西北工業大學

Parallel Computing

Threads and OpenMP

NORTH WESTERN POLYTECHNICAL UNIVERSITY

Outline

- Parallel Programming with OpenMP
  - See http://www.nersc.gov/nusers/help/tutorials/openmp/
  - Slides on OpenMP derived from U.Wisconsin tutorial.
- Summary

Parallel Programming in OpenMP

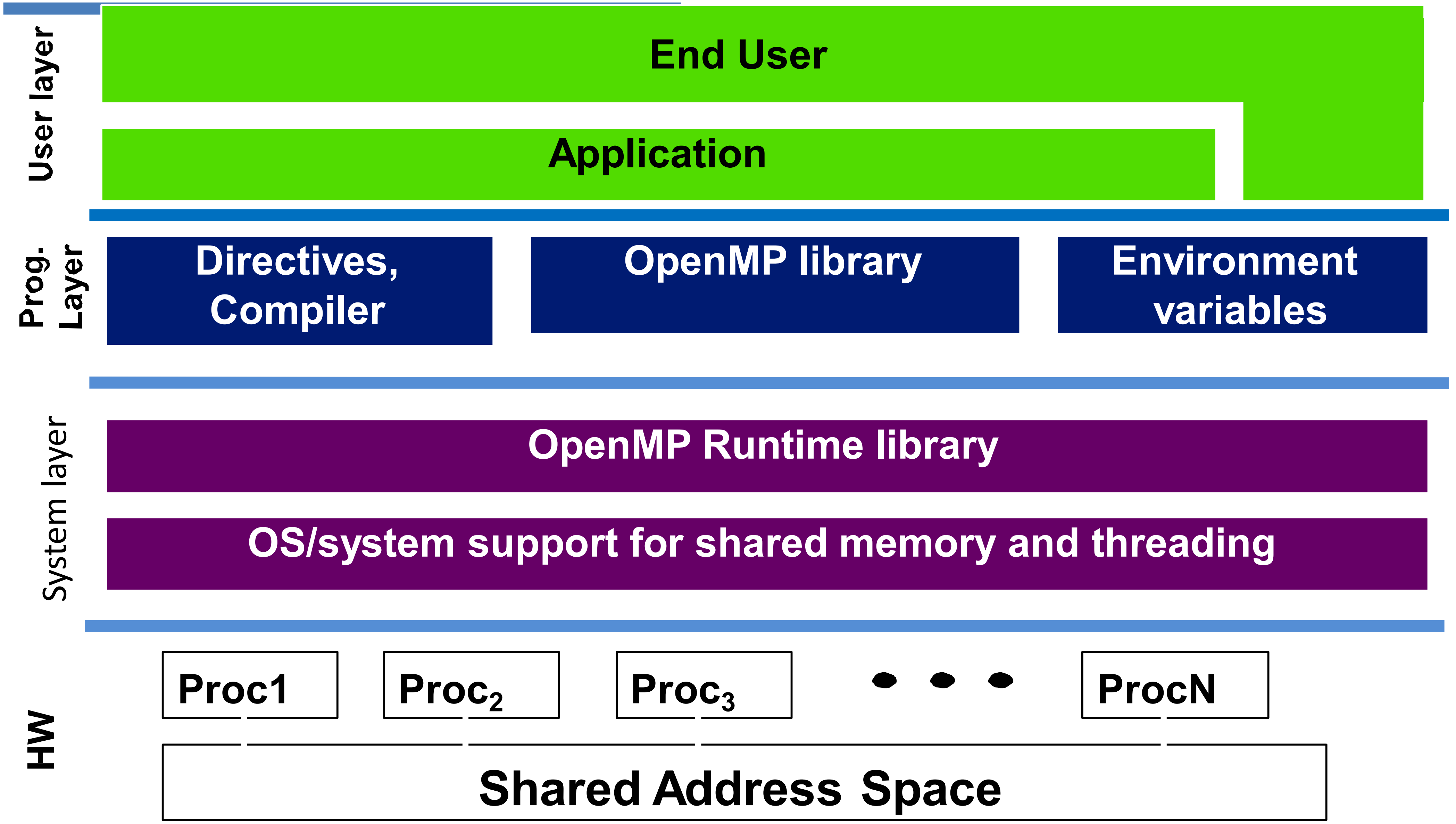
What is OpenMP

- **OpenMP = Open Multi-Parallelism**
- *OpenMP: An API for Writing Multithreaded Applications*
  - A set of compiler directives and library routines for parallel application programmers
  - Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
  - Standardizes established SMP practice + vectorization and heterogeneous device programming

OpenMP Programming Model

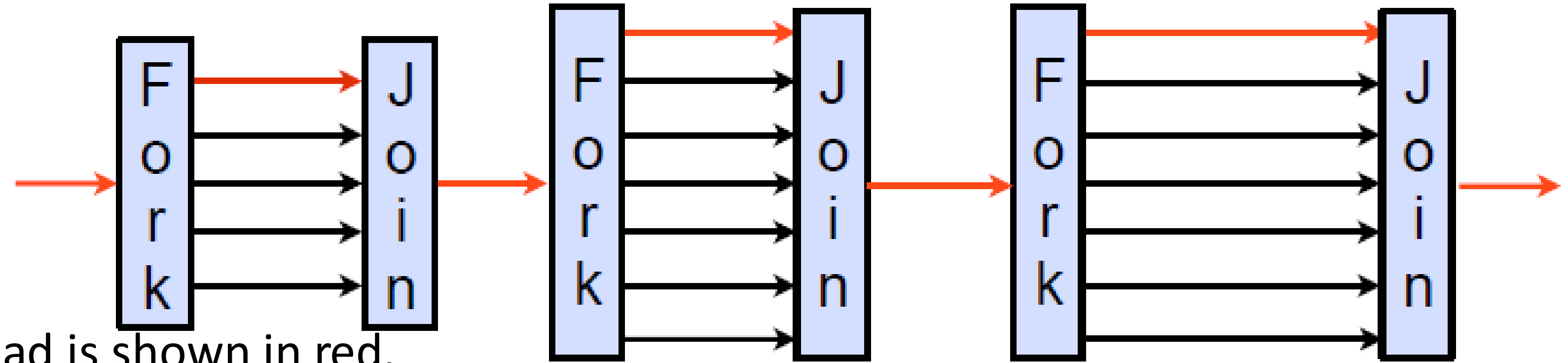
- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

OpenMP Basic Defs: Solution Stack



Fork-Join Parallelism

- OpenMP program begin as a single process: the *master thread*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by **FORK**.
  - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread



Master thread is shown in red.

Our first OpenMP Example

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

int main()
{
   omp_set_num_threads(5);
   #pragma omp parallel
   {
      int ID = omp_get_thread_num();
      printf("Hello (%d)\n", ID);
      printf(" world (%d)\n", ID);
   }
}
```

Set # of threads for OpenMP
In csh
setenv OMP_NUM_THREAD 8

Compile: g++ -fopenmp

hello.c Run: ./a.out

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create 4 threads in parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

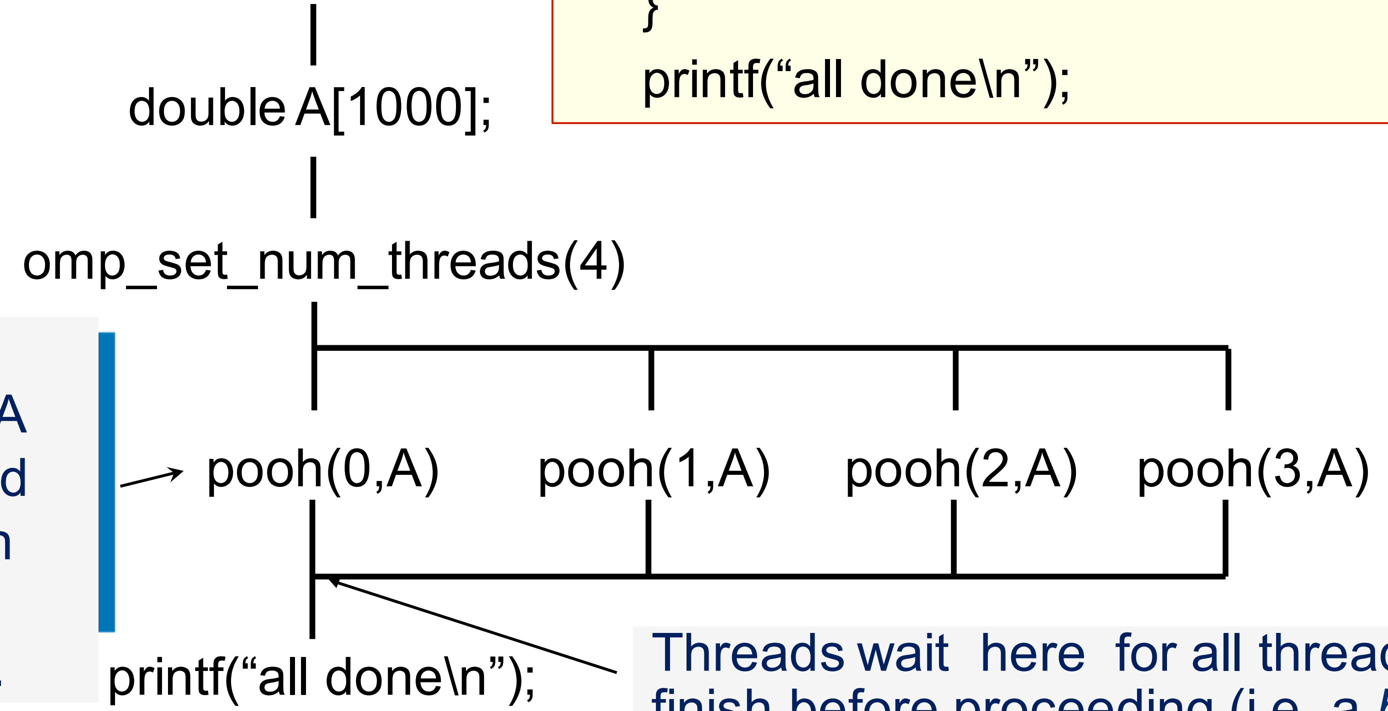
Each thread executes a copy of the code within the structured block

Each thread calls pooh(ID,A) for `ID` = `0` to `3`

Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

OpenMP: what the compiler does

```
#pragma omp parallel
num_threads(4)
{
    foobar ();
}
```

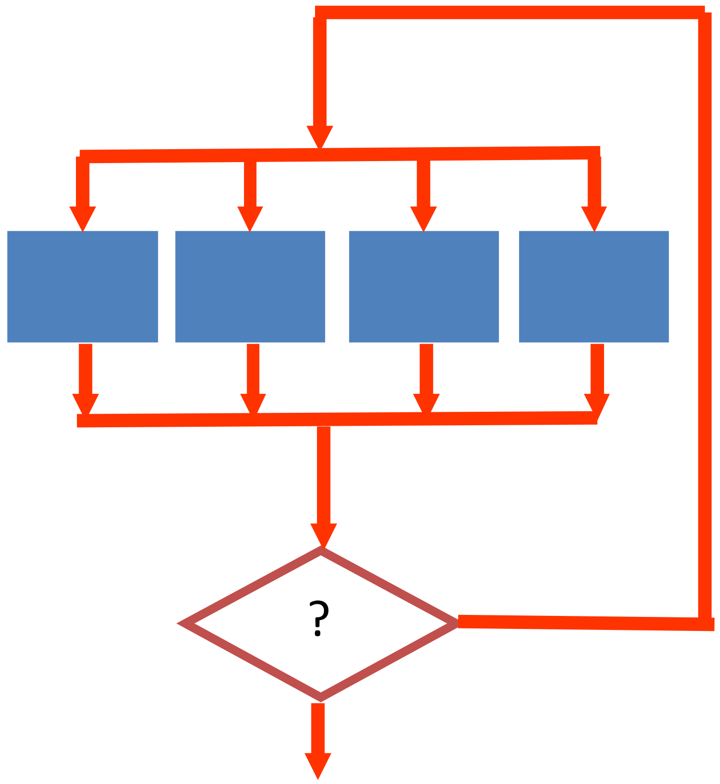
```
void thunk ()
{
     foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - Requires: No data dependencies (reads/write or write/write pairs) between iterations!

- Preprocessor calculates loop bounds for each thread directly from *serial* source



```
#pragma omp parallel for

for( i=0; i < 25; i++ ) {

  printf(“Foo”);

}
```

OpenMP* Pragma Syntax

- Most constructs in OpenMP* are compiler directives pragmas.
  - For C and C++, the pragmas take the form:

#pragma omp *construct [clause [clause]…]*

where *clause is one of the following:*

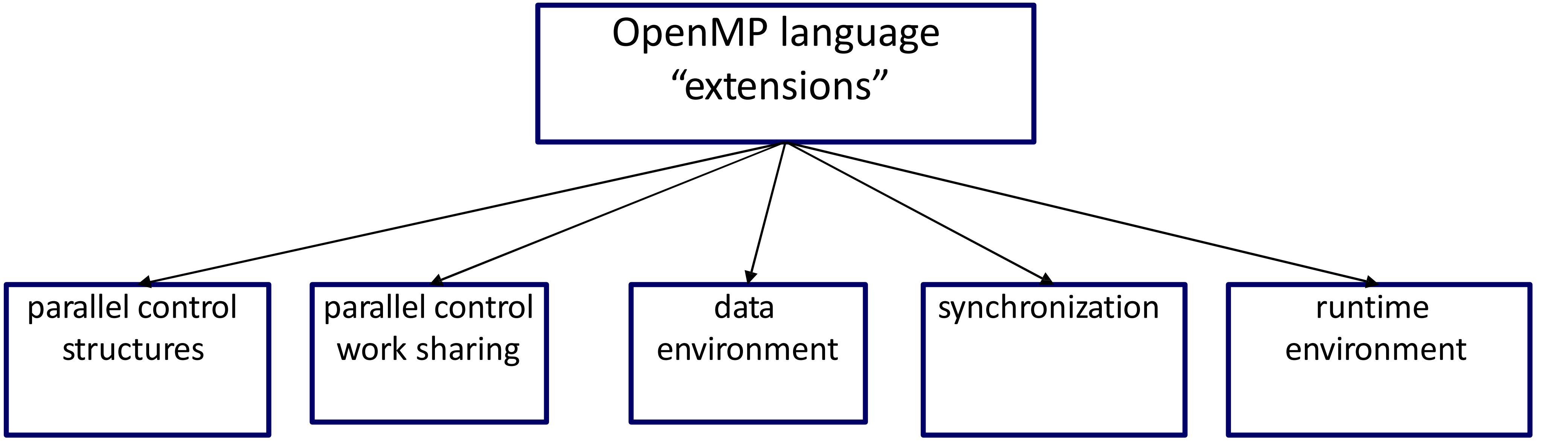
if(*[parallel :] scalar-expression)*
num_threads(*integer-expression)*
default(shared | none)
private(*list)*
firstprivate(*list)*
shared(*list)*
copyin(*list)*
reduction(*reduction-identifier : list)*
proc_bind(master | close | spread)

OpenMP Constructs

OpenMP’s constructs:

1. Parallel Regions
2. Worksharing (for/DO, sections, …)
3. Data Environment (shared, private, …)
4. Synchronization (barrier, flush, …)
5. Runtime functions/environment variables (omp_get_num_threads(), …)
6. tasks

OpenMP Constructs


OpenMP language "extensions"
parallel control structures
parallel control work sharing
data environment
synchronization
runtime environment

- governs flow of control in the program

parallel directive

- distributes work among threads

do
for
sections
directives

- specifies variables as shared or private

shared
private
clauses

- coordinates thread execution

critical
atomic
barrier
directive

- runtime functions
- environment variables

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

- scheduling

static, dynamic, guided

OpenMP: Structured blocks (C/C++)

Most OpenMP constructs apply to structured blocks.

- **Structured block: a block with one point of entry at the top and one point of exit at the bottom.**
- **The only "branches" allowed are STOP statements in Fortran and exit() in C/C++.**

```
#pragma omp parallel
{
more:  do_big_job(id);
         if(++count>1) goto more;
}
   printf(" All done \n");
```

A structured block

```
 if(count==1) goto more;
#pragma omp parallel
{
more:  do_big_job(id);
         if(++count>1) goto done;
}
done:     if(!really_done()) goto more;
```

Not A structured block

Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
  id = omp_thread_num();
  A[id] = big_compute(id);
}
```

```
#pragma omp for
for (I=0;I<N;I++) {
   res[I] = big_calc(I);
   A[I] = B[I] + res[I];
}
```

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    W(id) = garbage(id)
      res(id) = W(id)**2
      if(res(id) goto 10
C$OMP END PARALLEL
```

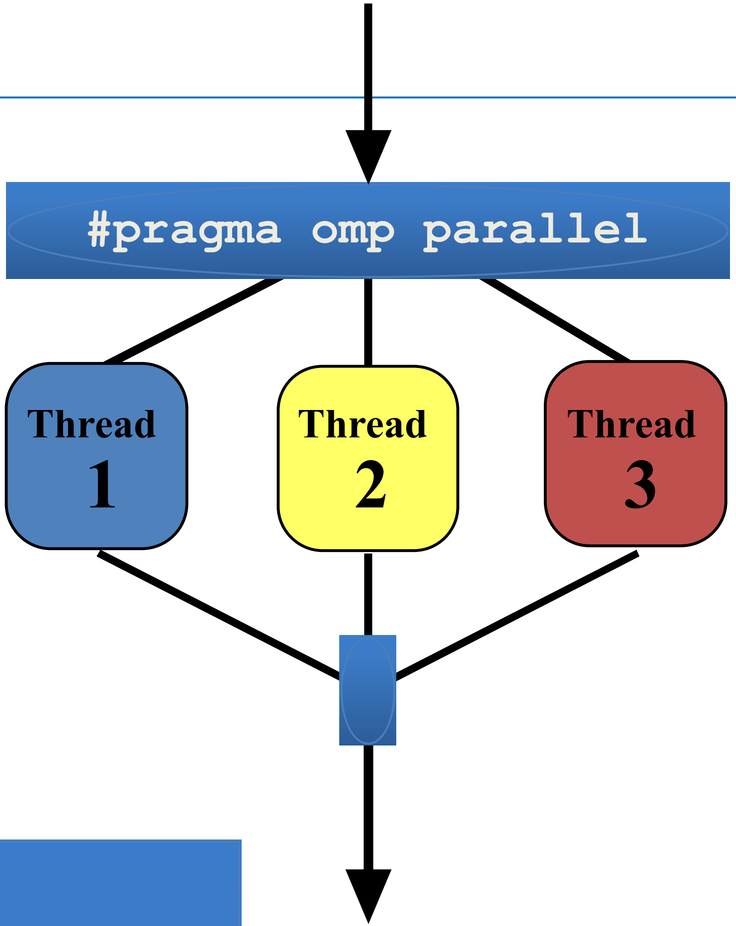
```
C$OMP PARALLEL DO
      do I=1,N
        res(I)=bigComp(I)
      end do
C$OMP END PARALLEL DO
```

PARALLEL PROGRAMMING IN OPENMP:

PARALLEL CONSTRUCT

Parallel Regions

- Defines parallel region over structured block of code
- Threads are created as **‘parallel’** pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise



C/C++ :

```
    #pragma omp parallel
        {
                block
        }
```

Parallel Regions

```
1   #pragma omp parallel
2   {
3       code block
4       work(…);
5   }
```

Line 1	Team of threads formed at parallel region
Lines 3-4	Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region
Line 5	All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads

parallel construct

- The syntax of the **parallel construct**

  When a thread encounters a **parallel construct, a team of threads is created to execute the parallel region.**

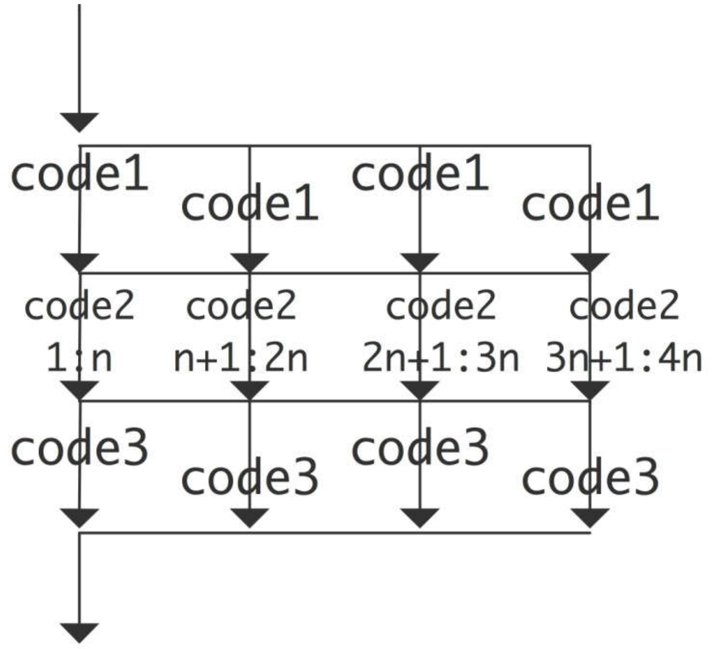
#pragma omp parallel *[clause[[,] clause] ...] new-line*
structured-block

where *clause is one of the following:*

if(*[parallel :] scalar-expression)*
num_threads(*integer-expression)*
default(shared | none)
private(*list)*
firstprivate(*list)*
shared(*list)*
copyin(*list)*
reduction(*reduction-identifier : list)*
proc_bind(master | close | spread)

parallel construct example:

- create a parallel region around the loop, and adjust the loop bounds:
  - Every thread has a thread Id to indentify itself
  - Threads execute same code statements



```
#pragma omp parallel {
    int threadnum = omp_get_thread_num();
    int numthreads = omp_get_num_threads();
    int low = N*threadnum/numthreads;
    int high = N*(threadnum+1)/numthreads;
   for (i=low; i<high; i++) //
              { do something with i }
}
```

Example: sum of n^2

- Sum squares of numbers from 1to N
- Smart way:

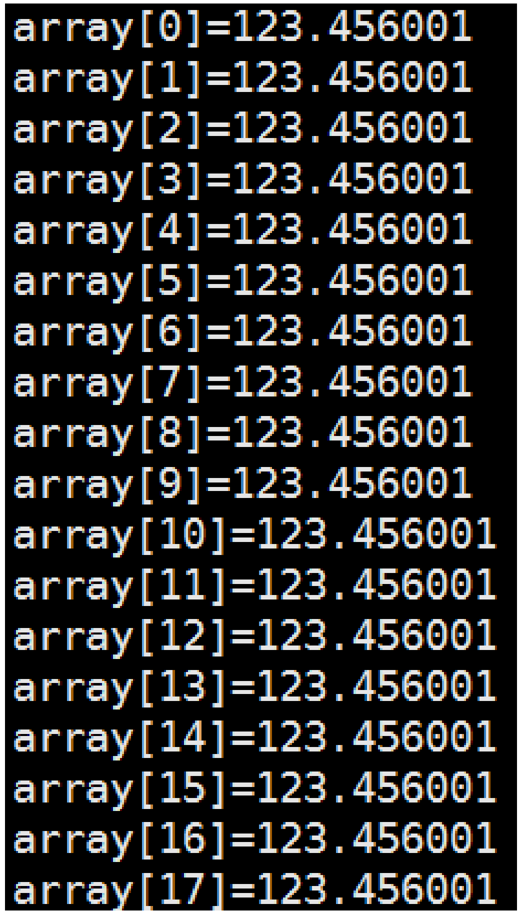
  sum = n(n + 1)(2n + 1)/6
- Dumb way:

  Split the range 1..N across threads
- Have each thread sum its squares Sum the results

```
#include <omp . h>
#include <stdio . h>
. . .
Int sum;
#pragma omp parallel
{
int myid , numthreads ,mysum, first , next , i ;
myid = omp_get_thread_num ( ) ;
numthreads=omp_get_num threads ( ) ;
mysum=0;
first=myidN/ numthreads+1;
ne x t=(myid+1)N/ numthreads+1;
f o r ( i=first ; i<next ; i++)
            mysum+=i*i ;
            sum += mysum; //?????
}
```

- **How Many Threads?**
- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the **IF** clause
  - Setting of the **NUM_THREADS** clause
  - Use of the **omp_set_num_threads()** library function
  - Setting of the **OMP_NUM_THREADS** environment variable
  - Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).
- Threads are numbered from 0 (master thread) to N-1

Example parallel



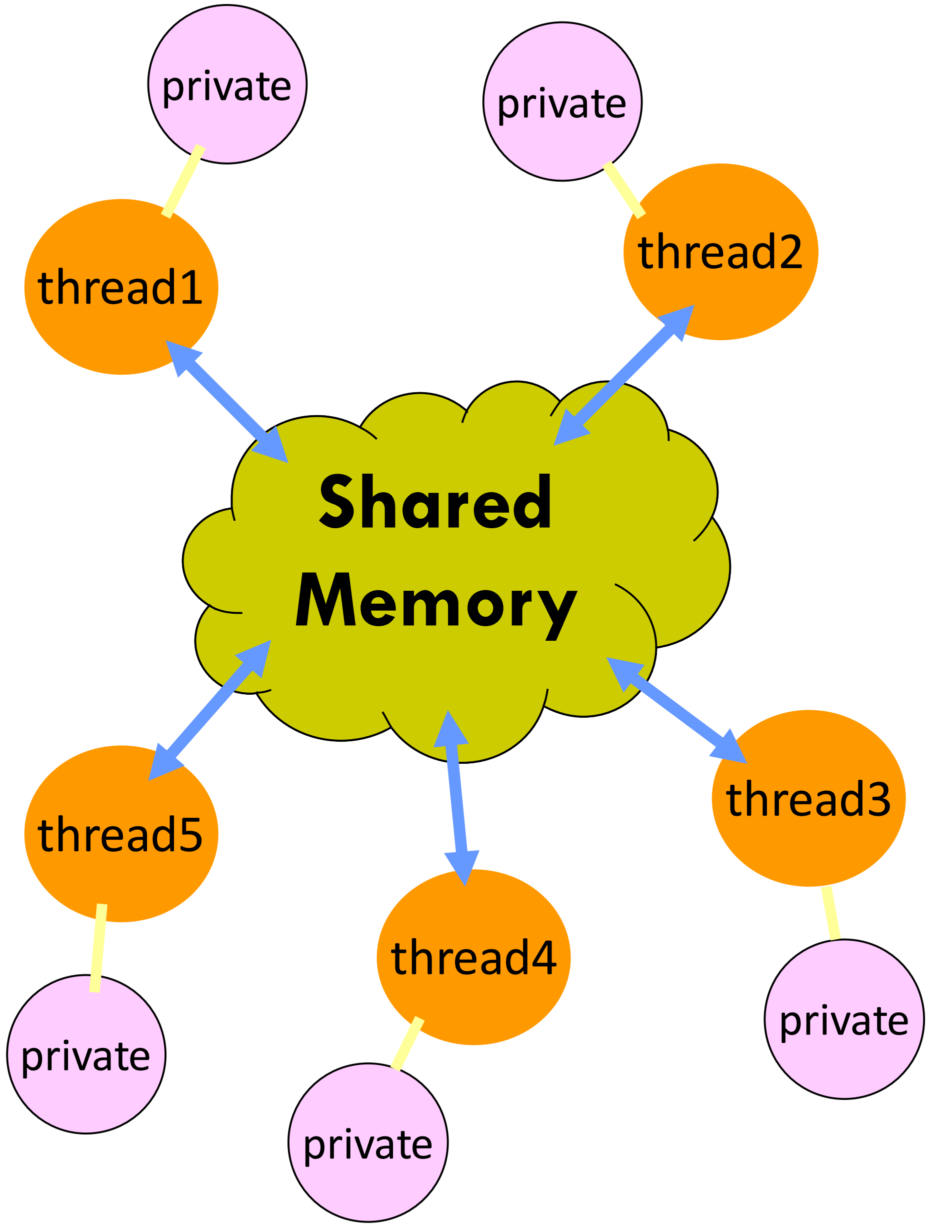
```
array[0]=123.456001
array[1]=123.456001
array[2]=123.456001
array[3]=123.456001
array[4]=123.456001
array[5]=123.456001
array[6]=123.456001
array[7]=123.456001
array[8]=123.456001
array[9]=123.456001
array[10]=123.456001
array[11]=123.456001
array[12]=123.456001
array[13]=123.456001
array[14]=123.456001
array[15]=123.456001
array[16]=123.456001
array[17]=123.456001
```

```
#include <omp.h>
void subdomain(float *x, int istart, int ipoints)
{
        int i;
        for (i = 0; i < ipoints; i++)
                x[istart+i] = 123.456;
}
void sub(float *x, int npoints){
        int iam, nt, ipoints, istart;
        #pragma omp parallel default(shared)private(iam,nt,ipoints,istart)
    {
         iam = omp_get_thread_num();
         nt =  omp_get_num_threads();
         ipoints = npoints / nt;    /* size of partition */
         istart = iam * ipoints;  /* starting array index */
        if (iam == nt-1)     /* last thread may do more */
             ipoints = npoints - istart;
             subdomain(x, istart, ipoints);
    }
}

int main(){
   float array[10000];
  sub(array, 10000);    return 0;
}
```

PARALLEL PROGRAMMING IN OPENMP:

DATA ENVIRONMENT

Data Environment



- OpenMP uses a shared-memory programming model
- But, not everything is shared...
  - Stack variables in functions called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE
  - Loop index variables are private (with exceptions)
    - C/C+: The first loop index variable in nested loops following a **`#pragma omp for`**

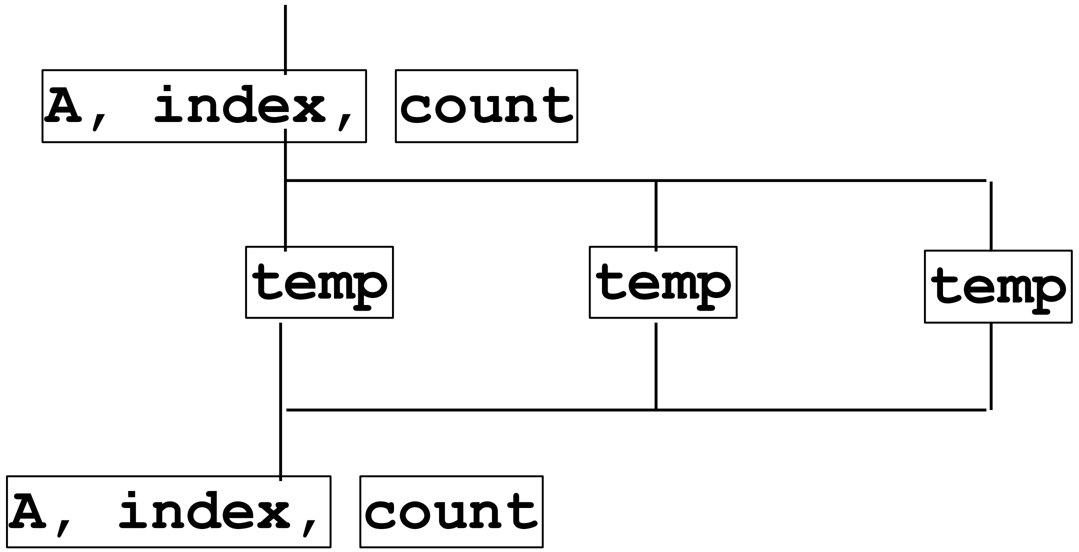
Data sharing: Examples

```
double A[10];
 int main() {
int index[10];
#pragma omp parallel
      work(index);
printf("%d\n", index[0]);;
}
```

```
extern double A[10];
 void work(int *index)
{  double temp[10];
    static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Data Environment constructs

- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
  - `PRIVATE`
  - `FIRSTPRIVATE`
  - `LASTPRIVATE`
  - `SHARED`
  - `DEFAULT`
  - `REDUCTION`
  - `COPYIN`

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

Data sharing:Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

DEFAULT(PRIVATE) *is Fortran only*

*All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

Data Scope Attributes

- The default status can be modified with

  `default (shared | none)`

- Scoping attribute clauses

  `shared(varname,…)`

  `private(varname,…)`

The shared clause declares one or more list items to be shared by tasks generated by a parallel, teams, or task generating construct.

The Private Clause

- Reproduces the variable for each thread
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

The Private Clause:example

- The private clause declares one or more list items to be private

```
#include <omp.h>
#include <stdio.h>
main()
{
int k = 100;
#pragma omp parallel for private(k)
        for ( k=0; k < 10; k++)
        {
                printf("k=%d\n", k);
        }
printf("last k=%d\n",k);
}
```

result：

```
k=6
k=7
k=8
k=9
k=0
k=1
k=2
k=3
k=4
k=5
last k=100
```

Firstprivate Clause

- Variables initialized from shared variable

  C++ objects are copy-constructed

```
int incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
      if ((I%2)==0) incr++;
      A(I)=incr;
}
```

```
int k = 100;
#pragma omp parallel for firstprivate(k)
    for (int i=0; i < 4; i++)
   {
       k+=i;
       printf("k=%d\n",k);
   }
printf(“last  k=%d\n", k);
```

```
result：
k=100
k=101
k=103
k=102
last k=100
```

lastprivate clause

- The lastprivate clause declares one or more list items to be private, and causes the corresponding original list item to be updated after the end of the region.

```
int k = 100;
#pragma omp parallel for firstprivate(k),lastprivate(k)
        for (int i=0; i < 4; i++)
        {
                k+=i;
                printf("k=%d\n",k);
        }
        printf(“last k=%d\n”, k);
```

```
result：
k=100
k=101
k=103
k=102
last k=103
```

Data Sharing:A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

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

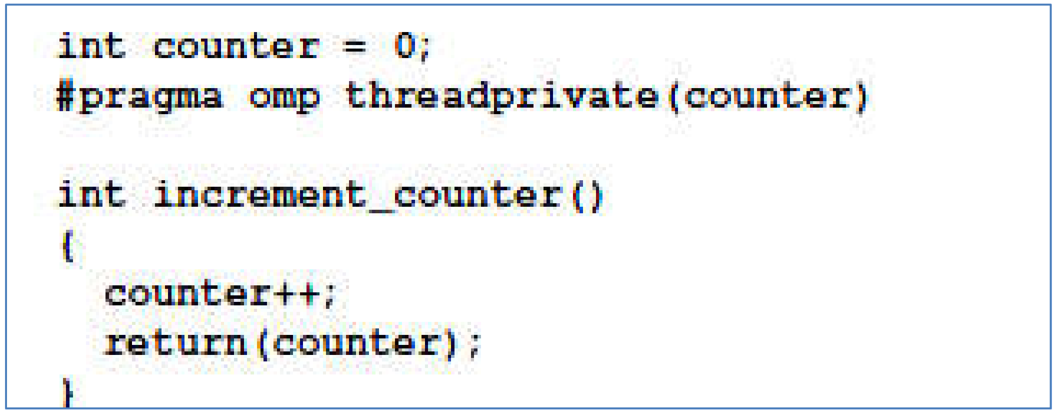
Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Threadprivate clause

- The threadprivate directive specifies that variables are replicated with each thread having its own copy
  - Each copy of a threadprivate variable is initialized once, in the manner specified by the program

#pragma omp threadprivate(*list*)

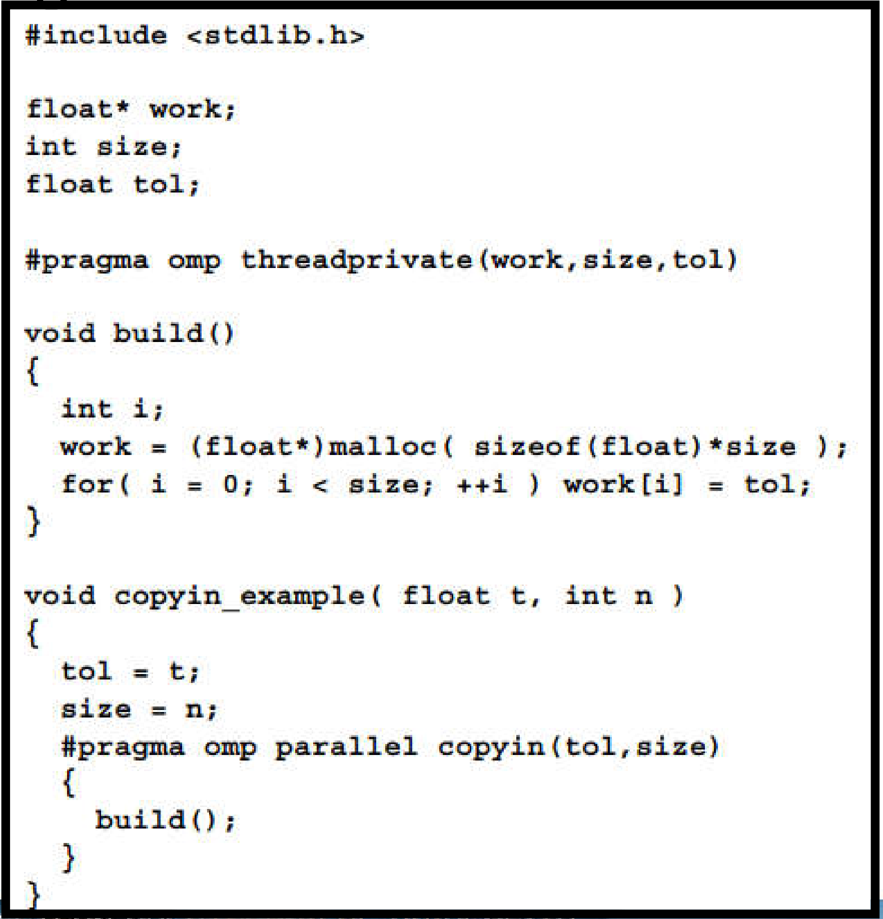


```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
  counter++;
  return(counter);
}
```

copyin Clause

- The copyin clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

copyin(*list*)



```
#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();
  }
}
```

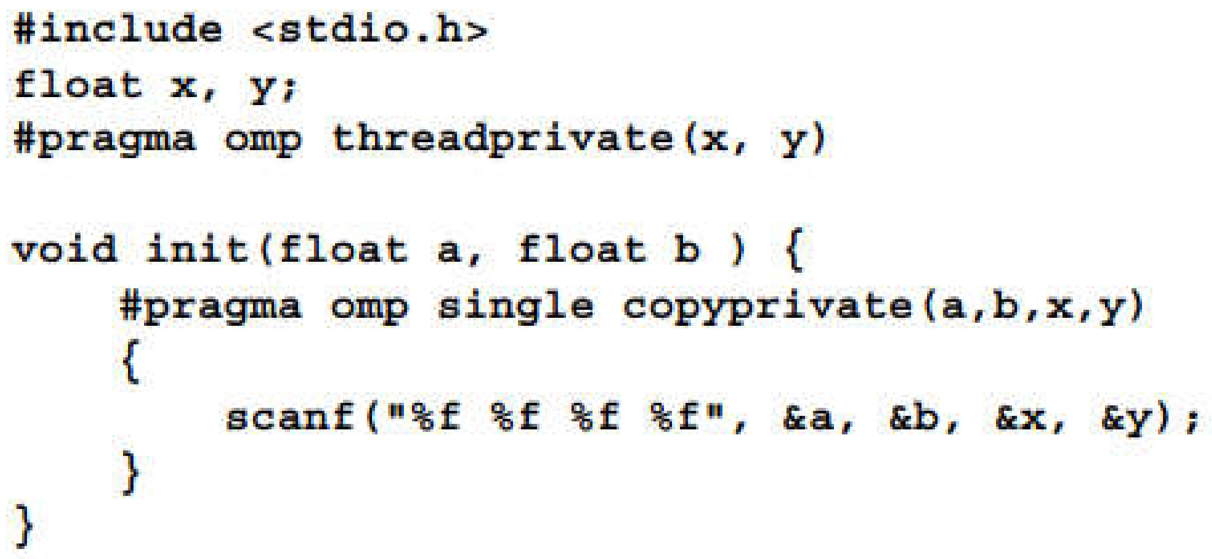
```
int main(int argc, char* argv[])
{
        int iterator;
        int  counter = 10；
        #pragma omp threadprivate(counter)
        printf("counter = %d\n", counter);
        #pragma omp parallel sections  copyin(counter)  private(iterator
        {
                #pragma omp section
                {
                        int count1;
                        for ( iterator = 0; iterator < 2; iterator++ )
                        {
                                count1 = increment_counter();
                                 printf("count1  = %d, thread_num  = %d\n", count1,omp_get_thread_num());
                        }
                }
                #pragma omp section
                {
                        int count2;
                        for ( iterator = 0; iterator < 4; iterator++ )
                        {
                                count2 = increment_counter();
                                printf("count2  = %d, thread_num  = %d\n", count2, omp_get_thread_num());
                        }
                }
        }
        printf("counter = %d， I  am thread %d\n", counter，omp_get_thread_num());
}
```

```
•    counter = 10
•    count1 = 11, thread_num = 0
•    count1 = 12, thread_num = 0
•    count2 = 11, thread_num = 1
•    count2 = 12, thread_num = 1
•    count2 = 13, thread_num = 1
•    count2 = 14, thread_num = 1
•    counter = 12. I  am thread 0
```

copyprivate Clause

- The copyprivate clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

copyprivate(*list*)



```
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter(){
        counter++;
        return(counter);
}
main(){
       #pragma omp parallel
        {
                int    count;
                #pragma omp single copyprivate(counter)
               {   counter = 50;   }
                count = increment_counter();
                printf("ThreadId: %ld, count = %ld\n", omp_get_thread_num(), count);
       }
}
```

result：

```
•   ThreadId: 4, count = 51
•   ThreadId: 1, count = 51
•   ThreadId: 5, count = 51
•   ThreadId: 3, count = 51
•   ThreadId: 6, count = 51
•   ThreadId: 0, count = 51
•   ThreadId: 2, count = 51
•   ThreadId: 7, count = 51
```

Example: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
      sum += a[i] * b[i];
   }
  return sum;
}
```

What is Wrong?

Protect Shared Data

- Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

OpenMP* Critical Construct

- `#pragma omp critical [(lock_name)]`

Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls `consum()` thereby protecting RES from race conditions

Naming the critical construct `RES_lock` is optional

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

OpenMP* Reduction Clause

```
reduction (op : list)
```

- The variables in “*list*” must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

Reduction Example

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
      sum += a[i] * b[i];
   }
```

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in “global” variable

C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

OpenMP data environment - summary

1. Variables are shared by default.
2. Global variables are shared by default.
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise.
4. Default scoping rule can be changed with default clause.

Data scope attribute clause description
private clause: declares the variables in the list to be private (not shared) to each thread.
firstprivate clause: declares variables in the list to be **private** *plus* the private variables are initialized to the value of the variable when the construct is encountered (”entered”).
lastprivate clause: declares variables in the list to be **private** *plus* the value of from the sequentially last iteration of the associated loops, or the lexically last section construct, is assigned to the original list item(s) after the end of the construct.
shared clause: declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables. Synchronization is generally advised if variables are updated.
reduction clause: performs a reduction on the scalar variables that appear in the list, with a specified operator.
default clause: allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.