InfoLab

IC621: Distributed and Parallel Computing

Lecture 07 : Shared-Memory Programming with OpenMP (I)

Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

OpenMP vs. Pthreads (I)

- **OpenMP : API for shared-memory parallel programming**
  - Pthreads are also API for shared-memory programming
- **Pthreads**
  - requires that the programmer explicitly specify the behavior of each thread
  - is a library of functions that can be linked to a C program
    - can be used with any C compiler
- **OpenMP**
  - allows the programmer to simply state that a block of code should be executed in parallel
    - the precise determination of the tasks and which thread should execute them is left to the compiler and the run-time system
  - requires compiler support for some operations

OpenMP vs. Pthreads (II)

- **Pthreads**
  - provides us with the power to program virtually any conceivable thread behavior
  - benefit : lower-level API
  - cost : it's up to us to specify every detail of the behavior of each thread

- **OpenMP**
  - allows the compiler and run-time system to determine some of the details of thread behavior
  - benefit: simpler to code some parallel behaviors
  - cost : some low-level thread interactions can be more difficult to program

Purpose of developing OpenMP

- Writing large-scale high-performance programs using APIs such as Pthreads was regarded to be too difficult
- OpenMP was explicitly designed to allow programmers to incrementally parallelize existing serial programs
- That is virtually impossible with MPI and fairly difficult with Pthreads

Directives-based API

- **`pragma`**
  - there are special preprocessor instructions known as `pragmas`
  - pragmas are typically added to a system to allow behaviors that aren't part of the basic C/C++ specification
  - compilers that don't support the `pragmas` are free to ignore them
  - syntax: **`# pragma`**
    - # : preprocessor directives
    - if a pragma won't fit on a single line, the newline needs to be "escaped" by a backslash \

- **Allowing a program that uses the pragmas to run on platforms that don't support them**
  - carefully written OpenMP program can be compiled and run on any system with a C compiler
    - regardless of whether the compiler supports OpenMP

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void);  /* Thread function */
6
7  int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 #  pragma omp parallel num_threads(thread_count)
12    Hello();
13
14    return 0;
15 }  /* main */
16
17 void Hello(void) {
18    int my_rank = omp_get_thread_num();
19    int thread_count = omp_get_num_threads();
20
21    printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23 }  /* Hello */
```

Compiling and running

- **Compiling**

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

- **Running**

```
$ ./omp_hello 4
```

- **Output**

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

or

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

A “Hello” program using OpenMP

- **OpenMP pragmas always begin with “`# pragma omp`”**
- **`parallel` directive**
  - specifies that the structured block of code that follows should be executed by multiple threads
- **Structured block**
  - *definition* : C statement or a compound C statement with one point of entry and one point of exit
  - it prohibits code that branches into or out of the middle of the structured block
  - the number of threads running it is determined by the run-time system
    - the algorithm used is fairly complicated
    - the system typically runs one thread on each available core

`num_threads` clause

- it allows the programmer to specify the number of threads that should execute the structured block

```
#  pragma omp parallel num_threads(thread_count)
```

- there may be system-defined limitations on the number of threads that a program can start
  - OpenMP Standard doesn't guarantee that `thread_count` threads will be actually started
  - most current systems can start hundreds or even thousands of threads

When the program reaches the parallel directive

- the original thread continues executing
- `thread_count-1` additional threads are started

Team

- the collection of threads executing the parallel block (master+slaves)
- master : the original thread
- slaves : the additional threads

Implicit barrier

- there is when the block of code is completed
  - i.e., when the threads return from the call to “Hello”
- a thread that has completed the block of code will wait for all the other threads in the team to complete the block
- when all the threads have completed the block
  - the slave threads will terminate
  - the master thread will continue executing the next codes

Local variables

- `my_rank` (rank or ID)
  - by calling **`omp_get_thread_num`**
  - the range is 0, 1, …, `thread_count – 1`
- `thread_count` (the number of threads)
  - by calling **`omp_get_num_threads`**

Error checking

- **If the compiler doesn't support OpenMP**
  - it will just ignore the `parallel` directive
  - however, the attempt to include `omp.h` and the calls to `omp_get_thread_num` and `omp_get_num_threads` will cause errors
- **Solution : using the preprocessor macro `_OPENMP`**

```
#ifdef _OPENMP
#  include <omp.h>
#endif
```

```
#  ifdef _OPENMP
      int my_rank = omp_get_thread_num();
      int thread_count = omp_get_num_threads();
#  else
      int my_rank = 0;
      int thread_count = 1;
#  endif
```

Example : trapezoidal rule

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     double  global_result = 0.0;
9     double  a, b;
10    int     n;
11    int     thread_count;
12
13    thread_count = strtol(argv[1], NULL, 10);
14    printf("Enter a, b, and n\n");
15    scanf("%lf %lf %d", &a, &b, &n);
16 #  pragma omp parallel num_threads(thread_count)
17    Trap(a, b, n, &global_result);
18
19    printf("With n = %d trapezoids, our estimate\n", n);
20    printf("of the integral from %f to %f = %.14e\n",
21       a, b, global_result);
22    return 0;
23 }  /* main */
```

```
25  void Trap(double a, double b, int n, double* global_result_p) {
26     double  h, x, my_result;
27     double  local_a, local_b;
28     int  i, local_n;
29     int my_rank = omp_get_thread_num();
30     int thread_count = omp_get_num_threads();
31
32     h = (b-a)/n;
33     local_n = n/thread_count;
34     local_a = a + my_rank*local_n*h;
35     local_b = local_a + local_n*h;
36     my_result = (f(local_a) + f(local_b))/2.0;
37     for (i = 1; i <= local_n-1; i++) {
38       x = local_a + i*h;
39       my_result += f(x);
40     }
41     my_result = my_result*h;
42
43  #  pragma omp critical
44     *global_result_p += my_result;
45  }  /* Trap */
```

Race condition

- multiple threads are attempting to access a shared resource
- at least one of the accesses is an update
- then, the accesses can result in an error
- e.g., `global_result += my result`

Critical section

- code executed by multiple threads that updates a shared resource
- shared resource can only be updated by one thread at a time
- e.g., mutexes and semaphore (in Pthreads)

`critical` directive

- tells the compiler that the system needs to arrange for the threads to have mutually exclusive access to the structured block of code

```
#  pragma omp critical
   global_result += my_result;
```

Scope of variables

- **Scope of a variable (in OpenMP)**
  - refers to the set of threads that can access the variable in a parallel block
- **Shared scope : a variable that can be accessed by all the threads in the team**
  - e.g., variables declared before a parallel block (`a, b, n, global_result, and thread_count`)
  - e.g., `*global_result_p`
- **Private scope: a variable that can only be accessed by a single thread**
  - variables of private scope have no need for the `critical` directive
  - e.g., variables used by each thread (`my_rank` and `thread_count`) are allocated from the thread's (private) stack

Reduction clause

- **Alternative code for trapezoidal rule**

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

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

- **It may actually be slower with multiple threads than one thread**

```
    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;
    }
```

Reduction variable

- a cleaner alternative that also avoids serializing execution of `Local_trap`
- we can specify that `global_result` is a *reduction* variable
- reduction operator : a binary operation (such as addition or multiplication)
- reduction : a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result
- note: all of the intermediate results of the operation should be stored in the same variable, i.e., the reduction variable

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

- `global_result` : reduction variable
- "+" : reduction operator (addition)
- the calls to `Local_trap` can take place in parallel

- **Syntax of the reduction clause**

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

- **Reduction operators (in C)**
  - `+, *, -, &, |, ^ ,&&, ||`
- **Subtraction is a bit problematic**
  - it isn't associative or commutative
  - e.g.,

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

  - two threads : thread 0 subtracts 1 and 2, thread 1 subtracts 3 and 4, then thread 0 will compute -3 and thread 1 will compute -4 and, of course, `-3 – (-7) = 4`
  - however, correct answer : -10

Reduction variables of float or double type

- the results may differ slightly when different numbers of threads are used
- it is due to the fact that floating point arithmetic isn't associative
- e.g., if a, b, and c are floats, then (a+b)+c may not be exactly equal to a+(b+c)

Variable in a reduction clause is shared

- private variable is created for each thread in the team
- threads' private variables are initialized to 0
- so, the reduction version is effectively identical to the non-reduction version

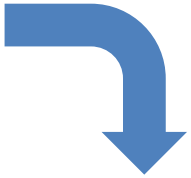
```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

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

The `parallel for` directive

- **Simply placing a directive immediately before the for loop**
  - 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;
```

Partitioning work among the threads

- partitioning is up to the system
- most systems use roughly a block partitioning
  - if there are `m` iterations,
  - then roughly the first `m/thread_count` are assigned to thread 0
  - the next `m/thread_count` are assigned to thread 1, and so on

Note: `approx` should be a reduction variable

- otherwise, it would have been an ordinary shared variable
- so, the body of the loop would be an unprotected critical section

```
approx += f(a + i*h);
```

Loop variable

- the default scope for all variables in a `parallel` directive is shared
- but, the default scope of the loop variable in a `parallel for` directive is private
  - each thread in the team has its own copy of `i`

Caveats

- **Simple method of parallelizing a serial program**
  - successively placing `parallel for` directives before each loop of a serial program
- **Caveat 1**
  - OpenMP will only parallelize for loops
  - it won't parallelize `while` loops or `do-while` loops
  - but, a `while` loop or a `do-while` loop can be converted to equivalent code that uses a `for` loop

Caveat 2

- OpenMP will only parallelize for loops for which the number of iterations can be determined
  - from the for statement itself (that is, the code for (. . . ; . . . ; . . .)),
  - prior to execution of the loop
- e.g., infinite loop cannot be parallelized
- e.g., the following loop cannot be parallelized, since the number of iterations can't be determined from the for statement alone
  - this `for` loop is also not a structured block, since the `break` adds another point of exit from the loop

```
for (i = 0; i < n; i++) {
   if ( . . . ) break;
   . . .
}
```

- **OpenMP only parallelizes `for` loops that are in canonical form**
  - variable `index` must have integer or pointer type
    - e.g., it can't be a float
  - expressions `start`, `end`, and `incr` must have a compatible type
    - e.g., if `index` is a pointer, then `incr` must have integer type
  - 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

```
for ( index = start  ;  index < end     ;  index++
                        index <= end        ++index
                        index >= end        index--
                        index > end         --index
                                            index += incr
                                            index -= incr
                                            index = index + incr
                                            index = incr + index
                                            index = index - incr )
```

Data dependences

- **Example : computing the first *n* fibonacci numbers**
  - the compiler will create an executable without complaint

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

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

  - with more than one thread, the results are unpredictable

1 1 2 3 5 8 0 0 0 0 *instead of* 1 1 2 3 5 8 13 21 34 55

What happened?

- the run-time system assigned
  - the computation of fibo[2], fibo[3], fibo[4], and fibo[5] to one thread
  - the computation of fibo[6], fibo[7], fibo[8], and fibo[9] to the other
- the second thread is using the values fibo[4] = 0 and fibo[5] = 0 to compute fibo[6]
  - then goes on to use fibo[5] = 0 and fibo[6] = 0 to compute fibo[7], and so on

Data dependences (sometimes a loop-carried dependence)

- OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a `parallel for` directive
  - it's up to the programmers to identify these dependences
- a loop in which the results of one or more iterations depend on other iterations cannot be correctly parallelized by OpenMP

Finding loop-carried dependences

- **We only need to worry about loop-carried dependences**
  - don't need to worry about more general data dependences
  - e.g., data dependence between Lines 2 and 3
    - however, no problem with the parallelization since the computation of x[i] and its subsequent use will always be assigned to the same thread

```
1       for (i = 0; i < n; i++) {
2          x[i] = a + i*h;
3          y[i] = exp(x[i]);
4       }
```

```
1   #   pragma omp parallel for num_threads(thread_count)
2       for (i = 0; i < n; i++) {
3          x[i] = a + i*h;
4          y[i] = exp(x[i]);
5       }
```

Observation

- at least one of the statements must write or update the variable in order for the statements to represent a dependence
- so, to detect a loop-carried dependence, we should only concern ourselves with variables that are updated by the loop body
- that is, we should look for variables that are read or written in one iteration, and written in another

no loop-carried dependence

```
1  #  pragma omp parallel for num_threads(thread_count)
2     for (i = 0; i < n; i++) {
3        x[i] = a + i*h;
4        y[i] = exp(x[i]);
5     }
```

loop-carried dependence

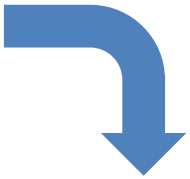
```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

Example : estimating π

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

- the update to factor in Line 7 in iteration k
- the subsequent increment of sum in Line 6 in iteration k+1

```
1         double factor = 1.0;
2         double sum = 0.0;
3         for (k = 0; k < n; k++) {
4            sum += factor/(2*k+1);
5            factor = -factor;
6         }
7         pi_approx = 4.0*sum;
```



```
1         double factor = 1.0;
2         double sum = 0.0;
3   #     pragma omp parallel for num_threads(thread_count) \
4            reduction(+:sum)
5         for (k = 0; k < n; k++) {
6            sum += factor/(2*k+1);
7            factor = -factor;
8         }
9         pi_approx = 4.0*sum;
```

- **Loop-carried dependence in estimating π**
  - if iteration k is assigned to one thread
  - iteration k+1 is assigned to another thread
  - then, no guarantee that the value of factor in Line 6 will be correct
- **We can fix the problem by examining the series**

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

  - in iteration k, the value of factor should be $(-1)^k$
    - +1 if k is even, and -1 if k is odd
  - so, if we replace the code

```
sum += factor/(2*k+1);
factor = -factor;
```

  - by

```
if (k % 2 == 0)
   factor = 1.0;
else
   factor = -1.0;
sum += factor/(2*k+1);
```

  - or, if you prefer the ?: operator

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*k+1);
```

  - we will eliminate the loop dependency

Things still aren't quite right

- e.g., two threads and n = 1000, the result is consistently wrong

```
With n = 1000 terms and 2 threads,
   Our estimate of pi = 2.97063289263385
With n = 1000 terms and 2 threads,
   Our estimate of pi = 3.22392164798593
```

- on the other hand, only one thread get a correct result

```
With n = 1000 terms and 1 threads,
   Our estimate of pi = 3.14059265383979
```

Reasons

- by default, any variable declared before the loop (except the loop variable) is shared among the threads
- `factor` is shared
  - thread 0 might assign it the value 1, but before it can use the value in the update to sum, thread 1 could assign it the value -1

Solution

- in addition to eliminating the loop-carried dependence
- we need to insure that each thread has its own copy of `factor` by adding a private clause to the `parallel for` directive

```
1          double sum = 0.0;
2  #       pragma omp parallel for num_threads(thread_count) \
3             reduction(+:sum) private(factor)
4          for (k = 0; k < n; k++) {
5             if (k % 2 == 0)
6                factor = 1.0;
7             else
8                factor = -1.0;
9             sum += factor/(2*k+1);
10         }
```

- the updates of one thread to `factor` won't affect the value of factor in another thread

Note

- the value of a variable with private scope is unspecified at the beginning of a parallel block or a parallel for block
- its value is also unspecified after completion of a parallel or parallel for block
- e.g., in the first `printf` statement and the final `printf`

```
1     int x = 5;
2  #  pragma omp parallel num_threads(thread_count) \
3        private(x)
4     {
5        int my_rank = omp_get_thread_num();
6        printf("Thread %d > before initialization, x = %d\n",
7              my_rank, x);
8        x = 2*my_rank + 2;
9        printf("Thread %d > after initialization, x = %d\n",
10             my_rank, x);
11    }
12    printf("After parallel block, x = %d\n", x);
```

`default(none)` clause

- Usually, we need to think about the scope of each variable in a `parallel` block or a `parallel for` block
  - rather than letting OpenMP decide on the scope of each variable
- `default(none)` clause will require that we specify the scope of each variable in the block and outside the block
- e.g.,
  - `sum` (reduction variable): has properties of both private and shared scope

```
        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);
        }
```

More about loops in OpenMP : Sorting

- **Bubble sort**
  - ➢ outer loop
    - first, finds the largest element in the list and stores it in `a[n-1]`
    - then, finds the next-to-the-largest element and stores it in `a[n-2]`,…
  - ➢ inner loop
    - compares consecutive pairs of elements in the current list
    - when a pair is out of order (`a[i] > a[i+1]`), it swaps them

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

- **Loop-carried dependences**
  - in outer loop: the contents of the current list depends on the previous iterations of the outer loop
  - in inner loop: the elements that are compared in iteration `i` depend on the outcome of iteration `i-1`

- **Keep in mind that**
  - even though we can always find loop-carried dependences
  - it may be difficult or impossible to remove them
  - `parallel for` directive is not a universal solution to the problem of parallelizing `for` loops

Odd-even transposition sort

- even phase (`phase % 2 == 0`)
  - each odd-subscripted element, `a[i]`, is compared to the element to its "left," `a[i-1]`
  - if they're out of order, they're swapped
- odd phase (`phase % 2 == 1`)
  - each odd-subscripted element is compared to the element to its right
  - if they're out of order, they're swapped
- it is guaranteed that after n phases, the list will be sorted

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Loop-carried dependence

- in outer loop: if phase 0 and phase 1 are executed simultaneously, the pair that's checked in phase 1 might be (7,8) rather than (9,6)
- in inner loop: no loop-carried dependences

Table 5.1 Serial Odd-Even Transposition Sort

	Subscript in Array						
Phase	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Code for odd-even transposition sort using OpenMP

```
1       for (phase = 0; phase < n; phase++) {
2          if (phase % 2 == 0)
3   #         pragma omp parallel for num_threads(thread_count) \
4                default(none) shared(a, n) private(i, tmp)
5             for (i = 1; i < n; i += 2) {
6                if (a[i-1] > a[i]) {
7                   tmp = a[i-1];
8                   a[i-1] = a[i];
9                   a[i] = tmp;
10               }
11            }
12         else
13  #         pragma omp parallel for num_threads(thread_count) \
14               default(none) shared(a, n) private(i, tmp)
15            for (i = 1; i < n-1; i += 2) {
16               if (a[i] > a[i+1]) {
17                  tmp = a[i+1];
18                  a[i+1] = a[i];
19                  a[i] = tmp;
20               }
21            }
22      }
```

- **Potential problem 1**
  - we need to be sure that all the threads have finished phase p before any thread starts phase p+1
  - however, none of the threads will proceed to the phase p+1, until all of the threads have completed the phase p
- **Potential problem 2**
  - OpenMP implementation may fork and join `thread_count` threads on each pass through the body of the outer loop
    - overhead associated with forking and joining the threads
  - e.g., the input list is of 20,000 elements

Table 5.2 Odd-Even Sort with Two `parallel for` Directives and Two `for` Directives (times are in seconds)

`thread_count`	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

Solution for the potential problem 2

- it would be superior to fork the threads once and reuse the same team of threads for each execution of the inner loops
- we can fork our team of `thread_count` threads before the outer loop with a `parallel` directive
- in the inner loop, we use a `for` directive such that OpenMP parallelizes the for loop with the existing team of threads

`for` directive

- it doesn't fork any threads
- it uses whatever threads have already been forked in the enclosing `parallel` block
- there is an implicit barrier at the end of the loop

```
1  #  pragma omp parallel num_threads(thread_count) \
2        default(none) shared(a, n) private(i, tmp, phase)
3     for (phase = 0; phase < n; phase++) {
4        if (phase % 2 == 0)
5  #        pragma omp for
6           for (i = 1; i < n; i += 2) {
7              if (a[i-1] > a[i]) {
8                 tmp = a[i-1];
9                 a[i-1] = a[i];
10                a[i] = tmp;
11             }
12          }
13       else
14 #        pragma omp for
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Scheduling loops

- **Exact assignment of loop iterations to threads is system dependent**
- **However, most OpenMP implementations use roughly a block partitioning**
  - this assignment might be less than optimal
  - e.g., suppose that the time required by the call to `f` is proportional to the size of the argument `i`

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

  - assign much more work to thread `thread_count-1` than it will assign to thread 0

A better assignment

- a cyclic partitioning of the iterations among the threads
- e.g., suppose `t = thread_count`
  - n = 10,000
  - single thread : 3.67 seconds
  - block partitioning (2 threads):
    2.76 seconds (x1.33)
  - cyclic partitioning (2 threads):
    1.84 seconds (x1.99)

Thread	Iterations
0	0, n/t, 2n/t, …
1	1, n/t + 1, 2n/t + 1, …
⋮	⋮
t − 1	t − 1, n/t + t − 1, 2n/t + t − 1, …

Scheduling : assigning iterations to threads

- a good assignment of iterations to threads can have a very significant effect on performance
- the **`schedule`** clause can be used to assign iterations in either a `parallel for` or a `for` directive

The schedule clause

- **Without the schedule clause**

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- **With the schedule clause**

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

Syntax of the `schedule` clause

```
schedule(<type> [, <chunksize>])
```

type	meaning
static	the iterations can be assigned to the threads before the loop is executed
dynamic *or* **guided**	the iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system
auto	the compiler and/or the run-time system determine the schedule
runtime	the schedule is determined at run-time

- `chunksize` is a positive integer
- only static, dynamic, and guided schedules can have a `chunksize`

The `static` schedule type

- **The system assigns chunks of `chunksize` iterations to each thread in a round-robin fashion**
  - e.g., schedule(static,1)

Thread 0:	0, 3, 6, 9
Thread 1:	1, 4, 7, 10
Thread 2:	2, 5, 8, 11

  - e.g., schedule(static,2)

Thread 0:	0, 1, 6, 7
Thread 1:	2, 3, 8, 9
Thread 2:	4, 5, 10, 11

  - e.g., schedule(static,4)

Thread 0:	0, 1, 2, 3
Thread 1:	4, 5, 6, 7
Thread 2:	8, 9, 10, 11

  - if `chunksize` is omitted, it becomes approximately `total_iterations/thread_count`

The `dynamic` and `guided` schedule types

- **The iterations are broken up into chunks of `chunksize` consecutive iterations**
- **`Dynamic`**
  - each thread executes a chunk
  - when a thread finishes a chunk, it requests another one from the run-time system
  - this continues until all the iterations are completed
  - if `chunksize` is omitted, `1` is used as it
- **`Guided`**
  - as chunks are completed, the size of the new chunks decreases
  - if no `chunksize` is specified, it decreases down to 1
  - e.g., n = 10,000 and `thread_count` = 2

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

The `runtime` schedule type

- **Environment variables**
  - named values that can be accessed by a running program
  - available in the program's environment
  - e.g., PATH, HOME, …

```
$ echo $PATH
```

- **The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop**
  - OMP_SCHEDULE : can take on any of the values that can be used for a static, dynamic, or guided schedule
  - e.g., in the bash shell

```
$ export OMP_SCHEDULE="static,1"
```

Which schedule?

- **There is some overhead associated with the use of a schedule clause**
  - the overhead is greater for dynamic schedules than static schedules
  - the overhead associated with guided schedules is the greatest of the three
- **If we're getting satisfactory performance without a schedule clause, we should go no further**
  - however, if we suspect that the performance of the default schedule can be substantially improved
  - then, we should probably experiment with some different schedules

Guidance for scheduling option

- if each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance

- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a static schedule with small `chunksizes` will probably give the best performance

- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology