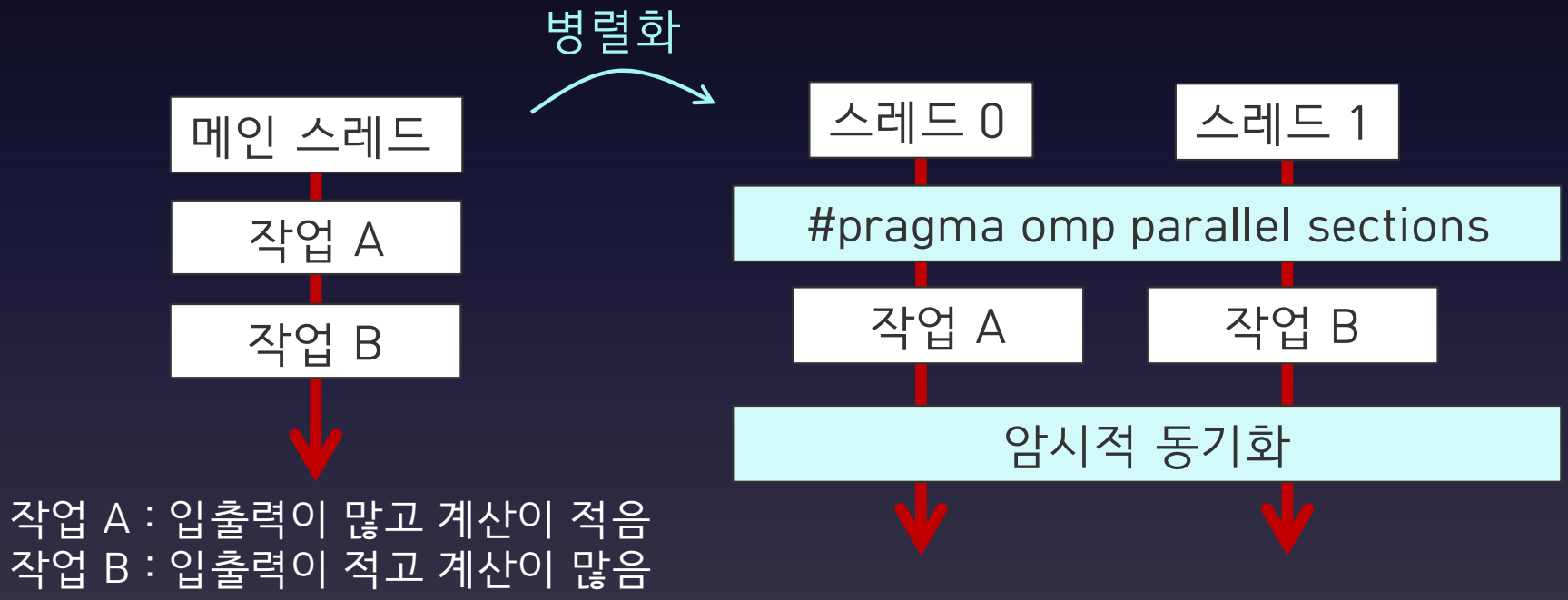
```
#pragma omp parallel
{
#pragma omp for

    for(int i=0; i<MAX; i++)
        fpData[i] = sqrt(fpData[i]);  //루트 계산
}

    //결과출력
    cout << "Data : " << fpData[0] << " "
         << fpData[1] << " " << fpData[2] << " "
         << fpData[3] << endl;
```

작업(Section)의 병렬화

- 서로 다른 작업을 여러 개의 스레드가 작업 종류별로 처리하는 것



- 작업B의 시간이 짧으므로 총 소요시간은 작업A의 소요시간과 같음

작업(Section)의 병렬화

- Example - 순차프로그램

```
void InputData(float* fpData,int nSize)
{
    for(int i=0; i<nSize; i++)
        fpData[i] = i+1;
}
void PrintData(float* fpData)
{
    cout << "Data : " << fpData[0] << " "
         << fpData[1] << " " << fpData[2] << " "
         << fpData[3] << endl;
}
void CalSqrt(float* fpData, int nSize)
{
    for(int i=0; i<nSize; i++)
        fpData[i] = sqrt(sqrt(fpData[i]));
}
void CalLog(float* fpData, int nSize)
{
    for(int i=0; i<nSize; i++)
        fpData[i] = log(log(fpData[i]));
}
```

```
int main(int argc, char *argv[])
{

    const int MAX  = 100000000;  //1억개 배열생성
    float* fpData1 = new float[MAX];
    float* fpData2 = new float[MAX];


    InputData(fpData1, MAX);
    InputData(fpData2, MAX);

    CalSqrt(fpData1, MAX);
    CalLog(fpData2, MAX);

    PrintData(fpData1);
    PrintData(fpData2);


    delete[] fpData1;
    delete[] fpData2
            ;
    return 0;
}
```

작업(Section)의 병렬화

- Example - 병렬화 프로그램

```
    chrono::system_clock::time_point tpStart, tpEnd;
    tpStart = chrono::system_clock::now();

#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        CalSqrt(fpData1, nMAX);
        #pragma omp section
        CalLog(fpData2, nMAX);
    }
}
    tpEnd = chrono::system_clock::now();

    dTime = chrono::duration_cast<chrono::nanoseconds>(tpEnd-
            tpStart).count() / 1e6;
    cout << "Elapsed Time: " << dTime << " ms" << endl;
```

#pragma omp parallel : 스레드 생성
#pragma omp sections : 작업분할로 수행할 영역 지정
#pragma omp section : 작업을 스레드에 배정함

피보나치 수열

- 앞선 두 수의 합이 다음 수가 되는 수열 규칙
  - 0 1 1 2 3 5 8 13 21 34 55 89 …
  - Fn = Fn-1 + Fn-2 (n>=2)

- 각 수를 생성하는데 필요한 일의 양이 다름
  - 첫 번째 수 : 한 번의 함수호출
  - 40 번째 수 : 496,740,421 함수호출

- 루프 병렬화 적용할 때 문제점은 ?

```
#include <iostream>
#include <math.h>
#include <time.h>

using namespace std;

int Fibonacci(int n)
{
    int x,y;

    if(n<2)
        return n;
    else
    {
        x = Fibonacci(n-1);
        y = Fibonacci(n-2);
        return (x+y);
    }
}

int main(int argc, char *argv[])
{
    const int nMAX = 41;
    int       i = 0, nFibNumber[nMAX] = { 0 };
    timespec  oStart, oEnd;

    clock_gettime(CLOCK_MONOTONIC, &oStart);

    for(int i=1; i<nMAX; i++)
        nFibNumber[i] = Fibonacci(i);

    clock_gettime(CLOCK_MONOTONIC, &oEnd);

    //시간 출력
    cout << "Ellapased Time : " << 1e3*(oEnd.tv_sec - oStart.tv_sec) +
            (oEnd.tv_nsec - oStart.tv_nsec) / 1e6 << " ms" << endl;

    //피보나치 수열 출력
    cout << "Fibonacci No : ";
    for(int i=1; i<nMAX; i++)
        cout << nFibNumber[i] << " ";

    return 0;
}
```

루프 병렬화: 피보나치 수열

```
 7 int Fibonacci(int n)
 8 {
 9     int x,y;
10
11     if(n<2)
12         return n;
13     else
14     {
15         x = Fibonacci(n-1);
16         y = Fibonacci(n-2);
17
18         return (x+y);
19     }
20 }
```

```
int main(int argc, char *argv[])
{
    const int nMAX = 41;
    int       i = 0, nFibNumber[nMAX] = { 0 };
    timespec  oStart, oEnd;

    clock_gettime(CLOCK_MONOTONIC, &oStart);

#pragma omp parallel
    {

#pragma omp for

        for(int i=1; i<nMAX; i++)
            nFibNumber[i] = Fibonacci(i);
    }

    clock_gettime(CLOCK_MONOTONIC, &oEnd);

    //시간 출력
    cout << "Ellapased Time : " << 1e3*(oEnd.tv_sec - oStart.tv_sec) +
            (oEnd.tv_nsec - oStart.tv_nsec) / 1e6 << " ms" << endl;

    //피보나치 수열 출력
    cout << "Fibonacci No : ";
    for(int i=1; i<nMAX; i++)
        cout << nFibNumber[i] << " ";

    return 0;
}
```

루프병렬화

- 스케줄 속성에 의한 분배

```
10 int main()
11 {
12     const int nMAX = 100;
13
14     int i;
15     int nppData[nMAX][4];
16
17     for(i=0; i<nMAX; i++)
18     {
19         for(int j=0; j<4; j++)
20             nppData[i][j] = 0;
21     }
22
23     //스레드 개수 지정
24     omp_set_num_threads(4);
25
26 #pragma omp parallel
27     {
28 #pragma omp for schedule(static,4)
29         for(i=0; i<nMAX; i++)
30         {
31             nppData[i][omp_get_thread_num()] = i;
32         }
33
34     }
35
36     for(i=0; i<nMAX; i++)
37     {
38         cout << nppData[i][0] << cout << " " << nppData[i][1] << cout << " "
39              << nppData[i][2] << cout << " " << nppData[i][3] << cout << endl;
40
41     }
```

고정식

한 스레드에 배분하는 작업량

루프병렬화

- 스케줄 속성에 의한 분배

```
10 int main()
11 {
12     const int nMAX = 100;
13
14     int i;
15     int nppData[nMAX][4];
16
17     for(i=0; i<nMAX; i++)
18     {
19         for(int j=0; j<4; j++)
20             nppData[i][j] = 0;
21     }
22
23     //스레드 개수 지정
24     omp_set_num_threads(4);
25
26 #pragma omp parallel
27     {
28 #pragma omp for schedule(dynamic,1)
29         for(i=0; i<nMAX; i++)
30         {
31             nppData[i][omp_get_thread_num()] = i;
32         }
33
34     }
35
36     for(i=0; i<nMAX; i++)
37     {
38         cout << nppData[i][0] << cout << " " << nppData[i][1] << cout << " "
39              << nppData[i][2] << cout << " " << nppData[i][3] << cout << endl;
40
41     }
42 }
```

준비상태에 따른 분배

루프병렬화

- 스케줄 속성에 의한 분배
  - #pragma omp for schedule(static, chunk_size)
  - #pragma omp for schedule(dynamic, chunk_size=1)
  - #pragma omp for schedule(guided, chunk_size) : dynamic과 같지만 배분량이 일정 비율로 줄어듦
  - #pragma omp for schedule(auto) : 컴파일러가 결정. OpenMP 3.0 이상