Shared Memory Parallelism

Introduction to Threads
- Exercise: Race condition

OpenMP Programming Model
- Scope of Variables: Exercise 1
- Synchronisation: Exercise 2

Scheduling
- Exercise: OpenMP scheduling

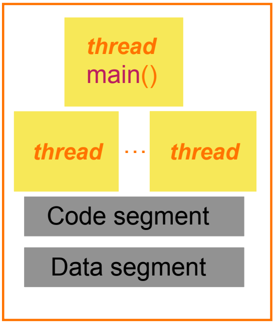
Reduction
- Exercise: Pi

Shared variables
- Exercise: CacheTrash

Tasks

Future of OpenMP

Processes and Threads


thread
main()
thread … *thread*
Code segment
Data segment

Modern operating systems load programs as processes
- Resource holder
- Execution

A process starts executing at its entry point as a thread

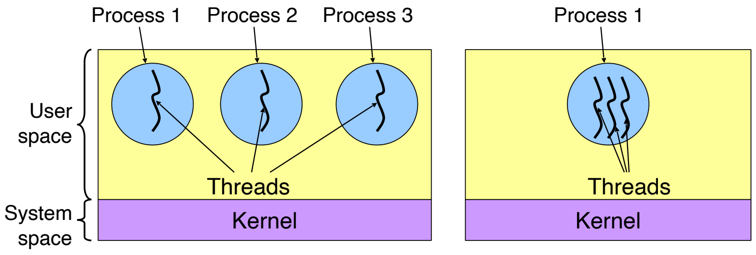
Threads can create other threads within the process

All threads within a process share code & data segments

Threads have lower overhead than processes

Threads: “processes” sharing memory

Process == address space
Thread == program counter / stream of instructions
Two examples
- Three processes, each with one thread
- One process with three threads



What Are Threads Good For?

- Overlapping computation and I/O
- Improving responsiveness of GUIs: thread waiting for keyboard input
- Improving performance through parallel execution
  - with the help of OpenMP

Fork/Join Programming Model

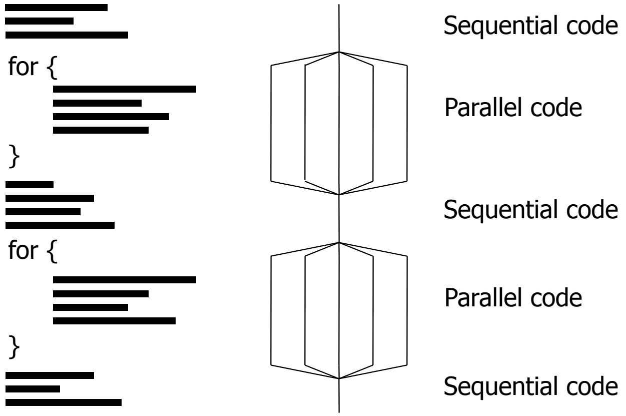
When program begins execution, only master thread active

Master thread executes sequential portions of program

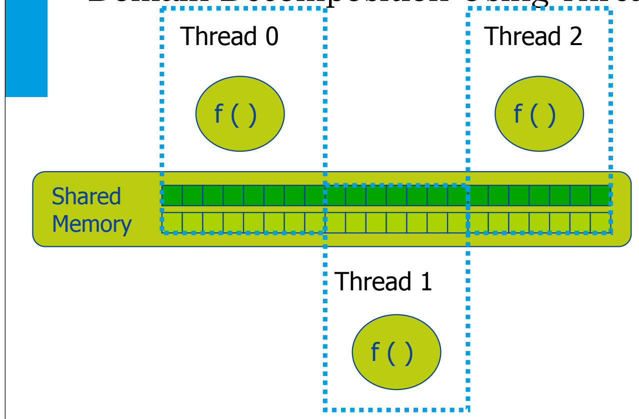
For parallel portions of program, master thread ***forks*** (creates or awakens) additional threads

At ***join*** (end of parallel section of code), extra threads are suspended or die

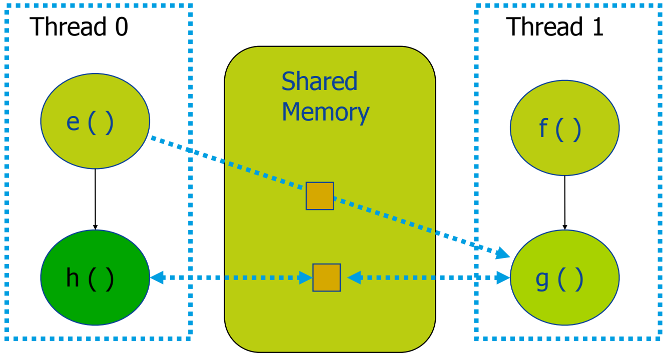
Relating Fork/Join to Code (OpenMP)

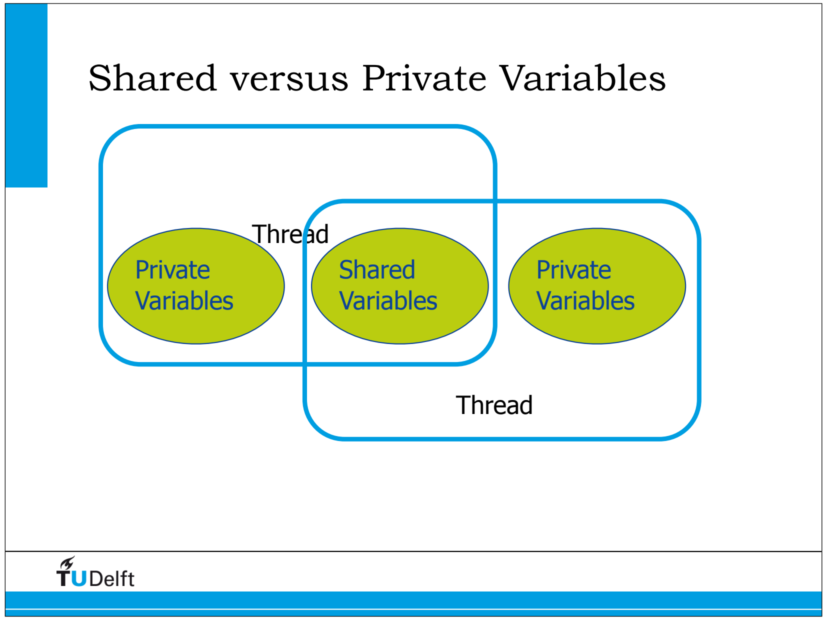


Domain Decomposition Using Threads



Task Decomposition Using Threads





Shared versus Private Variables

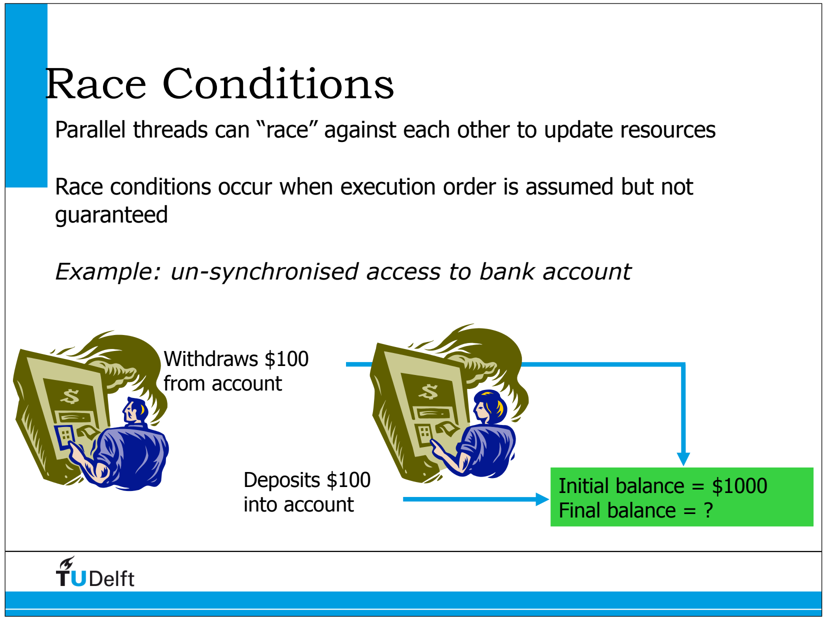
Thread

Private Variables

Shared Variables

Private Variables

Thread



Race Conditions

Parallel threads can "race" against each other to update resources

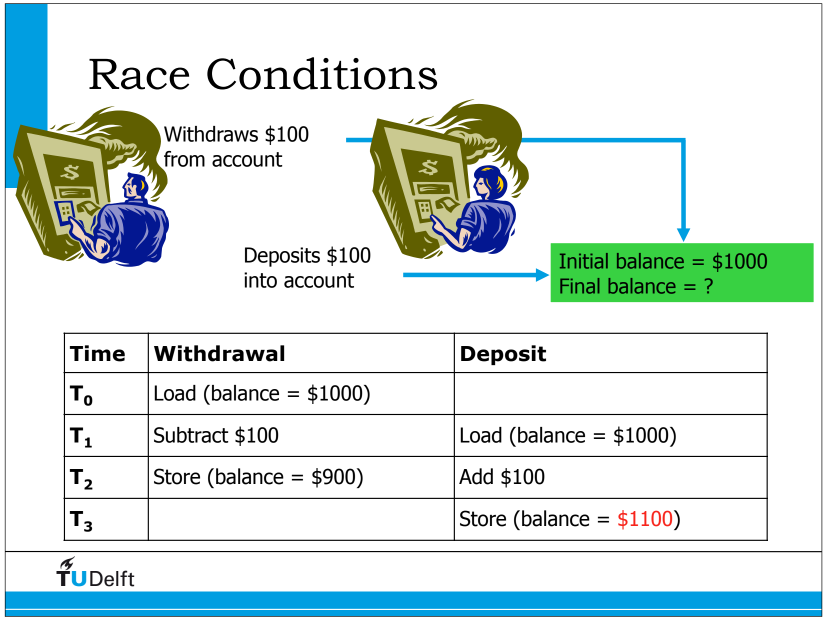
Race conditions occur when execution order is assumed but not guaranteed

Example: un-synchronised access to bank account

Withdraws $100 from account

Deposits $100 into account

Initial balance = $1000
Final balance = ?



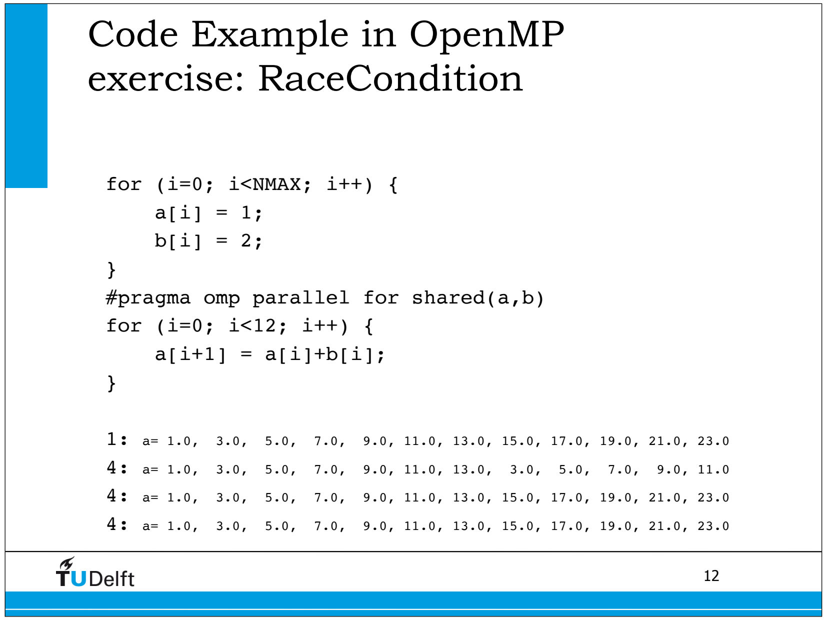
Race Conditions

Withdraws $100 from account

Deposits $100 into account

Initial balance = $1000
Final balance = ?

Time	Withdrawal	Deposit
T_0	Load (balance = $1000)	
T_1	Subtract $100	Load (balance = $1000)
T_2	Store (balance = $900)	Add $100
T_3		Store (balance = $1100)



Code Example in OpenMP
exercise: RaceCondition

```
for (i=0; i<NMAX; i++) {
    a[i] = 1;
    b[i] = 2;
}
#pragma omp parallel for shared(a,b)
for (i=0; i<12; i++) {
    a[i+1] = a[i]+b[i];
}

1: a= 1.0,  3.0,  5.0,  7.0,  9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0
4: a= 1.0,  3.0,  5.0,  7.0,  9.0, 11.0, 13.0,  3.0,  5.0,  7.0,  9.0, 11.0
4: a= 1.0,  3.0,  5.0,  7.0,  9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0
4: a= 1.0,  3.0,  5.0,  7.0,  9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0
```

Code Example in OpenMP

```
thread    computation
0         a[1] = a[0] + b[0]
0         a[2] = a[1] + b[1]
0         a[3] = a[2] + b[2]   <--| Problem
1         a[4] = a[3] + b[3]   <--| Problem
1         a[5] = a[4] + b[4]
1         a[6] = a[5] + b[5]   <--| Problem
2         a[7] = a[6] + b[6]   <--| Problem
2         a[8] = a[7] + b[7]
2         a[9] = a[8] + b[8]   <--| Problem
3         a[10] = a[9] + b[9] <--| Problem
3         a[11] = a[10] + b[10]
```

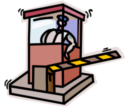
How to Avoid Data Races

Scope variables to be local to threads

- Variables declared within threaded functions
- Allocate on thread's stack
- TLS (Thread Local Storage)

Control shared access with critical regions

- Mutual exclusion and synchronization
- Lock, semaphore, event, critical section, mutex…



Examples variables

Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

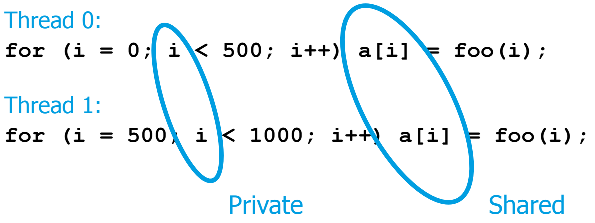
Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);
```



Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

Private

Shared

Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;   j = f(k);   m = g(k); ...
}

int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}

int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;   j = f(k);   m = g(k);
}
```

Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Task Decomposition

```
int e;
```

Static variable: Shared

```
main () {
   int x[10], j, k, m;   j = f(k);   m = g(k);
}
```

Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Task Decomposition

```
int e;
```

Heap variable: Shared

```
main () {
   int x[10], j, k, m;   j = f(x, k);   m = g(x, k);
}
```

Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;   j = f(k);   m = g(k);
}
```

Function's local variables: Private

Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Shared and Private Variables

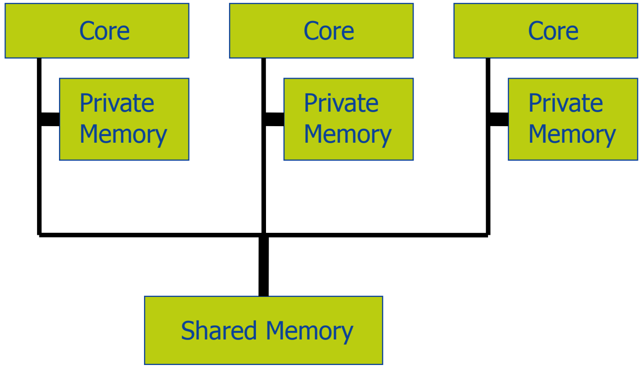
Shared variables

- Static variables
- Heap variables
- Contents of run-time stack at time of call

Private variables

- Loop index variables
- Run-time stack of functions invoked by thread

The Shared-Memory Model



What Is OpenMP?

Compiler directives for multithreaded programming

Easy to create threaded Fortran and C/C++ codes

Supports data parallelism model

Portable and Standard

Incremental parallelism

- ➡Combines serial and parallel code in single source

OpenMP is not …

Not Automatic parallelization

– User explicitly specifies parallel execution
– Compiler does **not** ignore user directives even if wrong

Not just loop level parallelism

– Functionality to enable general parallel parallelism

Not a new language

– Structured as extensions to the base
– Minimal functionality with opportunities for extension

Directive based

Directives are special comments in the language

– Fortran fixed form: `!$OMP, C$OMP, *$OMP`

– Fortran free form: `!$OMP`

Special comments are interpreted by OpenMP compilers

```
      w = 1.0/n
      sum = 0.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:sum)
      do I=1,n
        x = w*(I-0.5)
        sum = sum + f(x)
      end do
      pi = w*sum
      print *,pi
      end
```

Comment in Fortran
but interpreted by
OpenMP compilers

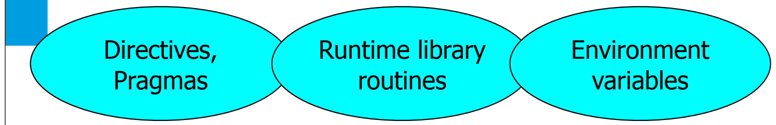
C example

`#pragma omp` directives in C

– Ignored by non-OpenMP compilers

```
  w = 1.0/n;
  sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
  for(i=0, i<n, i++) {
    x = w*((double)i+0.5);
    sum += f(x);
  }
  pi = w*sum;
  printf("pi=%g\n", pi);
}
```

Architecture of OpenMP



Control structures
Work sharing
Synchronization
Data scope attributes
- private
- shared
- reduction

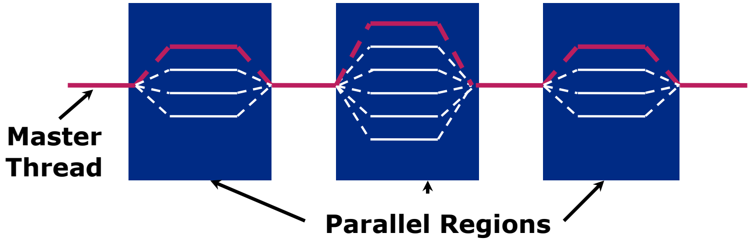
Orphaning

- Control & query routines
  - number of threads
  - throughput mode
  - nested parallism
- Lock API

- Control runtime
  - schedule type
  - max threads
  - nested parallelism
  - throughput mode

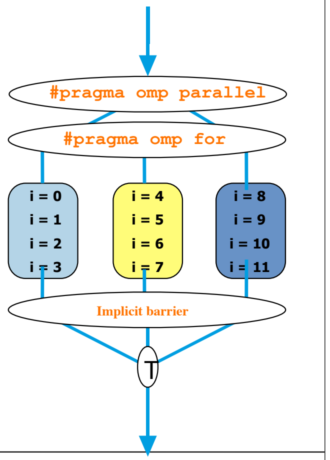
Programming Model

- Fork-join parallelism:
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: the sequential program evolves into a parallel program



Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
    for(i = 0; i < 12; i++)
       c[i] = a[i] + b[i]
```



Threads are assigned an independent set of iterations

Threads must wait at the end of work-sharing construct

Combining pragmas

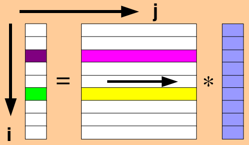
These two code segments are equivalent

```
#pragma omp parallel
{
     #pragma omp for
     for (i=0; i< MAX; i++) {
         res[i] = huge();
     }
}
```

```
#pragma omp parallel for
     for (i=0; i< MAX; i++) {
         res[i] = huge();
     }
```

Matrix-vector example

```
 #pragma omp parallel for default(none) \
           private(i,j,sum) shared(m,n,a,b,c)
 for (i=0; i<m; i++)
 {
     sum = 0.0;
     for (j=0; j<n; j++)
       sum += b[i][j]*c[j];
     a[i] = sum;
  }
```



TID = 0

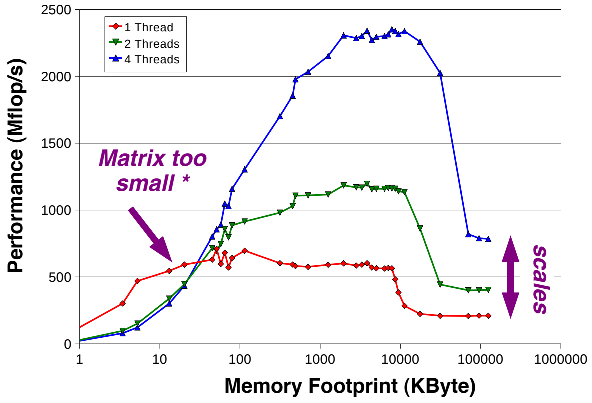
```
for (i=0,1,2,3,4)
  i = 0
sum = Σ  b[i=0][j]*c[j]
    a[0] = sum
  i = 1
sum = Σ  b[i=1][j]*c[j]
    a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
  i = 5
sum = Σ  b[i=5][j]*c[j]
    a[5] = sum
  i = 6
sum = Σ  b[i=6][j]*c[j]
    a[6] = sum
```

etc

Performance is matrix size dependent



OpenMP parallelization

OpenMP Team := Master + Workers

A Parallel Region is a block of code executed by all threads simultaneously

- The master thread always has thread ID 0
- Thread adjustment (if enabled) is only done before entering a parallel region
- Parallel regions can be nested, but support for this is implementation dependent
- An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially

A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Data Environment

OpenMP uses a shared-memory programming model

- **Most** variables are shared by default.
- Global variables are shared among threads
  C/C++: File scope variables, static

Not everything is shared, there is often a need for “local” data as well

Data Environment

... 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`

About Variables in OpenMP

Shared variables

Can be accessed by every thread thread. Independent read/write operations can take place.

Private variables

Every thread has it’s own copy of the variables that are created/destroyed upon entering/leaving the procedure. They are not visible to other threads.

serial code	parallel code
global	shared
auto local	local
static	use with care
dynamic	use with care

Data Scope clauses

attribute clauses

```
default(shared)

shared(varname,…)

private(varname,…)
```

The Private Clause

Reproduces the variable for each thread

- Variables are un-initialised; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, float *a, float
*x, 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;
     }
}
```

Synchronization

Barriers `#pragma omp barrier`

Critical sections `#pragma omp critical()`

Lock library routines

```
omp_set_lock(omp_lock_t *lock)

omp_unset_lock(omp_lock_t *lock)

....
```

OpenMP Critical Construct

`#pragma omp critical [(lock_name)]`

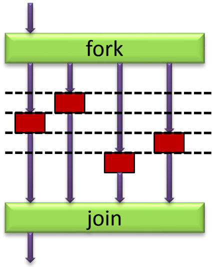
Defines a critical region on a structured block

All threads execute the code, but only one at a time. Only one calls `consum()` thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional, but may increase performance.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical(R1_lock)
    consum (B, &R1);
    A = bigger_job(i);
#pragma omp critical(R2_lock)
    consum (A, &R2);
  }
}
```

OpenMP Critical

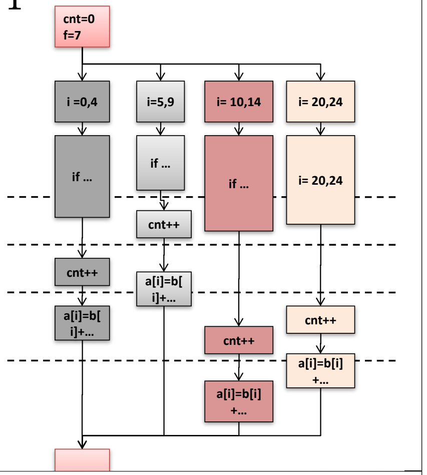


🟥 - critical region

All threads execute the code, but only one at a time. Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

Critical Example 1

```
cnt = 0;
f = 7;
#pragma omp parallel
{
   #pragma omp for
      for (i=0;i<20;i++){
         if(b[i] == 0){

            #pragma omp critical
               cnt ++;
         } /* end if */
         a[i]=b[i]+f*(i+1);
      } /* end for */
} /* omp end parallel */
```



OpenMP Single Construct

Only one thread in the team executes the enclosed code

The Format is:

```
#pragma omp single [nowait][clause, ..]{
        "block"
}
```

The supported clauses on the single directive are:

```
private (list)
firstprivate (list)
```

NOWAIT: the other threads will **not** wait at the end single directive

OpenMP Master directive

```
#pragma omp master {
        "code"
}
```

All threads but the master, skip the enclosed section of code and continue

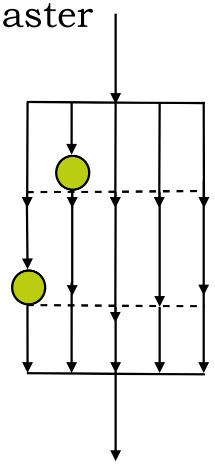
There is no implicit barrier on entry or exit !

```
#pragma omp barrier
```

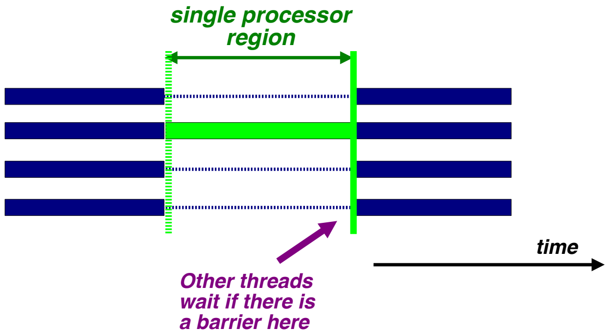
Each thread waits until all others in the team have reached this point.

Work Sharing: Single Master

```
#pragma omp parallel
{
     ....
#pragma omp single [nowait]
{
     ....
}
#pragma omp master
{
     ....
}
     ....
#pragma omp barrier
}
```



Single processor



Work Sharing: Orphaning

Worksharing constructs may be outside lexical scope of the parallel region

```
#pragma omp parallel
{
     ....
     dowork( )
     ....
}
 ....
```

```
void dowork( )
{
   #pragma omp for
   for (i=0; i<n; i++) {
       ....
   }
}
```

Scheduling the work

schedule (static | dynamic | guided | auto [, chunk]) schedule (runtime)

static [, chunk]

- Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
- In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads

Thread	0	1	2	3
*no chunk**	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

dynamic [, chunk]

- Fixed portions of work; size is controlled by the value of chunk
- When a thread finishes, it starts on the next portion of work

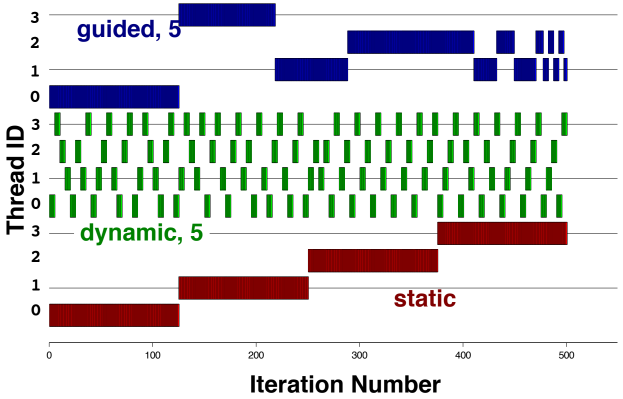
- **guided [, chunk]**
  - Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially

runtime

- Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE

Example scheduling

500 iterations on 4 threads



Environment Variables

The names of the OpenMP environment variables must be UPPERCASE
The values assigned to them are case insensitive

```
OMP_NUM_THREADS

OMP_SCHEDULE “schedule [chunk]”

OMP_NESTED { TRUE | FALSE }
```

Exercise: OpenMP scheduling

Download code from:
http://www.xs4all.nl/~janth/HPCourse/OMP_schedule.tar

Two loops

- Parallel code with omp sections
- Check what the auto-parallelisation of the compiler has done
- Insert OpenMP directives to try out different scheduling strategies
  - c$omp& schedule(runtime)
  - export OMP_SCHEDULE="static,10"
  - export OMP_SCHEDULE="guided,100"
  - export OMP_SCHEDULE="dynamic,1”

OpenMP Reduction Clause

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

The variables in "*list*" must be shared in the enclosing parallel region

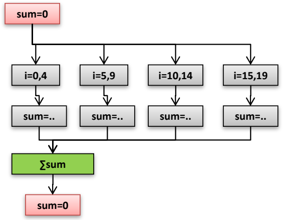
Inside parallel or work-sharing construct:

- 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 and commutative operators can be used with reduction
Initial values are the ones that make sense

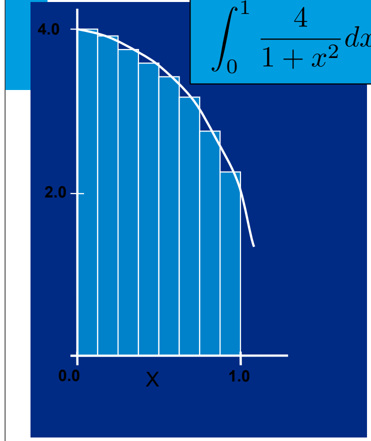
Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

FORTRAN:
intrinsic is one of MAX, MIN, IAND, IOR, IEOR
operator is one of +, *, -, .AND., .OR., .EQV., .NEQV.

Numerical Integration Example

$$\int_0^1 \frac{4}{1+x^2}dx = \pi$$



```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i<num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Numerical Integration to Compute Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i<num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Parallelize the numerical integration code using OpenMP

What variables can be shared?

`step, num_steps`

What variables need to be private?

`x, i`

What variables should be set up for reductions?

`sum`

Solution to Computing Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;
   step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
   for (i=0; i<num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

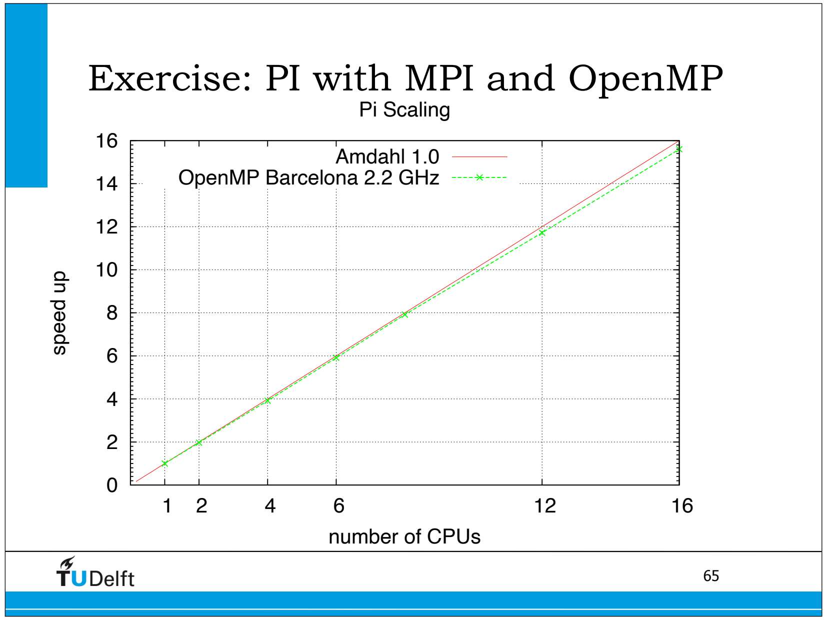
Let's try it out

Go to example MPI_pi.tar and work with openmp_pi2.c

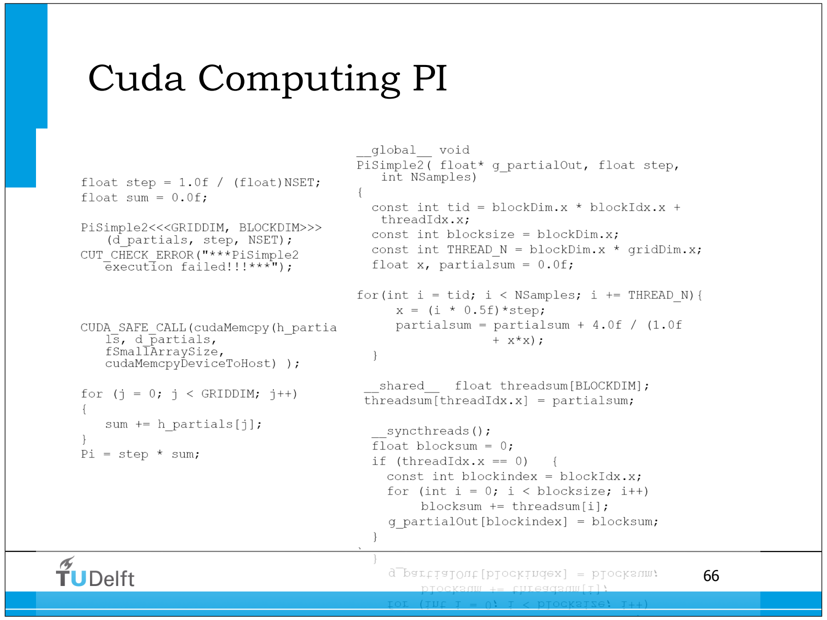
Explore the effect of the critical region.

Exercise: PI with MPI and OpenMP

cores	OpenMP
1	9.617728
2	4.874539
4	2.455036
6	1.627149
8	1.214713
12	0.820746
16	0.616482



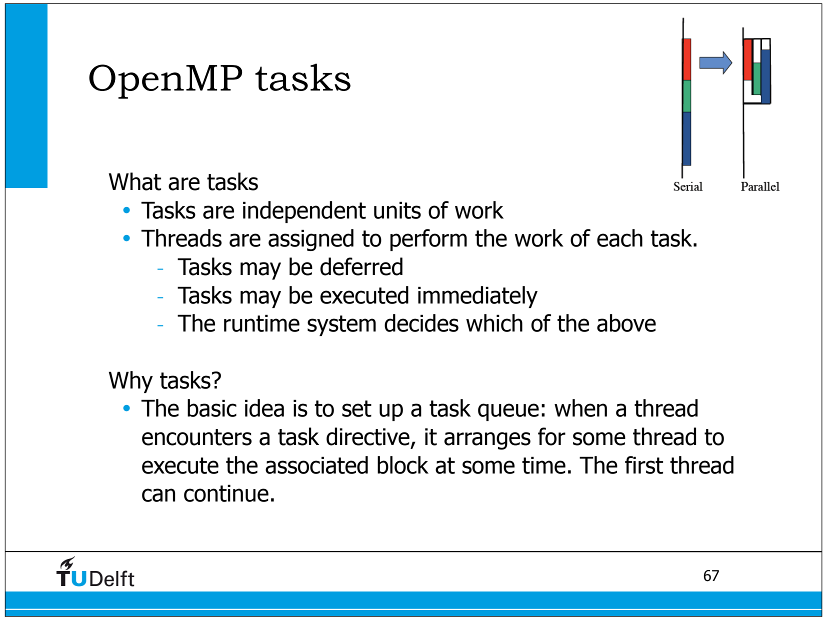
Exercise: PI with MPI and OpenMP



Cuda Computing PI

```
float step = 1.0f / (float)NSET;
float sum = 0.0f;

PiSimple2<<<GRIDDIM, BLOCKDIM>>>
   (d_partials, step, NSET);
CUT_CHECK_ERROR("***PiSimple2
   execution failed!!!***");



CUDA_SAFE_CALL(cudaMemcpy(h_partia
   ls, d_partials,
   fSmallArraySize,
   cudaMemcpyDeviceToHost) );

for (j = 0; j < GRIDDIM; j++)
{
   sum += h_partials[j];
}
Pi = step * sum;
```

```
__global__ void
PiSimple2( float* g_partialOut, float step,
   int NSamples)
{
  const int tid = blockDim.x * blockIdx.x +
   threadIdx.x;
  const int blocksize = blockDim.x;
  const int THREAD_N = blockDim.x * gridDim.x;
  float x, partialsum = 0.0f;

for(int i = tid; i < NSamples; i += THREAD_N){
     x = (i * 0.5f)*step;
     partialsum = partialsum + 4.0f / (1.0f
                    + x*x);
  }

 __shared__  float threadsum[BLOCKDIM];
 threadsum[threadIdx.x] = partialsum;

  __syncthreads();
  float blocksum = 0;
  if (threadIdx.x == 0)   {
    const int blockindex = blockIdx.x;
    for (int i = 0; i < blocksize; i++)
         blocksum += threadsum[i];
    g_partialOut[blockindex] = blocksum;
  }
```



OpenMP tasks

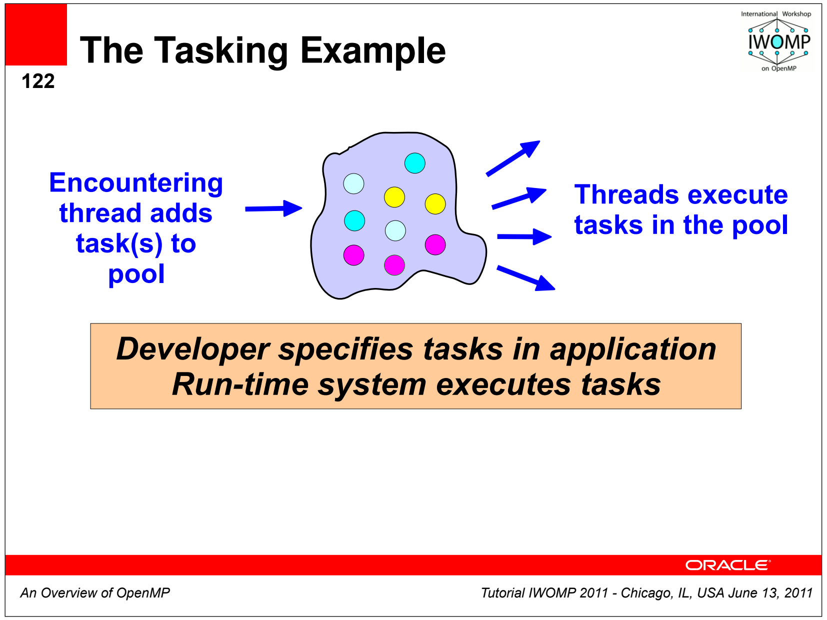
Serial
Parallel

What are tasks

- Tasks are independent units of work
- Threads are assigned to perform the work of each task.
  - Tasks may be deferred
  - Tasks may be executed immediately
  - The runtime system decides which of the above

Why tasks?

- The basic idea is to set up a task queue: when a thread encounters a task directive, it arranges for some thread to execute the associated block at some time. The first thread can continue.



The Tasking Example

International Workshop
IWOMP
on OpenMP
Encountering thread adds task(s) to pool
Threads execute tasks in the pool

Developer specifies tasks in application
Run-time system executes tasks

OpenMP tasks

Tasks allow to parallelize irregular problems
- Unbounded loops
- Recursive algorithms
- Manger/work schemes

A task has
- Code to execute
- Data environment (It owns its data)
- Internal control variables
- An assigned thread that executes the code and the data

OpenMP tasks

```
#pragma omp parallel  -> A parallel region creates a team of threads;
#pragma omp single
{                     -> One thread enters the execution
  ...
#pragma omp task
  { ... }             -> pick up threads „from the work queue‟
  …
#pragma omp taskwait
}                     -> the other threads wait at the end of the single
```

Do not have a good example to show the working of tasks.

Summary

First tune single-processor performance

Tuning parallel programs
- Has the program been properly parallelized?
  - Is enough of the program parallelized (Amdahl’s law)?
  - Is the load well-balanced?
- location of memory
  - Cache friendly programs: no special placement needed
  - Non-cache friendly programs
    - False sharing?
- Use of OpenMP
  - try to avoid synchronization (`barrier, critical, single, ordered`)

Plenty of Other OpenMP Stuff

Scheduling clauses

Atomic

Barrier

Master & Single

Sections

Tasks (OpenMP 3.0)

API routines

Compiling and running OpenMP

Compile with -openmp flag (intel compiler) or -fopenmp (GNU)

Run program with variable:

```
export OMP_NUM_THREADS=4
```

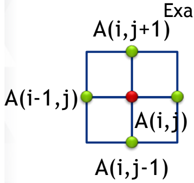
OpenMP 4.5 adds GPU support

Example: Jacobi Iteration

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

Common, useful algorithm

Example: Solve Laplace equation in 2D: $\nabla^2 f(x,y) = 0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

Jacobi Iteration

```
while ( err > tol && iter < iter_max ) {
  err=0.0;


  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

← Convergence Loop

← Calculate Next

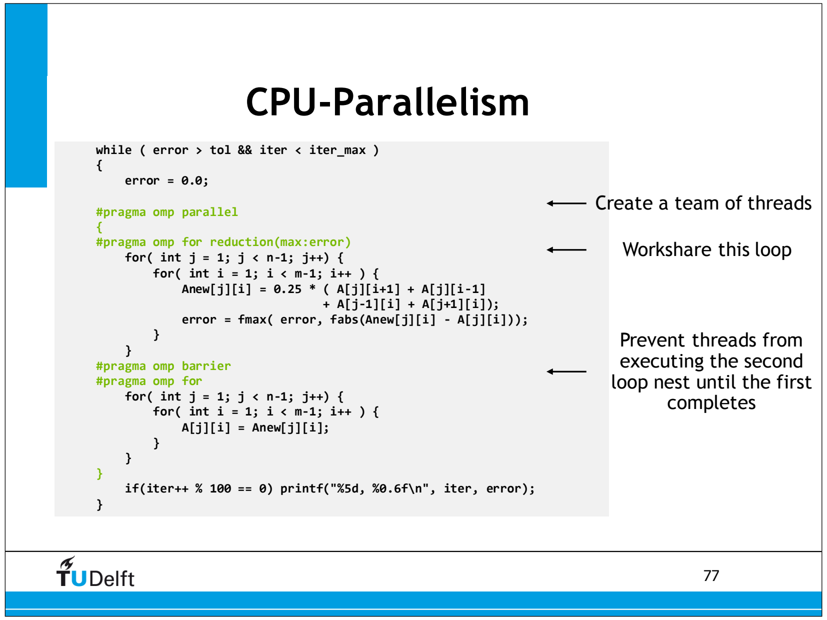
← Exchange Values

CPU-Parallelism

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

← Create a team of threads and workshare this loop across those threads.

← Create a team of threads and workshare this loop across those threads.



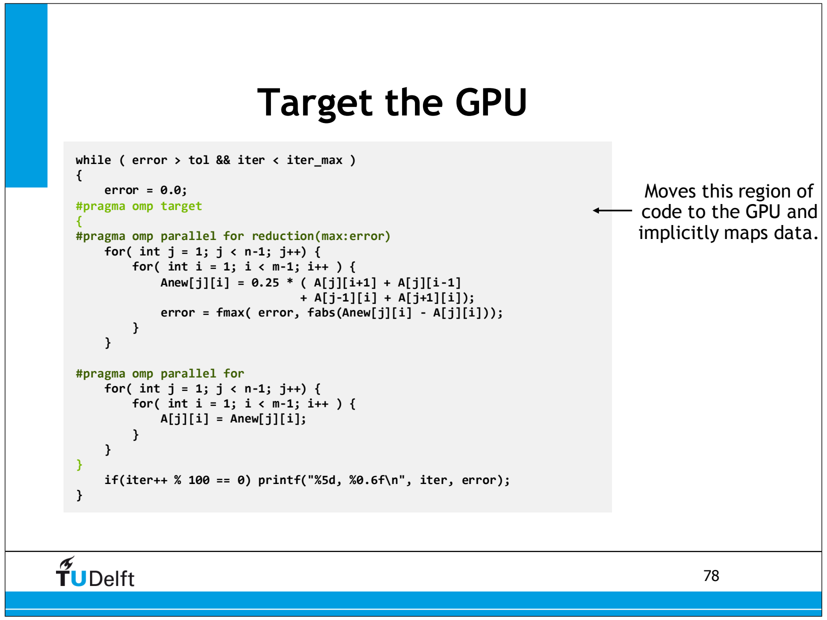
CPU-Parallelism

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp parallel
{
#pragma omp for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp barrier
#pragma omp for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Create a team of threads

Workshare this loop

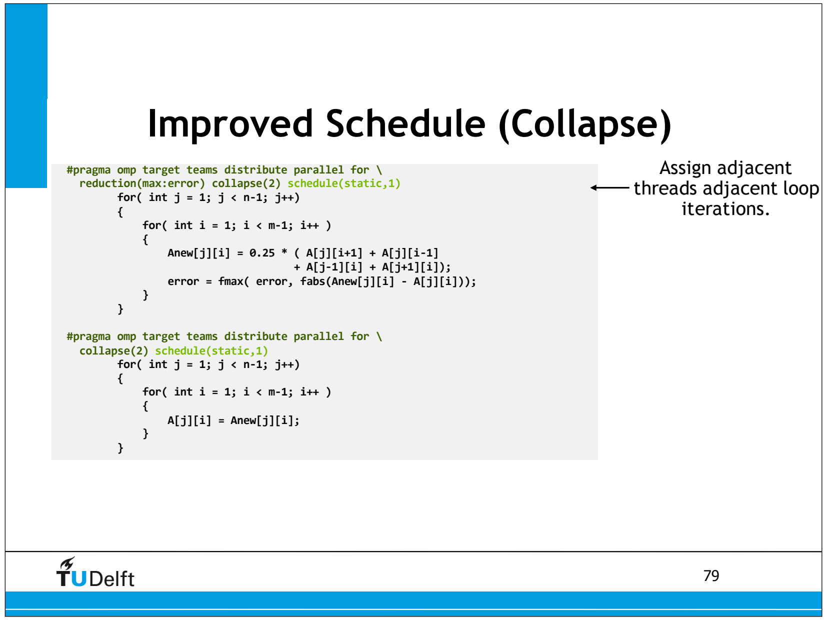
Prevent threads from executing the second loop nest until the first completes



Target the GPU

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target
{
#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Moves this region of code to the GPU and implicitly maps data.



Improved Schedule (Collapse)

```
#pragma omp target teams distribute parallel for \
  reduction(max:error) collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for \
  collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Assign adjacent threads adjacent loop iterations.