Course Information

- **Instructor:** Ekaterina Vladimirovna Bolgova
  - **E-mail:** *ekaterina_bolgova@itmo.ru*
- **Course language:** English.
- **Classes starts at** September 14, 2019.
- **Duration**: 1 semester.
- **Classes' types:**
  - Lectures (4): 14/09, 21/09, 28/09, 05/10
  - Labs (programming tasks): starts at October 19, 2019
- **Exam (2/1/0 question(s)):**
  - Submit all labs (code + report)
  - Make presentation (at least 1)
  - Attendance & activity
  - Bonus: if make workshop

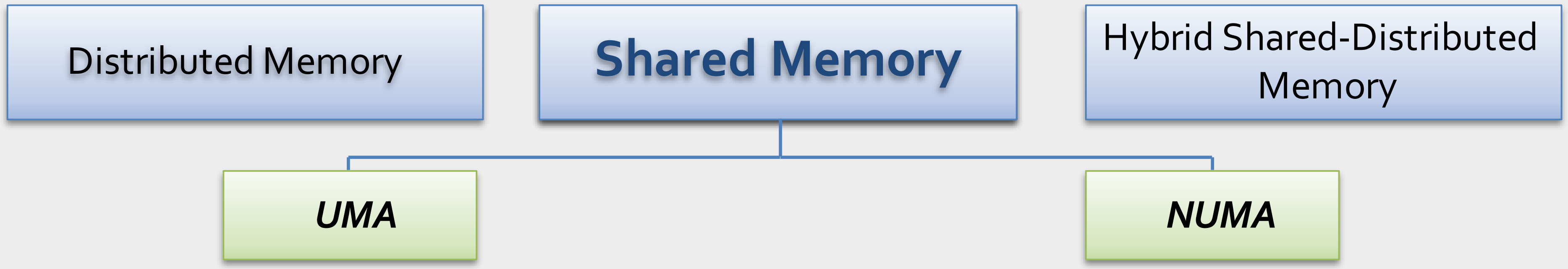
ITMO UNIVERSITY

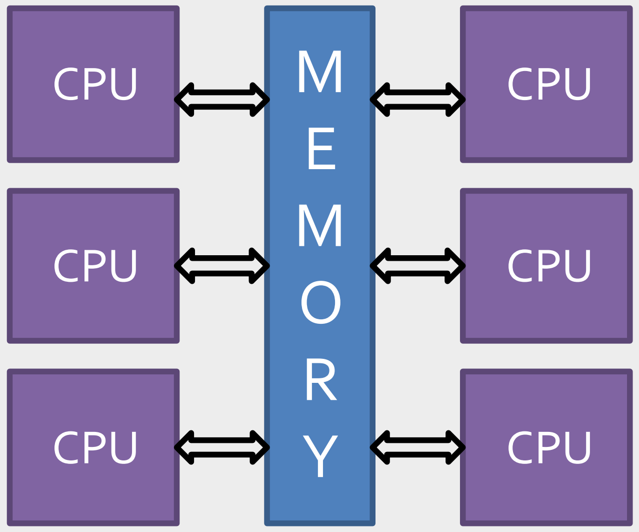
Basics of Parallel Programming with OpenMP

PhD Katerina Bolgova
eScience Research Institute & HPC Department

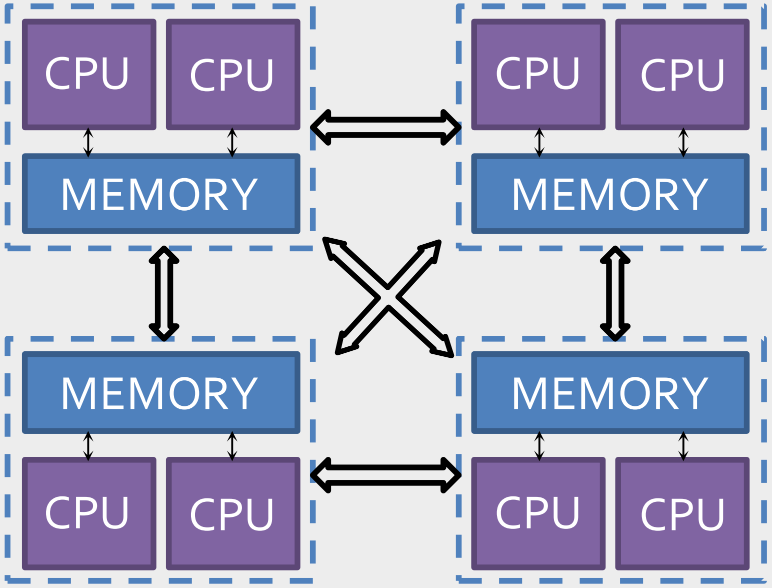
Background

- **What are Parallel Computer Memory Architectures?**



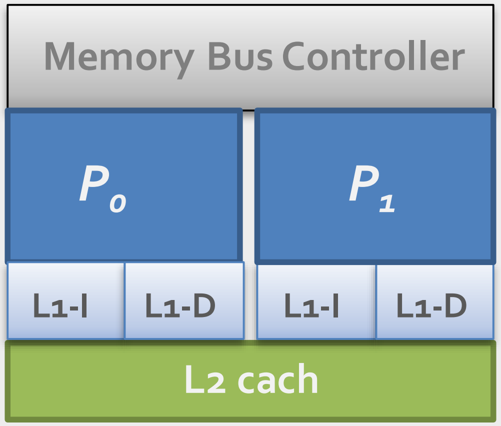


Symmetric Multiprocessor

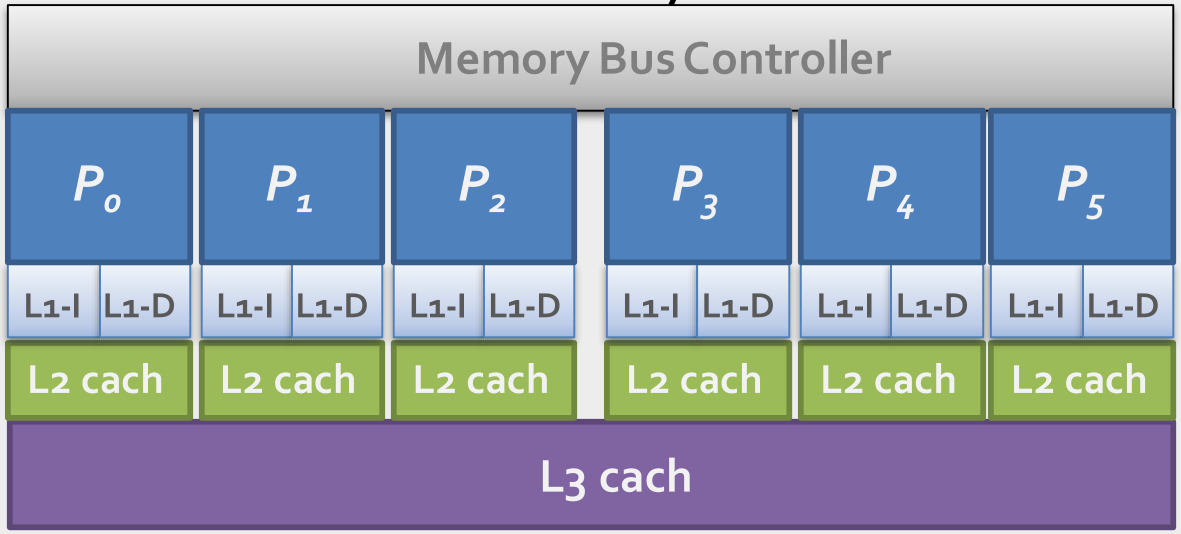


Background

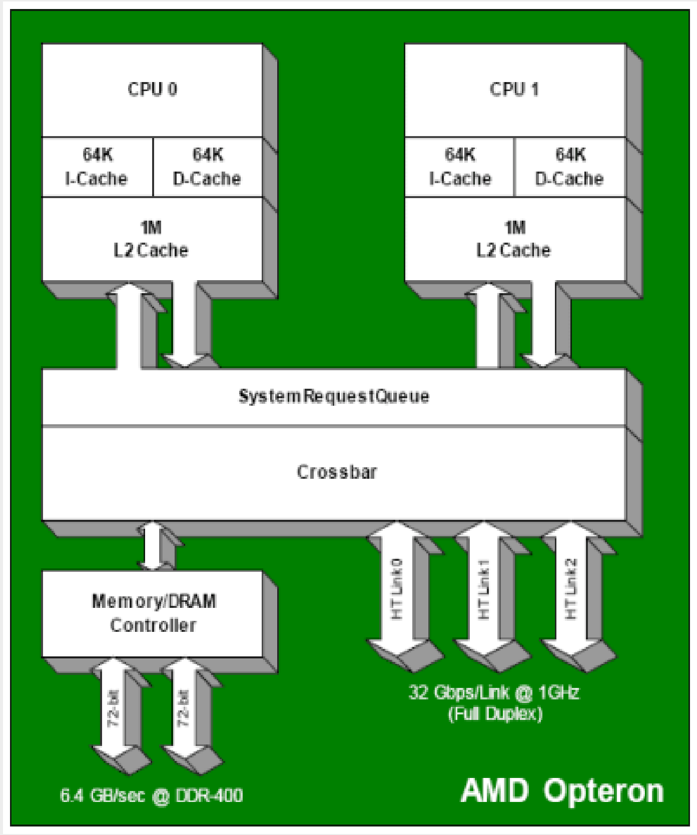
Intel Core 2 Duo



Intel Core i7



AMD Opteron



OpenMP Overview

OpenMP :

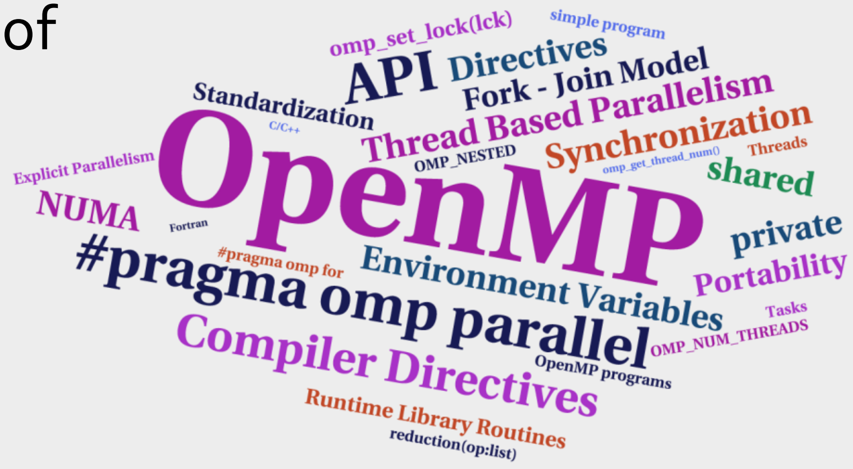
- is an Application Program Interface (**API**) that may be used to explicitly direct *multi-threaded, shared memory parallelism*
- is managed by the OpenMP Architecture Review Board (or *OpenMP ARB*)
- provides a portable, scalable model for developers of shared memory parallel applications.
- supports C/C++ and Fortran
- ***is a set of compiler directives and library routines for parallel application programmers***

Brief history

- The OpenMP standard specification started in 1997. (The version for Fortran appeared).
- In 1998 the C/C++ standard was released
- Since 2005, C and Fortran specifications have been released together
- The latest version (4.0) of the specification was released in July 2013

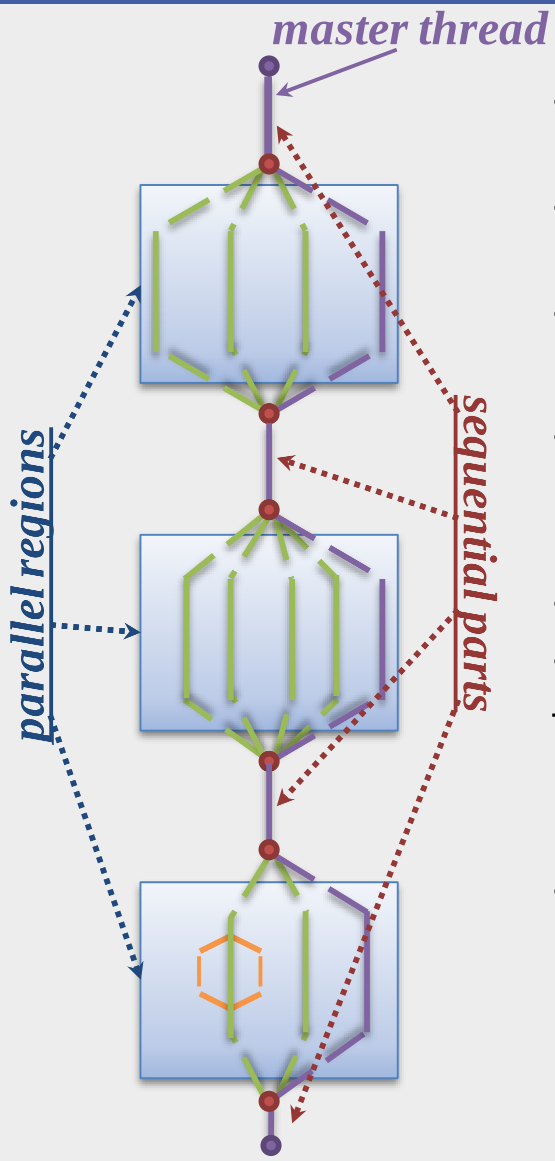
Advantages of OpenMP

- Provide a ***standard*** among a variety of shared memory architectures
- Establish a simple and limited set of directives for programming shared memory machines



- Significant parallelism can be implemented by using just 3 or 4 directives
- Provide capability to ***incrementally*** parallelize a serial program
- The API is specified for C/C++ and Fortran
- Public forum for API and membership
- Most major platforms have been implemented including Unix/Linux platforms and Windows

OpenMP Programming Model



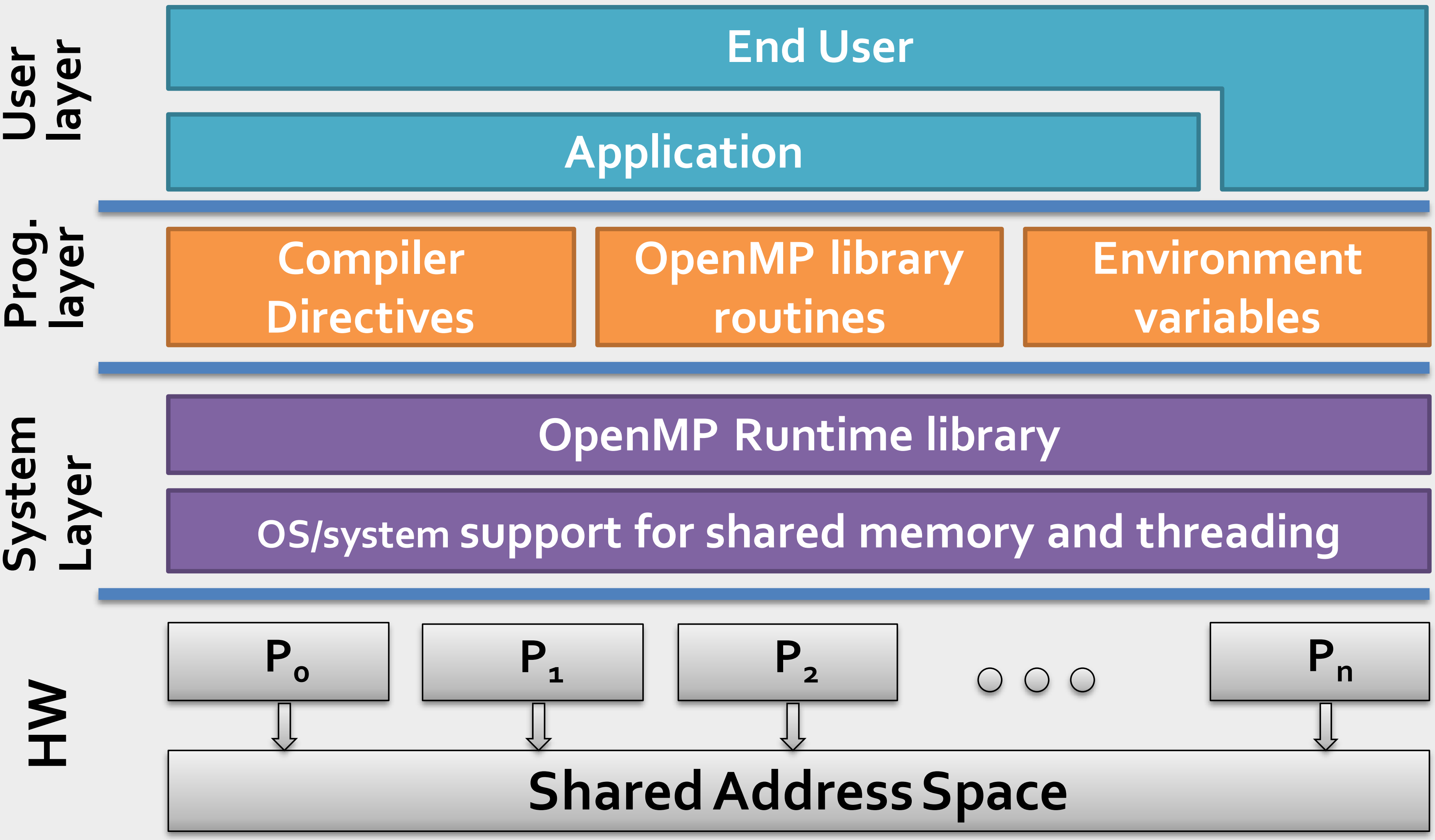
- OpenMP program begins as a single process: the ***master thread***
- **FORK:** the master thread then creates a ***team*** of parallel ***threads***
- Parallel execution
- **JOIN:** the team threads terminate, leaving only the master thread

- Parallelism added ***incrementally***
- The number of parallel regions and the threads that comprise them are arbitrary

- Each thread can spawn another team of threads (nested parallelism)

FORK-JOIN Model

OpenMP Solution Stack



Source: Tim Mattson, Introduction to OpenMP

OpenMP Directives Format

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

- Each directive starts with **#pragma omp**
- Compilers can ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided
- Directives are case-sensitive
- An OpenMP executable directive applies to at most one succeeding statement, which must be a structured block
- Long directive lines can be "continued" on succeeding lines with a backslash ("\") at the end of a directive line
- ***for C/C++ it needs to include the <omp.h> header file***

Example:

```
#pragma omp parallel default(shared)   \
                          private(beta,pi)
```

Thread Creation

```
#pragma omp parallel [clause ...]  newline
      structured_block
```

- When thread reaches this directive, it creates a *team* of threads
- All threads will execute code of this parallel region
- There is an implied barrier at the end of a parallel section
- The number of threads in the team remains constant for the duration of that parallel region

Example:

```
#include <omp.h>
int main(){
      #pragma omp parallel
      {
            int ID = omp_get_thread_num();
            cout<<" hello ("<< ID <<")\n";
            cout<<" world ("<< ID <<")\n";
      }
}
```

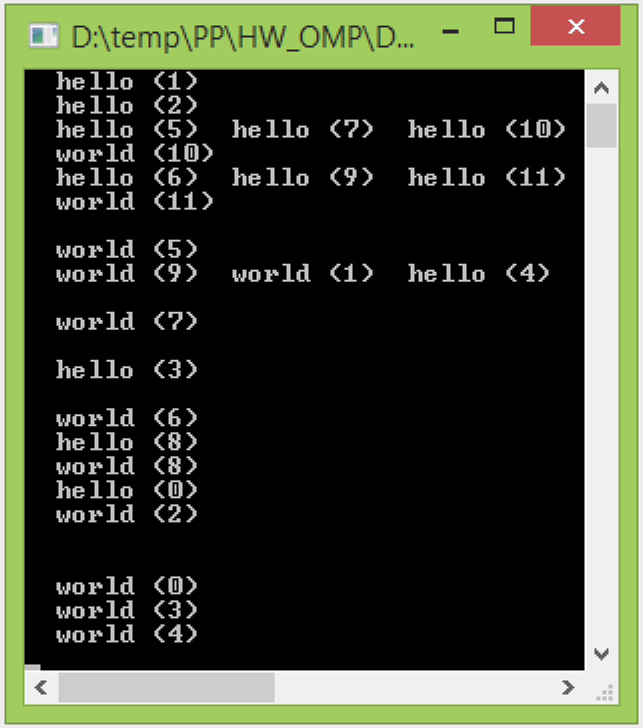
OpenMP include file

Parallel region

End of the Parallel region

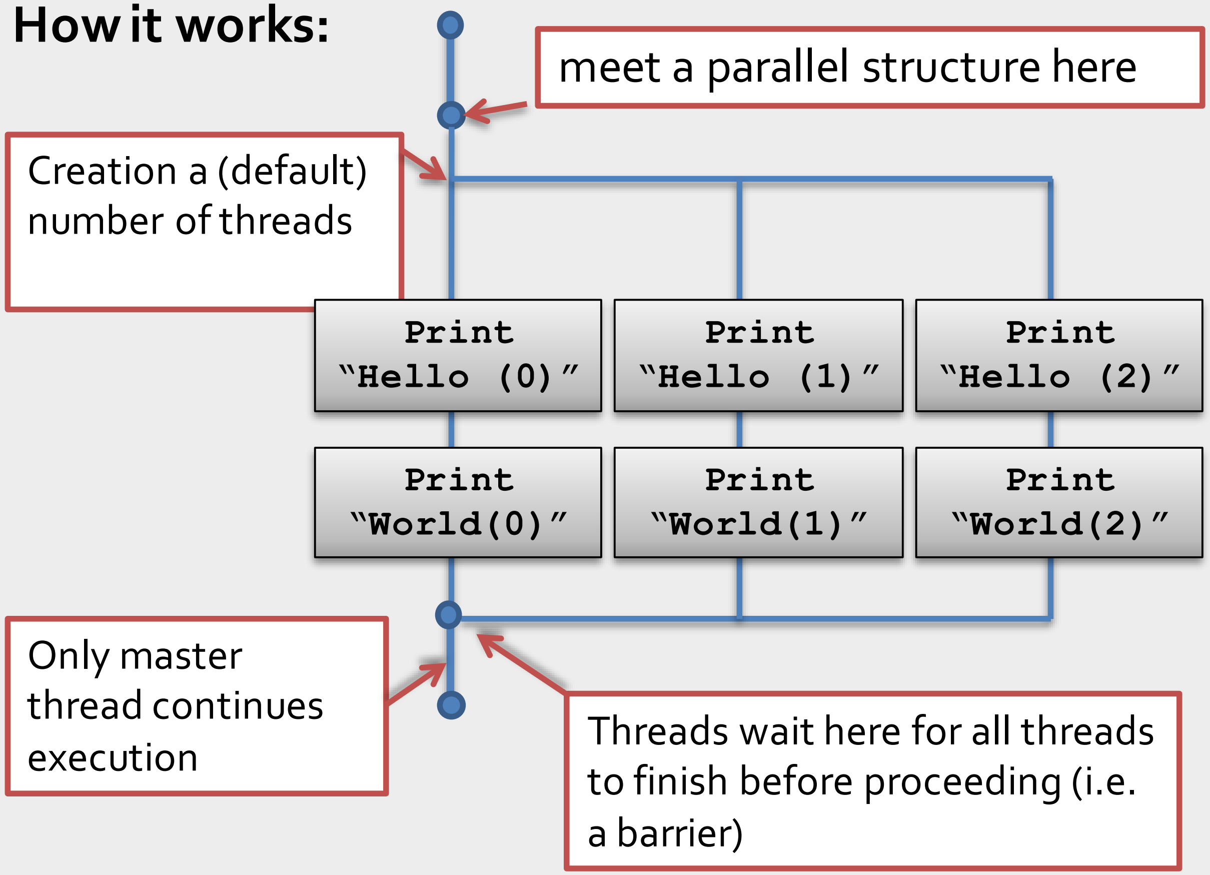
Thread Creation

Sample Output:



```
hello (1)
hello (2)
hello (5)  hello (7)  hello (10)
world (10)
hello (6)  hello (9)  hello (11)
world (11)

world (5)
world (9)  world (1)  hello (4)

world (7)

hello (3)

world (6)
hello (8)
world (8)
hello (0)
world (2)


world (0)
world (3)
world (4)
```

How it works:


meet a parallel structure here
Creation a (default) number of threads
Print "Hello (0)"
Print "Hello (1)"
Print "Hello (2)"
Print "World(0)"
Print "World(1)"
Print "World(2)"
Only master thread continues execution
Threads wait here for all threads to finish before proceeding (i.e. a barrier)

Clauses of Parallel Construct Overview

- **num_threads** (integer-expression **)** – explicitly determine the number of threads
- **private** (list **)** – each thread will have a local copy and use it as a temporary variable
- **firstprivate** (list) – like *private* except initialized to original value
- **shared** (list **)** – the data within a parallel region is shared
- **default (shared | none)** – allows the programmer to state that the default data scoping within a parallel region will be either *shared*, or *none*
- **reduction** (operator **:**list **)** – a safe way of joining work from all threads after construct
- **if** (scalar-expression **)** – This will cause the threads to parallelize the task only if a condition is met. Otherwise the block executes serially

Defining the Time of Program Execution

- Time in Seconds since a fixed point in the past:

```
double omp_get_wtime(void)
```

- *Scheme of using the **omp_get_wtime function**:*

```
double t_1, t_2, dt;
t_1 = omp_get_wtime();
...
t_2 = omp_get_wtime();
dt = t_2 - t_1;
```

Determining Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the *if* clause

```
#pragma omp parallel if (NMAX>LIMIT)
```

  - Setting of the ***num_threads*** clause

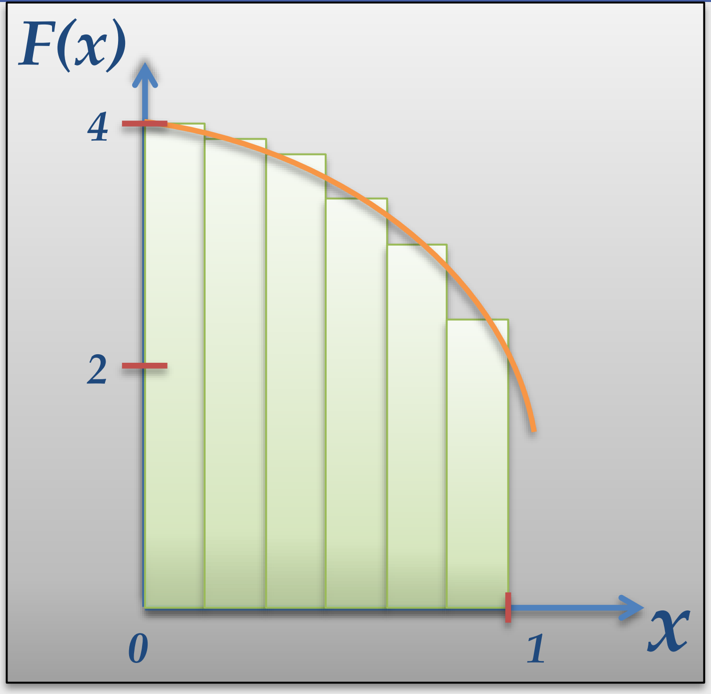
```
#pragma omp parallel num_threads(4)
```

  - Use of the **omp_set_num_threads()** library function

```
void omp_set_num_threads (int num_threads);
```

  - Setting of the **OMP_NUM_THREADS** environment variable

Sample: PI evaluation



We know that:

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

We can approximate the integral as a sum of rectangles:

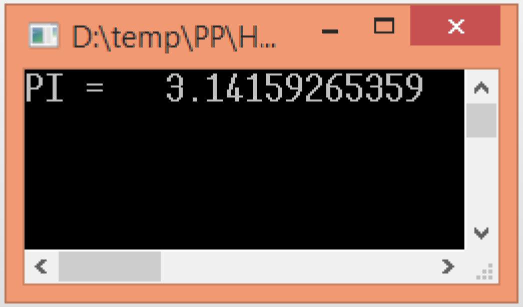
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi,$$ where

Δx – **rectangle width**

$F(x_i)$ – **rectangle height**

PI: Serial Program

```
static long cntSteps=10000000;
double step;

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

      step = 1.0/(double)cntSteps;

      for (i=0; i<cntSteps; i++)    {
                  x = (i + 0.5)*step;
                  sum = sum + 4.0/(1.0+ x*x);
      }
      pi = sum*step;

      cout << "PI = "<<setw(15)<<setprecision(12)<<pi;
}
```


D:\temp\PP\H...

```
PI =    3.14159265359
```

PI: Simple Parallel Program

```
#include <omp.h>
static long cntSteps=10000000;     double step;
#define thrdsCount = 2
void main() {
      double pi,sum[thrdsCount];
      int i;
      step = 1.0/(double)cntSteps;
   #pragma omp parallel  num_threads(thrdsCount)
   {
      int i,id; double x;
      id = omp_get_thread_num();
      for (i=id,sum[id]=0.0;i< cntSteps; i+=thrdsCount) {
            x = (i + 0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
      }
   }
      for(i=0, pi=0.0; i< thrdsCount;i++)
            pi += sum[i] * step;
      cout << "PI = "<<setw(15)<<setprecision(12)<<pi;
}
```

The SPMD design pattern

(Single Program Multiple Data)

PI: Simple Parallel Program

```
#include <omp.h>
static long cntSteps=10000000; double step;
#define thrdsCount = 2
void main () {
       double pi, sum[thrdsCount];
       int i;
       step = 1.0/(double)cntSteps;
   #pragma omp parallel num_threads(thrdsCount)
   {
      int i,id; double x;
       id = omp_get_thread_num();
       for (i=id, sum[id]=0.0;i<cntSteps; i+=thrdsCount) {
              x = (i + 0.5)*step;
              sum[id] += 4.0/(1.0 + x*x);
       }
   }
       for(i=0, pi=0.0; i<thrdsCount;i++)
              pi += sum[i] * step;
       cout << "PI = "<<setw(15)<<setprecision(12)<<pi;
}
```

#Th	T_p	S_p
1	0,79	1
2	0,51	1,55
4	0,46	1,72
8	0,43	1,84
16	0,36	2,19

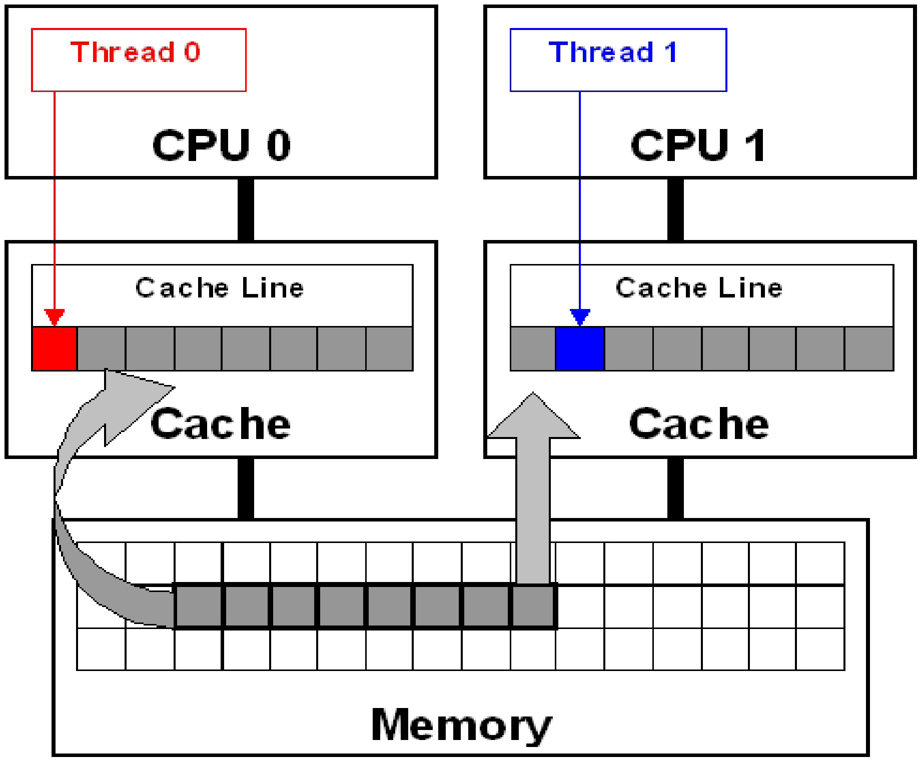
False sharing

- Modifying variables that reside on the same cache line by threads on different processors will cause the cache lines to "slosh back and forth" between threads.

- It's the reason of "**false sharing**"

- Combining scalars to an array program is a very common way to support creation of an SPMD
- But most likely the array elements are contiguous in memory and hence share cache lines

•Simple solution: Pad arrays



PI: Simple Parallel Program

```
#include <omp.h>
static long cntSteps=10000000; double step;
#define thrdsCount = 2
#define PAD 8 // assume 64 bytes L1 cache line size
void main() {
       double pi, sum[thrdsCount][PAD];
       int i;
       step = 1.0/(double)cntSteps;
   #pragma omp parallel num_threads(thrdsCount)
   {
       int i,id; double x;
       id = omp_get_thread_num();
       for(i=id, sum[id][0]=0.0;i<cntSteps; i+=thrdsCount){
              x = (i + 0.5)*step;
              sum[id][0] += 4.0/(1.0 + x*x);
       }
   }
       for(i=0, pi=0.0; i<thrdsCount;i++)
              pi += sum[i][0] * step;
       cout << "PI = "<<setw(15)<<setprecision(12)<<pi;
}
```

#Th	1st T_p	2nd T_p	S_p
1	0,79	0,78	1
2	0,51	0,41	1,9
4	0,46	0,23	3,39
8	0,43	0,14	5,57
16	0,36	0,13	6,0

Padding arrays doesn't resolve the "false sharing" problem

Work-Sharing in OpenMP

Types of Work-Sharing Constructs (WShCs):

- LOOP CONSTRUCT
- SECTIONS/SECTION CONSTRUCTS
- SINGLE CONSTRUCT
- WORKSHARE CONSTRUCT
- TASK CONSTRUCT

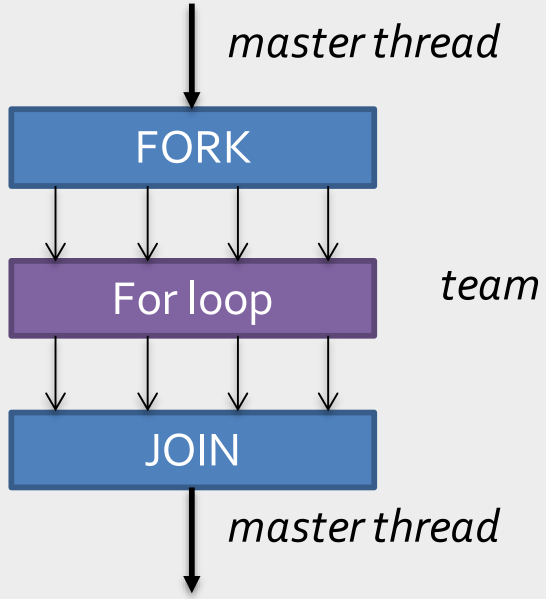
- A **WShC** divides the execution of code region among the members of the team
- **WShC**s do not launch new threads
- There is no implied barrier upon entry to a **WShC**, however there is an implied barrier at the end of a **WShC**.
- A **WShC** must be enclosed within a parallel region
- **WShC**s must be encountered by all members of a team or none at all

Loop Construct

Directive format:

```
#pragma omp for [clause,... ] new-line
      for-loop
```

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



- **for** directive specifies that the iterations of the loop (following it) must be executed in parallel by the team
- The loop variable (*i* in this case) is made "private" to each thread by default

Clauses of For Construct Overview

- **private** (list) – each thread will have a local copy and use it as a temporary variable
- **firstprivate** (list) – like *private* except initialized to original value
- **lastprivate** (list) – each variable from the sequentially last iteration of the associated loop is assigned to the variable's original object
- **reduction** (operator: list) – a safe way of joining work from all threads after construct
- **schedule (type[, chunk])** – describes how iterations of the loop are divided among the threads in the team **1. Task for report**
- **collapse (n)** — Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause
- **ordered** – the iterations of the loop must be executed as they would be in a serial program
- **nowait** – threads do not synchronize at the end of the parallel loop

Construct Comparison

Serial code:

```
for(i=0;i<n;i++)  {
        a[i] = a[i-1]+c[i];
}
```

Parallel region: SPMD pattern

```
#pragma omp parallel
{
    int id, i, Nths, istart, iend;
    id = omp_get_thread_num();
    Nths = 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-1] + c[i];
    }
}
```

Parallel region: worksharing

```
#pragma omp parallel
{
    #pragma omp for
    for(i=0;i<n;i++) {
        a[i] = a[i-1] + c[i];
    }
}
```

or

```
#pragma omp parallel for
    for(i=0;i<n;i++) {
        a[i] = a[i-1] + c[i];
    }
```

Sample of For construct

```
#include <omp.h>
#define NMAX 1000
main () {
   int i, j, sum;
   float a[NMAX][NMAX];
   <data initialization>

   #pragma omp parallel for shared(a) private(i,j,sum)
   {
     for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=0; j < NMAX; j++)
           sum += a[i][j];
        printf ("Sum of %d-string equals %f\n",i,sum);
    }
}
```

Data-Sharing Attribute Clauses

- Any variable in OpenMP is either **shared** or **private**
- Data-Sharing Attribute Clauses are used to explicitly define how variables should be scoped

- Data-Sharing Attribute Clauses include:
  - **shared,**
  - **private,**
  - **firstprivate,**
  - **lastprivate,**
  - **reduction,**
  - **default,**
  - **copyin,**
  - **copyprivate.**

- All the variables of the program are ***shared by default***

Data-Sharing Attribute Clauses

```
int sum = 0, a[NMAX], j;
    for (j=0; j < NMAX; j++)
           sum += a[j];
```

- The code combines values into a single variable (sum)
  - there is a true dependence between loop iterations that can't be trivially removed

- Such situation is called a ***Reduction*** and it is frequently used.

REDUCTION Clause

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

- A private copy of each list variable is created for each thread of a team
- Local copies are initialized depending on the "op" (1 for "*" and 0 for "+")
- At the end of the reduction the final value of each of the local copies are combined with the original global value using the operator specified
- The variables in "list" must be shared in the enclosing parallel region

PI: PP with Worksharing and Reduction

```
#include <omp.h>
static long cntSteps=10000000; double step;

void main() {
      double x, pi, sum = 0.0;
      int i;
      step = 1.0/(double)cntSteps;
  #pragma omp parallel
  {
      double x;
      #pragma omp for reduction(+:sum)
      for (i=0; i<cntSteps; i++)    {
                  x = (i + 0.5)*step;
                  sum += 4.0/(1.0+ x*x);
      }
  }
      pi = sum*step;
      cout << "PI = "<<setw(15)<<setprecision(12)<<pi;
}
```

Data-Sharing Attribute Clauses

SHARED Clause

```
Format: shared(list)
```

- declares variables in its list to be shared among all threads in the team
- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access shared variables

DEFAULT Clause

```
Format: default(shared|none)
```

- allows the user to specify a default scope for all variables in the lexical extent of any parallel region
- Using ***none*** as a default requires that the programmer explicitly scope all variables
- Specific variables can be exempted from the default using the shared clause

Data-Sharing Attribute Clauses

PRIVATE Clause

`Format: private(list)`

- declares variables in its list to be private in each thread of the team
- Variables declared private should be uninitialized for each thread
- The original object referenced by the variable has an indeterminate value upon entry to the construct, and has an indeterminate value upon exit from the construct

FIRSTPRIVATE Clause

`Format: firstprivate(list)`

- private clause with automatic initialization of the variables in its list
- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct

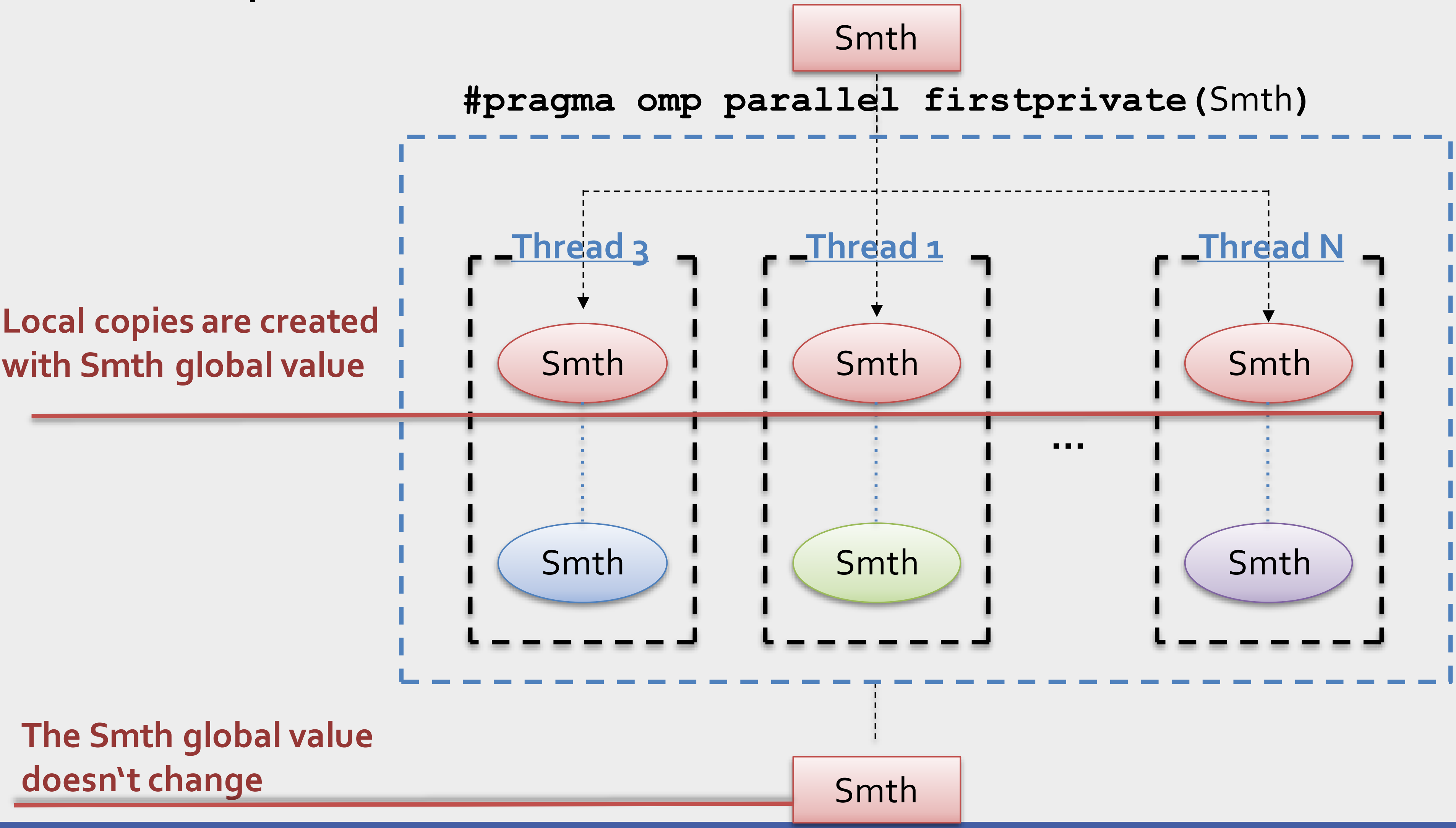
LASTPRIVATE Clause

`Format: lastprivate(list)`

- private clause with a copy from the last (sequentially) loop iteration or section to the original variable object

Data-Sharing Attribute Clauses

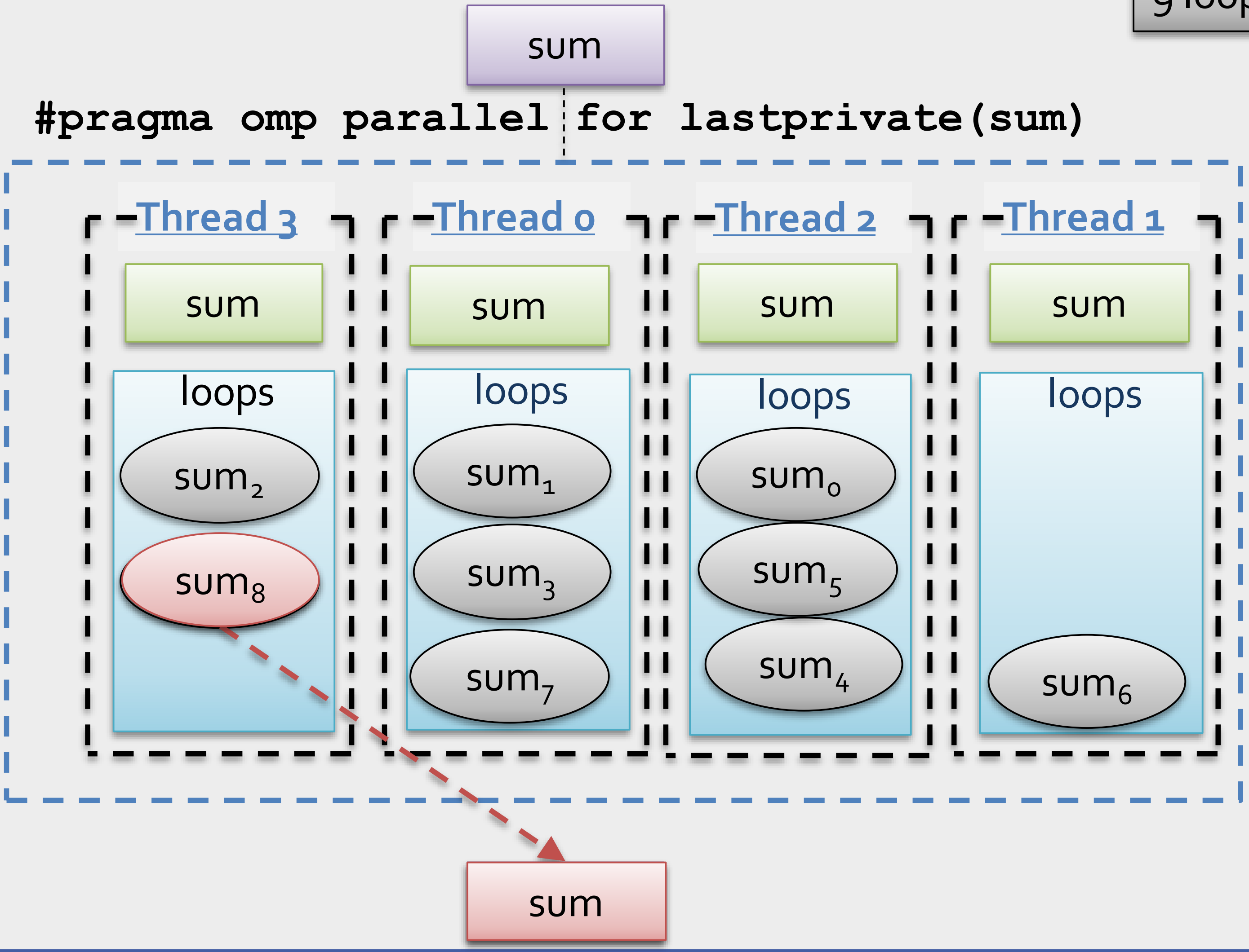
How the firstprivate works:


Smth
#pragma omp parallel firstprivate(Smth)
Thread 3
Thread 1
Thread N
Local copies are created with Smth global value
Smth
Smth
Smth
...
Smth
Smth
Smth
The Smth global value doesn't change
Smth

Data-Sharing Attribute Clauses

How the lastprivate works:

4 threads
9 loop iterations



Data-Sharing Attribute Clauses: Examples

```
int main()  {
   int b = 3;
   b += 7;
   printf ("b is %d\n”, b);
}
```

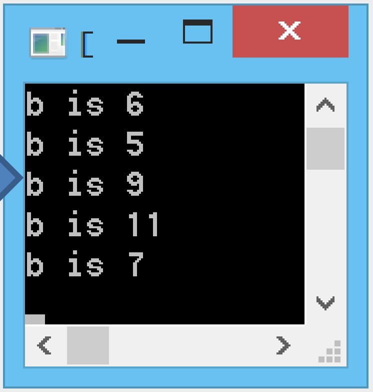
```
int b = 3;
#pragma omp parallel firstprivate(b)
{
   b += 7;
   printf ("b is %d\n”, b);
}
```

✓

Synchronization Constructs in OpenMP

Sample:

```
int b = 3;
#pragma omp parallel
{
    b += 1;
    printf ("b is %d\n”, b);
}
```



```
b is 6
b is 5
b is 9
b is 11
b is 7
```

- *Synchronization*: bringing one or more threads to a well defined and known point in their execution
- OpenMP includes:

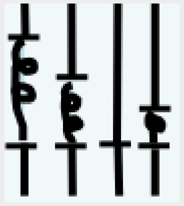
High level synchronization

- barrier
- critical
- atomic
- ordered

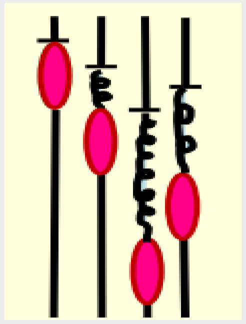
Low level synchronization

- locks
- flush

Barrier



Mutual exclusion



Synchronization in OpenMP

BARRIER Directive

```
Format: #pragma omp barrier
```

- synchronizes all threads in the
- When a ***barrier*** directive is reached, a thread will wait at that point until all other threads have reached that barrier
- All threads then resume executing in parallel the code that follows the barrier
- All threads in a team (or none) must execute the ***barrier*** region

Synchronization in OpenMP

CRITICAL Directive

```
Format: #pragma omp critical [name]
                   structured_block
```

- specifies a region of code that must be executed by only one thread at a time
- If a thread is currently executing inside a ***critical*** region and another thread reaches that ***critical*** region and attempts to execute it, it will block until the first thread exits that ***critical*** region
- Names act as global identifiers. Different ***critical*** regions with the same name (or are unnamed) are treated as the same region
- It is illegal to branch into or out of a CRITICAL block

Synchronization in OpenMP

ATOMIC Directive

```
Format: #pragma omp atomic
          x <operator> = <expression>
```

- specifies that a memory location must be updated atomically, rather than letting multiple threads attempt to write to it
- The directive applies only to a single, immediately following statement

Sample:

```
double A[NMAX];
#pragma omp parallel for
for (i = 0; i < NMAX; i++){
      big_computations(A);

      #pragma omp atomic
      sum += A[i];
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

Synchronization in OpenMP

Lock Functions

Initialize a simple/nested lock	`void omp_init_lock(omp_lock_t *lock);`
	`void omp_init_nest_lock(omp_nest_lock_t *lock);`

Wait until a lock is available	`void omp_set_lock(omp_lock_t *lock);`
	`void omp_set_nest_lock(omp_nest_lock_t *lock);`

Release a simple/nested lock	`void omp_unset_lock(omp_lock_t *lock);`
	`void omp_unset_nest_lock(omp_lock_t *lock);`

Remove a simple/nested lock	`void omp_destroy_lock(omp_lock_t *lock);`
	`void omp_destroy_nest_lock(omp_nest_lock_t *lock);`

Test a simple/nested lock	`int omp_test_lock(omp_lock_t *lock);`
	`int omp_test_nest_lock(omp_lock_t *lock);`

Synchronization in OpenMP

Lock Functions. Example

```
int A[NMAX], sum = 0, i;
for (i = 0; i < NMAX; i++)
  A[i] = i;
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel for
{
   for (i=0; i < NMAX; i++) {
      omp_set_lock (&lock);
      sum += A[i];
      omp_unset_lock (&lock);
   }
}
omp_destroy_lock (&lock);
```

HW: Tasks for report

- 2. Tell about:
  - Flush directive
  - Single directive
  - Master directive
  - Ordered directive
  - Sections Construct

- 3. Common Mistakes in OpenMP and How To Avoid Them (article)

Learning more about OpenMP

- OpenMP architecture review board URL, the "owner" of the OpenMP specification:
    - www.openmp.org
- OpenMP User's Group (cOMPunity) URL:
    - www.compunity.org
- The OpenMP reference card:
    - http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf

Getting Started. OpenMP vs. Visual Studio

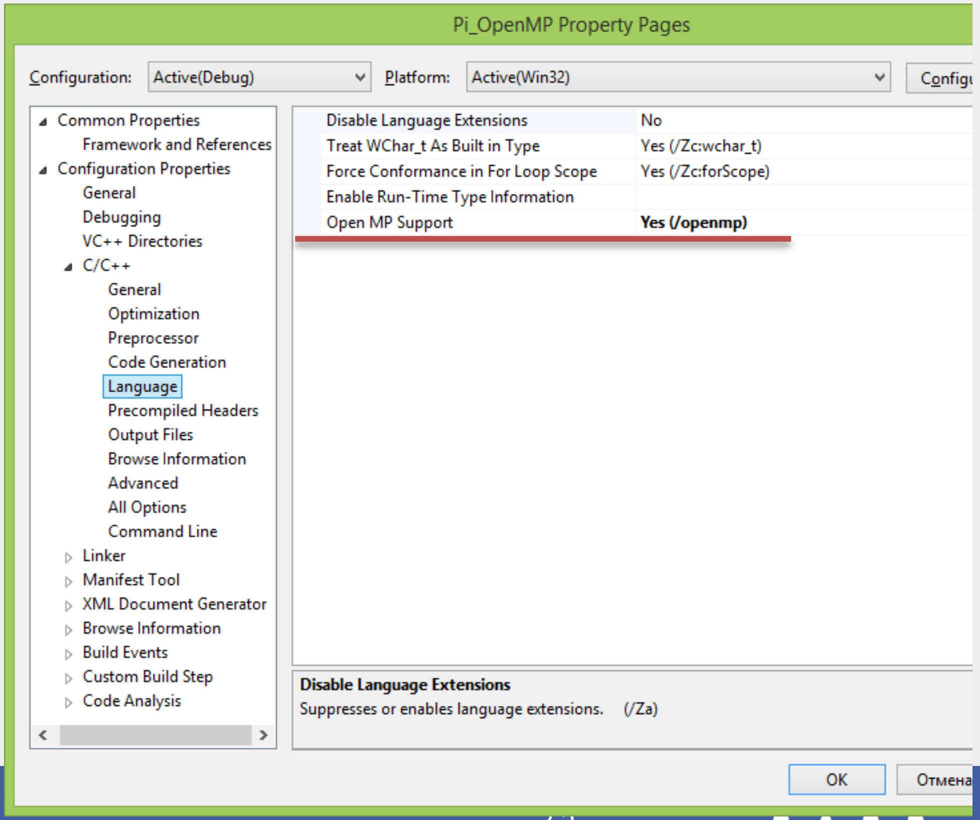
To use OpenMP with Visual Studio, you need:

- Visual Studio 2005 Professional or better
- A multi processor or multi core system to see speed improvement
- An algorithm to parallelize

Visual Studio uses OpenMP standard 2.0 and is located in the vcomp.dll

To use OpenMP:

- Include <omp.h>
- Enable OpenMP compiler switch in Project Properties



Disable Language Extensions	No
Treat WChar_t As Built in Type	Yes (/Zc:wchar_t)
Force Conformance in For Loop Scope	Yes (/Zc:forScope)
Enable Run-Time Type Information	
Open MP Support	**Yes (/openmp)**

Disable Language Extensions
Suppresses or enables language extensions. (/Za)