OpenMP: Basics of *Parallel For*

Laxmikant V. Kale

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Some Slides based on Prof. David Padua's lectures on OpenMP

Recap

- OpenMP directives are expressed as pragmas

  #pragma omp *construct clauses..*

- Corresponding syntax exists for FORTRAN, another popular language for performance-oriented programming, especially in science and engineering
- If directives are ignored by compilers, it is still (usually) a correct sequential program
  - I.e., the directives are only intended to help in speeding up the program by using multiple processors

parallel for Construct

- Specifies that the iterations of the immediately following loop **for** loop (C/C++) may be executed in parallel
- We haven't introduced clauses yet

```
#pragma omp parallel for [clauses]
    for (i = e1; e2; e3) {
        body of the loop
    }
```

- e1, e2 and e3 are expressions
- Only a restricted form of for loop is allowed
  - The total number of iterations must be known at the start of the loop
  - And the incrementing (e3) and continuation (e2) conditions must be correspondingly simple
  - You cannot write a while loop in the form of a for loop (which C/C++ permits)

Double Precision a*x + y (daxpy)

```
void daxpy(double *z, double a, double *x, double *y, int n) {



  for(int i=0; i<n; i++)
  {
        z[i] = a*x[i]+y[i];
  }
  return;
}
```

Double Precision a*x + y (daxpy)

```
void daxpy(double *z, double a, double *x, double *y, int n) {

  #pragma omp parallel for

  for(int i=0; i<n; i++)
  {
        z[i] = a*x[i]+y[i];
  }
  return;
}
```

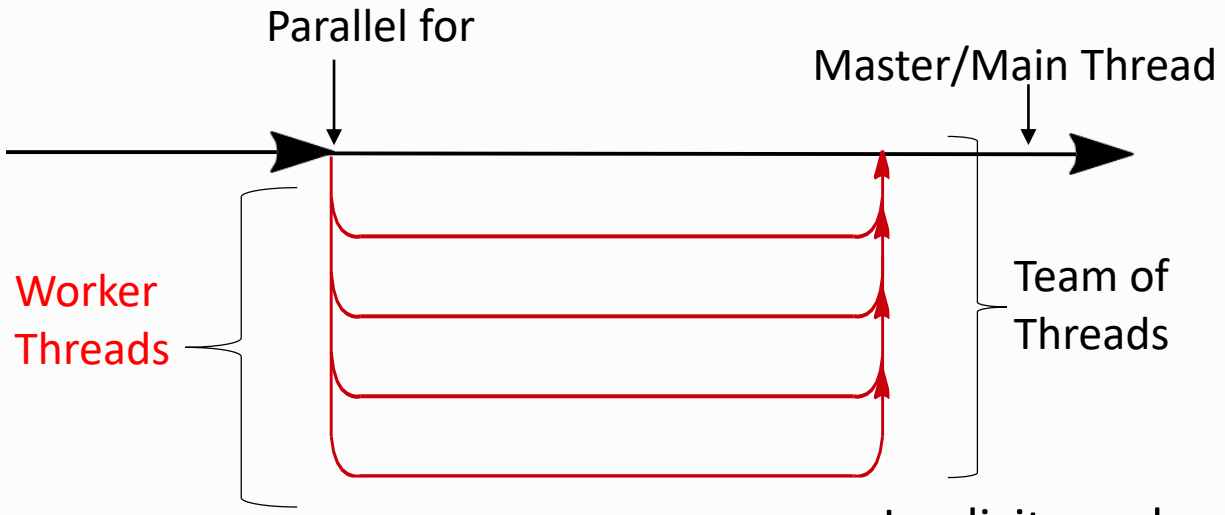
Parallel for: Execution

Master thread executes serial portion of code

Master thread enters *saxpy* function

Master thread encounters *parallel for* directive

Creates worker threads, and makes a team: workers + master



Master thread resumes execution after the **for** loop

Worker threads disappear

Implicit synchronization: wait for all threads to finish their allocation of iterations

Master and worker threads divide iterations of parallel **for** loop and execute them concurrently

Number of Threads

- You can control the number of threads OpenMP uses
  - Typically set equal to the number of cores (or SMT threads) or one less
- Use an environment variable

```
setenv OMP_NUM_THREADS 8 (on Unix with csh shell)
export OMP_NUM_THREADS =8 (with bash)
```

- Use *omp_set_num_threads()* function, to override the env variable

```
void saxpy(double *z, double a, double *x,
                    double *y, int n) {
      omp_set_num_threads(8);
      #pragma omp parallel for
      for(int i=0; i<n; i++)
      {
            z[i] = a*x[i]+y[i];
      }
      return;
}
```

Loop Scheduling

- Which thread will execute which iterations?
- This is called the loop schedule
- The default schedule assigns iterations to threads as evenly as possible (good enough for *saxpy*)
- You can have more control over scheduling
  - As we will discuss later

OpenMP: Enabling Parallelization

Private Variables: firstprivate and lastprivate

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Private Clause: Semantics and Limitations

- Each thread has its own copy of each variable declared private
- Copies of private variables have undefined values, when the loop starts
  - Except on the master thread
- The value of a private variable is unavailable to the master thread after a parallel loop terminates
  - Because it is unclear which thread's value to copy back into the master thread's copy
- `firstprivate` and `lastprivate` clauses provide additional functionality

firstprivate and lastprivate

- firstprivate (list) initializes each thread's copy of a private variable to the value of the master thread's copy, for all variables in list
- lastprivate (list) writes back to the master's copy the value contained in the private copy belonging to the thread that executed the sequentially last iteration of the loop, for all variables in list

`lastprivate` Example

- What's wrong with this example?
- The print statement, in sequential code, prints the value taken by tmp in the last iteration (i=n-1)

```
#pragma omp parallel for lastprivate (tmp)
  for(int i=0; i<n; i++)
     {
       tmp = x[i]*x[i]*3.1415
       z[i] = tmp*x[i]+y[i];
       t[i] = tmp *y[i];
     }
 printf("tmp = %f\n", tmp);
```

In parallel execution, the value of tmp outside (after) the loop is undefined

- Changing "private" to "lastprivate" does the right thing:
  - The value of tmp from the thread that executed the last iteration is copied to the main thread.

`firstprivate` Example

- What's wrong with this example?
- s is used as a scratchpad (like tmp was, in previous example)
- I.e. values of s[1] ..s[7] are assigned before use in each iteration
- But s[0] is only read.. It is assigned before the loop

```
float s[8];
s[0] = calculateBase(…);
#pragma omp parallel for firstprivate (s)
  for(int i=0; i<n; i++)
     {
        s[..] = ..;
        ..
        ..    = ..s[..];
     }
```

We want **s** to be private, to allow its use as scratchpad for temporary iteration-specific calculations.

But, we want s[0] to come from before the loop.

- Changing "private" to "firstprivate" does the right thing:
  - The value of s, including s[0], from the main thread is copied to all threads.

A variable can be in both lists

- What if we also want to print (say) s[7] after the loop?
- We can declare s as **`firstprivate`** as well as **`lastprivate`**

```
float s[8];
s[0] = calculateBase(…);
#pragma omp parallel for firstprivate (s) lastprivate(s)
  for(int i=0; i<n; i++)
     {
        s[..] = ..;
        ..
        ..    = ..s[..];
     }
Printf("last iteration's scratchpad value was: %f\n", s[7]);
```

OpenMP: Enabling Parallelization

Reduction Variables

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Example: Sum over an array

```
sum = 0;
for (i = 0; i < N; i++){
  sum += A[i];
}
```

- The operations on sum
  - Create a loop carried dependency
  - ”+=” implies both a read and write
  - **`sum = sum + A[i]`**
- But really, all that we are doing is “adding into” sum
  - It is a commutative-associative operation
  - We don’t care which thread or iterations add to **`sum`** first
- But that doesn’t help
  - Statements like “**`sum = sum + A[i]`**” are implemented a a series of machine instructions (load, store, add)
  - Interleaving of these instructions will create an inconsistent result

Enter OpenMP's "`reduction`" clause

- Reduction variables can be thought of as a special case of private variables
- Syntax:
  - reduction(op:varList)

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

- Op (for operation)
  - Can be one of : +, -, *, &&, ||, &, |, ^
- A loop may have multiple reduction clauses
- Reduction variables may be array sub-ranges: A[lowerBound:length]
  - e.g. A[53:10] 10 elements starting with A[53]

Reduction Example 2

```
sum = 0;
#pragma omp parallel for reduction(+:sa, sb) reduction(max:m)
for (i = 0; i < N; i++){
 sa += A[i];
 sb += B[i];
 if (A[i]>m) m = A[i];
 m = B[i]>m?B[i]:m;  // same result as: if (B[i]>m) m = B[i];
}
```

sa will be the sum over array A
sb will be the sum over array B
m will be the max over both arrays

Summary: Data Sharing

- Six clause types allow the programmer to specify how data is shared between threads executing a parallel do (*data scope clauses*):
  - private: private (list)
  - shared: shared(list)
  - default: default (private | shared | none)
  - reduction: reduction(intrinsic operator : list)
  - firstprivate: firstprivate(list)
  - lastprivate: lastprivate(list)

- default(private), for example, can be used to avoid large list of variables to be privatized
- default(none) : many programmers like to use this clause so they are forced to express each variable's sharing attribute: shared/private/lastprivate/firstprivate explicitly

Exploring and understanding further

- When in doubt, consult the standard:
  - https://www.openmp.org/specifications/
  - Consult the complete specification (we are using OpenMP 4.5)
  - Also associated example programs are provided there
- More resources at: https://www.openmp.org/

OpenMP: Enabling Parallelization

Private Variables

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Example Loop

- Is this parallelizable? Why?
  - Loop-carried dependence via tmp
  - How can you parallelize it?

```
#pragma omp parallel for
  for(int i=0; i<n; i++)
     {
       tmp[i] = x[i]*x[i]*3.1415
       z[i] = tmp[i]*x[i]+y[i];
       t[i] = tmp[i]*y[i];
     }
```

```
#pragma omp parallel for
  for(int i=0; i<n; i++)
     {
       tmp = x[i]*x[i]*3.1415
       z[i] = tmp*x[i]+y[i];
       t[i] = tmp *y[i];
     }
```

Idea 1: Use a separate tmp for each iteration

Idea 2: Eliminate tmp by re-computing it in the two statements

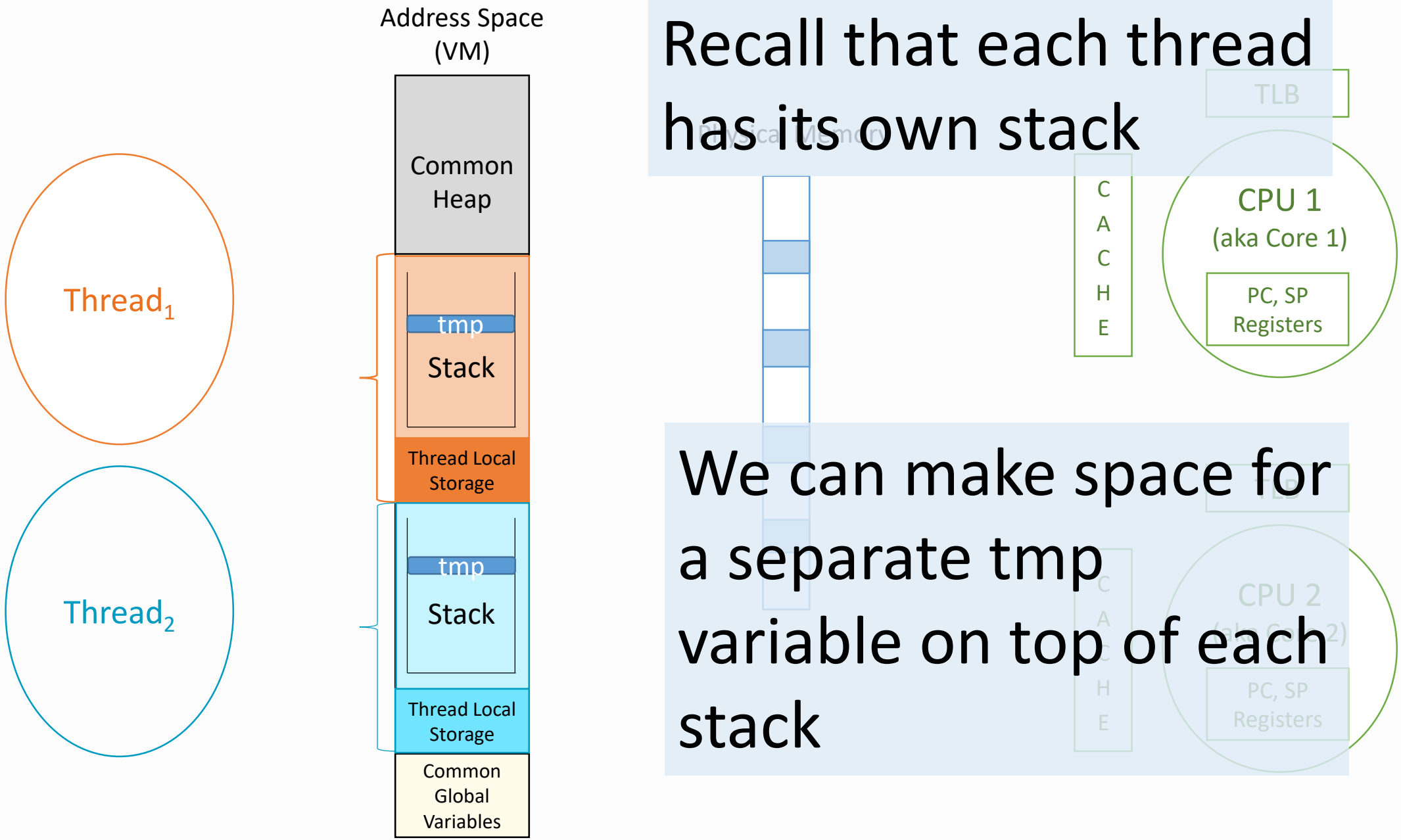
```
#pragma omp parallel for
  for(int i=0; i<n; i++)
     { z[i] = x[i]*x[i]*3.1415 *x[i]+y[i];
       t[i] = x[i]*x[i]*3.1415 *y[i];
     }
```

But there is a better solution!

Do we really need a separate variable (*tmp[i]*) for each iteration *i*?

A separate *tmp* for each ***thread*** will suffice

Recall that each thread has its own stack



We can make space for a separate tmp variable on top of each stack

Example Loop

- The compiler supports this idea with a “private” clause
- The keyword “private” is followed by a parenthesized list of variables, separated by commas

```
#pragma omp parallel for
  for(int i=0; i<n; i++)
     {
       tmp = x[i]*x[i]*3.1415
       z[i] = tmp*x[i]+y[i];
       t[i] = tmp *y[i];
     }
```

```
#pragma omp parallel for private (tmp)
  for(int i=0; i<n; i++)
     {
       tmp = x[i]*x[i]*3.1415
       z[i] = tmp*x[i]+y[i];
       t[i] = tmp *y[i];
     }
```

Shared and Private Variables and Defaults

- Shared variable: there exists one copy that all threads access
- Private variable: one copy for each thread
- Global variables are shared among threads
- Loop index variables are private by default
  - i does not have to be declared as private in the code below

```
void daxpy(double *z, double a, double *x, double *y, int n)
{
  #pragma omp parallel for

  for(int i=0; i<n; i++)
  {
        z[i] = a*x[i]+y[i];
  }
  return;
}
```

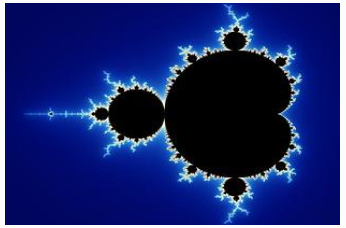
Shared and Private Variables and Defaults

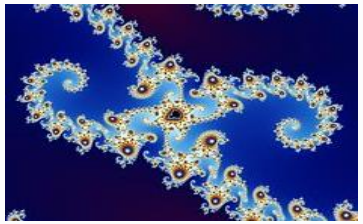
- Stack variables in function calls from parallel regions are thread-private

```
…
  #pragma omp parallel for

  for(int i=0; i<n; i++)
  {
        z[i] = f(x[i]);
  }

double f(double a){
  double b = sqrt(a);
  return a+b+b/5;
}
```

Since each thread has its own stack and function calls are implemented using stacks, each thread gets its own copy of the variable b

Example: Mandlebrot Sets





```
double x, y;
int i, j, m, n, maxiter;
int depth[200][300];
extern int mandel_val();
n = 300;
m = 200;
maxiter = 200;
```

```
#pragma omp parallel for private(j, x, y)
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)  {
        x = i/ (double) m;
        y = j/ (double) n;
        depth[i][j] = mandel_val(x, y, maxiter);
   }
```

Private Clause: Semantics and Limitations

- Each thread has its own copy of each variable declared private
- Copies of private variables have undefined values, when the loop starts
  - Except on the master thread
- The value of a private variable is unavailable to the master thread after a parallel loop terminates
  - Because it is unclear which thread's value to copy back into the master thread's copy
- `firstprivate` and `lastprivate` clauses provide additional functionality

firstprivate and lastprivate

- firstprivate (list) initializes each thread's copy of a private variable to the value of the master thread's copy, for all variables in list
- lastprivate (list) writes back to the master's copy the value contained in the private copy belonging to the thread that executed the sequentially last iteration of the loop, for all variables in list

`lastprivate` Example

- What's wrong with this example?
- The print statement, in sequential code, prints the value taken by tmp in the last iteration (i=n-1)

```
#pragma omp parallel for lastprivate (tmp)
  for(int i=0; i<n; i++)
     {
       tmp = x[i]*x[i]*3.1415
       z[i] = tmp*x[i]+y[i];
       t[i] = tmp *y[i];
     }
 printf("tmp = %f\n", tmp);
```

In parallel execution, the value of tmp outside (after) the loop is undefined

- Changing "private" to "lastprivate" does the right thing:
  - The value of tmp from the thread that executed the last iteration is copied to the main thread.

`firstprivate` Example

- What's wrong with this example?
- s is used as a scratchpad (like tmp was, in previous example)
- I.e. values of s[1] ..s[7] are assigned before use in each iteration
- But s[0] is only read.. It is assigned before the loop

```
float s[8];
s[0] = calculateBase(…);
#pragma omp parallel for firstprivate (s)
  for(int i=0; i<n; i++)
     {
        s[..] = ..;
        ..
        ..    = ..s[..];
     }
```

We want **s** to be private, to allow its use as scratchpad for temporary iteration-specific calculations.

But, we want s[0] to come from before the loop.

- Changing "private" to "firstprivate" does the right thing:
  - The value of s, including s[0], from themain thread is copied to all threads.

A variable can be in both lists

- What if we also want to print (say) s[7] after the loop?
- We can declare s as **`firstprivate`** as well as **`lastprivate`**
- is used as But s[0] is only read.. It is assigned before the loop

```
float s[8];
s[0] = calculateBase(…);
#pragma omp parallel for firstprivate (s) lastprivate(s)
  for(int i=0; i<n; i++)
     {
        s[..] = ..;
        ..
        ..    = ..s[..];
     }
Printf("last iteration's scratchpad value was: %f\n", s[7]);
```

Summary: Data Sharing

- Six clause types allow the programmer to specify how data is shared between threads executing a parallel do (*data scope clauses*):
  - private: private (list)
  - shared: shared(list)
  - default: default (private | shared | none) (ex: C/C++)
  - reduction: reduction(intrinsic operator : list) [NEXT LECTURE]
  - firstprivate: firstprivate(list)
  - lastprivate: lastprivate(list)

Reduction Example

```
double sum(double *values, int n){
        double s=0;
#pragma omp parallel for reduction (+:s)
        for(int i=0; i<n; i++) {
                s = s + values[i];
        }
        return;
}
```

Restructuring for Parallelization

Eliminating Apparent Dependences by Rewriting Portions of Code

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Sometimes, You Have to Tweak Your Code a Bit

- Remember the definition of loop-carried dependency
  - If two different iterations access the same location, and
  - One of the two accesses is a write
- We already noted that this is more strict than necessary, and we can deal with some “dependences” by privatizing the associated variables
- We will next see examples where rewriting the code (without changing what it is accomplishing in the end) can make a loop parallelizable
  - This happens because there are many ways of writing code
  - Sequential programmer has no reason to worry about which way will hinder parallelization

Restructuring Example 1

- Is this parallelizable? Why?
- Loop-carried dependence via t
  - We can handle it: privatize t
  - Note that t is defined before used in each iteration (twice)
- But that doesn't solve the problem
  - Array A values are being written and read
  - Which would be fine if each iteration touches distinct set of values …
  - But A[i+1] and A[i] create loop-carried dependence

```
#pragma omp parallel for??
 for(int i=0; i<n-1; i++)
 {
   t = A[i] + B[i];
   C[i] = t + t;
    t = D[i] - B[i];
   A[i+1] = t*t;
 }
```

Restructuring Example 1

```
for(int i=0; i<n-1; i++)
 {
   t = A[i] + B[i]
   C[i] = t + t
   t = D[i] - B[i]
   A[i+1] = t*t
 }
```

- Let us analyze what the loop is doing in detail
- Observations:
  - Arrays B and D are only read (not written into)
  - Array C is written into, but not read
  - All values in A except A[0] are changed at the end
  - C[i] is assigned values using the new value in A[i] (except for C[0], which uses old A[0])
    - I.e., stale values of A are not used in assigning to C
    - By "stale" we mean values that existed before the loop started
  - The variable t is used for two different purposes …
    - For assigning to C[i] and for assigning to A[i+1]

Restructuring Example 1 : Transformations

- Let us use two different variables
  - for the two uses of t
  - Assume they are declared inside the for body
- Now we notice that the 2 statements involving t and t1 can be interchanged
- Next, we can make two loops out of this
  - And both are parallel!

```
for(int i=0; i<n-1; i++)
 {int t, t1;
  t = A[i] + B[i];
  C[i] = t + t ;
  t1 = D[i] - B[i];
  A[i+1] = t1*t1;
 }
```

```
for(int i=0; i<n-1; i++)
 {int t1;
  t1 = D[i] - B[i];
  A[i+1] = t1*t1;
 }
for(int i=0; i<n-1; i++)
 {int t;
  t = A[i] + B[i];
  C[i] = t + t;
}
```

Restructuring Example 1: Parallel Code

```
#pragma omp parallel for
for(int i=0; i<n-1; i++)
 {int t1;
   t1 = D[i] - B[i];
   A[i+1] = t1*t1;
 }
#pragma omp parallel for
for(int i=0; i<n-1; i++)
 {int t;
   t = A[i] + B[i];
   C[i] = t + t;
}
```

Using the Loop Index Variable

```
j=1;
for (i=0; i<N; i++)
{
    j = j+7;
    A[i] = B[j]*5;
}
```

```
j=1;
#pragma omp parallel for private(j)
for (i=0; i<N; i++)
{
    j =8 + i*7;
    A[i] = B[j]*5;
}
```

Loop-carried dependence due to j

Eliminated using expressions
that are functions of index variable (i)

Use some simple algebra!