



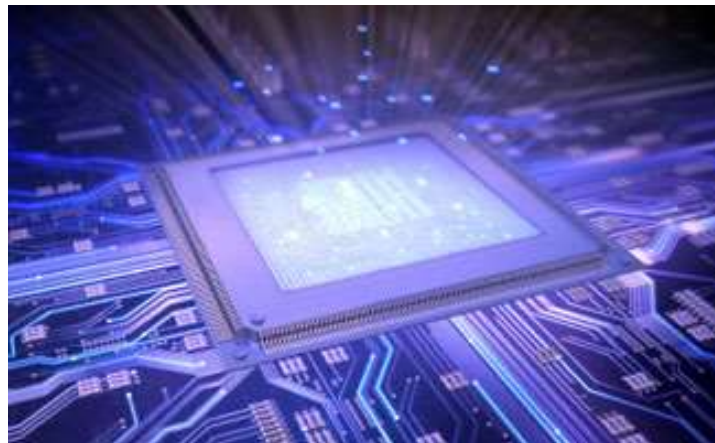
Parallel Computing

OpenMP II

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>



Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to **the set of threads that can access the variable in a parallel block.**

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has **shared** scope.
- A variable that can only be accessed by a single thread has **private** scope.
- The default scope for variables declared before a parallel block is shared.



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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

shared

private

```

void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
        *global_result_p += my_result;
} /* Trap */

```

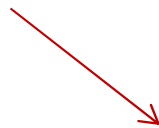
Do you remember
the trapezoidal?

We need this more complex version to add each thread's local calculation to get *global_result*.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Although we'd prefer this.

```
double Trap(double a, double b, int n);
```



```
global_result = Trap(a, b, n);
```

How about this:

```
double Local_trap(double a, double b, int n);
```

and we use it like this:

```
global_result = 0.0;  
# pragma omp parallel num_threads(thread_count)  
{  
#     pragma omp critical  
    global_result += Local_trap(double a, double b, int n);  
}
```

... we force the threads to execute sequentially.

It is now slower than a version with single thread!

How can we fix this?

We can avoid this problem by:

1. declaring a private variable inside the parallel block
2. moving the critical section after the function call

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;  /* private */

    my_result += Local_trap(double a, double b, int n);
# pragma omp critical
    global_result += my_result;
}
```

Can we do better?

Reduction operators

- A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result (we have seen that in MPI!).
- All of the intermediate results of the operation should be stored in the same variable: the **reduction variable**.

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```



+, *, -, &, |, ^, &&, ||

And the code becomes:

```
global_result = 0.0;  
# pragma omp parallel num_threads(thread_count) \  
  reduction(+: global_result)  
global_result += Local_trap(double a, double b, int n);
```

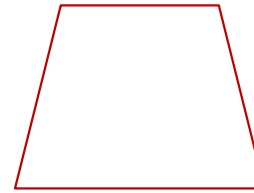
How Does OpenMP Do it?

- The reduction variable is shared.
- OpenMP create a local variable for each thread
- Those local variables are initialized to the identity value for the reduction operator
- When the parallel block ends, the values in the private variables are combined into the shared variable.

#pragma omp parallel for

- Forks a team of threads to execute the following structured block.
- The structured block following the parallel for directive **must be a for loop**.
- The system parallelizes the for loop by **dividing the iterations of the loop among the threads**.

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```



```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: approx)  
    for (i = 1; i <= n-1; i++)  
        approx += f(a + i*h);  
approx = h*approx;
```

In a loop that is parallelized with
parallel for the default scope of
loop index is private

Legal forms for parallelizable *for* statements

for	{			index++
				++index
			index < end	index--
			index <= end	--index
		index = start ;	index >= end ;	index += incr
			index > end	index -= incr
				index = index + incr
				index = incr + index
		index = index - incr		
	}			

Number of iterations **MUST** be known
prior to the loop execution.

OpenMP won't parallelize while loops or do-while loops.

Caveats

- The variable **index** must have integer or pointer type (e.g., it can't be a float).
- The expressions **start**, **end**, and **incr** must have a compatible type. For example, if **index** is a pointer, then **incr** must have integer type.


Caveats

- The expressions **start**, **end**, and **incr** must not change during execution of the loop.
- During execution of the loop, the variable **index** can only be modified by the "increment expression" in the for statement.

Data dependencies

```
fibonacci[ 0 ] = fibonacci[ 1 ] = 1;  
for (i = 2; i < n; i++)  
    fibonacci[ i ] = fibonacci[ i - 1 ] + fibonacci[ i - 2 ];
```

note 2 threads



```
fibonacci[ 0 ] = fibonacci[ 1 ] = 1;  
# pragma omp parallel for num_threads(2)  
for (i = 2; i < n; i++)  
    fibonacci[ i ] = fibonacci[ i - 1 ] + fibonacci[ i - 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes
we get this

1 1 2 3 5 8 0 0 0 0

What happened?

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.



Question

Do we have to worry about the following?
And why?

```
#pragma omp parallel for num_threads(2)
for( i = 0 ; i < n; i++) {
    x[i] = a + i*h;
    y[i] = exp(x[i]);
}
```

Estimating π

$$\pi = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \right] = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;  
double sum = 0.0;  
for (k = 0; k < n; k++) {  
    sum += factor/(2*k+1);  
    factor = -factor;  
}  
pi_approx = 4.0*sum;
```

OpenMP solution #1

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

Is this a good solution?

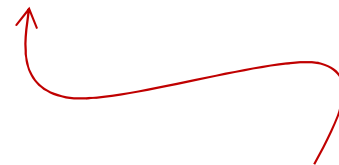
OpenMP solution #2

```
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```

How about this one?

OpenMP solution #3

```
# double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```



ensures factor has
private scope.

The default clause

- Lets the programmer specify the scope of each variable in a block.

`default (none)`

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The default clause

```
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+:sum) private(k, factor) \
    shared(n)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```

Conclusions

- To make the best use of OpenMP try to have programs with for-loops
- Scope is an error prone concept here. So be careful!
 - It may be a good idea to specify the scope of each variable in the parallel (or parallel for) block