OpenMP

OpenMP

Based on compiler directives, can use serial code

Defined by a group of hardware / software companies . The Fortran API was released October 28, 1997. The API C / C + + was released in 1998.

Since release OpenMP 4, even GPGPU is supported!

Portable / multi-platform, includes Unix and Windows NT

Available in C / C + + and Fortran

It can be very easy and simple to use

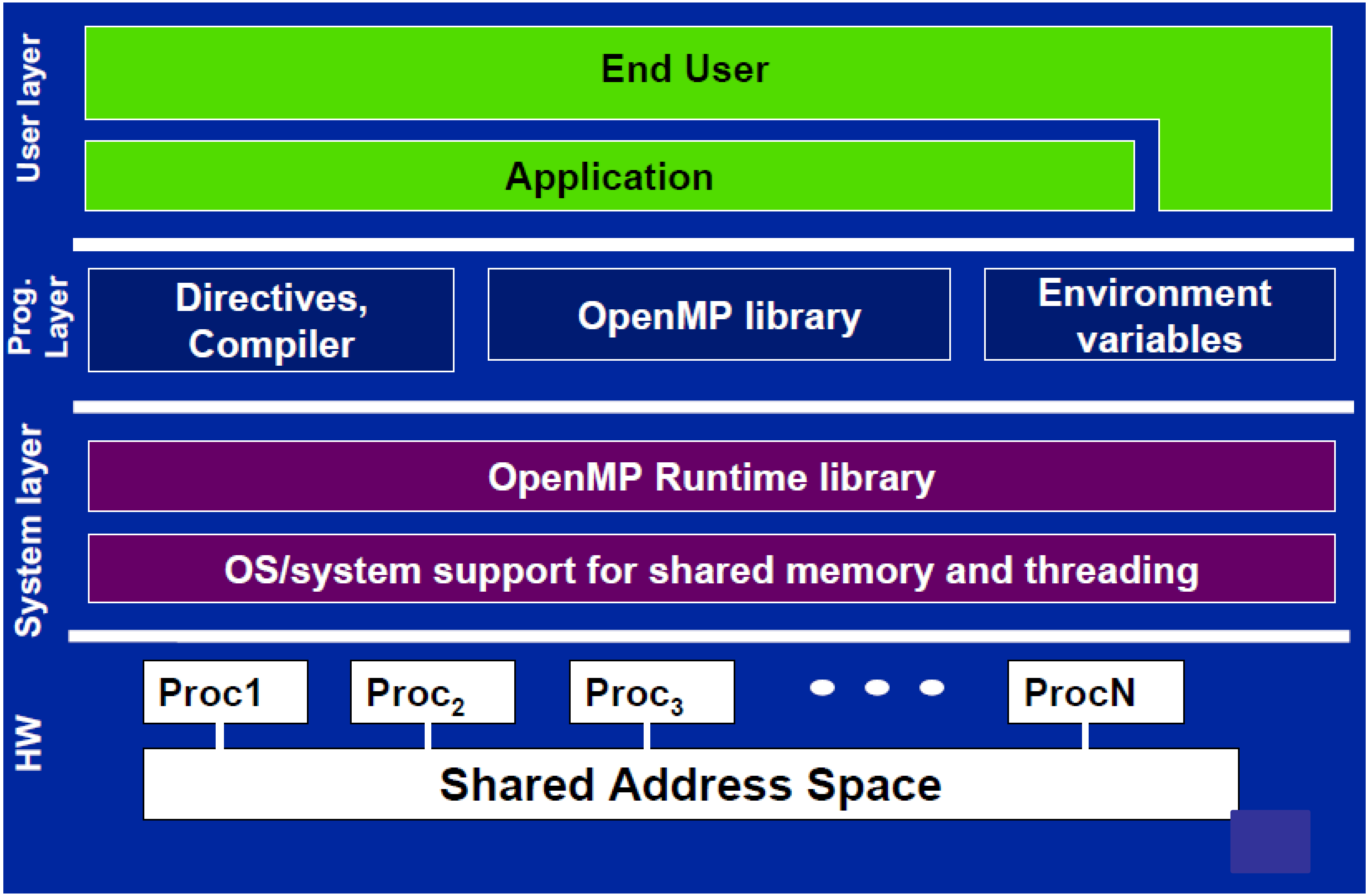
It allows an **incremental approach to parallelism** (i.e., the ability to "convert" a sequential program in parallel, "little by little")

Why OpenMP

- Incremental parallelism models
  - It is necessary to project parallelism to the entire program, but work-sharing directives, etc, can be applied to extend portions of code
  - Different strategies can by applied in the code
- Local parallelism model
  - Applications of work-sharing directives that distribute loop iterations among threads that are computationally costly and functions that can be executed in parallel (data/function decomposition)

Significant parallelism can be implemented by using just 3 or 4 directives

OpenMP Basic Defs: Solution Stack



Reference site (docs, tutorials, specifications, etc): **www.openmp.org**

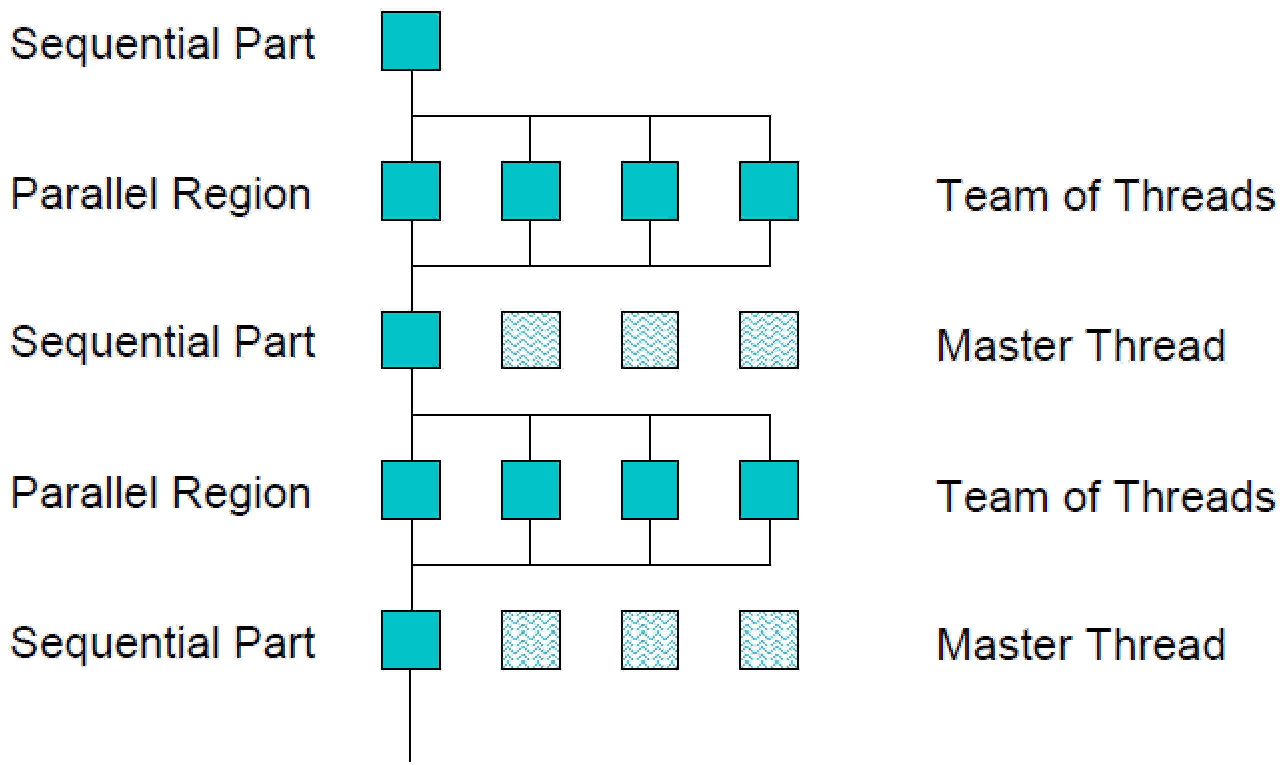
OpenMP vs MPI

Characteristic	***OpenMP***	***MPI***
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes

Conceptual model of execution

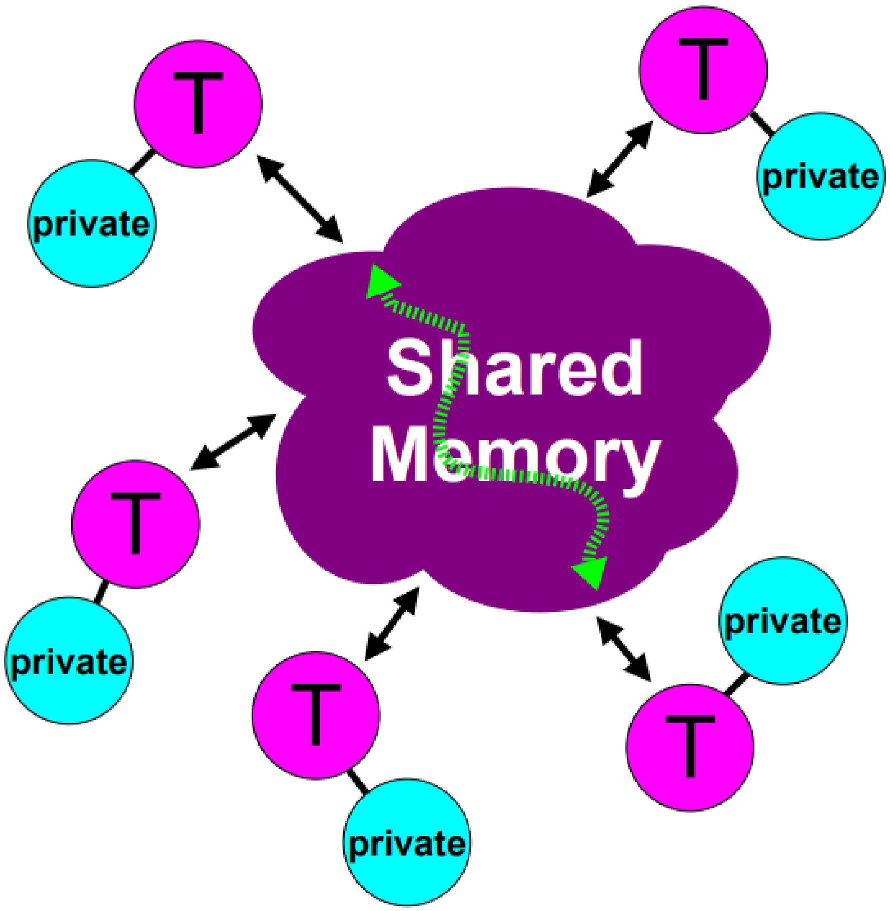
- The conceptual model of OpenMP consists in an alternate serial and parallel execution, called «fork/join»
  - At the beginning and the end there is a serial process (master process)
  - In some parts (parallel regions) some independent threads are launched (**fork**)
  - Each thread can execute concurrent operations
  - At the end of the parallel part, the master thread waits for all threads (**join**)
  - The master thread continues sequentially

Fork/Join Model



OpenMP Memory 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 only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but is mostly implicit



OpenMP: Presentation

- Complier directives
  - Considered as comments for supported languages, to maintain serial versions and non-compatible OpenMP compilers
- Library functions
  - Permit to link the code to OpenMP
    - Function prototypes need to be included
  - Permit to determine/change the number of adopted threads
- Environment variables

OpenMP Components

Directives

- *Parallel region*
- *Worksharing constructs*
- *Tasking*
- *Synchronization*
- *Data-sharing attributes*

Runtime Enviromnent

- *Number of threads*
- *Thread ID*
- *Dynamic thread adjustment*
- *Nested parallelism*
- *Schedule*
- *Active levels*
- *Thread limit*
- *Nesting level*
- *Ancestor thread*
- *Team size*
- *Wallclock timer*
- *Locking*

Environment variables

- *Number of threads*
- *Scheduling type*
- *Dynamic thread adjustment*
- *Nested parallelism*
- *Stacksize*
- *Idle threads*
- *Active levels*
- *Thread limit*

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives

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

Example

```
#pragma omp parallel num_threads(4)
```

- Function prototypes and types in the file:

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

- Most OpenMP constructs apply to a «structured block»
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - It’s ok to have an exit() within the structured block

Pragmas: Overridable preprocessing directives

- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax:

  `#pragma omp` <*rest of pragma*>

  – *In C/C*++

```
#pragma omp directive [clause[[,] clause]…]
Structured block
```

OpenMP Overview

How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions
  - Race condition: when the program's outcome changes as the threads are scheduled differently
- To control race conditions:
  - Use synchronization to protect data conflicts
- Synchronization is expensive, so:
  - Change how data is accessed to minimize the need for synchronization

Runtime library

`OMP_GET_NUM_THREADS()` – Returns the current number of threads

`OMP_GET_THREAD_NUM()` – Returns the id of this thread

`OMP_SET_NUM_THREADS(n)` – Set the desired number of threads

`OMP_IN_PARALLEL()` – Returns TRUE if inside parallel region

`OMP_GET_MAX_THREADS()` - Returns the number of possible threads

How many threads?

- The number of threads in a parallel region is determined by the following factors:

1. Use of NUM_THREADS clause
2. Use of the opm_set_num_threads() library function
3. Setting of the OMP_NUM_THREADS environment variable
4. The implementation default

Threads are numbered from 0 (master thread) to N-1

Identifying threads

- The OpenMP function for identifying threads is `omp_get_thread_num`
  - In C/C++: `int omp_get_thread_num(void)`

- Each thread receives a different return value

- The master thread receives 0
  - Other threads get: 1, 2, 3, … N-1 with threads

`omp_get_thread_num`

- How to use it?
  - Include the file header:

    `#include “omp.h”`

- Where?
  - In a parallel region

- Why?
  - To let each thread work on different data

- What happens out of the parallel region?
  - It returns 0

Changing the number of threads

- The OMP_NUM_THREADS environmental variable specifies the maximum number of threads that will be created in the parallel region
  - For efficiency reasons, the implementation can ignore the declaration

- At the command line:
  - `setenv OMP_NUM_THREADS 5` for csh
  - `export OMP_NUM_THREADS=5` for bash

Threads: fork and join

- The PARALLEL directive creates a parallel region (**<u>fork</u>**), where besides a **master thread** (which executes the serial code) a **variable number of threads** are created
  - We'll see how to define the number later
- From here, all threads execute the code **concurrently** and **independently**
  - Each thread executes the same code
- Only the master thread continues at the end of the parallel region (**<u>join</u>**)

```
#pragma omp parallel [clause list]
/* structured block */

```

Pthreads comparison

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) {
    int a;

    // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Comparison between OpenMP and the corresponding program written with Pthreads

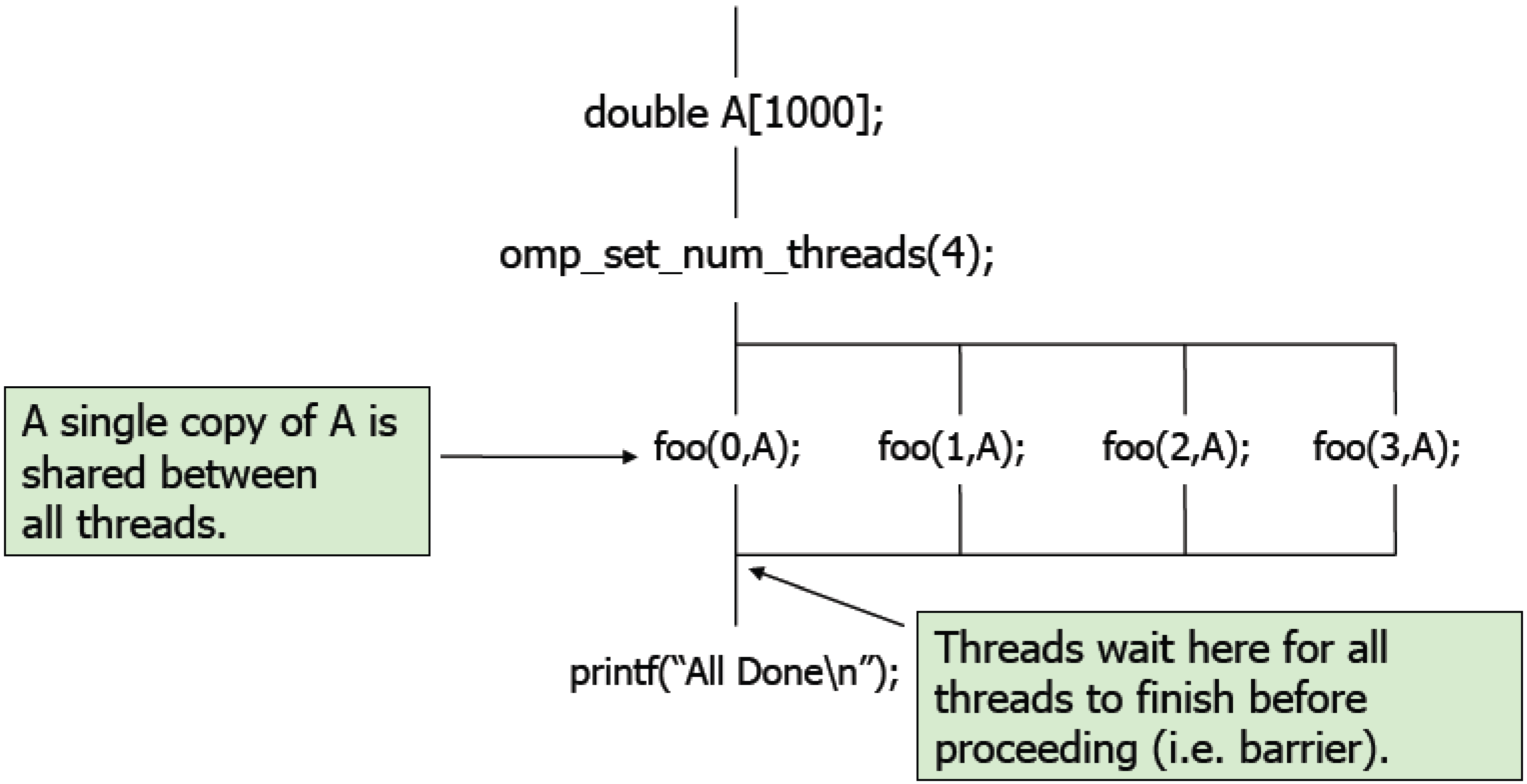
OpenMP: Parallel Regions

- For example, to create a 4-thread parallel region:
  - each thread calls foo(ID,A) for **ID = 0** to **3**

Each thread redundantly executes the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID =omp_get_thread_num();
   foo(ID,A);
}
printf("All Done\n");
```

OpenMP: Parallel Regions



Ciao Mondo! (Hello World!)

```
#include <omp.h>
main () {
int nthreads, tid;
```

To compile:
`gcc -fopenmp hello.c -o hello`

```
/* Fork a team of threads with each thread having a private tid
variable */
#pragma omp parallel private(tid)
  { /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
  {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
  }
} /* All threads join master thread and terminate */

} // main
```

Example

```
#pragma omp parallel
printf("Hello from %d\n", omp_get_thread_num())
```

With 5 threads

```
<bonaccor@poseidon ~/OMP-EX>./WriteThrNumC
Hello from 1
Hello from 2
Hello from 0
Hello from 3
Hello from 4
```

With 2 threads

```
<bonaccor@poseidon ~/OMP-EX>./WriteThrNumF
Hello from        0
Hello from        1
```

Thread: Execution context

- Every thread has its own **execution context**
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
  - static variables
  - dynamically allocated data structures in the heap
  - variables on the run-time stack
  - additional run-time stack for functions invoked by the thread

Shared and Private Variables

- **Shared variable**: has same address in the execution context of **every thread**
- **Private variable**: has different address in execution context of every thread

- A thread cannot access the private variables of another thread!

`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

```
void wrong(){
    int IS = 0;
#pragma parallel for private(IS)
    for(int J=1;J<1000;J++)
        IS = IS + J;
    printf("%i", IS);
}
```

Shared and Private Variables

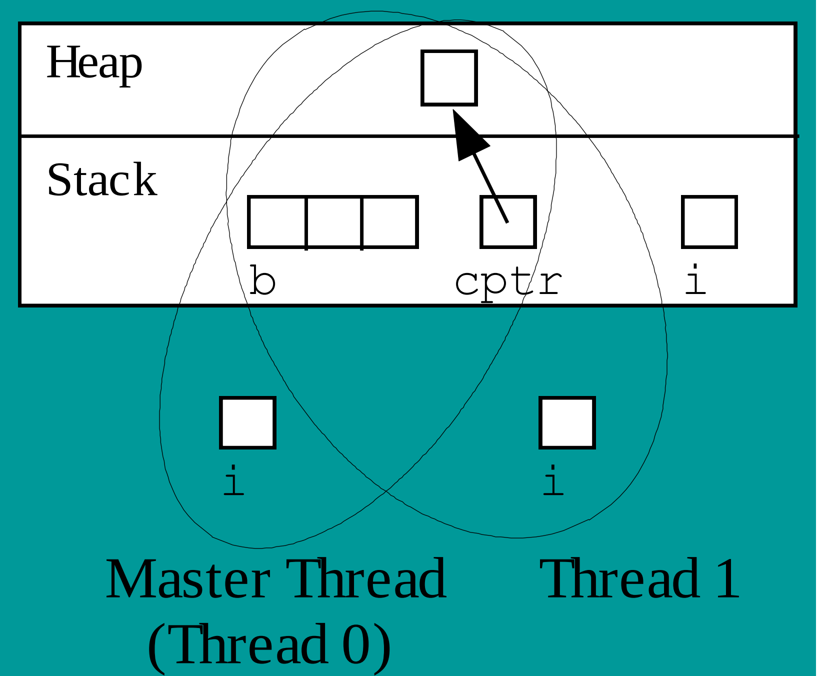
- private (list)
  - No storage association with original object
  - Alle references are to the local object
  - Values are undefined on entry and exit
- shared (list)
  - Data is accessible by all threads in the team
  - All threads access the same address space

C / C++ - General Code Structure

```
#include <omp.h>

main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

Beginning of parallel section. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
  {

  Parallel section executed by all threads
        .
        .
        .

  All threads join master thread and disband

  }

Resume serial code
      .
      .
      .

}
```

Shared and Private Variables

```
int main (int argc, char *argv[])
{
   int b[3];
   char *cptr;
   int i;

   cptr = malloc(1);
#pragma omp parallel for
   for (i = 0; i < 3; i++)
      b[i] = i;
```



By default, variables are of type **<u>shared</u>**, while loop variables are **<u>private</u>**

Dividing computation: Work-sharing Constructs

- Divides the execution of the enclosed region among the members of the team that encounter it
- Work-sharing constructs do not launch new threads
- No implied barrier upon entry to a work sharing construct
- However, there is implied barrier at the end of the work-sharing construct (unless **nowait** is used)

Work-sharing Constructs : Loop Parallelism …

- Loop level parallelism: parallelize only loops!
  - Easy to implement
  - Highly readable code
  - Less than optimal performance (sometimes)
  - Most often used in OpenMP

- C\C++ for loop directive
  - #pragma omp for
- These directives do not create a team of threads but assume there has already been a team forked.
- If not inside a parallel region shortcuts can be used.

  - #pragma omp parallel for

`parallel`, `for`

The difference between `parallel`, `parallel for` and `for` is as follows:

A team is the group of threads that execute currently

At the program beginning, the team consists of a single thread

A `parallel` directive splits the current thread into ***a new team*** of threads for the duration of the next block/statement, after which the team merges back into one

- `for` divides the work of the for-loop among the threads of the ***current team***. It does not create threads, it only divides the work amongst the threads of the currently executing team
- `parallel for` is a shorthand for two commands at once: `parallel` and `for`
- `parallel` creates a new team, and `for` splits that team to handle different portions of the loop

So, if your program **never contains a parallel directive**, there is never more than **one** thread; the master thread that starts the program and runs it, as in non-threading programs

Combined parallel/worksharing construct

- OpenMP shortcut: put the «parallel» and worksharing directive on the same line

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

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

These are equivalent

An example ...

For-loop with independent iterations

```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

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

```
$ cc -xopenmp source.c
$ export OMP_NUM_THREADS=5
$ ./a.out
```

... becomes

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
i=0-199	i=200-399	i=400-599	i=600-799	i=800-999
a[i]	a[i]	a[i]	a[i]	a[i]
+	+	+	+	+
b[i]	b[i]	b[i]	b[i]	b[i]
=	=	=	=	=
c[i]	c[i]	c[i]	c[i]	c[i]

The loop worksharing constructs

- The loop woksharing constructs splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

Loop construct name:

- **C/C++: for**
- **Fortran: do**

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause

Loop worksharing constructs

- A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

Careful! The behaviour at run-time might not be what expected!

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
   j +=2;
   A[i] = big(j);
}
```

Note: loop index "i" is private by default

No loop-carried dependency

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
   int j = 5 + 2*i;
   A[i] = big(j);
}
```

<u>Note</u>: While OpenMP is efficient and simple (for adding parallelism to an application), there are some things to keep in mind. For example, the <u>index variable</u> in the loop for external parallel-type is **<u>private</u>**, but the <u>index variables for nested</u> for loops are **<u>shared</u>** by default. However, when you have nested loops, usually you want the inner loop have private index variables. So, you use the **<u>private clause</u>** to specify these variables

Data Sharing: Firstprivate clause

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

```
void useless() {
    int tmp = 0;
 #pragma omp parallel for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
      tmp += j;
    printf("%d\n", tmp);
 }
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

Data Sharing: Lastprivate clause

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

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
   lastprivate(tmp)
   for (int j = 0; j < 1000; ++j)
      tmp += j;
   printf("%d\n", tmp);
}
```

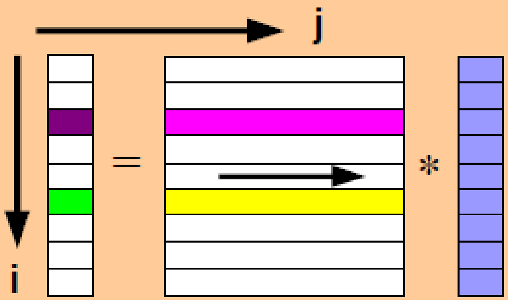
Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the "last sequential" iteration (i.e., for j=999)

Example: Matrix – vector product

`default(none)`:
we need to explicit the type of all variables

```
#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 =    b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

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

... etc ...

`If` Clause

- if (scalar expression)
  - Only execute in parallel if expression is true
  - Otherwise, execute serially

```
#pragma omp parallel if (n > some_threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

barrier/1

- Suppose we run each oh these two loops in parallel over i

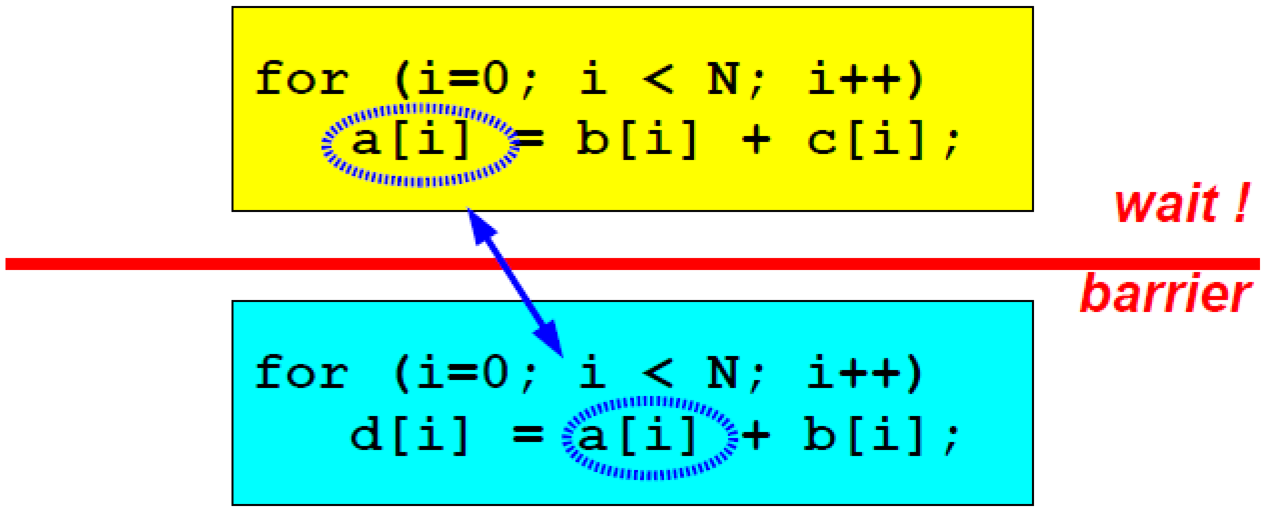
```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day): WHY?

barrier/2

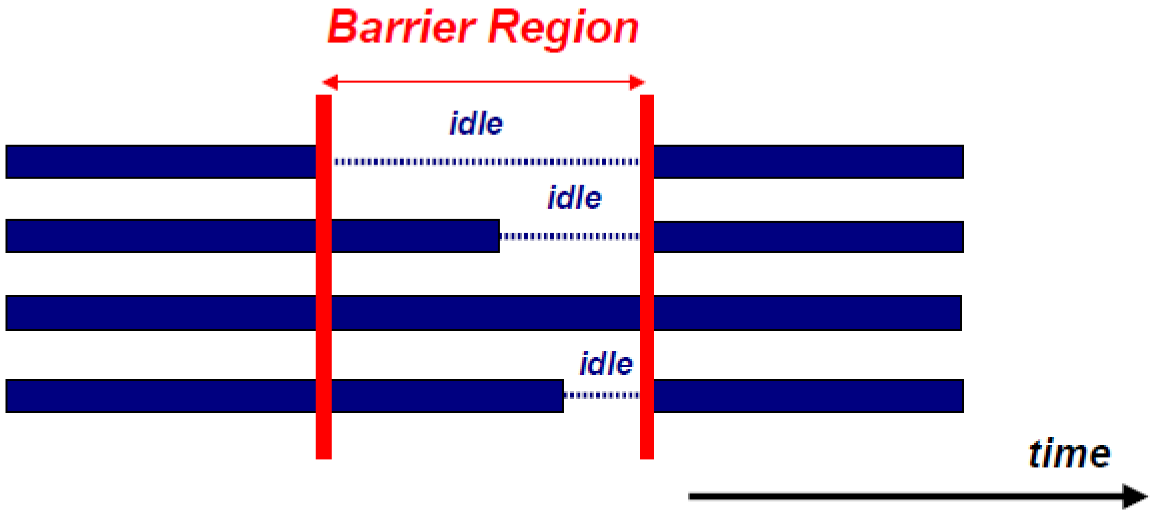
- We need to have updated all of a[] first, before using a[]



- All threads wait at the barrier point and only continue when ALL threads have reached the barrier point
- If there is the guarantee that the mapping of iterations onto threads is identical for both loops, there will be no data race

barrier/3

- Barrier syntax




```
#pragma omp barrier
```

Clausola `nowait`

- To minimize synchronization, some OpenMP directives/pragmas support the optional nowait clause
- If present, threads do not sync of that particular construct

```
#pragma omp for nowait
{
         :
}
```

Example

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];

#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

          ....
#pragma omp barrier
    scale = sum(a,0,n) + sum(z,0,n) + f;
          ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop
(work is distributed)

parallel loop
(work is distributed)

parallel region

synchronization

Statement is executed by all threads

Synchronization: Barrier

- Barrier: Each thread waits until 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_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

When to use barriers

- If data is updated asynchronously and data integrity is at risk
- Examples:
  - Between parts in the code that read and write the same section of memory
  - After one timestep/iteration in a solver
- Unfortunately, barriers tend to be expensive and also may not scale to large number of processors
- Locks can be used:

```
omp_init_lock(), omp_set_lock(),
omp_unset_lock(), omp_test_lock(),
omp_destroy_lock()
```

Getting timings

- Returns number of **physical processors (cores?)** available for use by the parallel program

```
int omp_get_num_procs (void)
```

- To get timings (in seconds since a fixed point in the past):

```
double omp_get_wtime();
```

Example: `omp_get_wtime`

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

int main(){
      double start = omp_get_wtime();
      sleep(10);
      double end = omp_get_wtime();

      printf(start: %.16g\n end: %.16g\n Time:
      %.16g\n”, start, end, end-start);

return 0;
}
```

Homework ☺

- Hello world!
- Set the number of threads equal to the number of cores of your “parallel” machine
- Print OpenMP settings (procs, etc)
- Play with changing shared and private variables in parallel sections and checking output ☺
- Summation of two vectors/ matrices (**ok**!)
- Copy one matrix to another (**swap** ☺)
- Summation of elements of a vector /matrix (<u>**oops**</u>!)
- Dot product (<u>**oops**</u>**!**)
- Take timings with different #threads (up to your machine max proc num)!
- …