OpenMP

Jeong-Gun Lee

Dept. of Computer Engineering, Hallym University

Email: Jeonggun.Lee@hallym.ac.kr

Shared Memory Programming

- Shared Memory Parallel Programming in the Multi-Core Era
- Desktop and Laptop
  - 2, 4, 8 cores and … ?

Shared Memory Programming

- Shared Memory Parallel Programming in the Multi-Core Era

- A single node in distributed memory clusters
  - Cluster node: 2 → 8 → 16 cores …
  - /proc/cpuinfo
- Shared memory hardware Accelerators
  - Nvidia GPUs: Thousand of processing units

Open Multi-Processing

OpenMP

https://en.wikipedia.org/wiki/OpenMP

From Wikipedia, the free encyclopedia

OpenMP (**Open Multi-Processing**) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran,[3] on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.[2][4][5]

OpenMP

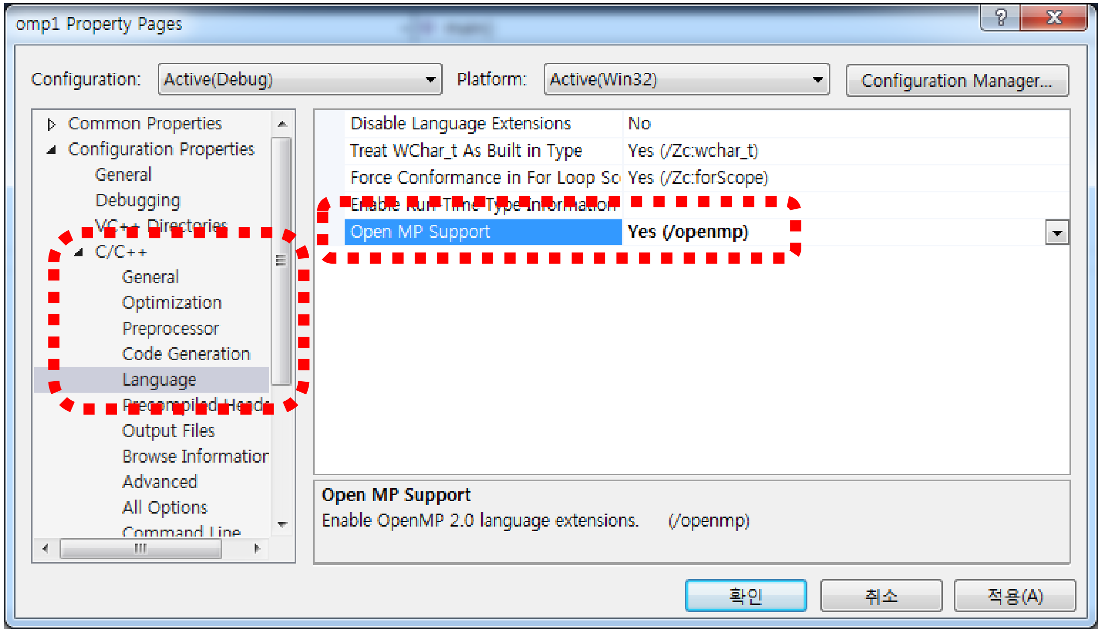
Original author(s)	OpenMP Architecture Review Board[1]
Developer(s)	OpenMP Architecture

It consists of a set of **compiler directives**, **library routines**, and **environment variables** that influence run-time behavior.

Basic

- Header file
  - #include <omp.h>
- Compile
  - gcc myomp.c –o myomp -fopenmp

Or



Hello Parallel World !

01_omp_hello_world.c

omp1.cpp

(Global Scope)

```c
#include <stdafx.h>
# include <stdio.h>
int main()
{
    #pragma omp parallel
    {
        printf("Hello world\n");
    }
    return 0;
}
```

```
E:₩OpenMP₩omp1₩Debug>omp1
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

시스템

등급:	7.4 Windows 체험 지수
프로세서:	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz 3.60 GHz
설치된 메모리(RAM):	16.0GB
시스템 종류:	64비트 운영 체제

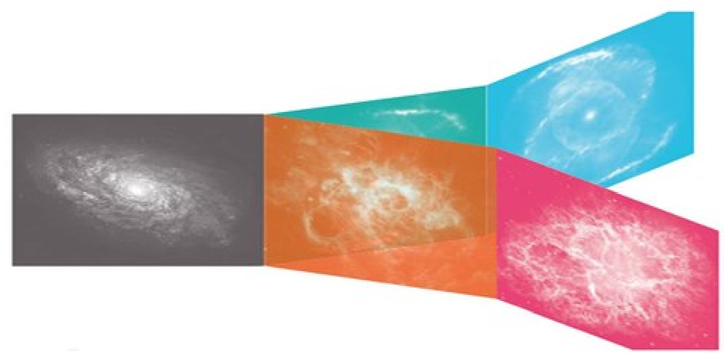
Hello Parallel World !

다중 우주론

다중 우주론은, 우주가 여러 가지 일어나는 일들과 조건에 의해 통상적으로 갈래가 나뉘어, 서로 다른 일이 일어나는 우주가 사람들이 알지 못하는 곳에서 동시에 진행되고 있다는 이론이다. 다중 우주는 급팽창 이론, M이론, 양자역학 등을 설명하는 데 유용한 이론으로 생각되며, 과학계뿐만이 아니라 예술이나 철학과도 관련이 있다. 위키백과

3단계 다중우주

양자 다중세계



양자역학의 '다세계 해석'에 따르면, 우주는 양자의 파동함수에 따라 끊임없이 갈라진다. 하나하나의 우주가 다중우주를 구성한다.

How is OpenMP typically used?

- OpenMP is usually used to **parallelize loops**:
  - Find your most time consuming loops (bottleneck).
  - Split them up between threads.

02_omp_SAXPY.c

Sequential Program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

How is OpenMP typically used?

- Single Program Multiple Data (SPMD)

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

Thread 0

```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 0;
  ub = 250;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 1

```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 250;
  ub = 500;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 2

```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 500;
  ub = 750;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 3

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  lb = 750;
  ub = 1000;
  for (i=lb;i<ub;i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

OpenMP **Fork-and-Join** model

```
printf("program begin\n");
N = 1000;
```

Serial

```
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
```

Parallel

```
M = 500;
```

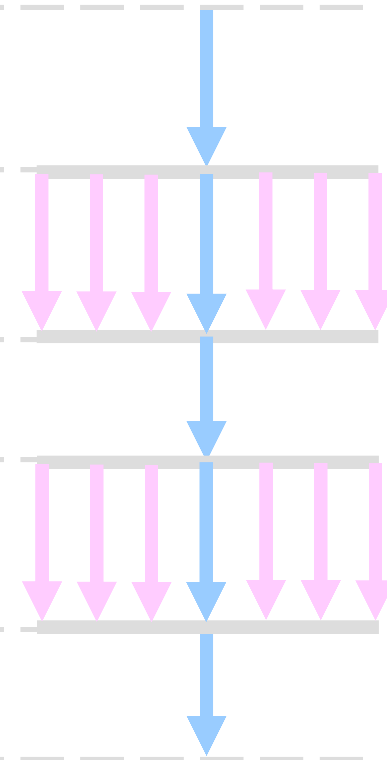
Serial

```
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];
```

Parallel

```
printf("program done\n");
```

Serial



OpenMP Constructs

- OpenMP’s constructs:
  - **Parallel Regions**
  - Worksharing (for, sections, …)
  - Data Environment (shared, private, …)
  - Synchronization (barrier, critical section, …)
  - Runtime functions/environment variables (omp_get_num_threads(), …)

OpenMP: Structured blocks (C/C++)

- Most OpenMP constructs apply to structured blocks.
  - ***Structured block***: a block with one point of entry at the top and one point of exit at the bottom.
  - The only "branches" allowed is exit() in C/C++.

```
#pragma omp parallel
{
more:  do_big_job(id);
       if(++count>1) goto more;
}
printf(" All done \n");
```

A structured block

```
if(count==1) goto more;
#pragma omp parallel
{
more:  do_big_job(id);
       if(++count>1) goto done;
}
done:    if(!really_done()) goto more;
```

<u>Not</u> A structured block

Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

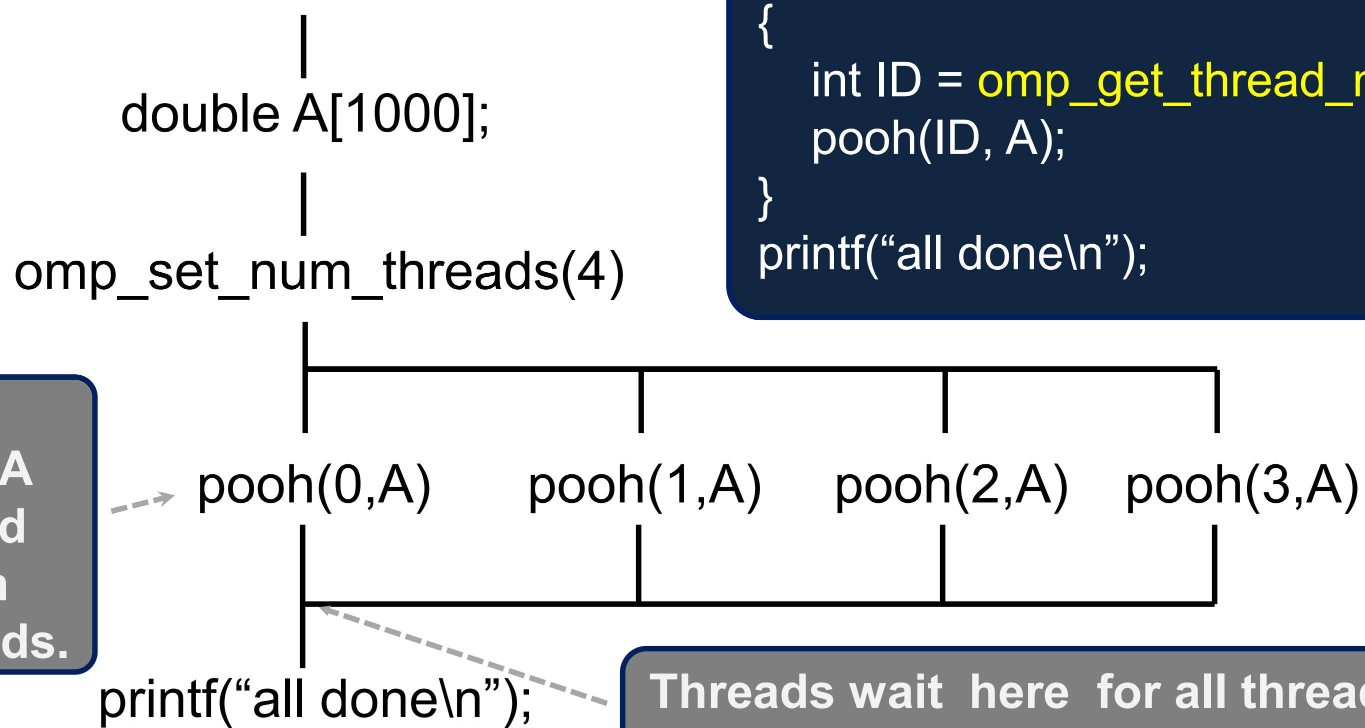
```
#pragma omp parallel
{
  id = omp_thread_num();
  A[id] = big_compute(id);
}
```

```
#pragma omp parallel for
for (I=0;I<N;I++) {
   res[I] = big_calc(I);
   A[I] = B[I] + res[I];
}
```

OpenMP **Parallel Regions**

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4); // API
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (I.e. a *barrier*)

num_threads

```
#pragma omp parallel num_threads(16)
printf("Hello, OpenMP!\n");
```

num_threads if

```
int i = 10;
#pragma omp parallel num_threads(8) if (i >= 10)
        printf("Hello, OpenMP!\n");
```

```
int i = 5;
#pragma omp parallel num_threads(8) if (i >= 10)
        printf("Hello, OpenMP!\n");
```

OpenMP Constructs

- OpenMP's constructs:
  - Parallel Regions
  - **Worksharing (for, sections, …)**
  - Data Environment (shared, private, …)
  - Synchronization (barrier, critical section, …)
  - Runtime functions/environment variables (omp_get_num_threads(), …)

OpenMP API: Combined parallel work-share

- OpenMP shortcut: Put the "parallel" and the work-share on the same line

```
int i;
double  res[MAX];
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```

```
int i;
double  res[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
  res[i] = huge();
}
```

These are equivalent

Schedule - static

```
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    clock_t startTime = clock();

    #pragma omp parallel
    {
        int threadCount = omp_get_num_threads();
        #pragma omp for schedule(static)
        for (int i = 0; i < threadCount * 5; i++)
        {
           Sleep(i * 100);
           printf("Thread Num: %d, counter = %i\n“, omp_get_thread_num(), i);
        }
        printf("Thread Num: %d, Finished\n", omp_get_thread_num());
    }
    printf("Elpase Time: %d\n", clock() - startTime);
}
```

Schedule - static

```
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    clock_t startTime = clock();

    #pragma omp parallel
    {
        int threadCount = omp_get_num_threads();
        #pragma omp for schedule(static, 2)
        for (int i = 0; i < threadCount * 5; i++)
        {
            Sleep(i * 100);
          printf("Thread Num: %d, counter = %i\n“, omp_get_thread_num(), i);
        }
        printf("Thread Num: %d, Finished\n", omp_get_thread_num());
    }
    printf("Elpase Time: %d\n", clock() - startTime);
}
```

Schedule - dynamic

```
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    clock_t startTime = clock();

    #pragma omp parallel
    {
        int threadCount = omp_get_num_threads();
        #pragma omp for schedule(dynamic)
        for (int i = 0; i < threadCount * 5; i++)
        {
            Sleep(i * 100);
          printf("Thread Num: %d, counter = %i\n“, omp_get_thread_num(), i);
        }
        printf("Thread Num: %d, Finished\n", omp_get_thread_num());
    }
    printf("Elpase Time: %d\n", clock() - startTime);
}
```

Schedule - guided

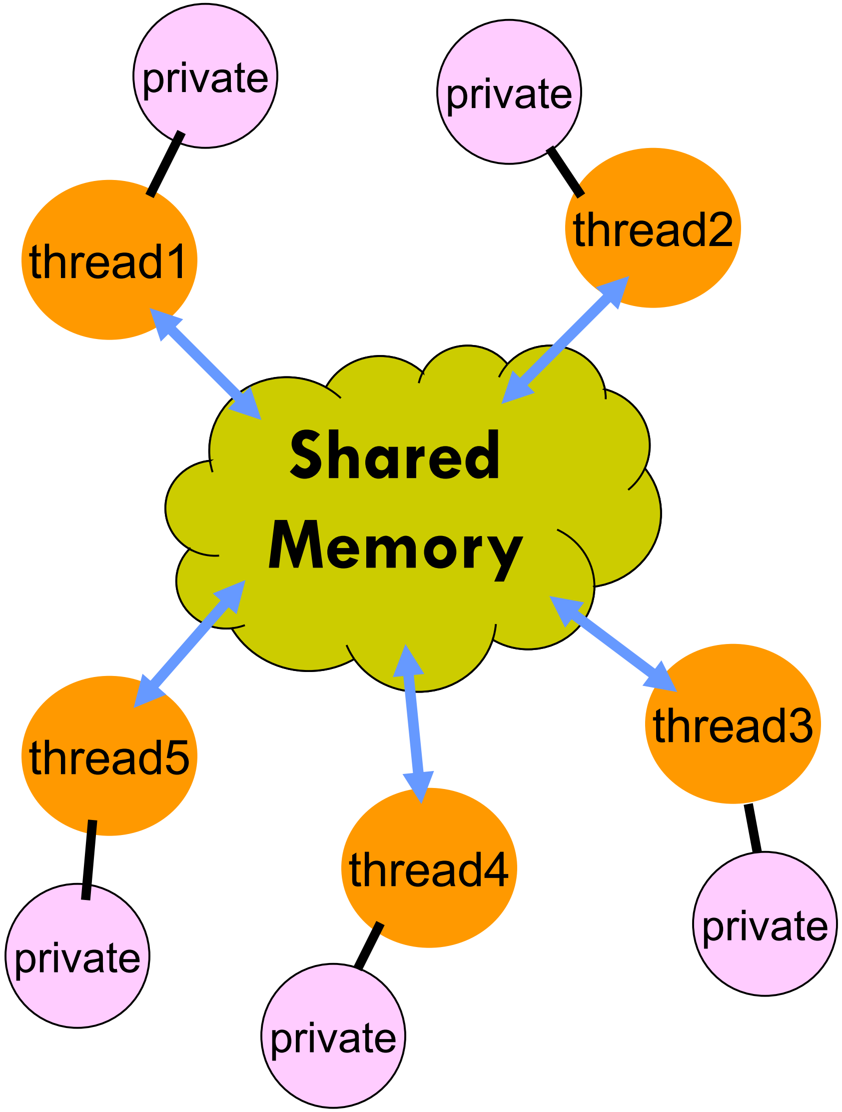
```
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    clock_t startTime = clock();

    #pragma omp parallel
    {
        int threadCount = omp_get_num_threads();
        #pragma omp for schedule(guided)
        for (int i = 0; i < threadCount * 5; i++)
        {
            Sleep(i * 100);
          printf("Thread Num: %d, counter = %i\n“, omp_get_thread_num(), i);
        }
        printf("Thread Num: %d, Finished\n", omp_get_thread_num());
    }
    printf("Elpase Time: %d\n", clock() - startTime);
}
```

OpenMP Constructs

- OpenMP's constructs:
  - Parallel Regions
  - Worksharing (for, sections, …)
  - **Data Environment (shared, private, …)**
  - Synchronization (barrier, critical section, …)
  - Runtime functions/environment variables (omp_get_num_threads(), …)

Shared Memory Model



- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the threads that owns it
- Data transfer is transparent to the programmer

Data Environment: Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Variables within a statement block are PRIVATE.

Private / shared

- Just single thread code

```
E:₩OpenMP₩omp1₩Debug>omp1
Elpase Time : 1420
pi = 3.14159265
```

```
double pi = 0.0;
const int iterationCount = 200000000;

clock_t startTime = clock();

for (int i = 0; i < iterationCount; i++)
{
        pi += 4 * (i % 2 ? -1 : 1) / (2.0 * i + 1.0);
}

printf("Elpase Time : %d\n", clock() - startTime);
printf("pi = %.8f\n", pi);
```

Private / shared

- Apply **OpenMP** constructs !

```
E:\OpenMP\omp1\Debug>omp1
Elpase Time : 1633
pi = 3.14099656
```

```
double pi = 0.0;
const int iterationCount = 200000000;
clock_t startTime = clock();

#pragma omp parallel
{
        #pragma omp for
        for (int i = 0; i < iterationCount; i++)
        {
                pi += 4 * (i % 2 ? -1 : 1) / (2.0 * i + 1.0);
        }
}

printf("Elpase Time : %d\n", clock() - startTime);
printf("pi = %.8f\n", pi);
```

Problems in shared memory

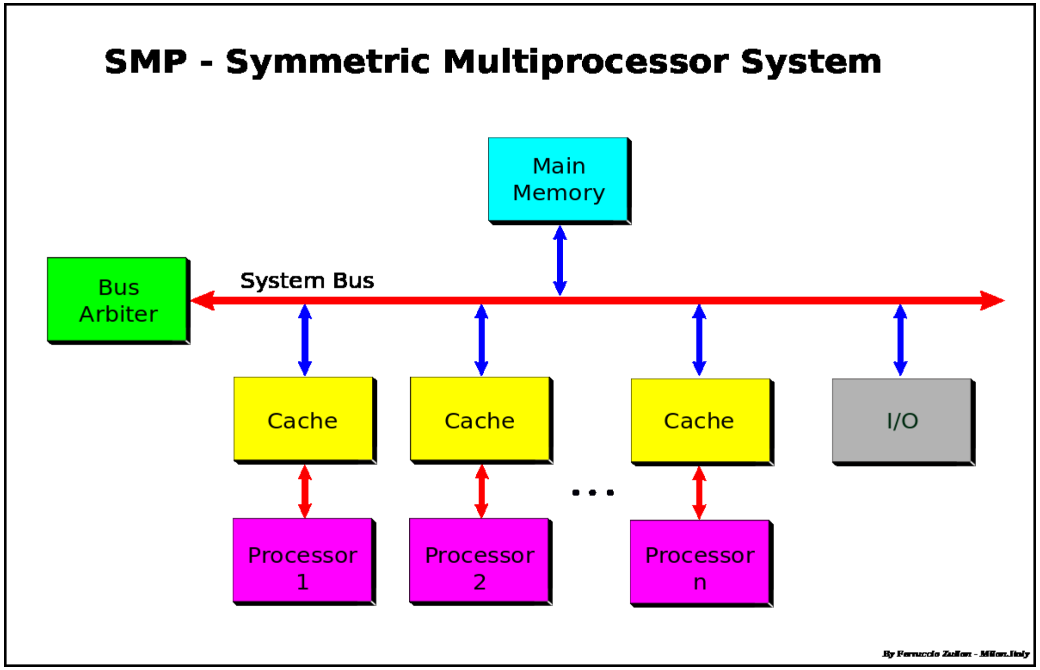
```
E:₩OpenMP₩omp1₩Debug>omp1
Elpase Time : 1420
pi = 3.14159265
```

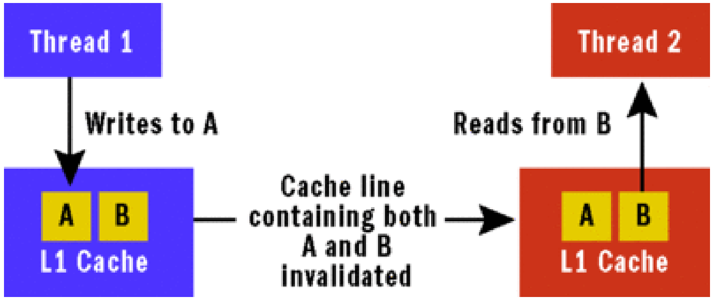
```
E:₩OpenMP₩omp1₩Debug>omp1
Elpase Time : 1633
pi = 3.14099656
```

- What's wrong ?
  - Incorrect Pi
    - Some **synchronization problem** of multiple threads
  - Execution time increases
    - ???

True / False Sharing

- Problem: Execution time increases ! → Why ?





Private / shared

- Solution for the problems

```
double pi = 0.0;
const int iterationCount = 200000000;
clock_t startTime = clock();

#pragma omp parallel
{
        #pragma omp for
        for (int i = 0; i < iterationCount; i++)
        {
                #pragma omp atomic
                pi += 4 * (i % 2 ? -1 : 1) / (2.0 * i + 1.0);
        }
}

printf("Elpase Time : %d\n", clock() - startTime);
printf("pi = %.8f\n", pi);
```

Private / shared

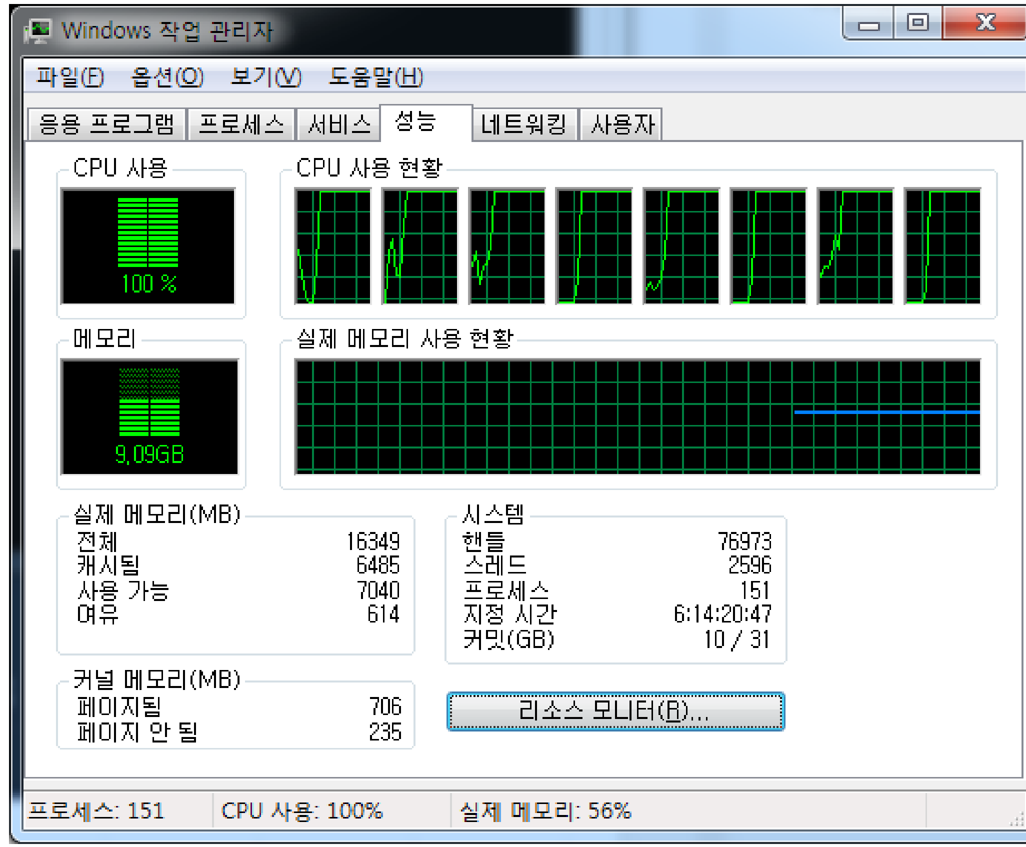
- Solution 1 for the problems

```
double pi = 0.0;
const int iterationCount = 200000000;
clock_t startTime = clock();

#pragma omp parallel
{
        #pragma omp for
        for (int i = 0; i < iterationCount; i++)
        {
                #pragma omp atomic
                pi += 4 * (i % 2 ? -1 : 1) / (2.0 * i + 1.0);
        }
}

printf("Elpase Time : %d\n", clock() - startTime);
printf("pi = %.8f\n", pi);
```

```
E:₩OpenMP₩omp1₩Debug>omp1
Elpase Time : 76159
pi = 3.14159265
```



Private / shared

- Solution2 for the problems

```
E:\OpenMP\omp1\Debug>omp1
Elpase Time : 371
pi = 3.14159265
```

```
double pi = 0.0;
const int iterationCount = 200000000;
clock_t startTime = clock();

#pragma omp parallel
{
        double temp = 0.0;
        #pragma omp for
        for (int i = 0; i < iterationCount; i++)
        {
                temp += 4 * (i % 2 ? -1 : 1) / (2.0 * i + 1.0);
        }
        #pragma omp atomic
        pi += temp;

}

printf("Elpase Time : %d\n", clock() - startTime);
printf("pi = %.8f\n", pi);
```

OpenMP Constructs

- OpenMP's constructs:
  - Parallel Regions
  - Worksharing (for, sections, …)
  - Data Environment (shared, private, …)
  - **Synchronization** (barrier, critical section, …)
  - Runtime functions/environment variables (omp_get_num_threads(), …)

Barrier Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum = lsum + A[i];
   }
   #pragma omp barrier
   printf(“Thread %d : lsum = %d\n”, omp_get_thread_num(),
lsum
}
```

Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum = lsum + A[i];
   }
   #pragma omp critical
   { sum += lsum; }
}
```

Threads wait their turn; only one thread at a time executes the critical section

Critical Construct

```
int sum1 = 0;
int sum2 = 0;

#pragma omp parallel for num_threads(4)
for (int i = 0; i < 20; i++)
{
        #pragma omp critical(my)
        sum1 += i;

        #pragma omp critical(your)
        {
                sum2 -= i;
                sum2 += i;
        }
}

printf("sum1 : %d\n", sum1);
printf("sum2 : %d\n", sum2);
```

Atomic Construct

```
int sum1 = 0;
int sum2 = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 20; i++)
{
       #pragma omp atomic
       sum1 += i;

       #pragma omp atomic
       sum2 -= i;

       #pragma omp atomic
       sum2 += i;
}

printf("sum1 : %d\n", sum1);
printf("sum2 : %d\n", sum2);
```

Reduction Clause

Shared variable

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
   sum = sum + A[i];
}
```

Critical .vs. Atomic .vs. Reduction

```
clock_t startTime = clock();
long long sum = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 10000000; i++)
{
        #pragma omp critical(my)
        sum += i;
}
printf("c
```

```
clock_t startTime = clock();
long long sum = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 10000000; i++)
{
        #pragma omp atomic
        sum += i;
}
printf("c
```

```
clock_t startTime = clock();
long long sum = 0;
#pragma omp parallel for num_threads(4) reduction(+:sum)
for (int i = 0; i < 10000000; i++)
{
        sum += i;
}
printf("critical, sum : %I64d, time : %d\n", sum, clock() - startTime);
```

OpenMP Constructs

- OpenMP's constructs:
  - Parallel Regions
  - Worksharing (for, sections, …)
  - Data Environment (shared, private, …)
  - Synchronization (barrier, critical section, …)
  - **Runtime functions/environment variables (omp_get_num_threads(), …)**

Runtime func./environment var.

```
printf("omp_get_num_procs()=%d\n", omp_get_num_procs());

omp_set_num_threads(2);
printf("omp_get_max_threads()=%d\n", omp_get_max_threads());

printf("omp_in_parallel()=%d\n", omp_in_parallel());
printf("\n[parallel region]\n");

#pragma omp parallel
{
    printf("omp_get_thread_num()=%d\n", omp_get_thread_num());
    #pragma omp barrier
    #pragma omp master
    {
        printf("\t[master]\n");
        printf("\tomp_in_parallel()=%d\n", omp_in_parallel());
        printf("\tomp_get_num_threads()=%d\n", omp_get_num_threads());
        printf("\tomp_get_dynamic()=%d\n", omp_get_dynamic());
    }
    #pragma omp barrier
    #pragma omp single
    {
        printf("\t[single]\n");
        printf("\tomp_get_thread_num()=%d\n", omp_get_thread_num());
    }
}
```

Summary

- OpenMP is great for parallel programming
  - It allows parallel programs to be written ***incrementally***.
  - Sequential programs can be enhanced with OpenMP directives, leaving the original program essentially intact.
  - Compared to MPI: you don't need to partition data and insert messages in OpenMP programs

Resources

OpenMP™

http://www.openmp.org
http://openmp.org/wp/resources

Lab

- Test all the OpenMP codes we have seen so far.
- Develop your Matrix Multiplication code with OpenMP !