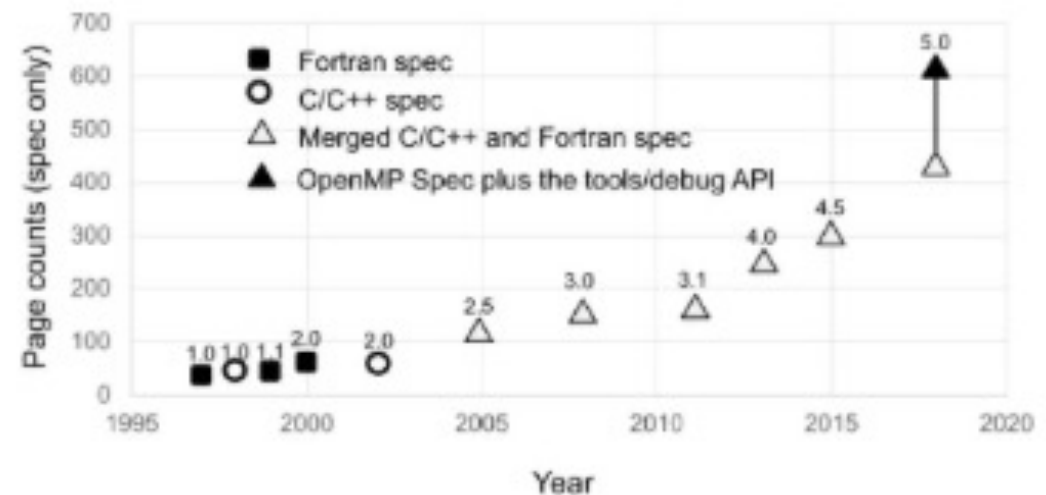




OpenMP

the common core

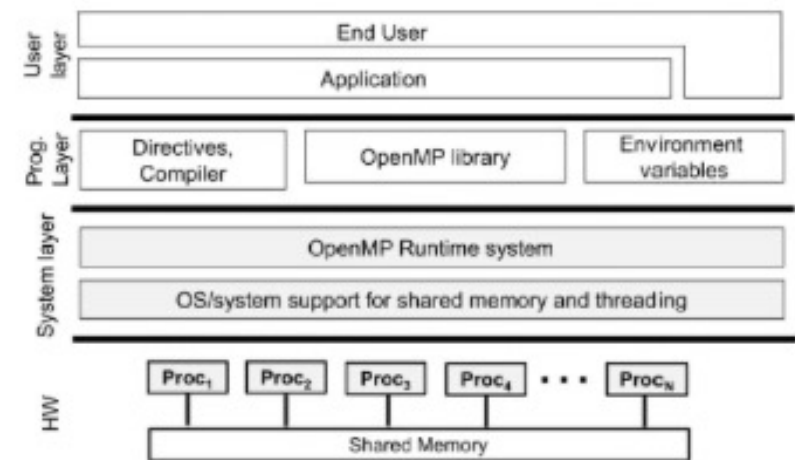


OpenMP Standard (version 2.5)

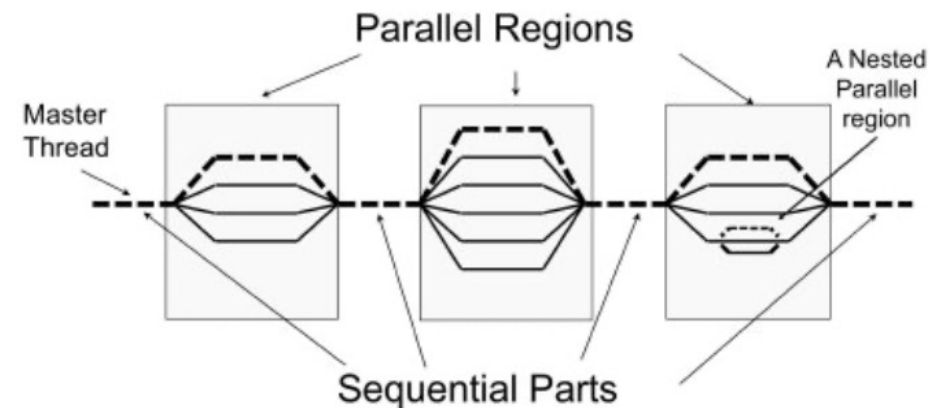
Introduction to OpenMP

- OpenMP is a standard to **Shared Memory (SM) parallel programming** (e.g., targets multi-core machines).

- Based on: Compiler directives, Library routines and Environment variables;
- Supports C/C++ and Fortran programming languages.



- Execution model is based on the **fork-join** model of parallel execution.
- Parallelism is specified through directives, added by the programmer to the code
 - The compiler implements the parallelism (e.g, POSIX Threads)

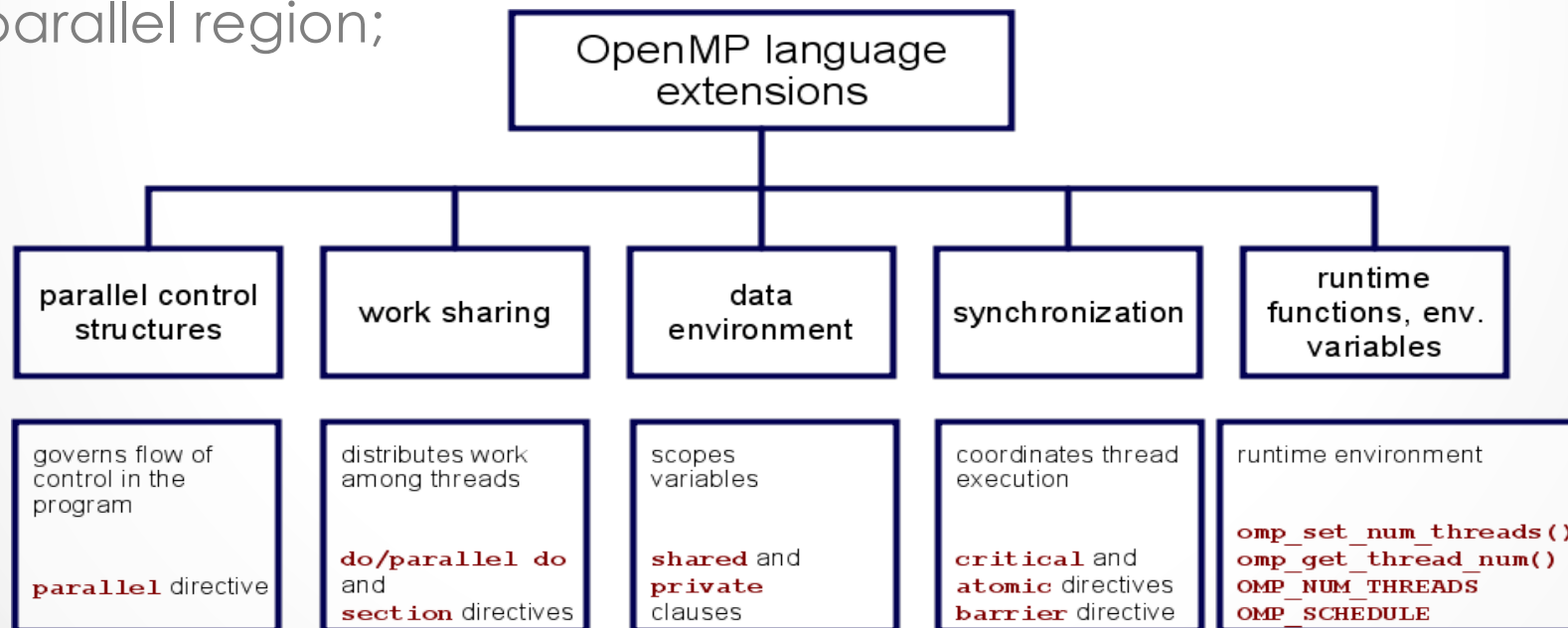


OpenMP considerations:

- It is the programmer's responsibility to ensure **correctness** and **efficiency** of parallel programs
 - OpenMP itself **does not** solve problems as :
 - **Data races**, deadlock or poor load balancing (among others).
 - But, offers routines to solve problems like load balancing.
- The creation/managing of threads is delegated to the compiler & OpenMP runtime:
 - + Easier to parallelize applications;
 - - Less control over threads behaviour.
- By default, the number of **parallel activities** is defined according to available resources
 - e.g. 2 cores -> 2 threads
 - SMT capability counts as additional cores

OpenMP: Programming Model

- An OpenMP program begins with a single thread (**master thread**).
- **Parallel regions** create a team of parallel activities;
- **Work-sharing** constructs generate work for the team to process;
- **Data sharing** clauses specify how variables are shared within a parallel region;



The OpenMP core constructs (1)

- OpenMP directives format for C/C++ applications:
 - `#pragma omp` directive-name [clause[[,] clause]...] new-line
- **Parallel Construct => required on every OpenMP program**
 - `#pragma omp parallel` Creates a team of threads.
- **Work-sharing Constructs**
 - `#pragma omp for` Assignment of loop iterations to threads.
 - ~~`#pragma omp sections` Assignment of blocks of code (section) to threads.~~
 - `#pragma omp single` Restricts a block of code to be executed by a single thread.

The OpenMP core constructs (2)

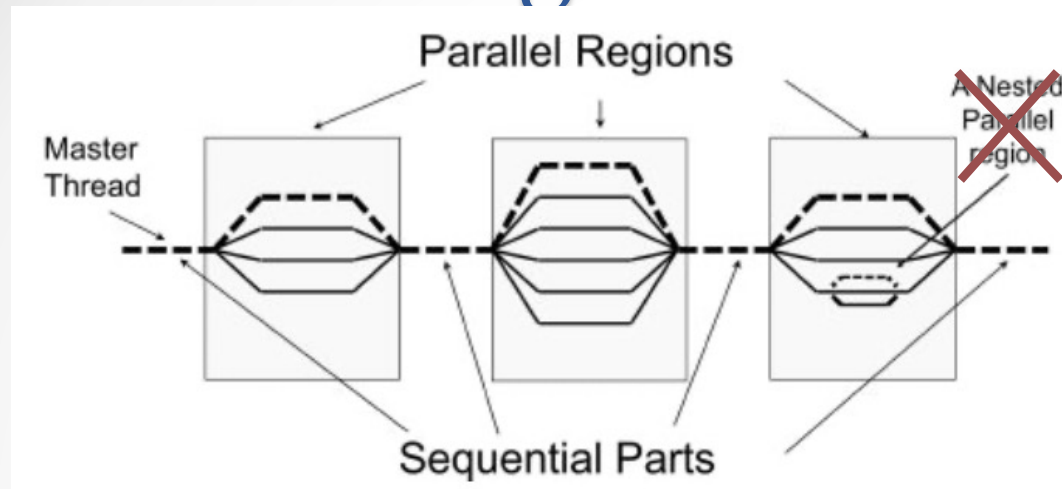
- Tasking Constructs (standard v2.5)

- `#pragma omp task` Creation of a task to be executed by threads in the team.

- Master & Synchronization Constructs

- ~~`#pragma omp master` Restricts a block of code to be executed only the master thread.~~
- `#pragma omp critical` Restricts the execution of a block of code to a single thread at a time.
- `#pragma omp barrier` Makes all threads in a team to wait for the remaining.
- `#pragma omp taskwait` wait for the completion of the current task child's.
- `#pragma omp atomic` Ensures that a specific storage location is managed atomically.
- ~~`#pragma omp flush` Makes a thread's temporary view of memory consistent with memory.~~
- ~~`#pragma omp ordered` Specifies a block of code in a loop region that will be executed in the order of the loop iterations.~~

Parallel Region



- When a thread encounters a parallel construct, a **team of threads** is created (**FORK**);
- The thread which encounters the parallel region becomes the **master** of the new team;
- **All threads** in the team (including the master) execute the region;
- At end of parallel region, all threads synchronise, and join master thread (**JOIN**).

Parallel region syntax

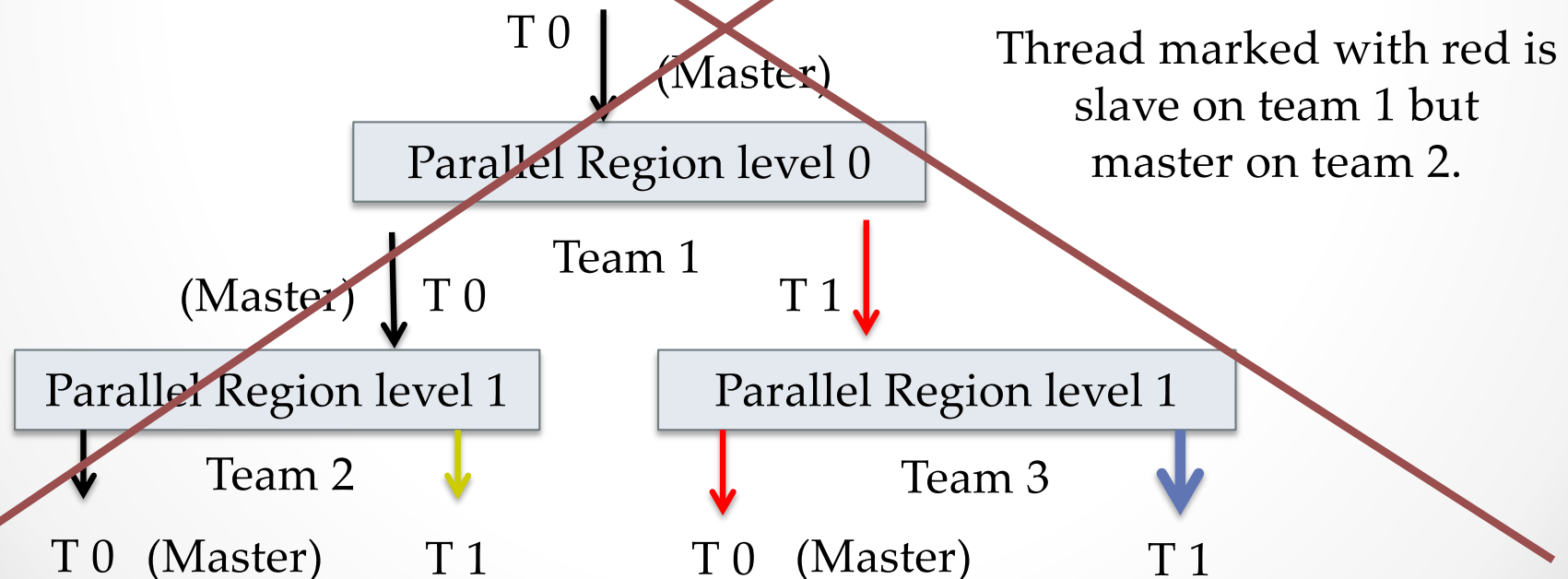
```
#pragma omp parallel [clauses]  
{  
    code_block  
}
```

Where clause can be:

```
if (scalar-expression)  
num_threads (integer-expression)  
private (list)  
firstprivate (list)  
shared (list)  
reduction (operator: list)
```


~~Nested Parallel Region => (outdated)~~ ~~use tasks instead (next lecture)~~

- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and becomes the master of this team;
- If **nested parallelism** is disabled, then no additional team of threads will be created.
- To enable/disable -> `omp_set_nested(x);`
- **Example:**



Work-Sharing: Loop Construct

- The *for* loop iterations are distributed across **threads in the team**;
 - The distribution is based on:
 - **chunk_size**, by default is 1;
 - **schedule** by default is **static**.
- Loop **schedule** (map of loop iterations to threads):
 - **Static** – Iterations divided into chunks of size **chunk_size** assigned to the threads in a team in a **round-robin** fashion;
 - **Dynamic** – the chunks are assigned to threads in the team as the threads request them;
 - **Guided** - similar to dynamic but the chunk size decreases during execution.
 - **Auto** – the selection of the scheduling strategy is delegated to the OpenMP implementation.

Loop work-sharing syntax

```
#pragma omp for[clauses]
{
    code_block
}
```

Where clause can be:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
schedule(kind[, chunk_size])
collapse(n)
ordered
nowait
```

Loop Constructors

- schedule(**static**) vs schedule(**dynamic**)
 - **Static** has lower **overhead**;
 - **Dynamic** has a better load balance approach;
 - Increasing the chunk size in the dynamic for:
 - Diminishing of the scheduling overhead;
 - Increasing the possibility of load balancing problems.
- Lets consider the following loop that we want to parallelize using 2 threads, being `void f(int i)` a given function

```
#pragma omp parallel for schedule ( ? )  
for(i = 0; i < 100; i++)  
    f(i);
```

What is the most appropriated type of scheduling?

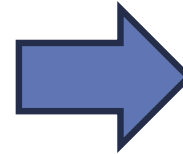
Parallel for with ordered clause

- `#pragma omp for schedule(static) ordered`
for (i = 0; i < N; ++i)
{
 ... // do something here (in parallel)
 `#pragma omp ordered`
 {
 printf("test() iteration %d\n", i);
 }
}

Parallel execution of code sections-use the task model instead (next lecture)

- Supports heterogeneous tasks: • Implementation with tasks:

```
#pragma omp parallel  
{  
  #pragma omp sections  
  {  
    #pragma omp section  
    {  
      taskA();  
    }  
    #pragma omp section  
    {  
      taskB();  
    }  
    #pragma omp section  
    {  
      taskC();  
    }  
  }  
}
```



```
#pragma omp parallel  
{  
  #pragma omp single  
  {  
    #pragma omp task  
    {  
      taskA();  
    }  
    #pragma omp task  
    {  
      taskB();  
    }  
    #pragma omp task  
    {  
      taskC();  
    }  
  }  
}
```

Task constructor (from OpenMP 2.5)

```
int fib(int n)
{
    int i, j;
    if (n<2) return n;
    else
    {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);

        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);

        #pragma omp taskwait
        return i+j;
    }
}

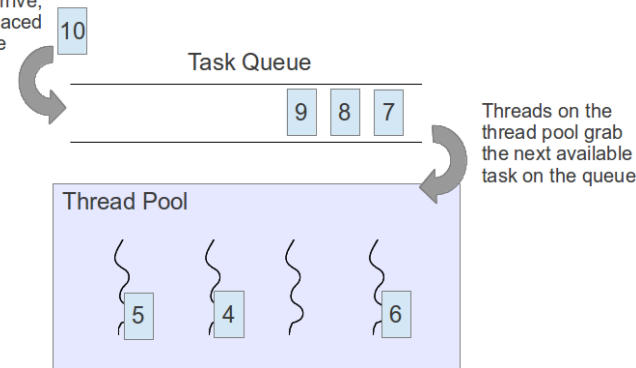
int main()
{
    int n = 10;

    omp_set_num_threads(4);

    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf ("fib(%d) = %d\n", n, fib(n));
    }
}
```

➤ Execution model:

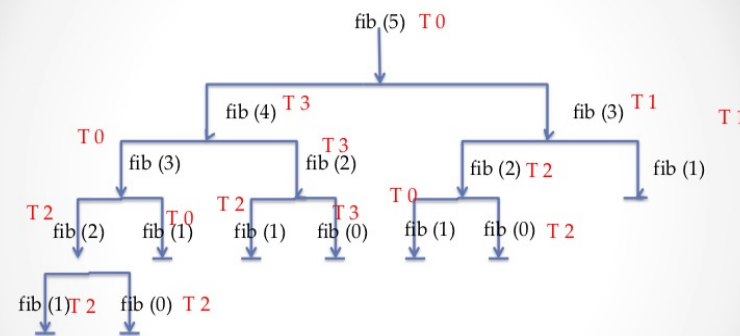
As tasks arrive,
they are placed
on a queue



- The **taskwait** directive ensures that the tasks generated are completed before the return statements.

- Only one thread executes the **single** directive, and hence the call to fib(n),
- **All four threads** will participate in executing the tasks generated.

Execution Tree Exemplified



Data Sharing

- What happens to variables in parallel regions?
 - Variables declared **inside** are **local** to each thread;
 - Variables declared **outside** are **shared**
- Data sharing clauses:
 - **private(varlist)** => each variable in varlist becomes private to each thread, initial values not specified.
 - **firstprivate(varlist)** => Same as private, but variables are initialized with the value outside the region.
 - **lastprivate(varlist)** => same as private, but the final value is the last loop iteration's value.
 - **reduction (op:var)** => same as lastprivate, but the final value is the result of reduction of private values using the operator "**op**".
- ~~• Directives for data sharing:~~
 - ~~◦ **#pragma omp threadlocal** => each thread gets a local copy of the value.~~
 - ~~◦ **copyin** clause copies the values from thread master to the others threads.~~

Synchronization Constructs:

- Critical regions (executed in mutual exclusion):
 - `#pragma omp critical [name]`
 `someFunctionCall();`
 - Restricts the execution of the associated structured blocks to a single thread at a time;
 - Works inter-teams (i.e., global lock)
 - An optional name may be used to identify the critical construct.
- Atomic Operations (fine-grain synchronization):
 - `#pragma omp atomic`
 `A[i] += x;`
 - The memory in will be updated atomically. It does not make the entire statement atomic; only the memory update is atomic.
 - A compiler might use special hardware instructions for **better** performance than when using **critical**.

Synchronization Constructs:

- Solving data races with critical VS atomic

```

5  int main(){
6      double result={0};
7
8      #pragma omp parallel for shared(result)
9      for(int i=0; i<1000000;i++) {
10         result+=sin(i);
11     }
12     printf("%f",result);
13 }
    
```

Data race

```

.L4: vxorpd %xmm0, %xmm0, %xmm0
     vcvtsi2sdl %ebx, %xmm0, %xmm0 ; convert to double
     call sin
     vmovsd 8(%rsp,%r12), %xmm1 ; load result into %xmm1
     vaddsd %xmm0, %xmm1, %xmm1 ; add
     vmovsd %xmm1, 8(%rsp,%r12) ; update result
     ...
     jne .L4
    
```

- Critical:** coarse-grain and pessimist

```

call GOMP_critical_start
vmovsd 8(%rsp,%r12), %xmm1 ; load result into %xmm1
vaddsd %xmm0, %xmm1, %xmm1
vmovsd %xmm1, 8(%rsp,%r12) ; update results
call GOMP_critical_end
    
```

Mutual exclusion: ensures that only one thread executes the critical region (e.g., blocks the calling thread if another thread is still executing the region)

- Atomic:** fine-grain and optimistic (note: simplified code)

```

.L4: vxorpd %xmm0, %xmm0, %xmm0
     vcvtsi2sdl %ebx, %xmm1, %xmm0
     call sin
.L5: vmovsd (%r12), %xmm1
     vaddsd %xmm0, %xmm1, %xmm1
     ...
     lock cmpxchgq %xmm1, (%r12) ; atomic comp & exchange
     cmpq %rax, %rdx
     jne .L5 ; repeat if update failed
     ...
     jmp .L4
    
```

Repeat until successful update

Avoid/reduce synchronisation

- Reduction of multiple values (in parallel):

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i<100; i++) {
    sum += array[i];
}
```

- Thread reuse across parallel regions

```
#pragma omp parallel {
    #pragma omp for
    for(int i = 0; i<100; i++)
        ...
    #pragma omp for
    for(int j = 0; j<100; j++)
        ...
}
```

Environment variables

- **OMP_SCHEDULE**
 - sets the *run-sched-var* ICV for the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types (i.e., **static**, **dynamic**, **guided**, and **auto**).
- **OMP_NUM_THREADS**
 - sets the *nthreads-var* ICV for the number of threads to use for parallel regions.
- **OMP_DYNAMIC**
 - sets the *dyn-var* ICV for the dynamic adjustment of threads to use for **parallel** regions.
- **OMP_NESTED**
 - sets the *nest-var* ICV to enable or to disable nested parallelism.
- **OMP_STACKSIZE**
 - sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- **OMP_WAIT_POLICY**
 - sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads.
- **OMP_MAX_ACTIVE_LEVELS**
 - sets the *max-active-levels-var* ICV that controls the maximum number of nested active parallel regions.
- **OMP_THREAD_LIMIT**
 - sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

OpenMP Routines

- **omp_set_num_threads / omp_get_num_threads**
- omp_get_max_threads
- **omp_get_thread_num**
- omp_get_num_procs.
- omp_in_parallel.
- omp_set_dynamic / omp_get_dynamic.
- omp_set_nested / omp_get_nested.
- omp_set_schedule / omp_get_schedule
- omp_get_thread_limit.
- omp_set_max_active_levels / omp_get_max_active_levels
- omp_get_level.
- omp_get_ancestor_thread_num.
- omp_get_team_size.
- omp_get_active_level
- **Locks**
 - void omp_init_lock(omp_lock_t *lock);
 - void omp_destroy_lock(omp_lock_t *lock);
 - void omp_set_lock(omp_lock_t *lock);
 - void omp_unset_lock(omp_lock_t *lock);
 - int omp_test_lock(omp_lock_t *lock);
- **Timers**
 - **double omp_get_wtime(void);**
 - double omp_get_wtick(void);