Programming Shared Address Space Platforms

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani

Pilani Campus

OpenMP

T1: ch 7

OpenMP

- OpenMP is an API for shared-memory parallelprogramming
  - “MP” in OpenMP stands for “multiprocessing”
- Both OpenMP and Pthreads are APIs for shared-memory programming
- Fundamental differences
  - Pthreads requires programmer to specify the behavior of each thread. In OpenMP, specify which block to be parallelized, compiler and runtime takes care of creating and executing.
  - Pthreads is a lower level and in OpenMP we can’t program low level thread interactions.
  - OpenMP allows programmers to incrementally parallelize serial programs where as this is virtually impossible with MPI and fairly difficult with Pthreads.

OpenMP

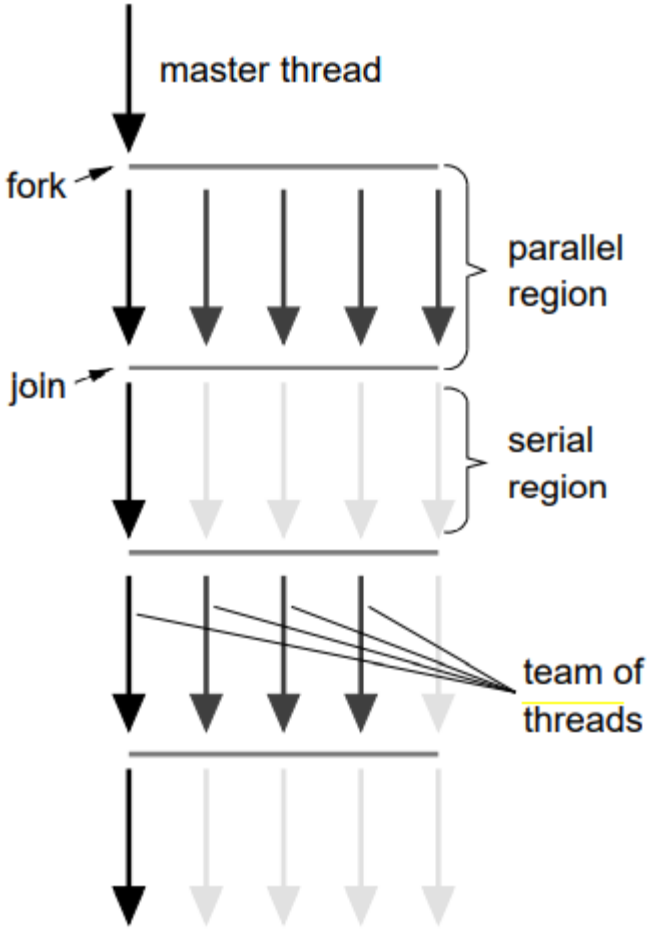
- OpenMP provides what's known as a "directives-based" shared-memory API
  - In C and C++ there are special preprocessor instructions known as pragmas
- Pragmas in C and C++ start with #pragma

```
# pragma omp parallel num threads(thread count)
```

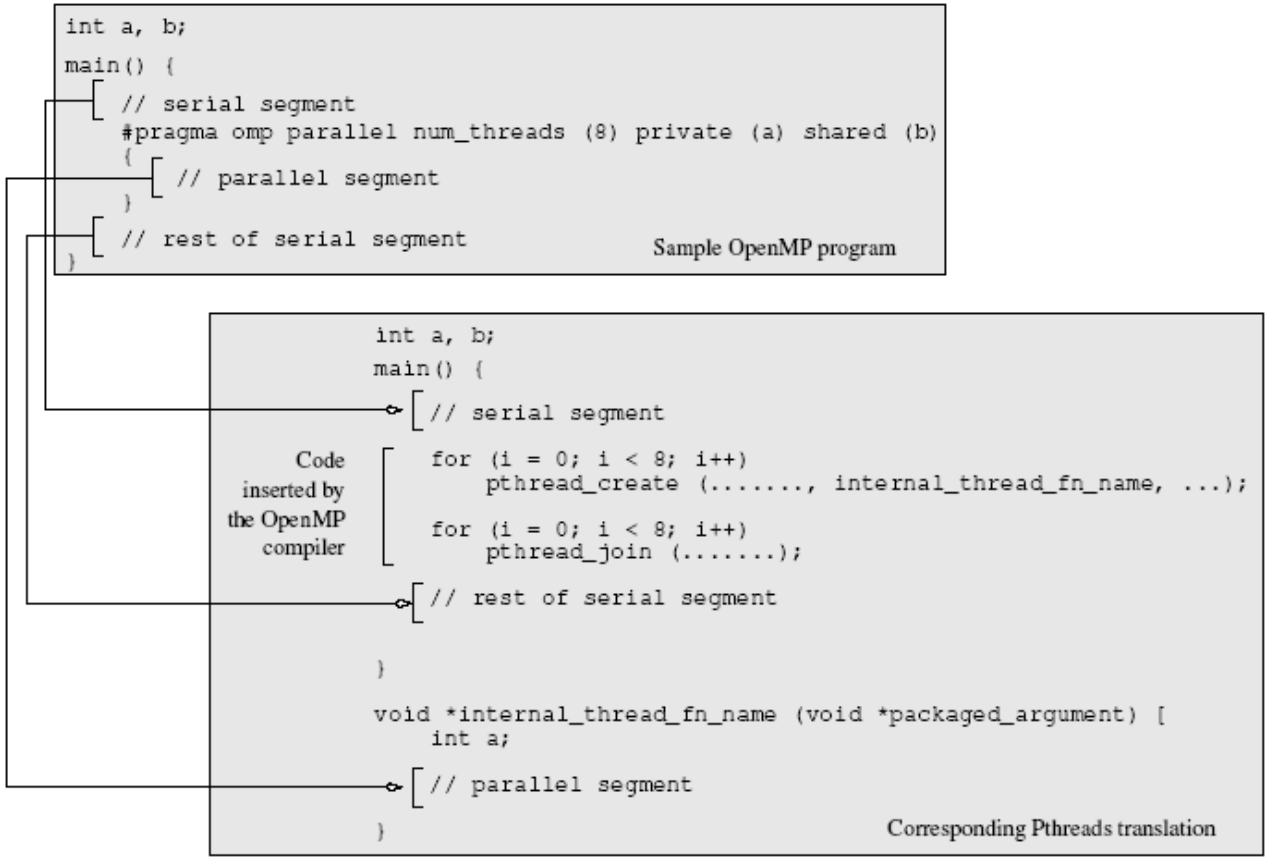
- Pragmas (like all preprocessor directives) are, by default, one line in length, so if a pragma won't fit on a single line, the newline needs to be "escaped"—that is, preceded by a backslash \
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

OpenMP Programming Model

- OpenMP directives in C and C++ are based on the #pragma compiler directives.
- A directive consists of a directive name followed by clauses.
  - #pragma omp directive [clause list]
- OpenMP programs execute serially until they encounter the parallel directive, which creates a group of threads.
  - #pragma omp parallel [clause list]
  - /* structured block */
- The main thread that encounters the parallel directive becomes the master of this group of threads and is assigned the thread id 0 within the group.
- Fork-join model



OpenMP Programming Model



A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP hello program

innovate achieve lead

```
29  int main(int argc, char* argv[]) {
30      int thread_count;
31
32      if (argc != 2) Usage(argv[0]);
33      thread_count = strtol(argv[1], NULL, 10);
34      if (thread_count <= 0) Usage(argv[0]);
35
36  #   pragma omp parallel num_threads(thread_count)
37      Hello(thread_count);
38
39      return 0;
40  }   /* main */
```

- OpenMP pragmas start with #pragma omp
- parallel directive specifies to create new threads
- Clauses add more details to directive. num_threads specifies how many threads to create

```
57  void Hello(int thread_count) {
58  #   ifdef _OPENMP
59          int my_rank = omp_get_thread_num();
60          int actual_thread_count = omp_get_num_threads();
61  #   else
62          int my_rank = 0;
63          int actual_thread_count = 1;
64  #   endif
65
66      if (my_rank == 0 && thread_count != actual_thread_count)
67          fprintf(stderr, "Number of threads started != %d\n", thread_count);
68      printf("Hello from thread %d of %d\n", my_rank, actual_thread_count);
69
70  }   /* Hello */
```

Original and new threads are called team. Original is called master. And new threads are called slaves.

There is implicit barrier after the parallel block. All threads must return and them master thread proceeds.

Compiling OpenMP Programs


```
$gcc -g -Wall -fopenmp -o omp_hello omp_hello . C
$./omp_hello 4
```

Clauses

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
- Conditional Parallelization: The clause if (scalar expression) determines whether the parallel construct results in creation of threads.
- Degree of Concurrency: The clause num_threads(integer expression) specifies the number of threads that are created.
- Data Handling: The clause private (variable list) indicates variables local to each thread. The clause firstprivate (variable list) is similar to the private, except values of variables are initialized to corresponding values before the parallel directive. The clause shared (variable list) indicates that variables are shared across all the threads.

Clauses


```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
   private (a) shared (b) firstprivate(c) {
 /* structured block */
}
```

- If the value of the variable is_parallel equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- The value of each copy of c is initialized to the value of c before the parallel directive.
- The default state of a variable is specified by the clause default (shared) or default (none).

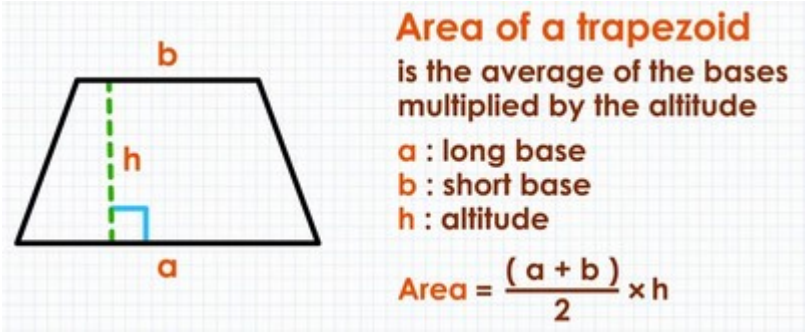
Clauses

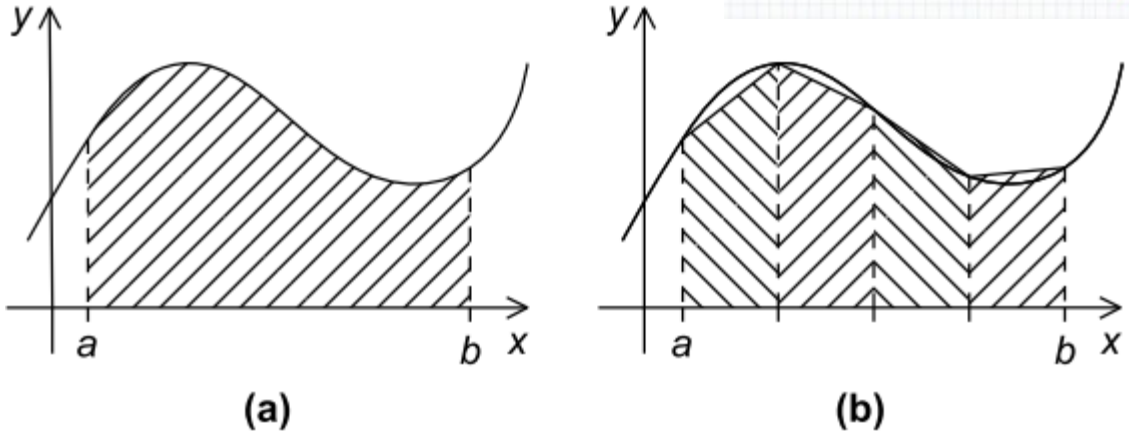
- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+, *, -, &, |, ^, &&,` and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

Example: The Trapezoidal Rule

- Trapezoidal rule is used to approximate the area between the graph of a function, y = f(x), two vertical lines, and the x-axis
- Divide the interval on the x-axis into n equal subintervals.
  - Then approximate the area lying between the graph and each subinterval by a trapezoid

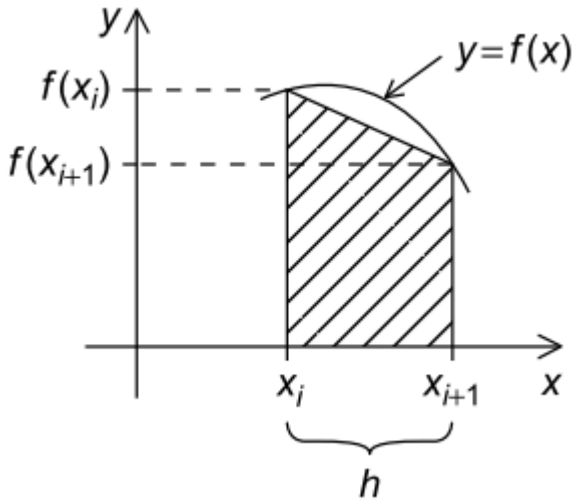




The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

Example: The Trapezoidal Rule

- Area of one trapezoid $=(h/2)[f(x_i)+f(x_{i+1})]$ where h is $x_{i+1} - x_i$
- If the vertical lines bounding the region are x = a and x = b, then $h = (b-a)/n$
  - divide into n equal subintervals
  - $x_0 = a$, $x_1 = a+h$, $x_2 = a + 2h$, ... , $x_{n-1} = a + (n-1)h$, $x_n = b$
  - Area= $f(x_0)+f(x_1)(h/2) + f(x_1)+f(x_2)(h/2) + \ldots. + f(x_{n-1})+f(x_n)\ (h/2)$
  - $h(f(x_0)/2+f(x_1)+f(x_2)+\ldots\ldots+f(x_{n-1})+f(x_n)/2)$



One trapezoid

Pseudo-code for a Serial program

- Area
  - $h(f(x_0)/2+f(x_1)+f(x_2)+\ldots\ldots+f(x_{n-1})+f(x_n)/2)$

```
/* Input: a, b, n */
h = (b−a)/n;
approx_area = (f(a) + f(b))/2.0;
for (i = 1; i <= n−1; i++) {
        x_i = a + i*h;
        approx_area += f(x_i);
}
approx_area = h*approx_area;
```

Example: The Trapezoidal Rule

innovate achieve lead

```
72  /*--------------------------------------------------------------------
73   * Function:    Trap
74   * Purpose:     Use trapezoidal rule to estimate definite integral
75   * Input args:
76   *    a: left endpoint
77   *    b: right endpoint
78   *    n: number of trapezoids
79   * Return val:
80   *    approx:  estimate of integral from a to b of f(x)
81   */
82  double Trap(double a, double b, int n, int thread_count) {
83     double  h, approx;
84     int  i;
85
86     h = (b-a)/n;
87     approx = (f(a) + f(b))/2.0;
88  #  pragma omp parallel for num_threads(thread_count) \
89        reduction(+: approx)
90     for (i = 1; i <= n-1; i++)
91       approx += f(a + i*h);
92     approx = h*approx;
93
94     return approx;
95  }  /* Trap */
```

```
59  /*--------------------------------------------------------------------
60   * Function:    f
61   * Purpose:     Compute value of function to be integrated
62   * Input arg:   x
63   * Return val:  f(x)
64   */
65  double f(double x) {
66     double return_val;
67
68     return_val = x*x;
69     return return_val;
70  }  /* f */
```

Specifying Concurrent Tasks in OpenMP

- The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks. OpenMP provides two directives
  - for and sections - to specify concurrent iterations and tasks.

for Directive

- The for directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
1   #pragma omp for [clause list]
2       /* for loop */
```

- *for* loop should not have control statements that allow the loop to exit prematurely
  - break statement
  - return statement
  - exit statement
  - goto statement
  - continue statement is allowed because its execution doesn't effect the number of loops

Finding Loop-carried Dependences

- OpenMP compilers don't check for dependences among iterations in a loop
  - It's up to us, the programmers to identify these dependences
- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP

```
1    fibo[0] = fibo[1] = 1;
2    for (i = 2; i < n; i++)
3        fibo[i] = fibo[i-1] + fibo[i-2];
4
5    fibo[0] = fibo[1] = 1;
6    # pragma omp parallel for num threads(thread count)
7    for (i = 2; i < n; i++)
8        fibo[i] = fibo[i-1] + fibo[i-2];
9
```

  - only need to worry about loop-carried dependences. We don't need to worry about more general data dependences

More About Loops In OPENMP: Sorting

- Serial bubble sort algorithm for sorting a list of integers:

```
138  void Bubble_sort(
139          int  a[]  /* in/out */,
140          int  n    /* in     */) {
141      int list_length, i, temp;
142
143      for (list_length = n; list_length >= 2; list_length--)
144          for (i = 0; i < list_length-1; i++)
145              if (a[i] > a[i+1]) {
146                  temp = a[i];
147                  a[i] = a[i+1];
148                  a[i+1] = temp;
149              }
150
151  }  /* Bubble_sort */
```

- Are there loop-carried dependencies?
  - Outer loop:
    - contents of the current list depends on the previous iterations of the outer loop
  - Inner loop:
    - In iteration i the elements that are compared depend on the outcome of iteration i-1.

How can we remove loop-carried dependencies?

Odd-even Transposition Sort

innovate achieve lead

- Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but that has more opportunities for parallelism

```
137 void Odd_even_sort(
138         int  a[]  /* in/out */,
139         int  n    /* in     */) {
140     int phase, i, temp;
141
142     for (phase = 0; phase < n; phase++)
143         if (phase % 2 == 0) { /* Even phase */
144             for (i = 1; i < n; i += 2)
145                 if (a[i-1] > a[i]) {
146                     temp = a[i];
147                     a[i] = a[i-1];
148                     a[i-1] = temp;
149                 }
150         } else { /* Odd phase */
151             for (i = 1; i < n-1; i += 2)
152                 if (a[i] > a[i+1]) {
153                     temp = a[i];
154                     a[i] = a[i+1];
155                     a[i+1] = temp;
156                 }
157     }
158 }  /* Odd_even_sort */
159
```

Suppose a = {9, 7, 8, 6}.

	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

Is there a loop-carried dependency in <u>outer</u> for loop?
Yes. Elements in current iteration are input for next itr.

Is there a loop-carried dependency in inner for loop?

For example, in an even phase loop, variable i will be odd, so for two distinct values of i, say i = j and i = k (i.e. j+2), the pairs {j − 1,j} and {k − 1, k} will be disjoint. The comparison and possible swaps of the pairs (a[j−1], a[j]) and (a[k−1], a[k]) can therefore proceed simultaneously.

Odd-even Transposition Sort

innovate achieve lead

```
161  void Odd_even(int a[], int n) {
162     int phase, i, tmp;
163  #  ifdef DEBUG
164     char title[100];
165  #  endif
166
167     for (phase = 0; phase < n; phase++) {
168        if (phase % 2 == 0)
169  #        pragma omp parallel for num_threads(thread_count) \
170              default(none) shared(a, n) private(i, tmp)
171           for (i = 1; i < n; i += 2) {
172              if (a[i-1] > a[i]) {
173                 tmp = a[i-1];
174                 a[i-1] = a[i];
175                 a[i] = tmp;
176              }
177           }
178        else
179  #        pragma omp parallel for num_threads(thread_count) \
180              default(none) shared(a, n) private(i, tmp)
181           for (i = 1; i < n-1; i += 2) {
182              if (a[i] > a[i+1]) {
183                 tmp = a[i+1];
184                 a[i+1] = a[i];
185                 a[i] = tmp;
186              }
187           }
188  #     ifdef DEBUG
189        sprintf(title, "After phase %d", phase);
190        Print_list(a, n, title);
191  #     endif
192     }
193  }  /* Odd_even */
```

We need to be sure that all the threads have finished phase p before any thread starts phase p + 1. However the parallel for directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase.

What about the overhead associated with forking and joining the threads in each iteration of the outer loop?

Odd-even Transposition Sort

innovate achieve lead

```
160  void Odd_even(int a[], int n) {
161     int phase, i, tmp;
162
163  #  pragma omp parallel num_threads(thread_count) \
164        default(none) shared(a, n) private(i, tmp, phase)
165     for (phase = 0; phase < n; phase++) {
166        if (phase % 2 == 0)
167  #        pragma omp for
168           for (i = 1; i < n; i += 2) {
169              if (a[i-1] > a[i]) {
170                 tmp = a[i-1];
171                 a[i-1] = a[i];
172                 a[i] = tmp;
173              }
174           }
175        else
176  #        pragma omp for
177           for (i = 1; i < n-1; i += 2) {
178              if (a[i] > a[i+1]) {
179                 tmp = a[i+1];
180                 a[i+1] = a[i];
181                 a[i] = tmp;
182              }
183           }
184     }
185  }  /* Odd_even */
```

Time in secs for sorting 20,000 elements

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

It is better to fork the threads once and reuse the same team of threads for each execution of the inner loops. OpenMP provides directives that allow us to do just this. We can fork our team of thread count threads before the outer loop with a *parallel* directive. Then, rather than forking a new team of threads with each execution of one of the inner loops, we use a *for* directive, which tells OpenMP to parallelize the for loop with the existing team of threads.

The for directive, unlike the parallel for directive, doesn't fork any threads. It uses whatever threads have already been forked in the enclosing parallel block

Scheduling Loops

innovate achieve lead

- Most OpenMP implementations use a block partitioning:
  - if there are n iterations in the serial loop, then in the parallel loop the first n/thread count are assigned to thread 0, the next n/thread count are assigned to thread 1, and so on.

```
1   sum = 0.0;
2   for (i = 0; i <= n; i++)
3   sum += f(i);
4
```

```
1  double f(int i) {
2      int j, start = i*(i+1)/2, finish = start + i;
3      double return val = 0.0;
4      for (j = start; j <= finish; j++) {
5          return val += sin(j);
6      }
7  return return val;
8  } /*f*/
```

  - suppose that the time required by the call to *f* is proportional to the size of the argument
  - Then a block partitioning of the iterations will assign much more work to thread count − 1 than it will assign to thread 0
  - A better assignment of work to threads: cyclic partitioning

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 Loops

innovate achieve lead

```
1  double f(int i) {
2      int j, start = i*(i+1)/2, finish = start + i;
3      double return val = 0.0;
4      for (j = start; j <= finish; j++) {
5          return val += sin(j);
6      }
7  return return val;
8  } /*f*/
```

n=10000	Time (secs)	Speedup
Single thread	3.67	
Two thread - block partitioning	2.76	1.33
Two thread - cyclic partitioning	1.84	1.99

- The call f(i) calls the sine function i times, and, for example, the time to execute f(2i) requires approximately twice as much time as the time to execute f(i)
- A good assignment of iterations to threads can have a very significant effect on performance
- In OpenMP, assigning iterations to threads is called scheduling, and the *schedule* clause can be used to assign iterations in either a parallel for or a for directive

```
1  sum = 0.0;
2  # pragma omp parallel for num threads(thread count) \
3  reduction(+:sum) schedule(static,1)
4  for (i = 0; i <= n; i++)
5      sum += f(i);
6
```

The *schedule* Clause

- In general, the schedule clause has the form

  schedule(<type> [,<chunksize>])
- The type can be any one of the following
  - *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.
  - *runtime*. The schedule is determined at run-time. In this case the environment variable OMP_SCHEDULE determines the scheduling class and the chunk size.
- chunksize
  - The chunksize is a positive integer. Only static, dynamic, and guided schedules can have a chunksize

static schedule type

- For a static schedule, the system assigns chunks of chunksize iterations to each thread in a round-robin fashion
- Suppose we have 12 iterations, and three threads.
  - schedule(static,1)
    - Thread 0: 0, 3, 6, 9
    - Thread 1: 1, 4, 7, 10
    - Thread 2: 2, 5, 8, 11
  - schedule(static,2)
    - Thread 0: 0, 1, 6, 7
    - Thread 1: 2, 3, 8, 9
    - Thread 2: 4, 5, 10, 11
  - schedule(static,4)
    - Thread 0: 0, 1, 2, 3
    - Thread 1: 4, 5, 6, 7
    - Thread 2: 8, 9, 10, 11

Thus the clause schedule(static, total iterations/thread count) is the default schedule used by most implementations of OpenMP. The chunksize can be omitted. If it is omitted, the chunksize is approximately total iterations/thread count.

dynamic schedule type

- In a dynamic schedule, the iterations are broken up into chunks of *chunksize* consecutive iterations
  - Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. This continues until all the iterations are completed
  - The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

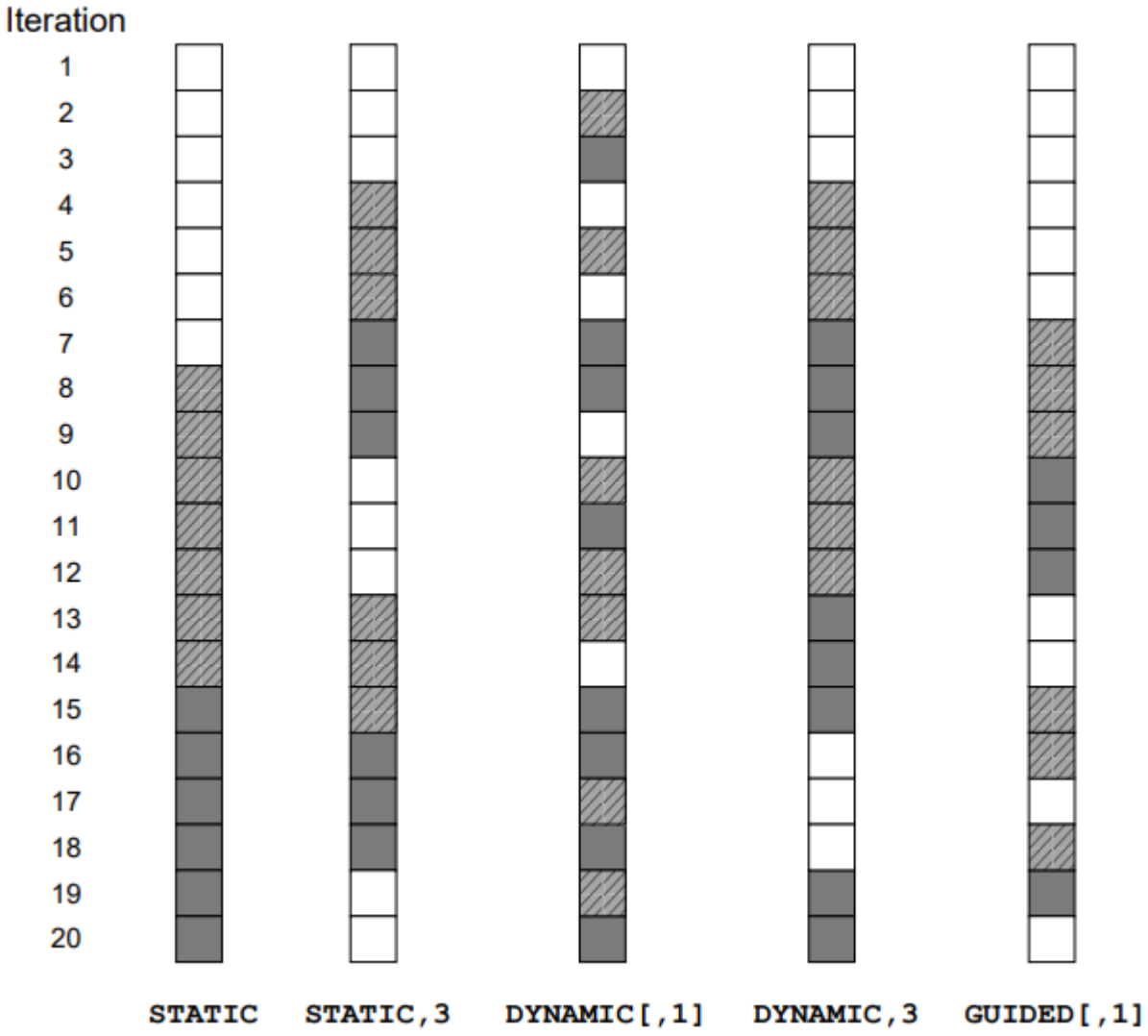
Because of a number of reasons, ranging from heterogeneous computing resources to non-uniform processor loads, equally partitioned workloads take widely varying execution times. For this reason, OpenMP has a dynamic scheduling class

guided schedule type

- In dynamic type, with 100 iterations with a chunk size of 5 there are 20 chunks. If there are 16 threads, in the best case, 16 threads get one chunk each and the remaining four threads get two chunks.
  - this assignment results in considerable idling(also referred to as an edge effect)
  - Solution is to reduce the chunk size as we proceed through the computation
  - The value of chunk-size defaults to one if none is specified.

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

Loop Schedules in OpenMP.



The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for DYNAMIC and GUIDED is one. If a chunksize is specified, the last chunk may be shorter. Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

Which schedule?

- Overhead is greater for dynamic schedules than static schedules, and the overhead associated with guided schedules is the greatest of the three
- Guidelines
  - 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, explore a variety of scheduling options.
    - The schedule(runtime) clause can be used and the different options can be explored by running the program with different assignments to the environment variable OMP SCHEDULE

Synchronization Across Multiple for Directives

- How to have a sequence of *for*-directives within a parallel construct that do not execute an implicit barrier at the end of each *for* directive
  - OpenMP provides a clause - *nowait*, which can be used with a for directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete

```
#pragma omp parallel
    {
        #pragma omp for nowait
            for (i = 0; i < nmax; i++)
                if (isEqual(name, current_list[i])
                    processCurrentName(name);
        #pragma omp for
            for (i = 0; i < mmax; i++)
                if (isEqual(name, past_list[i])
                    processPastName(name);
    }
```

Variable *name* needs to be looked up in two lists - current_list and past_list. If the name exists in a list, it must be processed accordingly. The name might exist in both lists. In this case, there is no need to wait for all threads to complete execution of the first loop before proceeding to the second loop. Consequently, we can use the nowait clause to save idling and synchronization overheads.

The *sections* Directive

- The *for* directive is suited to partitioning iteration spaces across threads
- There are three tasks (taskA, taskB, and taskC) that are independent of each other and therefore can be assigned to different threads
  - OpenMP supports such non-iterative parallel task assignment using the *sections* directive

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

```
#pragma omp parallel
{
 #pragma omp sections
 {
  #pragma omp section
  {
   taskA();
  }
  #pragma omp section
  {
   taskB();
  }
  #pragma omp section
  {
   taskC();
  }
 }
}
```

Merging parallel directives together

```
#pragma omp parallel sections
{
 #pragma omp section
  {
   taskA();
  }
  #pragma omp section
  {
   taskB();
  }
  #pragma omp section
  {
   taskC();
  }
 }
```

Synchronization Constructs in OpenMP

- #pragma omp barrier
  - On encountering this directive, all threads in a team wait until others have caught up, and then release
- Single Thread Executions: The single and master Directives
  - #pragma omp single [clause list]
    - On encountering the single block, the first thread enters the block. All the other threads proceed to the end of the block. They wait at the end of the single block. This directive is useful for computing global data as well as performing I/O
  - #pragma omp master
    - The master directive is a specialization of the single directive in which only the master thread executes the structured block

Synchronization Constructs in OpenMP

- Critical Sections: The critical and atomic Directives
  - critical regions - regions that must be executed serially, one thread at a time
  - OpenMP provides a critical directive for implementing critical regions

    #pragma omp critical [(name)]
  - The critical directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name. If a thread is already inside a critical section (with a name), all others must wait until it is done before entering the named critical section

```
1  #pragma omp parallel sections
2  {
3      #pragma parallel section
4      {
5        /* producer thread */
6        task = produce_task();
7        #pragma omp critical ( task_queue)
8        {
9           insert_into_queue(task);
10       }
11     }
12     #pragma parallel section
13     {
14       /* consumer thread */
15       #pragma omp critical ( task_queue)
16       {
17          task = extract_from_queue(task);
18       }
19       consume_task(task);
20     }
21 }
```

Synchronization Constructs in OpenMP

- Critical Sections: atomic Directive
  - If a critical section consists of an update to a single memory location, for example, incrementing or adding to an integer.
  - The atomic directive specifies that the memory location update in the following instruction should be performed as an atomic operation. The update instruction can be one of the following forms:

```
1 x binary_operation = expr
2 x++
3 ++x
4 x--
5 --x
binary_operation is one of {+, *, -, /, &, ,||,<<,>>}
```

  - Many processors provide a special load-modify-store instruction.
  - A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

In-Order Execution: The *ordered* Directive

- In many circumstances, it is necessary to execute a segment of a parallel loop in the order in which the serial version would execute it

```
1    #pragma omp ordered
2        structured block
```

```
1  cumul_sum[0] = list[0];
2  #pragma omp parallel for private (i) \
3   shared (cumul_sum, list, n) ordered
4   for (i = 1; i < n; i++)
5   {
6   /* other processing on list[i] if needed */
7    #pragma omp ordered
9    {
10     cumul_sum[i] = cumul_sum[i-1] + list[i];
11   }
12  }
```

  - To compute the cumulative sum of i numbers of a list, we can add the current number to the cumulative sum of i-1 numbers of the list. This loop must, however, be executed in order.
  - if large portions of a loop are enclosed in ordered directives, corresponding speedups suffer as the ordered directive represents an ordered serialization point in the program.

OpenMP Library Functions

- Controlling Number of Threads and Processors

```
1  #include <omp.h>
2  void omp_set_num_threads (int num_threads);
3  int omp_get_num_threads ();
4  int omp_get_max_threads ();
5  int omp_get_thread_num ();
6  int omp_get_num_procs ();
7  int omp_in_parallel();
```

- omp_set_num_threads sets the default number of threads and is overided by num_threads clause. Must be called outside parallel block. Requires omp_set_dynamic() to be called.
- omp_get_num_threads() returns the number of threads participating in a team
- omp_get_max_threads() returns the maximum number of threads that could possibly be created by a parallel directive encountered, which does not have a num_threads clause
- omp_get_thread_num returns a unique thread id. This integer lies between 0 (for the master thread) and omp get_num_threads() -1.
- omp_get_num_procs() returns the number of processors available to execute the threaded program at that point.
- omp_in_parallel returns a non-zero value if called from within the scope of a parallel region, and zero otherwise.

OpenMP Library Functions

- Controlling and Monitoring Thread Creation

```
1  #include <omp.h>
2  void omp_set_dynamic (int dynamic_threads);
3  int omp_get_dynamic ();
4  void omp_set_nested (int nested);
5  int omp_get_nested ();
```

- omp_set_dynamic() allows the programmer to dynamically alter the number of threads. If the value dynamic_threads evaluates to zero, dynamic adjustment is disabled, otherwise it is enabled. The function must be called outside the scope of a parallel region. The corresponding state, i.e., whether dynamic adjustment is enabled or disabled, can be queried using the function omp_get_dynamic, which returns a non-zero value if dynamic adjustment is enabled, and zero otherwise.
- The omp_set_nested enables nested parallelism if the value of its argument, nested, is non-zero, and disables it otherwise. When nested parallelism is disabled, any nested parallel regions subsequently encountered are serialized. The state of nested parallelism can be queried using the omp_get_nested function, which returns a non-zero value if nested parallelism is enabled, and zero otherwise.

OpenMP Library Functions

- Mutual Exclusion

```
1   #include <omp.h>
2
3   void omp_init_lock (omp_lock_t *lock);
4   void omp_destroy_lock (omp_lock_t *lock);
5   void omp_set_lock (omp_lock_t *lock);
6   void omp_unset_lock (omp_lock_t *lock);
7   int omp_test_lock (omp_lock_t *lock);
```

- OpenMP provides support for critical sections and atomic updates and also for an explicit lock.
- Before a lock can be used, it must be initialized using the omp_init_lock(). discarded using the function omp_destroy_lock().
- Once a lock has been initialized, it can be locked and unlocked using the functions omp_set_lock and omp_unset_lock.
- The function omp_test_lock() can be used to attempt to set a lock. If the function returns a non-zero value, the lock has been successfully set, otherwise the lock is currently owned by another thread.

Similar to recursive mutexes in Pthreads, OpenMP also supports nestable locks that can be locked multiple times by the same thread.

```
1   #include <omp.h>
2
3   void omp_init_nest_lock (omp_nest_lock_t *lock);
4   void omp_destroy_nest_lock (omp_nest_lock_t *lock);
5   void omp_set_nest_lock (omp_nest_lock_t *lock);
6   void omp_unset_nest_lock (omp_nest_lock_t *lock);
7   int omp_test_nest_lock (omp_nest_lock_t *lock);
```

Locks

- OpenMP provides several mechanisms for mutual exclusion in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Simulate message-p

Uses critical and atomic dir

```
34  int main(int argc, char* argv[
35      int thread_count;
36      int send_max;
37      struct queue_s** msg_queues;
38      int done_sending = 0;
39
40      if (argc != 3) Usage(argv[0]);
41      thread_count = strtol(argv[1], NULL
42      send_max = strtol(argv[2], NULL, 1
43      if (thread_count <= 0 || send_max
44
45      msg_queues = malloc(thread_count*sizeof(struct queue_node_s*));
46
47  #   pragma omp parallel num_threads(thread_count) \
48          default(none) shared(thread_count, send_max, msg_queues, done_sending)
49      {
50          int my_rank = omp_get_thread_num();
51          int msg_number;
52          srandom(my_rank);
53          msg_queues[my_rank] = Allocate_queue();
54
55  #       pragma omp barrier /* Don't let any threads send messages  */
56                             /* until all queues are constructed     */
57
58          for (msg_number = 0; msg_number < send_max; msg_number++) {
59              Send_msg(msg_queues, my_rank, thread_count, msg_number);
60              Try_receive(msg_queues[my_rank], my_rank);
61          }
62  #       pragma omp atomic
63          done_sending++;
```

```
90  void Send_msg(struct queue_s* msg_queues[], int my_rank,
91          int thread_count, int msg_number) {
92      // int mesg = random() % MAX_MSG;
93      int mesg = -msg_number;
94      int dest = random() % thread_count;
95  #   pragma omp critical
96      Enqueue(msg_queues[dest], my_rank, mesg);
97  }
98
99
100
```

```
103 void Try_receive(struct queue_s* q_p, int my_rank) {
104     int src, mesg;
105     int queue_size = q_p->enqueued - q_p->dequeued;
106
107     if (queue_size == 0) return;
108     else if (queue_size == 1)
109 #       pragma omp critical
110         Dequeue(q_p, &src, &mesg);
111     else
112         Dequeue(q_p, &src, &mesg);
113     printf("Thread %d > received %d from %d\n", my_rank
114 }   /* Try_receive */
```

pragma omp critical(**name**)
If named critical regions are not used, then only one thread can enter both functions.

Simulate message-passing using OpenMP: Locks

- In this example there is a different queue for each thread.
- Even with named critical regions, only one thread will be allowed when accessing its queue.
- The alternative is to use locks.

Simulate message-passing using OpenMP: Locks

```
92  void Send_msg(struct queue_s* msg_queues[], int my_rank,
93        int thread_count, int msg_number) {
94     // int mesg = random() % MAX_MSG;
95        int mesg = -msg_number;
96        int dest = random() % thread_count;
97        struct queue_s* q_p = msg_queues[dest];
98        omp_set_lock(&q_p->lock);
99        Enqueue(q_p, my_rank, mesg);
100       omp_unset_lock(&q_p->lock);
```

```
107 void Try_receive(struct queue_s* q_p, int my_rank) {
108       int src, mesg;
109       int queue_size = q_p->enqueued - q_p->dequeued;
110
111       if (queue_size == 0) return;
112       else if (queue_size == 1) {
113          omp_set_lock(&q_p->lock);
114          Dequeue(q_p, &src, &mesg);
115          omp_unset_lock(&q_p->lock);
116       } else
117          Dequeue(q_p, &src, &mesg);
```

```
82  struct queue_s* Allocate_queue() {
83        struct queue_s* q_p = malloc(sizeof(struct queue_s));
84        q_p->enqueued = q_p->dequeued = 0;
85        q_p->front_p = NULL;
86        q_p->tail_p = NULL;
87        omp_init_lock(&q_p->lock);
88        return q_p;
89  }  /* Allocate_queue */
```

If we use critical directive, irrespective of which queue is used, all thread have to wait.

Now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue, since different message queues have different locks.

Environment Variables in OpenMP

- OMP_NUM_THREADS
  - This environment variable specifies the default number of threads created upon entering a parallel region. The number of threads can be changed using either the omp_set_num_threads function or the num_threads clause in the parallel directive.
- OMP_DYNAMIC
  - This variable, when set to TRUE, allows the number of threads to be controlled at runtime using the omp_set_num threads function or the num_threads clause
- OMP_NESTED
  - This variable, when set to TRUE, enables nested parallelism
- OMP_SCHEDULE
  - This environment variable controls the assignment of iteration spaces associated with *for* directives that use the runtime scheduling class

Nested Parallel Directives

```
1   #pragma omp parallel for default(private) shared (a, b, c, dim) \
2                            num_threads(2)
3       for (i = 0; i < dim; i++) {
4       #pragma omp parallel for default(private) shared (a, b, c, dim) \
5                            num_threads(2)
6           for (j = 0; j < dim; j++) {
7               c(i,j) = 0;
8               #pragma omp parallel for default(private) \
9                            shared (a, b, c, dim) num_threads(2)
10              for (k = 0; k < dim; k++) {
11                  c(i,j) += a(i, k) * b(k, j);
12              }
13          }
14      }
```

- The code as written only generates a logical team of threads on encountering a nested parallel directive. The newly generated logical team is still executed by the same thread corresponding to the outer parallel directive. To generate a new set of threads, nested parallelism must be enabled using the OMP_NESTED environment variable. If the OMP_NESTED environment variable is set to FALSE, then the inner parallel region is serialized and executed by a single thread.

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You