Lecture 5

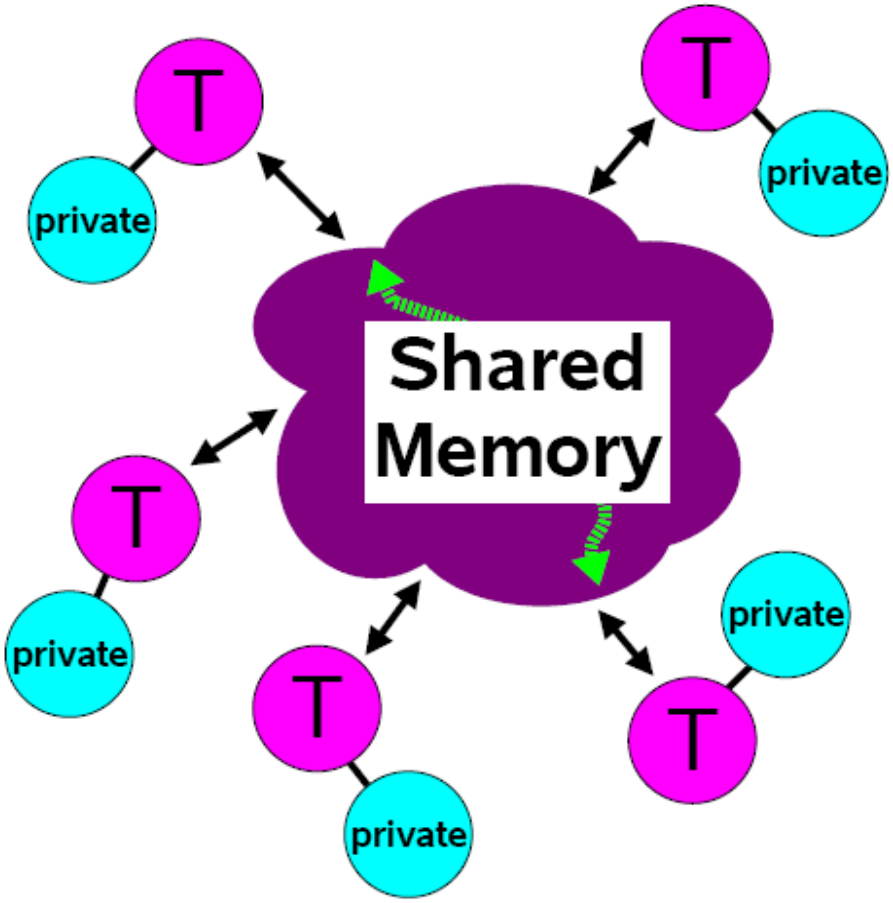
OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

Data Scope Clauses

- **SHARED (list)**
- **PRIVATE (list)**
- **FIRSTPRIVATE (list)**
- **LASTPRIVATE (list)**
- **DEFAULT (list)**
- **THREADPRIVATE (list)** (it is actually a directive, not clause)
- **COPYIN (list)**
- **REDUCTION (operator | intrinsic : list)**

Review: Shared Memory Model



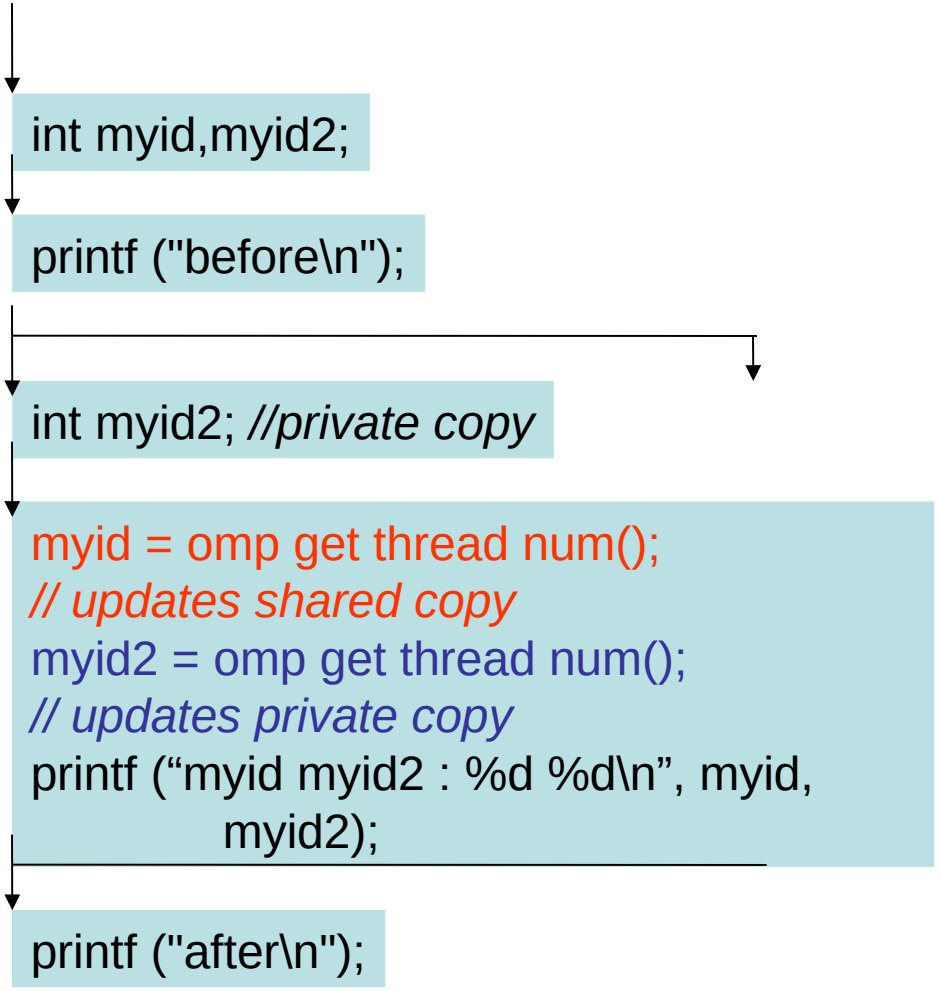
Programming Model

- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can be accessed only by the threads that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

Data Scope Example (shared vs private)

All sample codes are in /home/syam/ces745/openmp/Fortran/data-scope/

```
#include <omp.h>
main()
{
    int  myid, myid2;
    printf ("before\n");
    #pragma omp parallel private(myid2)
        {
        myid = omp_get_thread_num();
        myid2 = omp_get_thread_num();
        printf ("myid myid2: %d %d \n", myid,
              myid2);
        }
    printf ("after\n");
}
```



```
[~/ces745/openmp/Fortran/data-scope] ./scope
 before
 myid myid2 :          5          0
 myid myid2 :         26          8
 myid myid2 :         21         13
 myid myid2 :         21         17
 myid myid2 :         21         24
 myid myid2 :         21         30
 myid myid2 :         21         15
 ……..
 ……..

 myid myid2 :         21         31
 myid myid2 :         21         14
 myid myid2 :         21         25
 myid myid2 :         21         12
 myid myid2 :         21         11
 myid myid2 :         21          1
 myid myid2 :         21         18
 after
[~/ces745/openmp/Fortran/data-scope]
```

Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Data Environment:
Example storage attributes

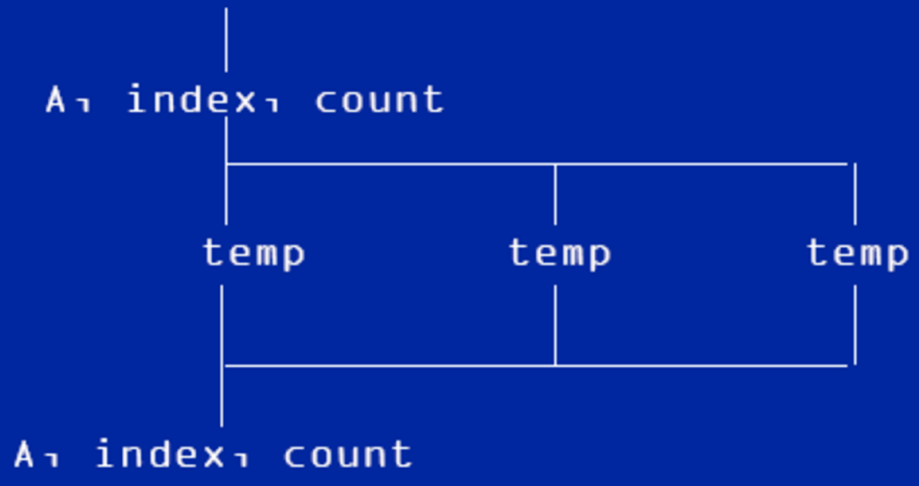
```
#include <omp.h>
float A[10];

main() {
 int index[10];
 input();
 #pragma omp parallel
   work(index);
 printf ("%d\n",index[1]);
}
```

```
void work(int* index1)
{
 float temp[10];
 static int count;
 ...
}
```

A, index[] and count are shared by all threads.

temp and index1 are local to each thread.



Data Environment:
Changing storage attributes

- One can selectively change storage attributes constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE*
- The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
  - LASTPRIVATE
- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

All the clauses on this page only apply to the *lexical extent* of the OpenMP construct.

Private Clause

- **private(*var*) creates a local copy of *var* for each thread**
  - The value is uninitialized
  - Private copy is **not** storage associated with the original

```
#include <omp.h>
// Wrong code
main()
{
int IS = 0;

#pragma omp parallel for private(IS)
    for (int j=0; j<1000; j++)
        IS = IS + j;

printf (“%d\n”, IS);
}
```

IS was not initialized

/home/syam/ces745/openmp/Fortran/data-scope/private-11.f90

```
program wrong
use omp_lib
integer :: myid, nthreads
IS=10
!$omp parallel private(IS, myid)
  myid = omp_get_thread_num()
  nthreads = omp_get_num_threads()
  print *, IS
  !$omp do
    do j=1, 10
       IS = IS + j
       print *, j, IS, nthreads, myid
    end do
  !$omp end do
 !$omp end parallel
IS = IS + 1
print *, IS
end
```

```
#include <omp.h>
// Buggy code
main()
{
int myid, nthreads;
int IS=10;
#pragma omp parallel private(IS, myid)
{
  myid = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf (“%d\n”, IS);
  #pragma omp for
    for (int j=1; j<=10; j++) {
       IS = IS + j;
       printf (“%d %d %d %d\n”), j, IS,
              nthreads, myid;
    }
}
IS = IS + 1;
printf (“%d\n”, IS);
}
```

[~] export OMP_NUM_THREADS=4
[~] ./private-11

```
        0
        0
        4    4205628        4        1
        0
        1    4205625        4        0
        5    4205633        4        1
        2    4205627        4        0
        3    4205630        4        0
        6    4205639        4        1
        0
        9    4205633        4        3
       10     4205643        4        3
        7    4205631        4        2
        8    4205639        4        2
       11
```

Different IS from threads

Firstprivate Clause

- **Firstprivate is a special case of private**
  - Initializes each private copy with the corresponding value from the master thread.

```
#include <omp.h>
// Wrong code, slightly better
main()
{
int IS = 0;

#pragma omp parallel for firstprivate(IS)
    for (int j=0; j<1000; j++)
        IS = IS + j;

printf ("%d\n", IS);
}
```

Each thread will get its own IS initialized to zero.

/home/syam/ces745/openmp/Fortran/data-scope

private-3.f90

```
program good
use omp_lib
integer x, y
x = 1
y = 2
!$omp parallel private(x) firstprivate(y)
  x = 3
  y = y+2
  print *, omp_get_thread_num(), x, y
!$omp end parallel

print *, x, y
end
```

private-4.f90

```
! without use omp_lib
program wrong

integer x, y
x = 1
y = 2
!$omp parallel private(x) firstprivate(y)
  x = 3
  y = y+2
  print *, omp_get_thread_num(), x, y
!$omp end parallel

print *, x, y
end
```

use omp_lib module (head file) is important in intel Fortran compiler

private-3.f90

```
[syam@saw-login1:] ./private-3_ifort
           0           3           4
           3           3           4
           1           3           4
           2           3           4
           1           2
```

private-4.f90

```
[syam@saw-login1:~] ./private-4_ifort
 0.0000000E+00           3           4
 0.0000000E+00           3           4
 0.0000000E+00           3           4
 0.0000000E+00           3           4
           1           2
```

Lastprivate Clause

- **Lastprivate passes the value of a private variable from the last iteration to a global variable**

```
#include <omp.h>
// Still wrong code
main()
{
int IS = 0;

#pragma omp parallel for firstprivate(IS) lastprivate(IS)
    for (int j=0; j<1000; j++)
        IS = IS + j;

printf ("%d\n", IS);
}
```

Each thread will get its own IS initialized to zero.

IS will have the value from the thread handling the last iteration (j=999)

/home/syam/ces745/openmp/Fortran/data-scope/first-last.f90

```
use omp_lib
integer :: myid, nthreads
integer :: x, y
x=10
 print *, "x, j, nthreads, myid"
!$omp parallel do private(myid) firstprivate(x) lastprivate(j, x)
  do j=1, 10
    myid = omp_get_thread_num()
    nthreads = omp_get_num_threads()
    x = x + j
    print *, x, j, nthreads, myid
  end do
!$omp end parallel do

y = x +1
print *, "x, j, y"
print *, x, j, y
end
```

```
#include <omp.h>
main()
{
int myid, nthreads;
int x, y, j;
x=10;
printf ("x, j, nthreads, myid\n");
#pragma omp parallel for private(myid) firstprivate(x) lastprivate(j, x)
  for (j=1; j<=10; j++) {
    myid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    x = x + j;
    printf (“%d %d %d %d\n”, x, j, nthreads, myid);
  }

y = x +1;
printf ("x, j, y\n");
printf (“%d %d %d\n”, x, j, y);
}
```

OpenMP:
Another data environment example

- Here's an example of PRIVATE and FIRSTPRIVATE

```
variables A,B, and C = 1
#pragma omp parallel private(B) firstprivate(C)
```

- Inside this parallel region ...
  - "A" is shared by all threads; equals 1
  - "B" and "C" are local to each thread.
    - B's initial value is undefined
    - C's initial value equals 1
- Outside this parallel region ...
  - Original values of "B" and "C" are restored

OpenMP:
Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to specify)
- To change default: DEFAULT(PRIVATE)
  - *each* variable in *static* extent of the parallel region is made private as if specified in a private clause
  - mostly saves typing
- DEFAULT(NONE): *no* default for variables in static extent. Must list storage attribute for each variable in static extent

Requirement for home assignments!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

OpenMP:
Default Clause Example

```
      itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      ……..
C$OMP END PARALLEL
```

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      ……..
C$OMP END PARALLEL
```

These two codes are equivalent

Changing default scoping rules: C vs Fortran

- Fortran

  default (shared | private | firstprivate | none)

  index variables (serial, and *parallel do*) are private

- C/C++

  default (shared | none)

  - no default (private): many standard C libraries are implemented using macros that reference global variables

  - serial loop index variables are shared, *parallel for* index variables are private

Default (none): helps catch scoping errors

Default scoping rules in Fortran

```
subroutine caller(a, n)
Integer n, a(n), i, j, m
m = 3

!$omp parallel do
   do i = 1, n
      do j = 1, 5
         call callee(a(j), m, j)
      end do
   end do
end

subroutine callee(x, y, z)
common /com/ c
Integer x, y, z, c, ii, cnt
save cnt

cnt = cnt +1
do ii = 1, z
    x = y +z
end do
end
```

Variable	Scope	Is Use Safe?	Reason for Scope
a	shared	yes	declared outside par construct
n	shared	yes	declared outside par construct
i	private	yes	parallel loop index variable
j	private	yes	Fortran seq. loop index var
m	shared	yes	declared outside par construct
x	shared	yes	actual param. is a, which is shared
y	shared	yes	actual param. is m, which is shared
z	private	yes	actual param. is j, which is private
c	shared	yes	in a common block
ii	private	yes	local stack var of called subrout
cnt	shared	no	local var of called subrout with save attribute

Default scoping rules in C

```
void caller(int a[ ], int n)
{
   int i, j, m=3;

#pragma omp parallel for
   for ( i = 0; i<n; i++){
      int k = m;
      for(j=1; j<=5; j++){
         callee(&a[i], &k, j);
      }
   }
extern int c;

void callee(int *x, int *y, int z)
{
  int ii;
  static int cnt;

   cnt++;
   for(ii=0; ii<z; ii++){
     *x = *y +c;
}
```

Variable	Scope	Is Use Safe?	Reason dor Scope
a	shared	yes	declared outside par construct
n	shared	yes	declared outside par construct
i	private	yes	parallel loop index variable
j	shared	no	loop index var, but not in Fortran
m	shared	yes	declared outside par construct
k	private	yes	auto var declared inside par constr.
x	private	yes	Value parameter
*x	shared	yes	actual param. is a, which is shared
y	private	yes	Value parameter
*y	private	yes	actual param. is k, which is private
z	private	yes	Value parameter
c	shared	yes	declared as extern
ii	private	yes	local stack var of called subrout
cnt	shared	no	declared as static

OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - reduction (op : list)
- The variables in "list" must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
  - pair wise "op" is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

reduction(operator|intrinsic:var1[,var2])

- Allows safe global calculation or comparison.
- A private copy of each listed variable is created and initialized depending on operator or intrinsic (e.g., 0 for +).
- Partial sums, local mins etc. are determined by the threads in parallel.
- Partial sums are added together from one thread at a time to get gobal sum.
- Local mins are compared from one thread at a time to get gmin.

- At the end of the region for which the reduction clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the operator specified.

```
sum = 0.0
c$omp do shared(x) private(i)
c$omp& reduction(+:sum)
do i = 1, N
sum = sum + x(i)
end do
```

```
gmin = 1e30
c$omp do shared(x) private(i)
c$omp& reduction(min:gmin)
do i = 1,N
gmin = min(gmin,x(i))
end do
```

reduction(operator|intrinsic:var1[,var2])

- Listed variables must be shared in the enclosing parallel context.

- In Fortran
  - operator can be **+, *, -, .and., .or., .eqv., .neqv.**
  - intrinsic can be **max, min, iand, ior, ieor**

- In C
  - operator can be **+, *, -, &, ^, |, &&, ||**
  - pointers and reference variables are not allowed in reductions!

OpenMP:
Reduction example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
        int i;
        double ZZ, func(), res=0.0;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++){
                ZZ = func(I);
                res = res + ZZ;
        }
}
```

Reduction Directive

```
#include <omp.h>
// Correct code using reduction
main()
{
int IS = 0;

#pragma omp parallel for shared(IS) reduction(+:IS)
    for (int j=0; j<1000; j++)
        IS = IS + j;

printf (“%d\n”, IS);
}
```

```
[reduction]$ ./para-reduction-2
 Before Par Region: I= 1  J= 1  K= 1

 Thread  0         I= 0  J= 0  K= 0
 Thread  1         I= 1  J= 1  K= 1

 Operator           +     *    MAX
 After Par Region:  I= 2  J= 0  K= 1
[reduction]$
```

```
 PROGRAM REDUCTION
       USE omp_lib
       IMPLICIT NONE
       INTEGER tnumber
       INTEGER I,J,K
       I=1
       J=1
       K=1
       PRINT *, "Before Par Region: I=",I," J=", J," K=",K
       PRINT *, ""
```

/home/syam/ces745/openmp/Fortran/data-scope/reduction

```
!$OMP PARALLEL PRIVATE(tnumber) REDUCTION(+:I) REDUCTION(*:J)
REDUCTION(MAX:K)
       tnumber=OMP_GET_THREAD_NUM()
       I = I+tnumber
       J = J*tnumber
       K = MAX(K,tnumber)
       PRINT *, "Thread ",tnumber, "        I=",I," J=", J," K=",K
!$OMP END PARALLEL

       PRINT *, ""
       print *, "Operator           +     *    MAX"
       PRINT *, "After Par Region:  I=",I," J=", J," K=",K

       END PROGRAM REDUCTION
```

Scope clauses that can appear in a parallel construct

- *shared* and *private* explicitly scope specific variables
- *firstprivate* and *lastprivate* perform initialization and finalizing of privatized variables
- *default* changes the default rules used when variables are not explicitly scoped
- *reduction* explicitly identifies reduction variables

General Properties of Data Scope Clauses

- A variable in a data scoping clause cannot refer to a portion of an object, but must refer to the entire object (e.g., not an individual array element but the entire array)

- A directive may contain multiple shared and private scope clauses; however, an individual variable can appear on at most a single clause (e.g., a variable cannot be declared as both shared and private). Exception: firstprivate & lastprivate

- Data references to variables that occur within the lexical extent of the parallel loop are affected by the data scope clauses; however, references from subroutines invoked from within the loop are not affected

Threadprivate directive

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.

C example (wrong code)

```
int istart, iend;
const int N=10000;

int main () {
int iarray[N];
#pragma omp parallel private(iam,
nthreads, chunk, istart, iend)
  {
  nthreads = omp_get_num_threads();
  iam = omp_get_thread_num();
  istart = iam*(N/n);
  iend =  (iam+1)*(N/n) - 1;
  if (iam == nthreads-1)
      iend = N-1;
  work(iarray);
  }
}
```

```
void work(int *iarray) {

for (int i=istart; i<=iend; i++)
    iarray[i] = i*i;

return;
}
```

Problem:
Private clause applies only within the lexical scope of the parallel region. References to istart, iend from within the work function directly access the shared instances of the global variables which are undefined.

C example (correct code)

```
int istart, iend;
#pragma omp threadprivate(istart,iend)
const int N=10000;

int main () {
int iarray[N];
#pragma omp parallel private(iam,
nthreads, chunk, istart, iend)
  {
  nthreads = omp_get_num_threads();
  iam = omp_get_thread_num();
  istart = iam*(N/n);
  iend =  (iam+1)*(N/n) - 1;
  if (iam == nthreads-1)
      iend = N-1;
  work(iarray);
  }
}
```

```
void work(int *iarray) {

for (int i=istart; i<=iend; i++)
    iarray[i] = i*i;

return;
}
```

Solution:
Using the threadprivate directive. It effectively behaves like a private clause except that it applies to the entire program. Both the main program and the subroutine access the same threadprivate copy of the variables.

Copyin clause

```
#include <stdlib.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void build()
{
   int i;
   work = (float*)malloc( sizeof(float)*size );
   for( i = 0; i < size; ++i ) work[i] = tol;
}
void copyin_example( float t, int n )
{
   tol = t;
   size = n;
   #pragma omp parallel copyin(tol,size)
   {
      build();
   }
}
```

The copyin clause is used to initialize threadprivate data upon entry to a parallel region. The value of the threadprivate variable in the master thread is copied to the threadprivate variable of each other team member.

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - atomic
  - critical section
  - barrier
  - flush
  - ordered
  - single
  - master

We discuss this here, but it really isn't a synchronization construct. It's a work-sharing construct that includes synchronization.

We discus this here, but it really isn't a synchronization construct.

Synchronization categories

- **Mutual Exclusion Synchronization**
  critical
  atomic

- **Event Synchronization**
  barrier
  ordered
  master

- **Custom Synchronization**
  flush
  (lock – runtime library)

OpenMP: Synchronization

- Only one thread at a time can enter a critical section.

```
sum = 0.0;
#omp pragma parallel for shared(A,sum)
  for (i=0; i<Niters; i++)
    {
     #omp critical
        sum = sum + A[i];
    }
```

Critical Directive

```
#include <omp.h>
// Correct code using critical region
main() {
int IS = 0;
#pragma omp parallel shared(IS)
{
   int IS_loc = 0;
   #pragma omp for
   for (int j=0; j<1000; j++)
        IS_loc = IS_loc + j;
   #pragma omp critical
   IS = IS + IS_loc;
}
printf ("%d\n", IS);
}
```

IS_loc will hold partial (per-thread) result

Critical region is needed to correctly add up partial results from different threads

Named Critical Sections

A named critical section must synchronize with other critical sections of the same name but can execute concurrently with critical sections of a different name.

```
        cur_max = min_infinity;
        cur_min = plus_infinity;
#pragma omp parallel for
        {
         for (i=0; i<n; i++)
           {
            if (a[i]>cur_max)
              {
#pragma omp critical (MAXLOCK)
               if (a[i]>cur_max)
                    cur_max = a[i];
               }
                                    if (a[i]<cur_min)
                                      {
                                #pragma omp critical (MINLOCK)
                                       if (a[i]<cur_min)
                                          cur_min = a[i];
                                       }
                                   } // for loop
                                } // parallel for
```

not sufficient; used for efficiency

OpenMP: Synchronization

- **Atomic** is a special case of a critical section that can be used for certain simple statements.
- It applies only to the update of a memory location (the update of X in the following example)

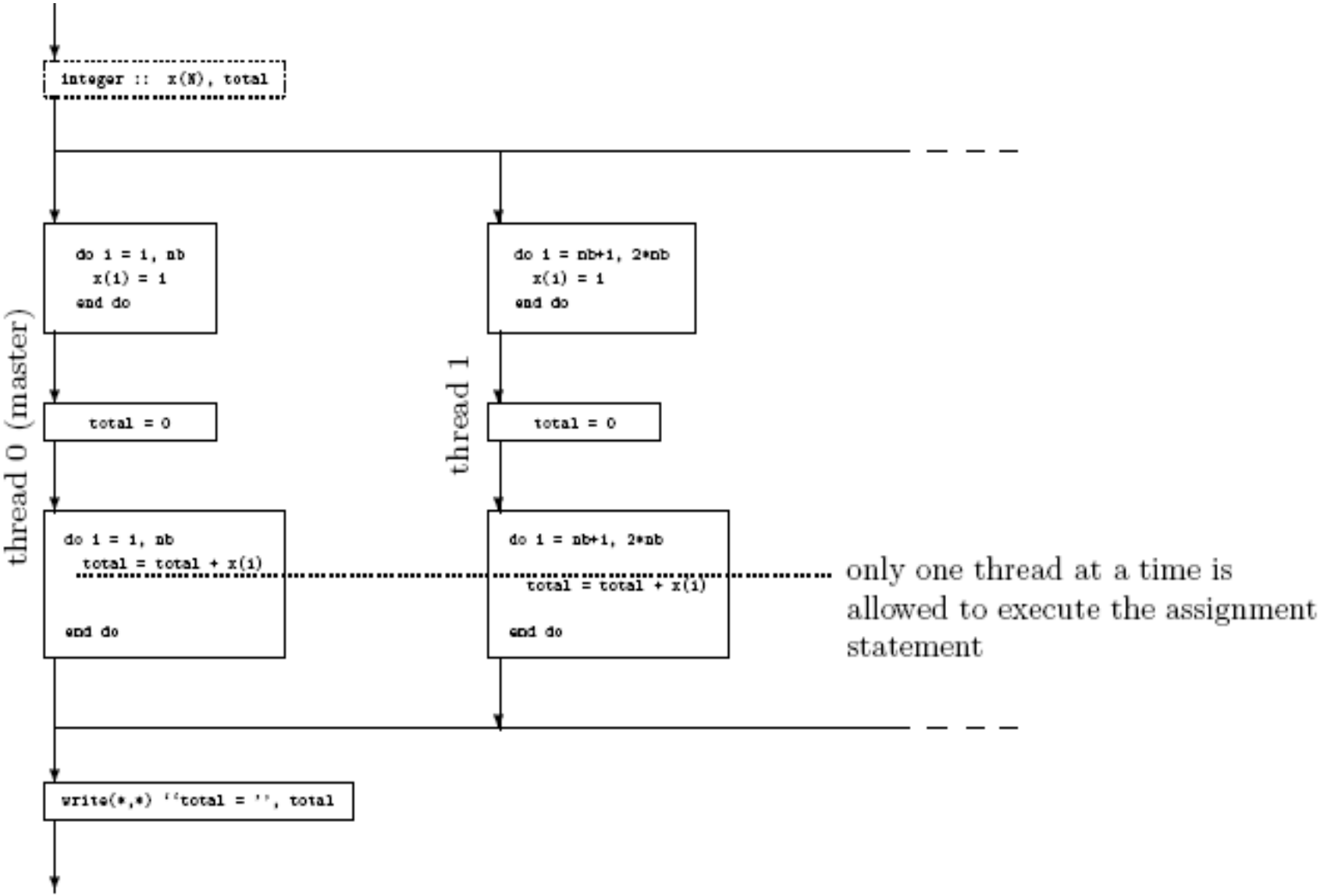
```
X = 0.0;
#pragma omp parallel shared(X)
{
  float B = doit();
  #pragma omp atomic
     X = X + B;
}
```

```
program sharing_par2
use omp_lib
  implicit none
  integer, parameter :: N = 500000
  integer(selected_int_kind(17)) :: x(N)
  integer(selected_int_kind(17)) :: total
  integer :: i
  total = 0
  !$omp parallel
    !$omp do
       do i = 1, N
         x(i) = i
       end do
     !$omp end do
     !$omp do
        do i = 1, N
          !$omp atomic
          total = total + x(i)
        end do
    !$omp end do
  !$omp end parallel
  write(*,*) "total = ", total
end program
```

```
! Parallel code with openmp do directives
! Synchronized with atomic directive
! which give correct answer, but cost more
```

```
[~] ./sharing-atomic-par2
 total =       1250000025000000
[~] ./sharing-atomic-par2
 total =       1250000025000000
[~] ./sharing-atomic-par2
 total =       1250000025000000
```

/home/syam/ces745/openmp/Fortran/synch/sharing-atomic-par2.f90


integer :: x(N), total
thread 0 (master)
thread 1
do i = 1, nb
x(i) = 1
end do
do i = nb+1, 2*nb
x(i) = 1
end do
total = 0
total = 0
do i = 1, nb
total = total + x(i)
end do
do i = nb+1, 2*nb
total = total + x(i)
end do
only one thread at a time is allowed to execute the assignment statement
write(*,*) ''total = '', total

OpenMP: Synchronization

- Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc3(id);
}
```

implicit barrier at the end of a for work-sharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Barriers are used to synchronize the execution of multiple threads within a parallel region, not within a work-sharing construct.

Ensure that a piece of work has been completed before moving on to the next phase

```
!$omp parallel private(index)
        index = generate_next_index()
        do while (index .ne. 0)
            call add_index (index)
            index = generate_next_index()
        enddo

        ! Wait for all the indices to be generated
!$omp barrier
        index = get_next_index()
        do while (index .ne. 0)
            call process_index (index)
            index = get_next_index()
        enddo
!omp end parallel
```

OpenMP: Synchronization

- The single construct denotes a block of code that is executed by only one thread.
- A barrier and a flush are implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```