LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Parallel Programming Using OpenMP

Instructor: Haidar M. Harmanani

Spring 2021

OpenMP
Enabling HPC since 1997

OpenMP Recap

- Approaches to Parallelism Using OpenMP:
  - Fine-grained parallelism (loop-level)
  - Coarse-grained parallelism (task and section level)
- Fine-grained parallelism
  - Individual loops parallelized where each thread is assigned a unique range of the loop index
  - Multiple threads are spawned inside a parallel loop after parallel loop execution is serial
  - Easy to implement
- Coarse-grained parallelism
  - Based on parallel regions parallelism where multiple threads are started for parallel regions

Examples

Simple Example 1

Write a program that prints either "A race car" or "A car race" and maximize the parallelism

Simple Example 1

```
#include <stdlib.h>
#include <stdio.h>

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

          printf("A ");
          printf("race ");
          printf("car ");

   printf("\n");
   return(0);
}
```

Output?

Parallel Simple Example 1

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
     printf("A ");
     printf("race ");
     printf("car ");
}
    printf("\n");  return(0);
}
```

8 threads

```
A race car A race car A race car A
race car A race car A race car A
race car A race car
```

Parallel Simple Example 1 Using `Single`

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
    #pragma omp single
    {
       printf("A ");
       printf("race ");
       printf("car ");
     }
    }
    printf("\n");  return(0);
}
```

Output?

Parallel Simple Example 1 Using `Tasks`

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
    #pragma omp single
    {
       printf("A ");
       #pragma omp task
       printf("race ");
       #pragma omp task
       printf("car ");
     }
    }
    printf("\n");  return(0);
}
```

Output?

A race car

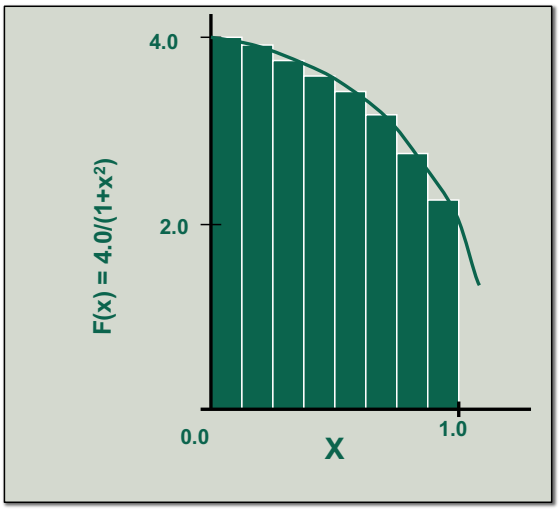
Parallel Simple Example 1 Using Tasks

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
  {
      #pragma omp single
      printf("A ");

      #pragma omp task
      printf("race ");

      #pragma omp task
      printf("car ");

      printf("\n");
return(0);
}
```

And Now?

A race car race car race race race race race race car car car car car car

Example 2: Compute PI



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Compute PI : Serial Code

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

```
static long num_steps = 100000;
double step;
void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```


Compute PI : Serial Code

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

```
double f(double x) {
return (double)4.0 / ((double)1.0 + (x*x));
}
void computePi() {
double h = (double) 1.0 / (double) iNumIntervals;
double sum = 0, x;

...

for (int i = 1; i <= iNumIntervals; i++) {
    x = h * ((double)i - (double)0.5);
    sum += f(x);
}

myPi = h * sum;
}
```

Compute PI : || Code

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

```
double f(double x) {
return (double)4.0 / ((double)1.0 + (x*x));
}
void computePi() {
double h = (double)1.0 / (double)iNumIntervals;
double sum = 0, x;

#pragma omp parallel for private(x) reduction(+:sum)

for (int i = 1; i <= iNumIntervals; i++) {
    x = h * ((double)i - (double)0.5);
    sum += f(x);
}

myPi = h * sum;
}
```


Compute PI : || Code

```
long num_steps=100000; double step;

void main()
{  int i;
   double x, sum = 0.0, pi;

    step = 1./(double)num_steps;
    start = clock();
    #pragma omp parallel for private(x) reduction (+:sum)
    for (i=0; i<num_steps; i++)
    {
      x = (i + .5)*step;
      sum = sum + 4.0/(1.+ x*x);
    }

    pi = sum*step;
    stop = clock();

    printf("The value of PI is %15.12f\n",pi);
    printf("Time to calculate PI was %f seconds\n",((double)(stop - start)/1000.0));
    return 0;
}
```

Calculate Pi by integration

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

# Threads	Runtime [sec.]	Speedup
1	0.002877	1.00
2	0.001777	1.62
4	0.002988	0.96
8	0.002050	1.40
16	0.105787	1.26

Number of Iterations: 1000,000
PI value: 3.141593

Architecture: Intel i7, 3 GHz, running Mac OS 10.11.3 with 16GB RAM

Useful MacOS/Linux Commands

```
To compile
     gcc –Wall –fopenmp –o pi pi.c

To set the number of threads to 4 using OMP_NUM_THREADS:
     In the bash shell, type: export OMP_NUM_THREADS=4
     in the c shell, type: setenv OMP_NUM_THREADS 4

You can set the number of threads to different values (2, 8, etc) using the same command
To run the OpenMP example code, simply type ./pi
You can use the time command to evaluate the program's runtime:
     Not very accurate but will do!
          voyager-2:~ haidar$ export OMP_NUM_THREADS=4
          voyager-2:~ haidar$ /usr/bin/time -p ./pi
           The value of PI is  3.141592653590
           The time to calculate PI was 18037.175000 seconds
           real 6.52
           user 17.97
           sys 0.06
You can compare the running time of the OpenMP version with the serial version by compiling
the serial version and repeating the above analysis
```

Useful MacOS/Linux Commands

- From within a shell, global adjustment of the number of threads:
  - export OMP_NUM_THREADS=4
  - ./pi
- From within a shell, one-time adjustment of the number of threads:
  - OMP_NUM_THREADS=4 ./pi
- Intel Compiler on Linux: asking for more information:
  - export KMP_AFFINITY=verbose
  - export OMP_NUM_THREADS=4
  - ./pi

Example 2: Recursive Fibonacci

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

{
   [...]
   fib(input);
   [...]
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return
}  x+y;
```

Recursive Fibonacci: Discussion

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

{
   [...]
   fib(input);
   [...]
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return
}  x+y;
```

- Only one Task / Thread should enter `fib()` from `main()`, it is responsible for creating the two initial work tasks
- `taskwait` is required, as otherwise `x` and `y` would be lost

Recursive Fibonacci: Attempt 0

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

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

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

What's wrong here?

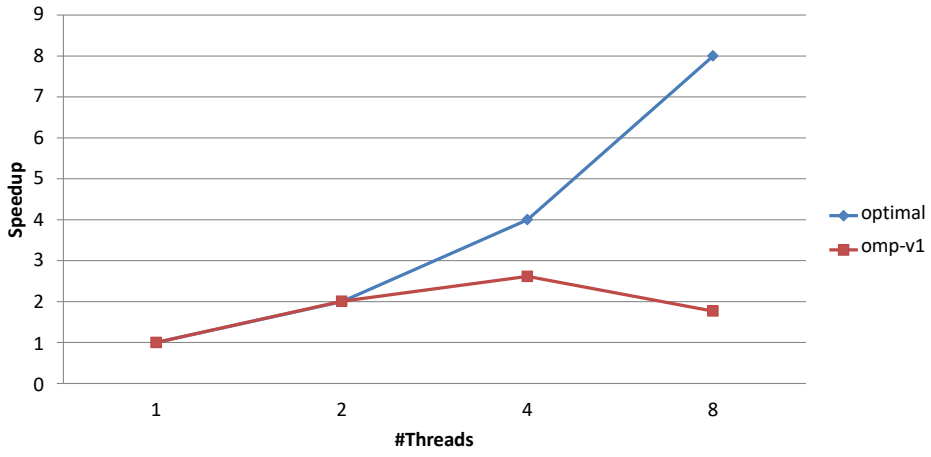
x and y are private. Can't use values of private variables outside of tasks

Recursive Fibonacci: Attempt 1

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

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

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


Recursive Fibonacci: Attempt 1



Task creation overhead prevents better scalability

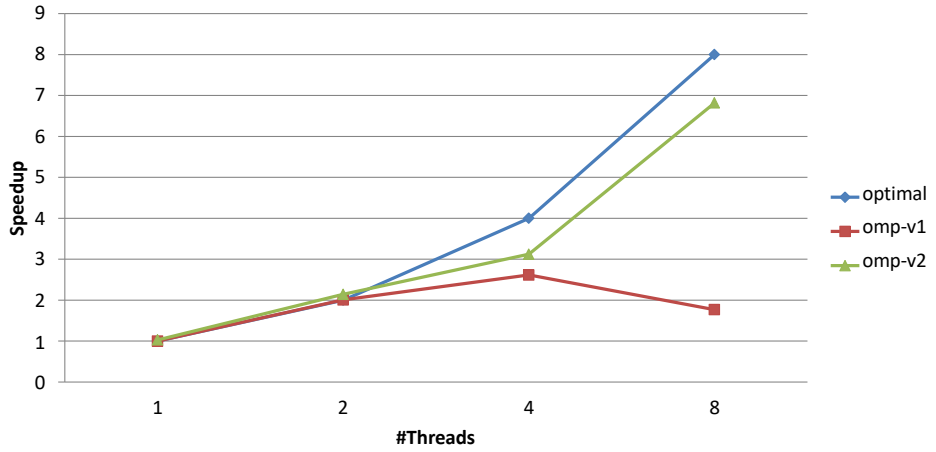
Recursive Fibonacci: Attempt 2

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

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
 if (n < 2) return n;
 int x, y;
 #pragma omp task shared(x) if(n > 30)
 {
    x = fib(n - 1);
 }
 #pragma omp task shared(y) if(n > 30)
 {
    y = fib(n - 2);
 }
 #pragma omp taskwait
    return x+y;
 }
```

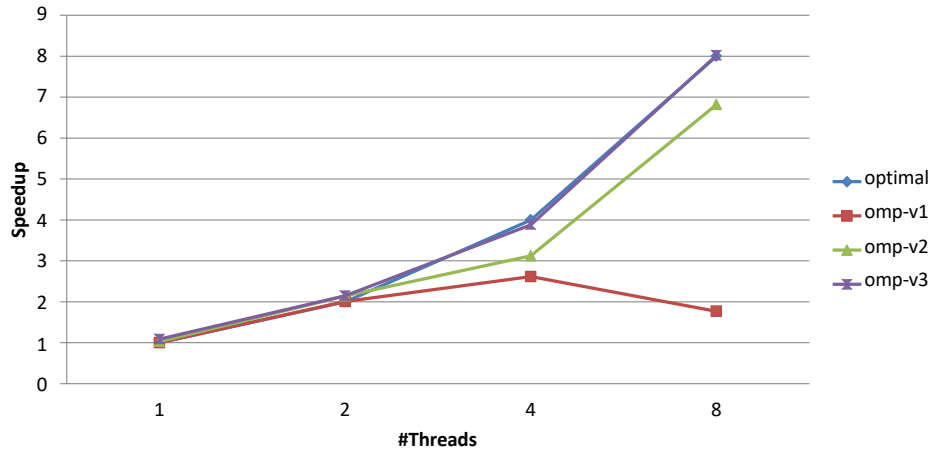
Don't create yet another task once a certain (small enough) n is reached

Recursive Fibonacci: Attempt 2



Overhead persists when running with 4 or 8 threads

Recursive Fibonacci: Attempt 3



Recursive Fibonacci: Attempt 3

```
int main(int argc,   char* argv[])
{
   [...]
#pragma  omp parallel
{
#pragma  omp single
{
   fib(input);
}
}
   [...]
}
```

Skip the OpenMP overhead once a certain n is reached (no issue w/ production compilers)

```
int fib(int n)  {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
   int x, y;
#pragma omp task shared(x)
   {
      x = fib(n - 1);
   }
#pragma omp task shared(y)
   {
      y = fib(n - 2);
   }
#pragma omp taskwait
      return x+y;
```

Example: Linked List using Tasks

- A typical C code that implements a linked list pointer chasing code is:

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

- Can we implement the above code using tasks in order to parallelize the application?

Parallel Tree Traversal: What is Wrong?

```
void traverse (Tree *tree)
{
   #pragma omp task
   if(tree->left)
     traverse(tree->left);

   #pragma omp task
   if(tree->right)
      traverse(tree->right);
   process(tree);
}
```

Simple Example: || Linked List

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
   #pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
   #pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single private(e)
{
   for(e=ml->first;e;e=e->next)
   #pragma omp task
      process(e);
}
```

Good solution – e is private

List Traversal

```
List ml; //my_list
#pragma omp parallel
{
   Element *e;
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

Simple Example: || Linked List

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single "while loop" thread creates a task for each instance of processwork()

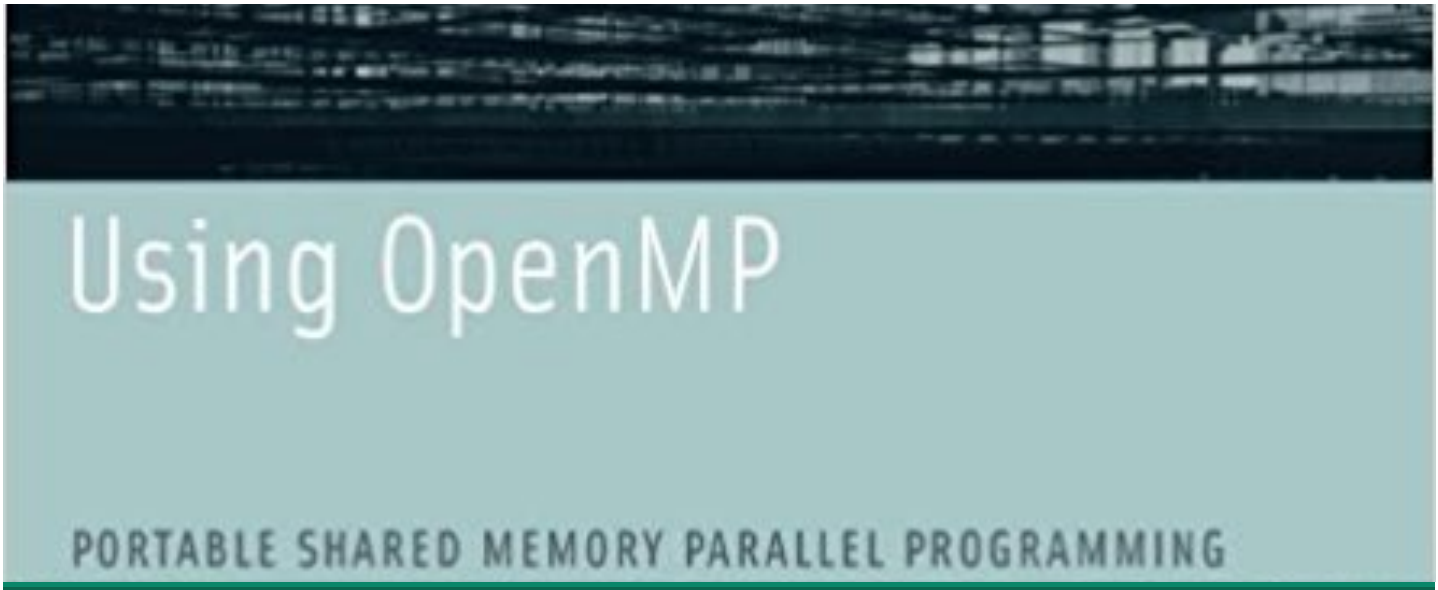
Task Construct: Linked List

- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop
  - Lets call this thread "L"
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task, construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

```
#pragma omp parallel
{
   #pragma omp single
   {  // block 1
      node * p = head;
      while (p) {
      //block 2
      #pragma omp task private(p)
         process(p);
      p = p->next;  //block 3
      }
   }
}
```

Example: Linked List using Tasks

```
#pragma omp parallel
{
 #pragma omp single nowait
   {
     while (p != NULL) {
       #pragma omp task firstprivate(p)
       {
         processwork(p);
       }
       p = p->next;
     }
    }
 }
```

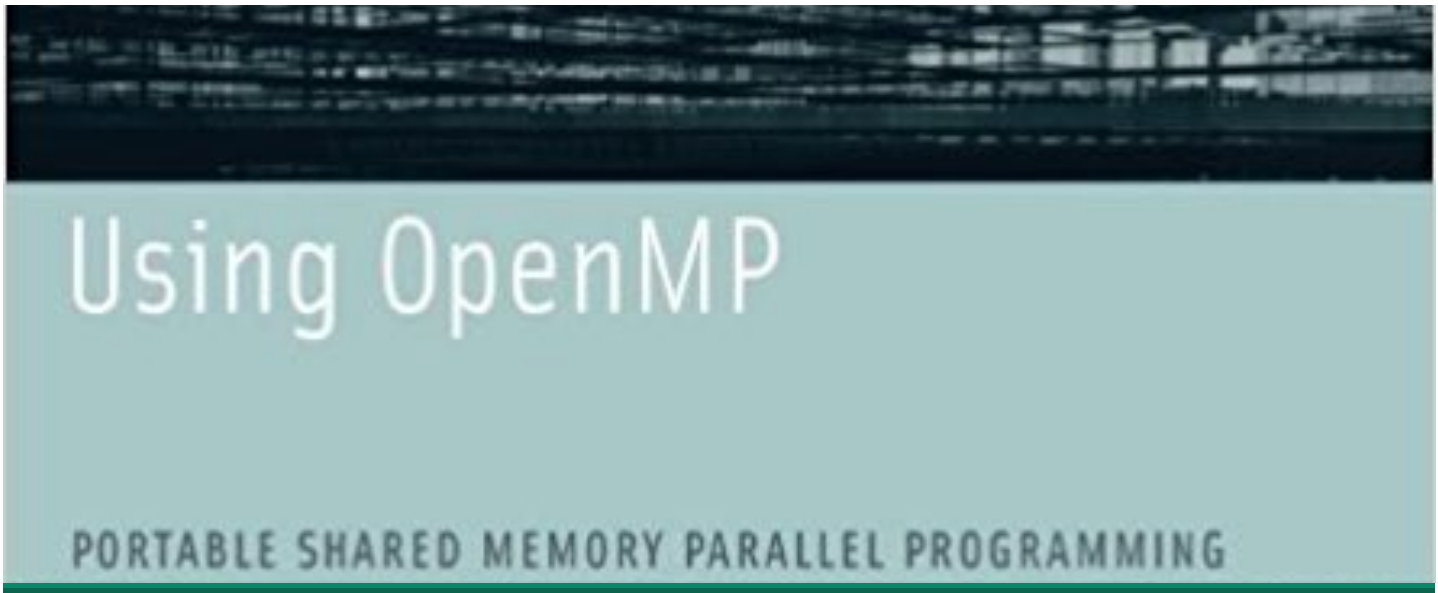
Environment Variables

Environment Variables in OpenMP

- OMP_NUM_THREADS: This environment variable specifies the default number of threads created upon entering a parallel region.
- OMP_SET_DYNAMIC: Determines if the number of threads can be dynamically changed.
- OMP_NESTED: Turns on nested parallelism.
- OMP_SCHEDULE: Scheduling of for-loops if the clause specifies runtime

Environment Variables

Name	Possible Values	Most Common Default
OMP_NUM_THREADS	Non-negative Integer	1 or #cores
OMP_SCHEDULE	“schedule [, chunk]“	„static, (N/P)“
OMP_DYNAMIC	{TRUE | FALSE}	TRUE
OMP_NESTED	{TRUE | FALSE}	FALSE
OMP_STACKSIZE	“size [B | K | M | G]“	-
OMP_WAIT_POLICY	{ACTIVE | PASSIVE}	PASSIVE
OMP_MAX_ACTIVE_LEVELS	Non-negative Integer	-
OMP_THREAD_LIMIT	Non-negative Integer	1024
OMP_PROC_BIND	{TRUE | FALSE}	FALSE
OMP_PLACES	Place List	-
OMP_CANCELLATION	{TRUE | FALSE}	FALSE
OMP_DISPLAY_ENV	{TRUE | FALSE}	FALSE
OMP_DEFAULT_DEVICE	Non-negative Integer	-



OMP False Sharing

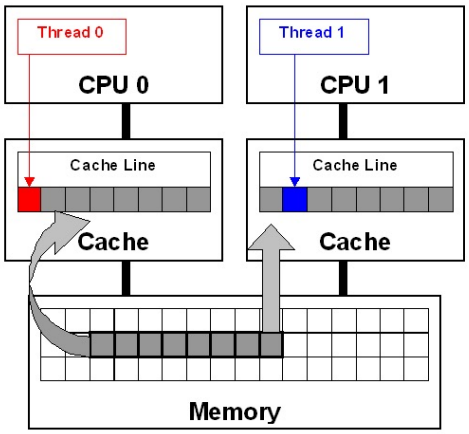
False Sharing: OMP Example

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
   int me = omp_get_thread_num();
   sum_local[me] = 0.0;
   #pragma omp for
   for (i = 0; i < N; i++)
      sum_local[me] += x[i] * y[i];
   #pragma omp atomic
      sum += sum_local[me];
}
```

Potential for false sharing on array sum_local

False Sharing: OMP Example

- The `sum_local` array is dimensioned according to the number of threads and is small enough to fit in a single cache line
- When executed in parallel, the threads modify different, but adjacent, elements of `sum_local`
- The cache line is invalidated for all processors

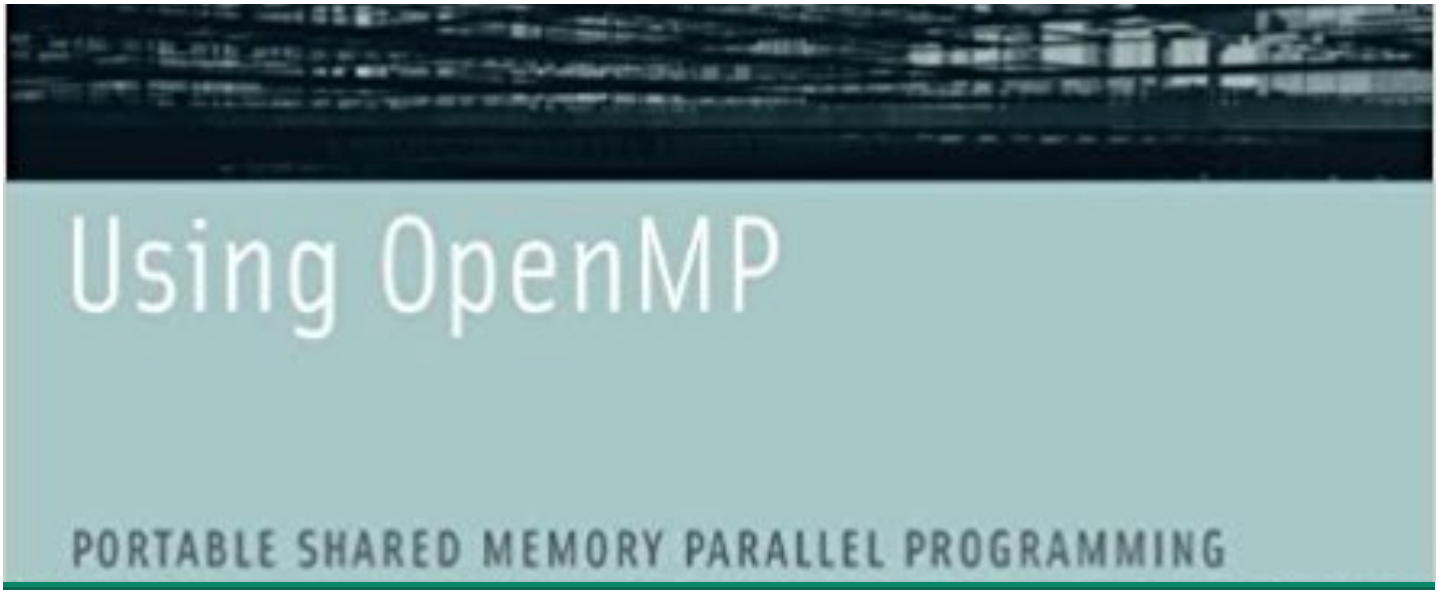


False Sharing: Solution

- Solution: ensure that variables causing false sharing are spaced far enough apart in memory that they cannot reside on the same cache line
- See: https://software.intel.com/sites/default/files/m/d/4/1/d/8/3-4-MemMgt_-_Avoiding_and_Identifying_False_Sharing_Among_Threads.pdf

Nesting `parallel` Directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
- If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.



OMP Library Functions

API?

- You can use OpenMP set of pragmas
  - Safe as your code can be analyzed as regular serial code by your compiler if you do not use the -openmp flag.
  - It's flexible, as you can define a lot of parameters in the code or from command line.
- But in some cases, you may want to write code aware of the OpenMP execution : use the `OpenMP API` on top of pragmas.

```
#include <omp.h>
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```


OpenMP Library Functions

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - `omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()`

A lock implies a memory fence of all thread visible variables

- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - `omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()`

Synchronization: Lock Functions

- Protect resources with locks.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
     id = omp_get_thread_num();
     tmp = do_lots_of_work(id);
     omp_set_lock(&lck);
        printf("%d %d", id, tmp);
     omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

- In addition, all lock routines also have a nested lock counterpart for recursive mutexes.