CISC 372: Parallel Computing
OpenMP, Part 2

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Private vs. shared variables

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be
    - private: all threads get their own private copy of the variable (in addition to the original), or

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be
    - private: all threads get their own private copy of the variable (in addition to the original), or
    - shared: one shared variable

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be
    - private: all threads get their own private copy of the variable (in addition to the original), or
    - shared: one shared variable
- specify what you want by clauses of the form
  - `shared(u1,u2,...)`
  - `private(v1,v2,...)`

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be
    - private: all threads get their own private copy of the variable (in addition to the original), or
    - shared: one shared variable
- specify what you want by clauses of the form
  - `shared(u1,u2,...)`
  - `private(v1,v2,...)`
- some (obvious) points
  - the `u1`,`u2`,... and `v1`,`v2`,... must all be visible at this point

Private vs. shared variables

- if a variable is declared within the parallel region...
  - all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region...
  - you have a choice: variable can be
    - private: all threads get their own private copy of the variable (in addition to the original), or
    - shared: one shared variable
- specify what you want by clauses of the form
  - `shared(u1,u2,...)`
  - `private(v1,v2,...)`
- some (obvious) points
  - the `u1`,`u2`,... and `v1`,`v2`,... must all be visible at this point
  - a variable cannot be both `shared` and `private`

The `default` clause

The `default` clause

- sets the default for private vs. shared in the parallel region

The `default` clause

- sets the default for private vs. shared in the parallel region
- `default(none)`
  - no default
  - every variable used in parallel region must be explicitly listed in `shared` or `private`

The `default` clause

- sets the default for private vs. shared in the parallel region
- `default(none)`
  - no default
  - every variable used in parallel region must be explicitly listed in `shared` or `private`
- `default(shared)`
  - if not listed, the variable is shared

The `default` clause

- sets the default for private vs. shared in the parallel region
- `default(none)`
  - no default
  - every variable used in parallel region must be explicitly listed in `shared` or `private`
- `default(shared)`
  - if not listed, the variable is shared
- there are rules specifying what happens if you don't have a default clause
  - but ignore them for now
  - explicitly declare every variable used in the region as either `private` or `shared`

hello2.c

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

int main (int argc, char *argv[]) {
  int nthreads, tid;

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0) { // only master
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } // end of parallel region
}
```

`num_threads()`: requesting the number of threads

`num_threads()`: requesting the number of threads

- you can request that the team have a specified number of threads

`num_threads()`: requesting the number of threads

- you can request that the team have a specified number of threads
- clause: `num_threads(expr)`
  - where `expr` is an expression which evaluates to a positive integer

`num_threads()`: requesting the number of threads

- you can request that the team have a specified number of threads
- clause: `num_threads(expr)`
  - where `expr` is an expression which evaluates to a positive integer
- the runtime system may give you the requested number of threads
  - or it may give you fewer

`num_threads()`: requesting the number of threads

- you can request that the team have a specified number of threads
- clause: `num_threads(expr)`
  - where `expr` is an expression which evaluates to a positive integer
- the runtime system may give you the requested number of threads
  - or it may give you fewer
- if you really need to know how many there are, ask
  - `int omp_get_num_threads()`

Initializing private variables

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region
- `x` is declared private

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region
- `x` is declared private
- when control enters the parallel region, what is the initial value of `x`?

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region
- `x` is declared private
- when control enters the parallel region, what is the initial value of `x`?
- answer: undefined, `even on master thread`
- try it! see `initial.c`

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region
- `x` is declared private
- when control enters the parallel region, what is the initial value of `x`?
- answer: undefined, `even on master thread`
- try it! see `initial.c`
- if you want the private `x` to be initialized with the value the original `x` had:
  - use `firstprivate`

Initializing private variables

- suppose `x` is an integer variable declared before entering a parallel region
- value of `x` is 5 upon reaching the parallel region
- `x` is declared private
- when control enters the parallel region, what is the initial value of `x`?
- answer: undefined, `even on master thread`
- try it! see `initial.c`
- if you want the private `x` to be initialized with the value the original `x` had:
  - use `firstprivate`
- clause: `firstprivate(v1,v2,...)`
  - declares `v1`,`v2`,... to be not only private, but to be initialized with global value
  - see `initial2.c`

Constructs, regions, and modification of original variable of `private`

Constructs, regions, and modification of original variable of `private`

OpenMP 4.0 Sec. 1.2.2:

> **construct**. *An OpenMP executable directive . . . and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.*

Constructs, regions, and modification of original variable of `private`

OpenMP 4.0 Sec. 1.2.2:

> **construct**. *An OpenMP executable directive . . . and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.*
>
> **region**. *All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.*

Constructs, regions, and modification of original variable of `private`

OpenMP 4.0 Sec. 1.2.2:

> **construct**. *An OpenMP executable directive . . . and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.*
>
> **region**. *All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.*

Sec. 2.14.3.3, `private` clause:

> *The value . . . of the original list item will change only*
>
> - *if accessed and modified via pointer,*
> - *if possibly accessed in the region but outside of the construct, [or]*
> - *as a side effect of directives or clauses[.]*

Constructs, regions, and modification of original variable of `private`

OpenMP 4.0 Sec. 1.2.2:

> **construct**. *An OpenMP executable directive . . . and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, in the lexical extent of an executable directive.*
>
> **region**. *All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.*

Sec. 2.14.3.3, `private` clause:

> *The value . . . of the original list item will change only*
>
> - *if accessed and modified via pointer,*
> - *if possibly accessed in the region but outside of the construct, [or]*
> - *as a side effect of directives or clauses[.]*

- beware! when you access a "private" variable outside of the construct
  - you may be accessing the original copy; see `semiprivate.c`

Work-sharing

Work-sharing

- you usually don't want all threads in the team to do the same thing

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team
  - kinds of work-sharing constructs

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team
  - kinds of work-sharing constructs
    - for loops: distribute iterations to team members

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team
  - kinds of work-sharing constructs
    - for loops: distribute iterations to team members
    - sections: distribute independent code bocks (work units)

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team
  - kinds of work-sharing constructs
    - for loops: distribute iterations to team members
    - sections: distribute independent code bocks (work units)
    - single: let only one thread execute a block

Worksharing constructs: `for` loops

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team
  - barrier at end of loop

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team
  - barrier at end of loop
  - in general, everything else is unspecified
    - how the iterations are distributed among the team members
    - the order in which the iterations are executed
    - what happens concurrently

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team
  - barrier at end of loop
  - in general, everything else is unspecified
    - how the iterations are distributed among the team members
    - the order in which the iterations are executed
    - what happens concurrently
- syntactic restrictions on the `for` statement:

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team
  - barrier at end of loop
  - in general, everything else is unspecified
    - how the iterations are distributed among the team members
    - the order in which the iterations are executed
    - what happens concurrently
- syntactic restrictions on the `for` statement:
  - `init-expr`: `var = expr`, integer type
  - `relop` is one of: `<`, `<=`, `>`, `>=`
  - `b` is a loop-invariant integer expression
  - `incr-expr` has one of a few forms; see OpenMP 4.0 Standard, Section 2.6

Allowed forms for increment expression in `for` loop

- `++var`
- `var++`
- `--var`
- `var--`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

Allowed forms for increment expression in `for` loop

- `++var`
- `var++`
- `--var`
- `var--`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

where `incr` is a loop invariant integer expression

Allowed forms for increment expression in `for` loop

- `++var`
- `var++`
- `--var`
- `var--`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

where `incr` is a loop invariant integer expression

- i.e., throughout one execution of the loop
  - `incr` will have the same value each time control reaches the top of the loop

Allowed forms for increment expression in `for` loop

- `++var`
- `var++`
- `--var`
- `var--`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

where `incr` is a loop invariant integer expression

- i.e., throughout one execution of the loop
  - `incr` will have the same value each time control reaches the top of the loop
- however `incr` could have different values in different loop executions

Loop invariant expressions

```
for (i=0; i<n; i++) {
  /* no writes to n */
}
```

Is `n` loop invariant?

Loop invariant expressions

```
for (i=0; i<n; i++) {
  /* no writes to n */
}
```

Is `n` loop invariant? Yes

```
for (i=0; i<n; i++) {
  for (j=0; j<i; j++) {
    /* no writes to i,j,n */
  }
}
```

Is `i` invariant of inner loop?

Loop invariant expressions

```
for (i=0; i<n; i++) {
  /* no writes to n */
}
```

Is `n` loop invariant? Yes

```
for (i=0; i<n; i++) {
  for (j=0; j<i; j++) {
    /* no writes to i,j,n */
  }
}
```

Is `i` invariant of inner loop? Yes

```
for (i=0; i<n; i++) {
  int max = i;
  for (j=0; j<max; j++) {
    ...
    if (a[j]>max) max = a[j];
    ...
  }
}
```

Is `max` invariant of inner loop?

Loop invariant expressions

```
for (i=0; i<n; i++) {
  /* no writes to n */
}
```

Is `n` loop invariant? Yes

```
for (i=0; i<n; i++) {
  for (j=0; j<i; j++) {
    /* no writes to i,j,n */
  }
}
```

Is `i` invariant of inner loop? Yes

```
for (i=0; i<n; i++) {
  int max = i;
  for (j=0; j<max; j++) {
    ...
    if (a[j]>max) max = a[j];
    ...
  }
}
```

Is `max` invariant of inner loop?
Probably not

Combining constructs

Combining constructs

```
#omp parallel
#omp for
S
```

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly
- this is useful when you have just one construct inside a parallel region

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly
- this is useful when you have just one construct inside a parallel region
- clauses must be unambiguous

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly
- this is useful when you have just one construct inside a parallel region
- clauses must be unambiguous
  - if a clause is applicable only to `parallel`, fine

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly
- this is useful when you have just one construct inside a parallel region
- clauses must be unambiguous
  - if a clause is applicable only to `parallel`, fine
  - if a clause if appicable only to `for`, fine

Combining constructs

```
#omp parallel
#omp for
S
```

can be abbreviated

```
#omp parallel for
S
```

- other constructs can be abbreviated similarly
- this is useful when you have just one construct inside a parallel region
- clauses must be unambiguous
  - if a clause is applicable only to `parallel`, fine
  - if a clause if appicable only to `for`, fine
  - if a clause is applicable to `parallel` and `for`
    - if it has the same meaning for each (e.g., `shared`), no problem
    - otherwise, undefined behavior

Clauses for the `for` loop directive

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end
- `reduction(...)`
  - apply some associative and commutative operation (like +) across all iterations for some variable

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end
- `reduction(...)`
  - apply some associative and commutative operation (like +) across all iterations for some variable
- `ordered`: declares that an `ordered` construct may occur in loop body

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end
- `reduction(...)`
  - apply some associative and commutative operation (like +) across all iterations for some variable
- `ordered`: declares that an `ordered` construct may occur in loop body
- `schedule`: options to control how iterations are distributed to threads

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end
- `reduction(...)`
  - apply some associative and commutative operation (like +) across all iterations for some variable
- `ordered`: declares that an `ordered` construct may occur in loop body
- `schedule`: options to control how iterations are distributed to threads
- `nowait`: remove the barrier at the end of the loop

Clauses for the `for` loop directive, cont.

- `collapse(`n`)`: apply directive to next n loops in a loop nest

Clauses for the `for` loop directive, cont.

- `collapse(n)`: apply directive to next n loops in a loop nest
  - n is an expression that evaluates to a positive integer
  - iteration space of the n loops is collapsed into a single space
  - the iterations in the resulting space are distributed to threads
  - all initializers, incremeters, and conditions must be invariant under all loops
    - i.e., they must remain constant throughout the entire loop nest

Clauses for the `for` loop directive, cont.

- `collapse(n)`: apply directive to next n loops in a loop nest
  - n is an expression that evaluates to a positive integer
  - iteration space of the n loops is collapsed into a single space
  - the iterations in the resulting space are distributed to threads
  - all initializers, incremeters, and conditions must be invariant under all loops
    - i.e., they must remain constant throughout the entire loop nest

Correct:

```
#pragma omp for collapse(2)
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    a[i][j] = 2*b[i][j];
```

Clauses for the `for` loop directive, cont.

- `collapse(n)`: apply directive to next n loops in a loop nest
  - n is an expression that evaluates to a positive integer
  - iteration space of the n loops is collapsed into a single space
  - the iterations in the resulting space are distributed to threads
  - all initializers, incremeters, and conditions must be invariant under all loops
    - i.e., they must remain constant throughout the entire loop nest

Correct:

```
#pragma omp for collapse(2)
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    a[i][j] = 2*b[i][j];
```

Incorrect:

```
#pragma omp for collapse(2)
for (i=0; i<n; i++)
  for (j=i; j<m; j++)
    a[i][j] = 2*b[i][j];
```

Question 1

Assume `a` and `b` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=0; i<n && a[i]>0; i++)
  b[i] = b[i] - a[i];
```

Question 1

Assume `a` and `b` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=0; i<n && a[i]>0; i++)
  b[i] = b[i] - a[i];
```

No (non-standard condition)

Question 2

Assume `a` and `b` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i++)
  b[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

Question 2

Assume `a` and `b` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i++)
  b[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

No (data race)

Question 3

Assume `a`, `b`, and `c` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i++)
  c[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

Question 3

Assume `a`, `b`, and `c` are disjoint arrays.
Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i++)
  c[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

Yes

Question 4

Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i+=k)
  c[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

Question 4

Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=1; i<n; i+=k)
  c[i] = b[i] - a[i] + b[i-1] - a[i-1]
```

Yes