ECE 432/532
Programming for Parallel Processors

We want to parallelize this loop.

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

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

Assignment of work
using cyclic partitioning.

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}  /* f */
```

Our definition of function *f*.

Results

- f(i) calls the sin function *i* times.

- Assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).

- n = 10,000
  - one thread
  - run-time = 3.67 seconds.

Results

- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33

- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99

The Schedule Clause

- Default schedule:

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

- Cyclic schedule:

```
      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);
```

schedule (type , chunksize)

- Type can be:
  - 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.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The *chunksize* is a positive integer.

The Static Schedule Type

```
schedule(static,1)
```

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

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

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

The Static Schedule Type

```
schedule(static,4)
```

$$\text{Thread } 0: \quad 0,1,2,3$$
$$\text{Thread } 1: \quad 4,5,6,7$$
$$\text{Thread } 2: \quad 8,9,10,11$$

The Dynamic Schedule Type

- The iterations are also 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.

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.

- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.

- If no chunksize is specified, the size of the chunks decreases down to 1.

- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

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

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.

- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

The Runtime Schedule Type

- To understand schedule(runtime) we need **environment variables**.

- Some commonly used environment variables are PATH, HOME, and SHELL.
  - The PATH variable specifies which directories the shell should search when it's looking for an executable
  - The HOME variable specifies the location of the user's home directory
  - The SHELL variable specifies the location of the executable for the user's shell.

```
$ echo $PATH
```

The Runtime Schedule Type

- Set **variables**.

```
$ export TEST_VAR="hello"

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

Queues

- Can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.

- A natural data structure to use in many multithreaded applications.

- For example, suppose we have several "producer" threads and several "consumer" threads.

  - Producer threads might "produce" requests for data.

  - Consumer threads might "consume" the request by finding or generating the requested data.

Message-Passing

- Each thread could have a shared message queue, and when one thread wants to "send a message" to another thread, it could enqueue the message in the destination thread's queue.

- A thread could receive a message by dequeuing the message at the head of its message queue.

Message-Passing

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
```

Sending Messages

```
mesg = random();
dest = random() % thread_count;
#    pragma omp critical
Enqueue(queue, dest, my_rank, mesg);
```

Receiving Messages

```
queue_size = enqueued - dequeued;
if (queue_size == 0) return;
else if (queue_size == 1)
#    pragma omp critical
     Dequeue(queue, &src, &mesg);
else
     Dequeue(queue, &src, &mesg);
Print_message(src, mesg);
```
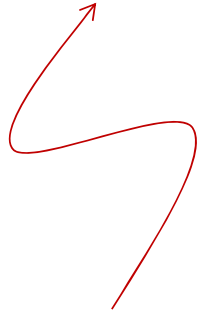
Termination Detection

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
    return TRUE;
else
    return FALSE;
```

Is it ok?

Termination Detection

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;
```

each thread increments this after
completing its for loop

Startup (1)

- When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.

- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

Startup (2)

- One or more threads may finish allocating their queues before some other threads.

- We need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.

- After all the threads have reached the barrier all the threads in the team can proceed.

```
# pragma omp barrier
```

The Atomic Directive (1)

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.

```
# pragma omp atomic
```

- Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

The Atomic Directive (2)

- Here <op> $+$, $*$, $-$, $/$, $\&$, $\wedge$, $|$, $<<$, or $>>$

- 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.

Critical Sections

- OpenMP provides the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously.

- However, the names are set during compilation, and we want a different critical section for each thread's queue.
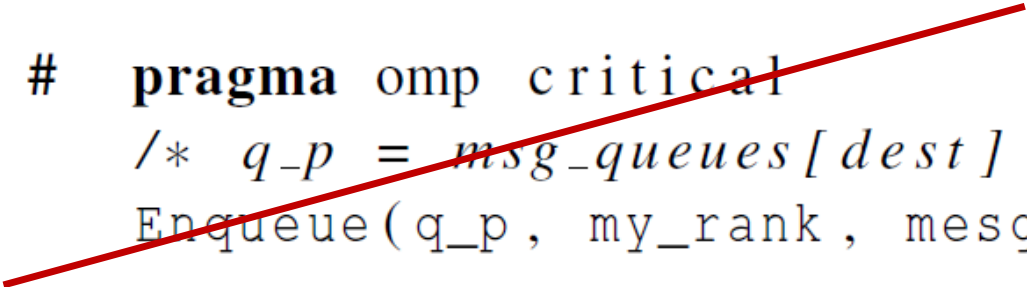
Locks

- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

Locks

```
/* Executed by one thread */
Initialize the lock data structure;

. . .

/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;

. . .

/* Executed by one thread */
Destroy the lock data structure;
```

Using Locks in the Message-Passing Program

```
#   pragma omp critical
    /* q_p = msg_queues[dest] */
    Enqueue(q_p, my_rank, mesg);
```

```
    /* q_p = msg_queues[dest] */
    omp_set_lock(&q_p->lock);
    Enqueue(q_p, my_rank, mesg);
    omp_unset_lock(&q_p->lock);
```

Using Locks in the Message-Passing Program

```
#   pragma omp critical
    /*  q_p = msg_queues[my_rank] */
    Dequeue(q_p, &src, &mesg);


    /*  q_p = msg_queues[my_rank] */
    omp_set_lock(&q_p->lock);
    Dequeue(q_p, &src, &mesg);
    omp_unset_lock(&q_p->lock);
```

Some Caveats

1. You shouldn't mix the different types of mutual exclusion for a single critical section.

```
#   pragma omp atomic          #   pragma omp critical
    x += f(y);                      x = g(x);
```

2. There is no guarantee of fairness in mutual exclusion constructs.

```
        while(1) {
            . . .
#           pragma omp critical
            x = g(my_rank);
            . . .
        }
```

Some Caveats

3. It can be dangerous to "nest" mutual exclusion constructs.

```
#   pragma omp critical
    y = f(x);

    . . .
    double f(double x) {
#       pragma omp critical
        z = g(x);   /* z is shared */

        . . .
    }
```

```
#   pragma omp critical(one)
    y = f(x);

    . . .
    double f(double x) {
#       pragma omp critical(two)
        z = g(x);   /* z is global */
        . . .
    }
```

Deadlock

Matrix-vector multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

$=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Matrix-vector multiplication

```
#   pragma omp parallel for num_threads(thread_count)    \
        default(none) private(i, j) shared(A, x, y, m, n)
    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }
```

Run-times and efficiencies
of matrix-vector multiplication
(times are in seconds)

| Threads | Matrix Dimension | | | | | |
| | $8,000,000 \times 8$ | | 8000×8000 | | $8 \times 8,000,000$ | |
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275