OpenMP Directives:

THE*parallel for*DIRECTIVE

- As an alternative to our explicit parallelization of the trapezoidal rule, OpenMP provides the parallel for directive. Using it, we can parallelize the serial trapezoidal rule

```
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;
```

- By simply placing a directive immediately before the for loop:

```
h       = (b-a)/n;
approx = (f(a) + f(b))/2.0;
# pragmaomp parallel for num threads(thread count) n
reduction(+: approx)
for (i = 1; i <= n-1; i++)
        approx += f(a + i*h);
        approx = h*approx;
```

- Like the ***parallel*** directive, the ***parallel for*** directive forks a team of threads to execute the following structured block. However, the structured block following the ***parallel for*** directive must be a ***for***loop.
- With the ***parallel for*** directive the system parallelizes the ***for*** loop by dividing the iterations of the loop among the threads.
- The ***parallel for*** directive is therefore very different from the ***parallel*** directive, because in a block that is preceded by a ***parallel*** directive.
- Most systems use roughly a block partitioning, that is, if there are
  m iterations, then roughly the first m/thread_count are assigned to thread 0, thenext m/thread_count are assigned to thread 1, and so on.
- The default scope for all variables in a ***parallel*** directive is shared, but in our ***parallel for*** if the loop variable i were shared,

the variable update, i++, would also be an unprotected critical section. Hence, in a loop that is parallelized with a ***parallel for*** directive, the default scope of the loop variable is ***private***; in our code, each thread in the team has its own copy of ***i***.

Caveats

- It may be possible to parallelize a serial program that consists of one large ***for*** loop by just adding a single ***parallel for*** directive. It may bepossible to incrementally parallelize a serial program that has many ***for*** loops bysuccessively placing ***parallel for*** directives before each loop.

- There are several caveats associated with the use of the ***parallel for*** directive. OpenMP will only parallelize ***for*** loops. It won't parallelize ***while loops*** or ***do-while*** loops. This may not seem to be too much of a limitation, since any code that uses a ***while loop*** or
  a***do-while*** loop can be converted to equivalent code that uses a for loop instead.

- 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 (. . . ; . . . ; . . .)),
  and
- prior to execution of the loop.

- For example, the "infinite loop"

```
for ( ; ; )
{
        . . .
}
```

  cannot be parallelized.

- Similarly, the loop

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

- 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.
- In fact, OpenMP will only parallelize for loops that are in canonical form. Loops in canonical form take one of the forms shown in fig below. The variables and expressions in this template are subject to some fairly obvious restrictions:
  - 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.
  - 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.

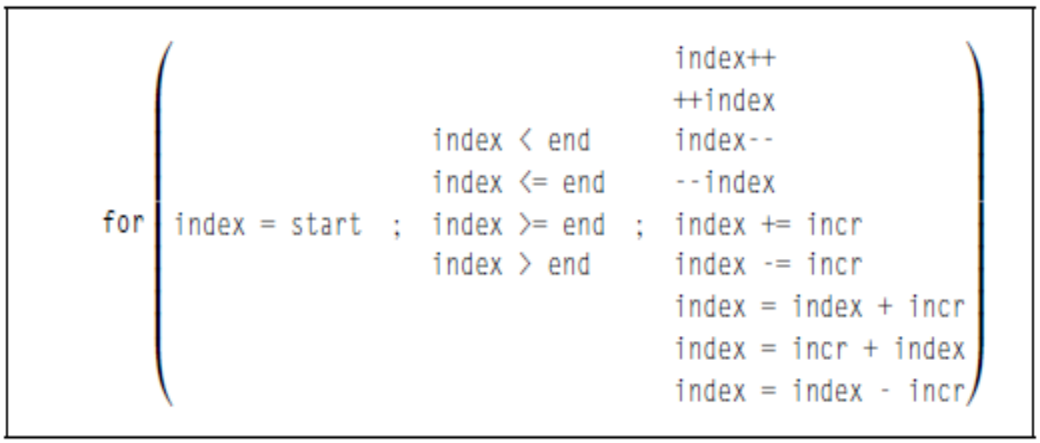

Fig: Legal forms for parallelizable for statements.