CSCI-GA.3033-012
Multicore Processors: Architecture & Programming

Lecture 7: OpenMP: Control vs. Simplicity Tradeoff

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

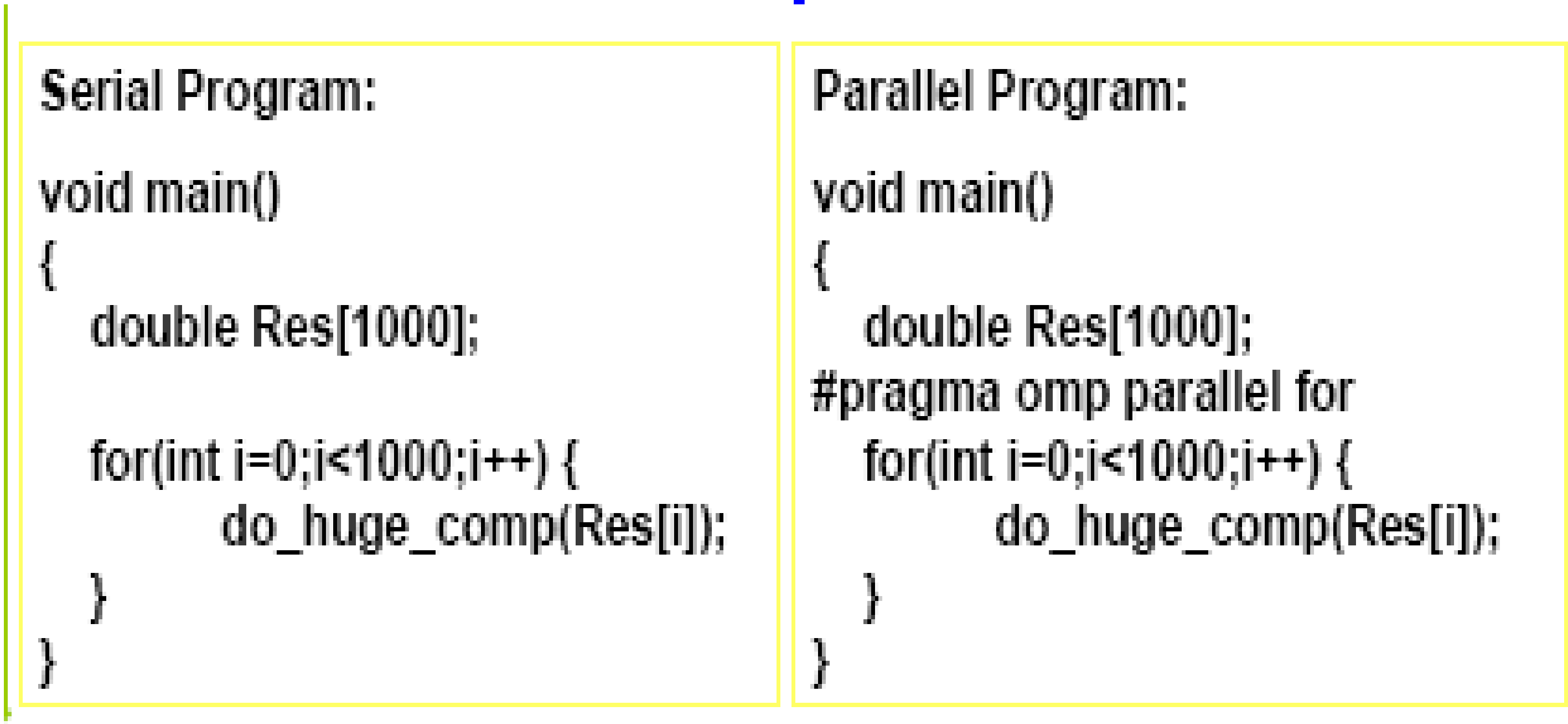
Small and Easy Motivation

```
int main() {


  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```

Small and Easy Motivation

```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

Simple!



Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

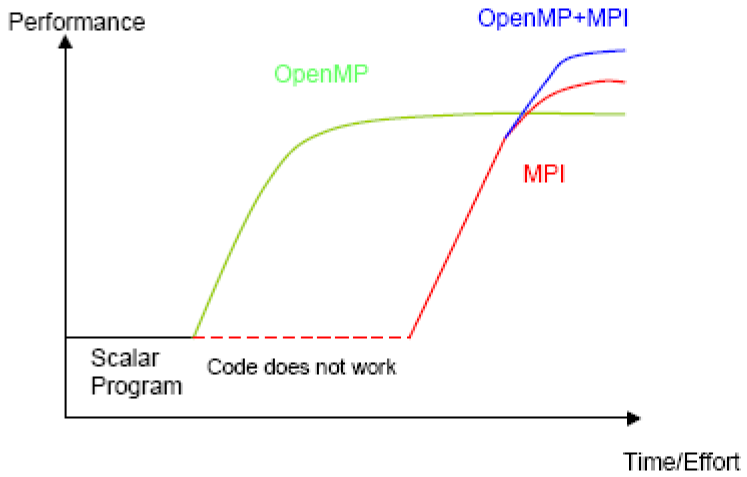
Parallel Program:

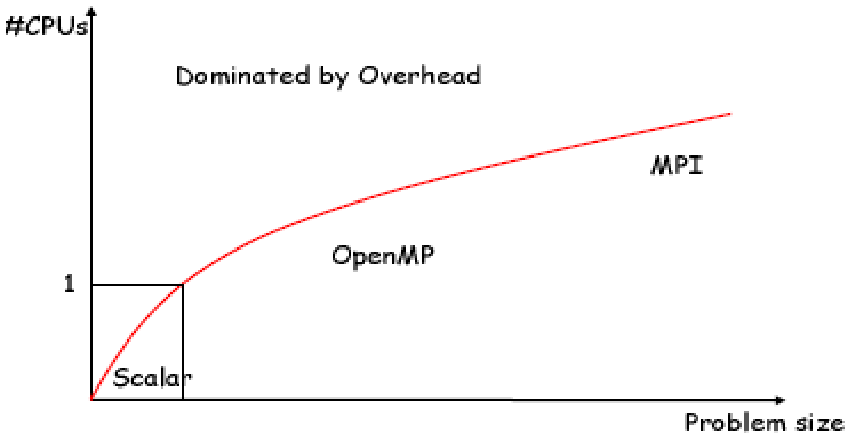
```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence

OpenMP is a small API that hides cumbersome threading calls with simpler *directives*

Interesting Insights About OpenMP





These insights are coming from HPC folks though!

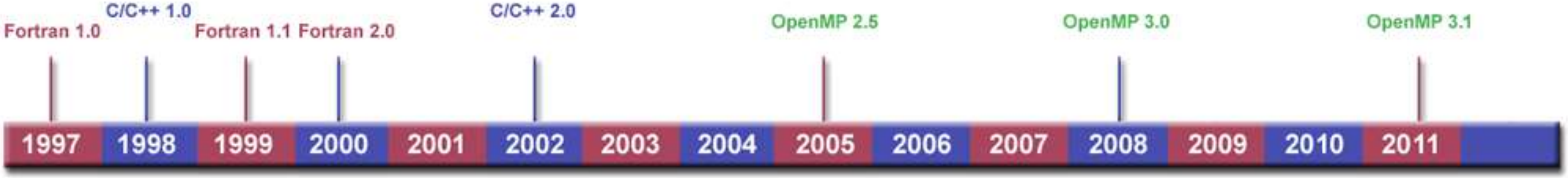
Source: www.sdsc.edu/~allans/cs260/lectures/OpenMP.ppt

OpenMP In a Nutshell

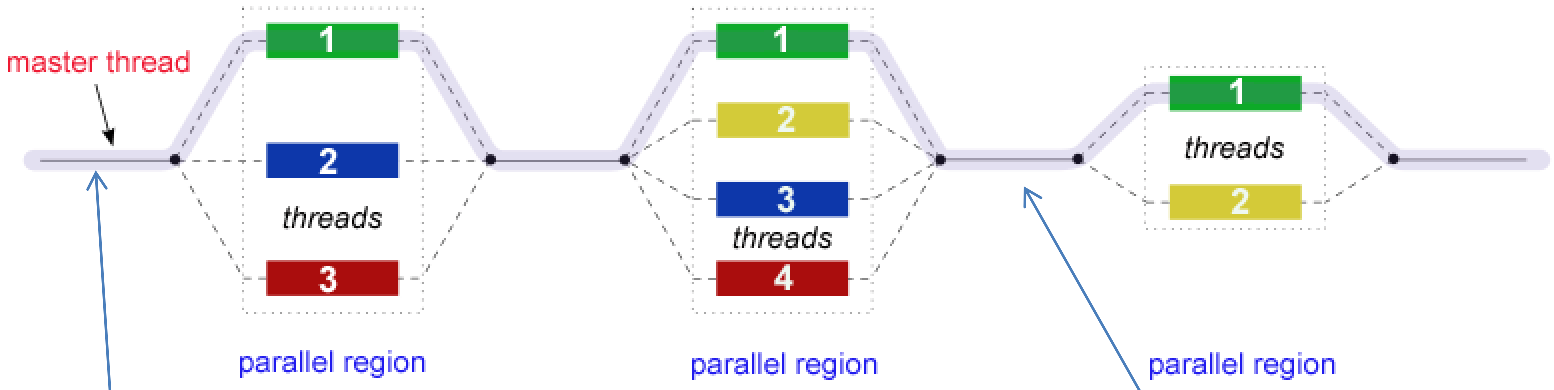
- API
- Multithreaded programming
- Assumes shared memory multiprocessor
- Works with C/C++, and Fortran
  - example: gcc -fopenmp
- Consists of:
  - compiler directives
  - library routines
  - environment variables
- Strengths: portability, simplicity, and scalability
- Weakness: less control to the programmer

Goals of OpenMP

- Standardization among platforms/architectures
- Easy of use
- Portability



OpenMP uses the fork-join model of parallel execution.

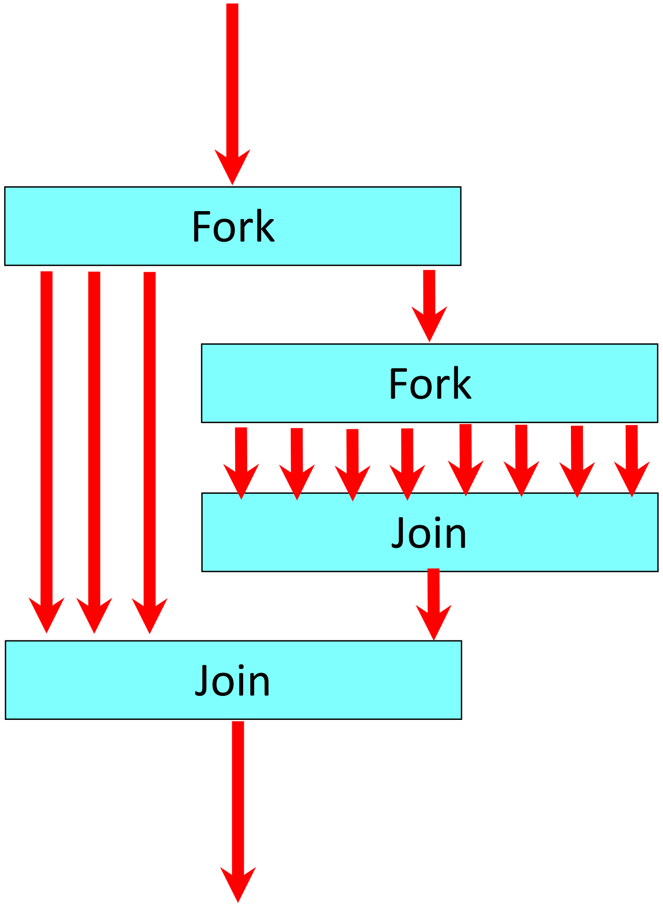


All OpenMP programs begin with a single thread: **master thread** (ID = 0)

FORK: the master thread then creates a team of parallel ***threads***.

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate

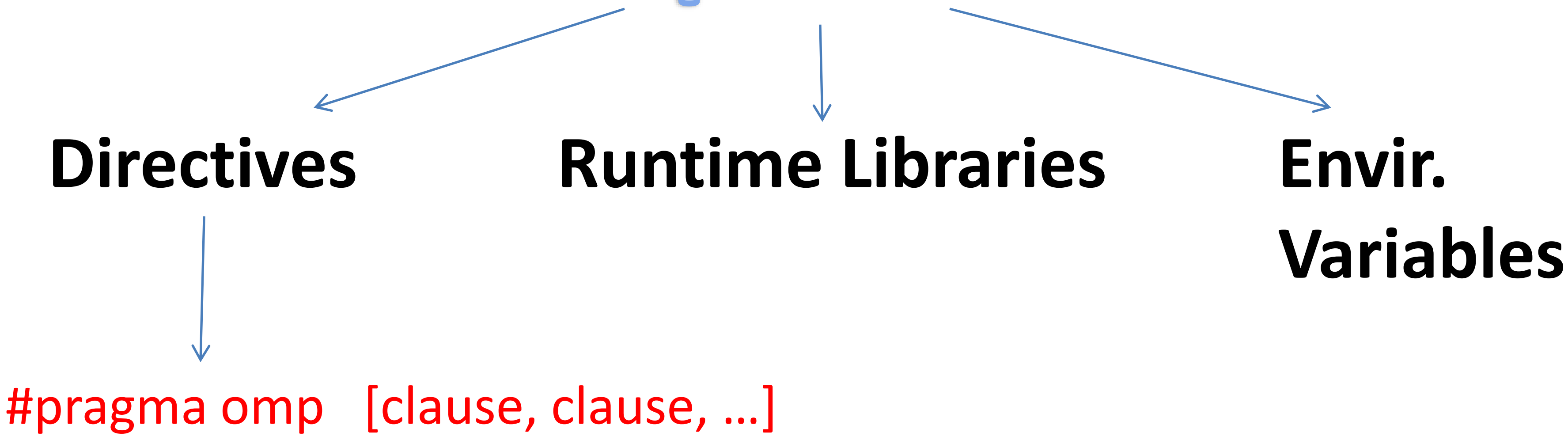
Isn't Nested Parallelism Interesting?



Important!

- The following are implementation dependent:
  - Nested parallelism
  - Dynamically alter number of threads
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.
- Threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time. The programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.

OpenMP



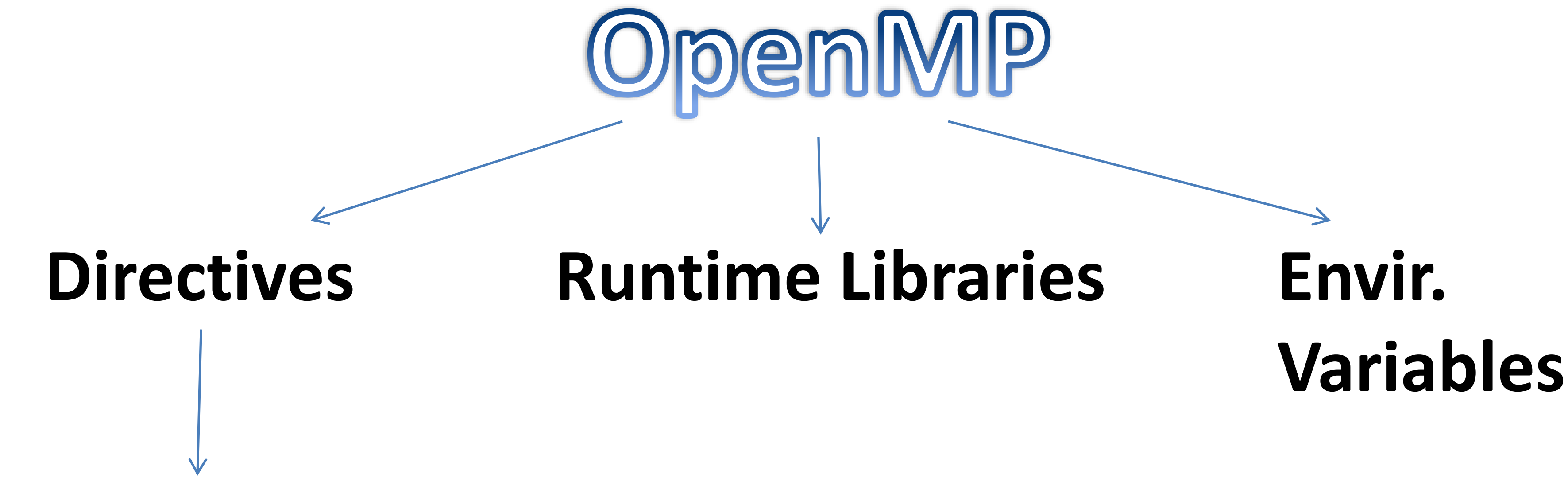
Directives

Runtime Libraries

Envir. Variables

#pragma omp [clause, clause, …]

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.



OpenMP

Directives

Runtime Libraries

Envir. Variables

#pragma omp [clause, clause, …]

Example:

#pragma omp parallel *[clause ...] newline*
if *(scalar_expression)*
private *(list)*
shared *(list)*
default (shared | none)
firstprivate *(list)*
reduction *(operator: list)*
copyin *(list)*
num_threads *(integer-expression)*

structured_block

```
#include <omp.h>

main () {
    int nthreads, tid;

    /* 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 */
    }
```

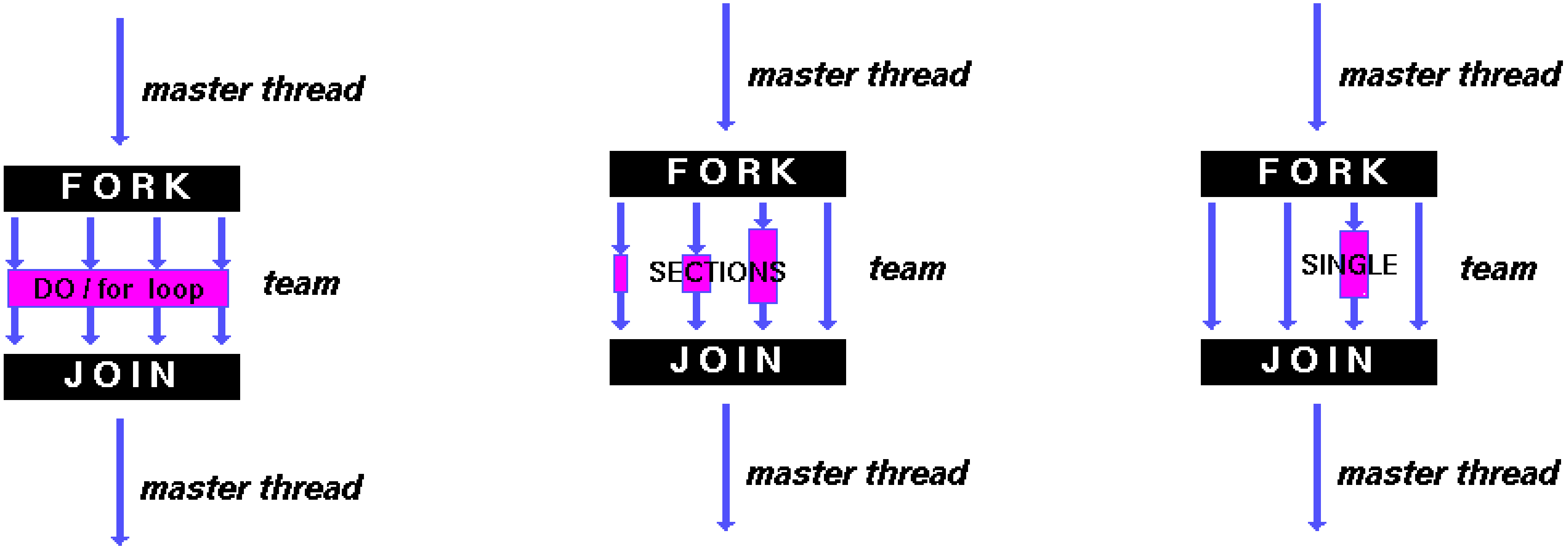
Runtime Libraries

```
#include <omp.h>

main () {
    int nthreads, tid;

    /* 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 */
    }
```

How Many Threads?

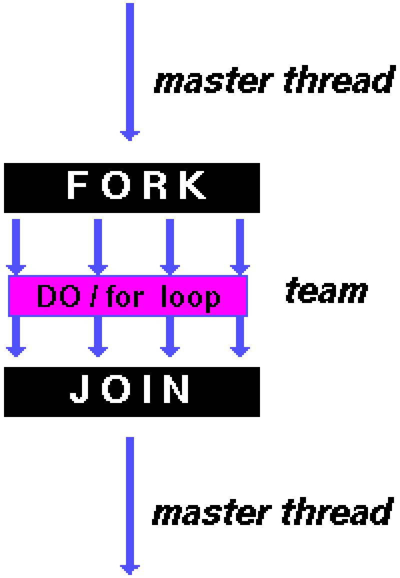
- Setting of the **NUM_THREADS** clause
- Use of the **omp_set_num_threads()** library function
- Setting of the **OMP_NUM_THREADS** environment variable
- Implementation default - usually the number of cores.
- Threads are numbered from 0 (master thread) to N-1

Dividing Work Among Threads



Important: Work sharing directives do NOT launch new threads.

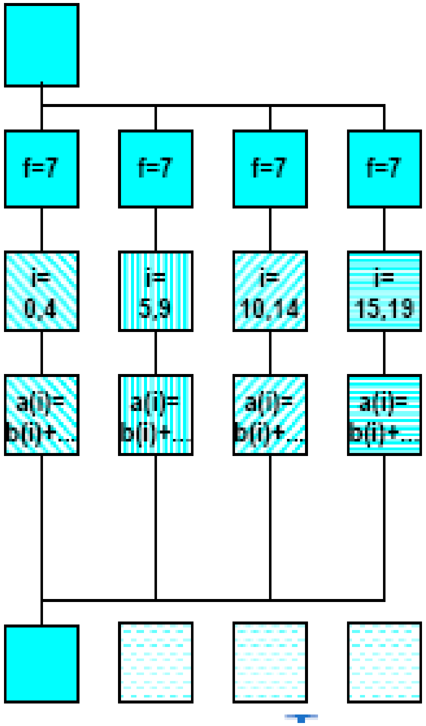
Dividing Work Among Threads



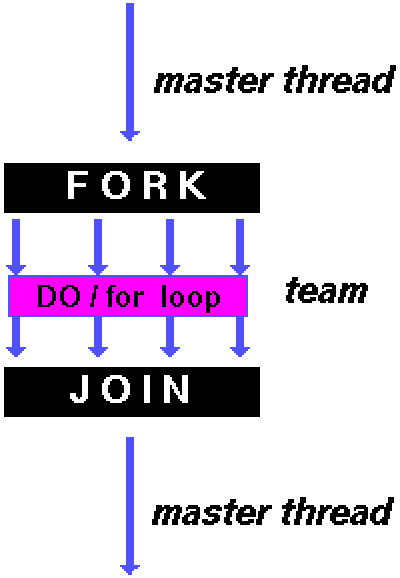
```
#pragma omp for [clause ...]
                        schedule (type [,chunk])
                        ordered private (list)
                        firstprivate (list)
                        lastprivate (list)
                        shared (list)
                        reduction (operator: list)
                        collapse (n)
                        nowait
for_loop
```

- Number of iterations must be known in advance
- No gotos into or out of the loop.

```
#pragma omp parallel private(f)
{
   f=7;
#pragma omp for
        for (i=0; i<20; i++)
        a[i] = b[i] + f * (i+1);
} /* omp end parallel */
```

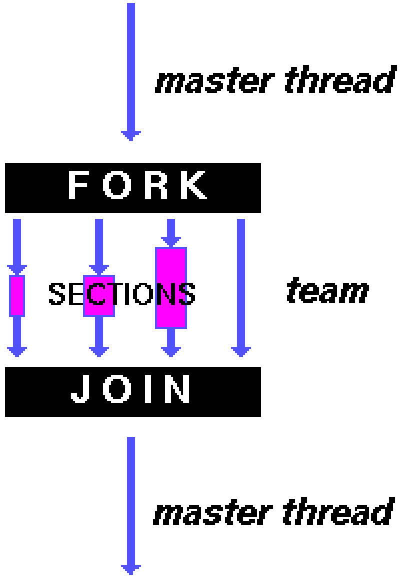


Dividing Work Among Threads



```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
            int i, chunk;
            float a[N], b[N], c[N];

            for (i=0; i < N; i++)
                        a[i] = b[i] = i * 1.0;

            chunk = CHUNKSIZE;

            #pragma omp parallel shared(a,b,c,chunk) private(i)
            {
            #pragma omp for schedule(dynamic,chunk) nowait
                        for (i=0; i < N; i++) c[i] = a[i] + b[i];
   } /* end of parallel section */
}
```

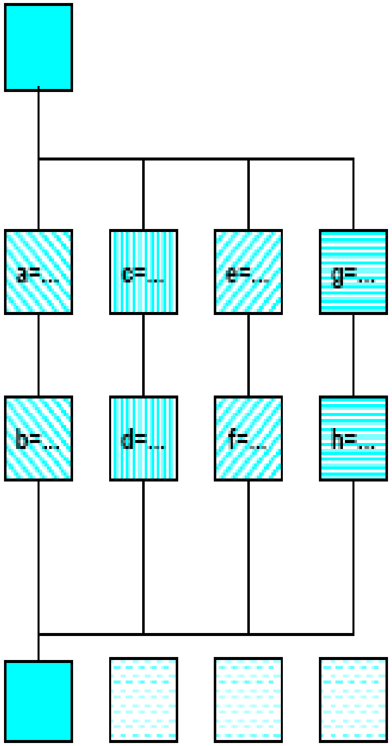
Dividing Work Among Threads



```
#pragma omp sections [clause ...]

{
 #pragma omp section

                structured_block

#pragma omp section

                structured_block
}
```

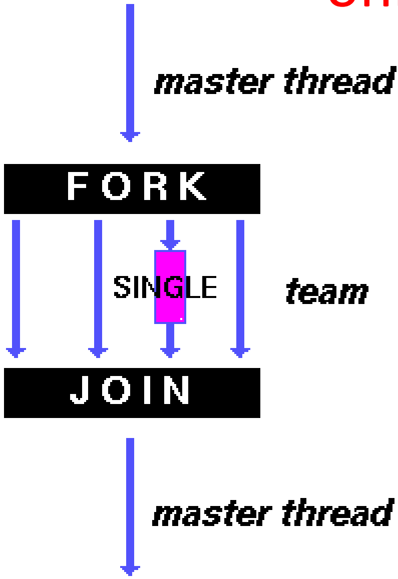
Implicit barrier here, unless you use nowait.

```
#pragma omp parallel
{
#pragma omp sections
    {{ a=...;
    b=...; }
#pragma omp section
    { c=...;
    d=...; }
#pragma omp section
    { e=...;
    f=...; }
#pragma omp section
    { g=...;
    h=...; }
} /*omp end sections*/
} /*omp end parallel*/
```



Dividing Work Among Threads

Specifies that the enclosed code is to be executed by only one thread in the team.



```
#pragma omp single [clause ...]

   structured_block
```

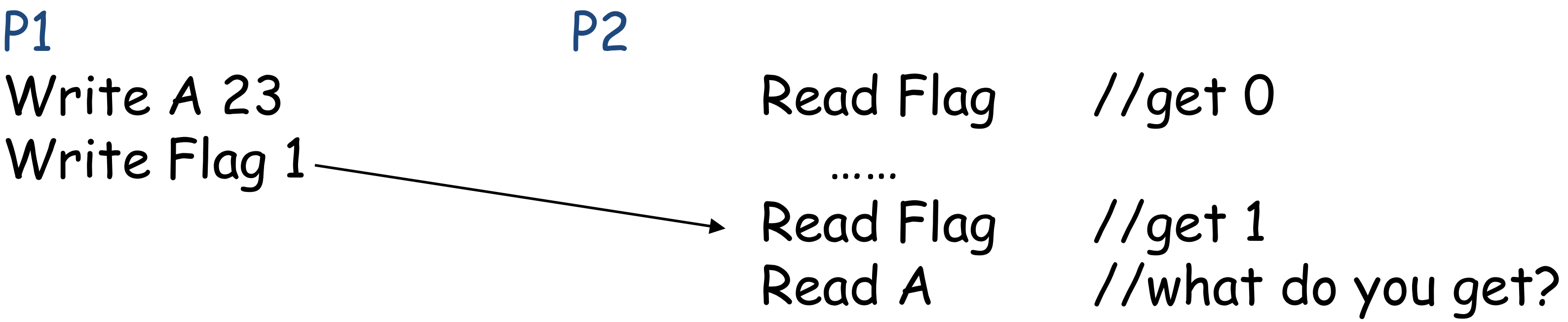
Example About Consistency Model

Code:

Initially A = Flag = 0

```
P1                              P2
A = 23;                         while (Flag != 1) {;}
Flag = 1;                       ... = A;
```

Possible execution sequence on each processor:



```
P1                  P2
Write A 23                      Read Flag     //get 0
Write Flag 1                      ……
                                Read Flag     //get 1
                                Read A        //what do you get?
```

Do You see the problem?

Example About Consistency Model

Code:

Initially A = Flag = 0

```
P1                        P2
A = 23;
flush;                    while (Flag != 1) {;}
Flag = 1;                 ... = A;
```

Execution:

- P1 writes data into *A*
- Flush waits till write to *A* is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of *A* even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.

What Does OpenMP Say Here?

#pragma omp flush *(list)*

- Thread-visible variables are written back to memory at this point.
- A bit complicated scenario arises if two threads execute the flush at the same time with common variables between them.
- If you do not specify a list, then all variables will be flushed

Synchronization: Critical Directive

Enclosed code

– executed by all threads, but

– **restricted to only one thread at a time**

- C/C++:

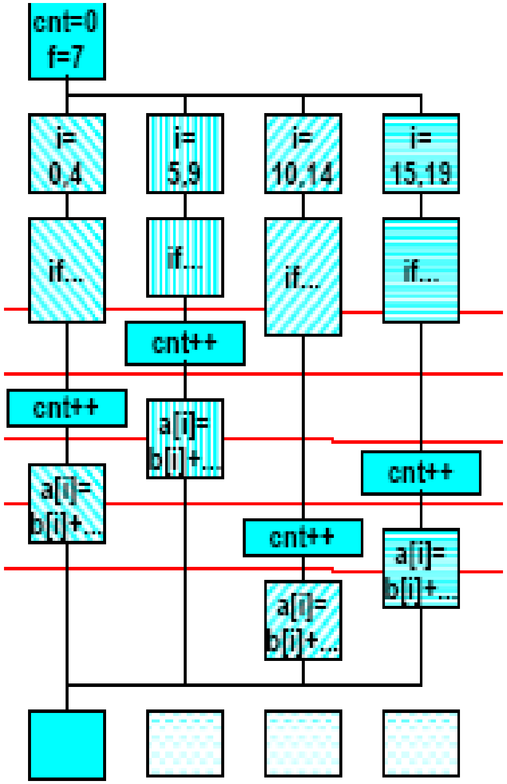
#pragma omp critical [（ name）]

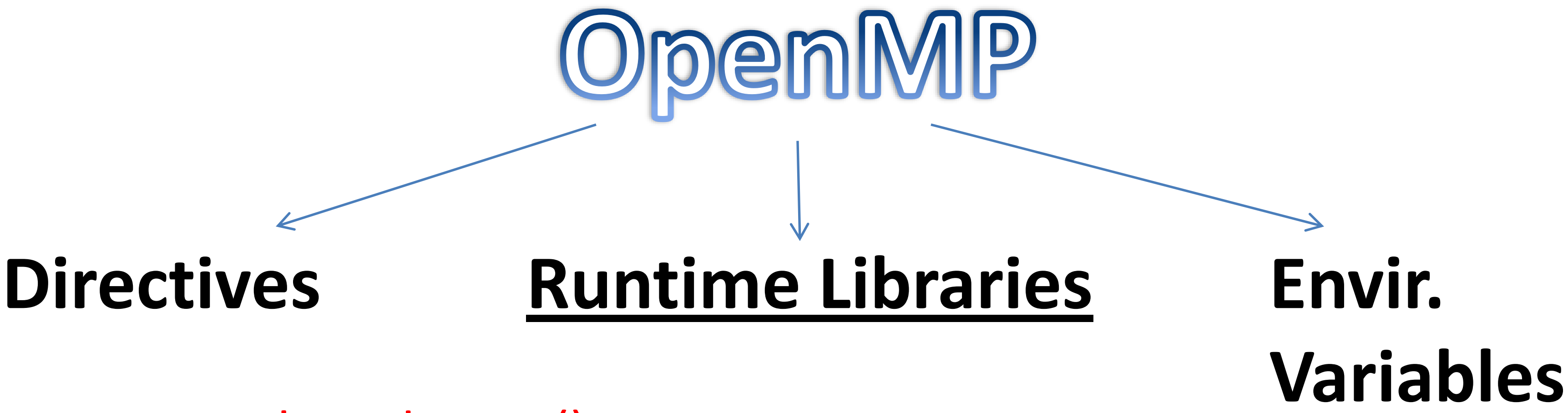
structured-block

- A thread waits at the beginning of a critical region until no other

thread in the team is executing a critical region with the same name.

```
cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<20; i++) {
            if (b[i] == 0) {
#pragma omp critical
                    cnt ++;
            } /* endif */
    a[i] = b[i] + f * (i+1);
    } /* end for */
} /*omp end parallel */
```





OpenMP

Directives

Runtime Libraries

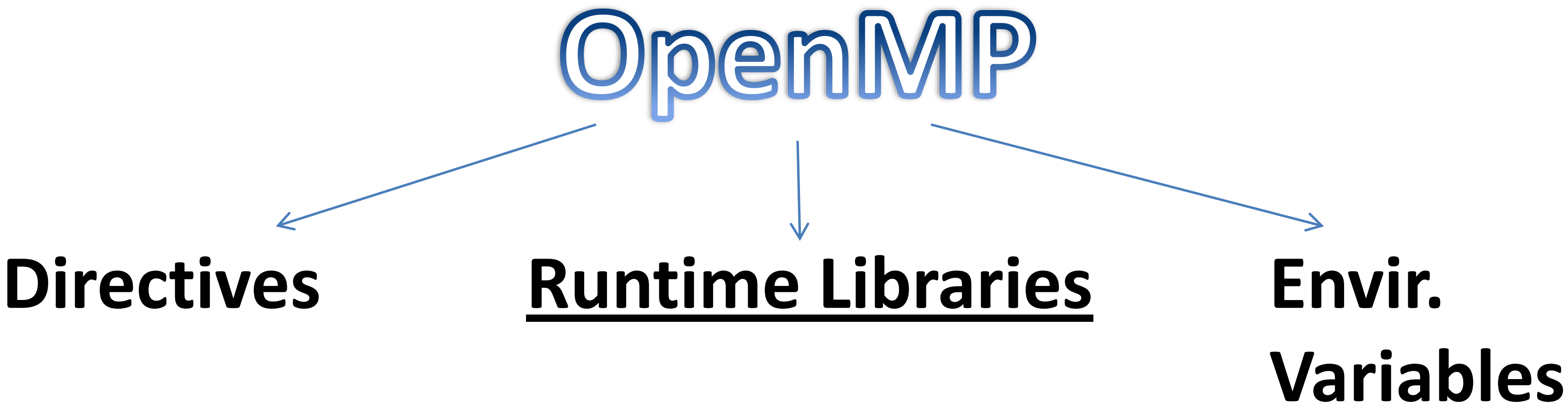
Envir. Variables

omp_get_thread_num()

- The number of threads remains unchanged for a parallel region
- The above function returns an integer
- Master thread has ID 0
- Different parallel regions may have different number of threads.

omp_get_num_threads()

- Returns, as integer, the number of threads in the current parallel region.



OpenMP

Directives

Runtime Libraries

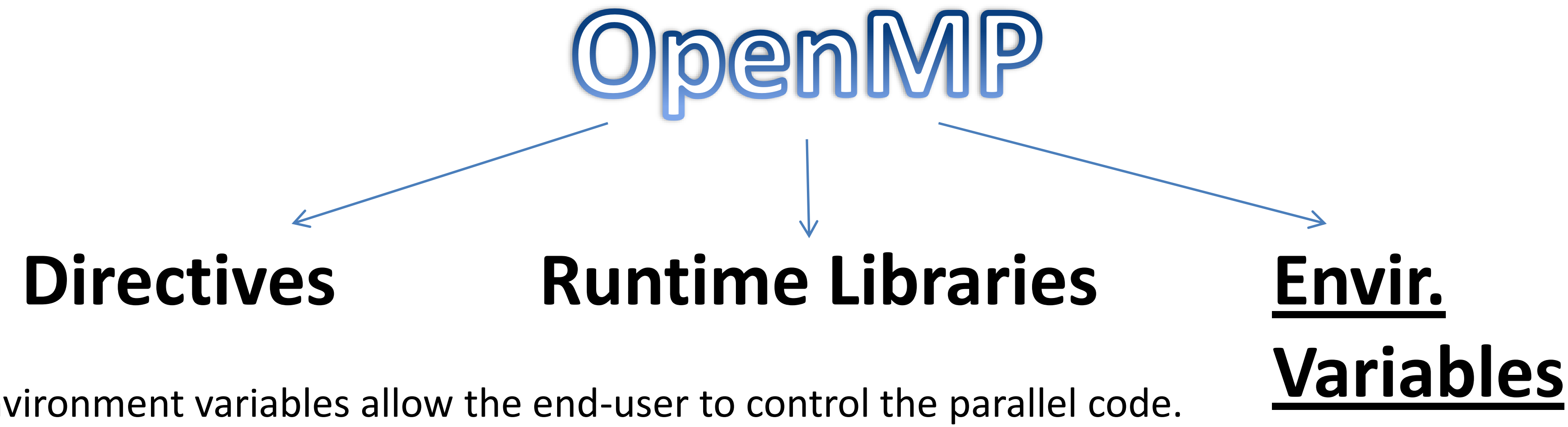
Envir. Variables

Are we in a parallel region?

OMP_IN_PARALLEL()

How many processors in the system?

OMP_GET_NUM_PROCS()



OpenMP

Directives

Runtime Libraries

Envir. Variables

- Environment variables allow the end-user to control the parallel code.
- All environment variables names are uppercase.

Example:

OMP_NUM_THREADS

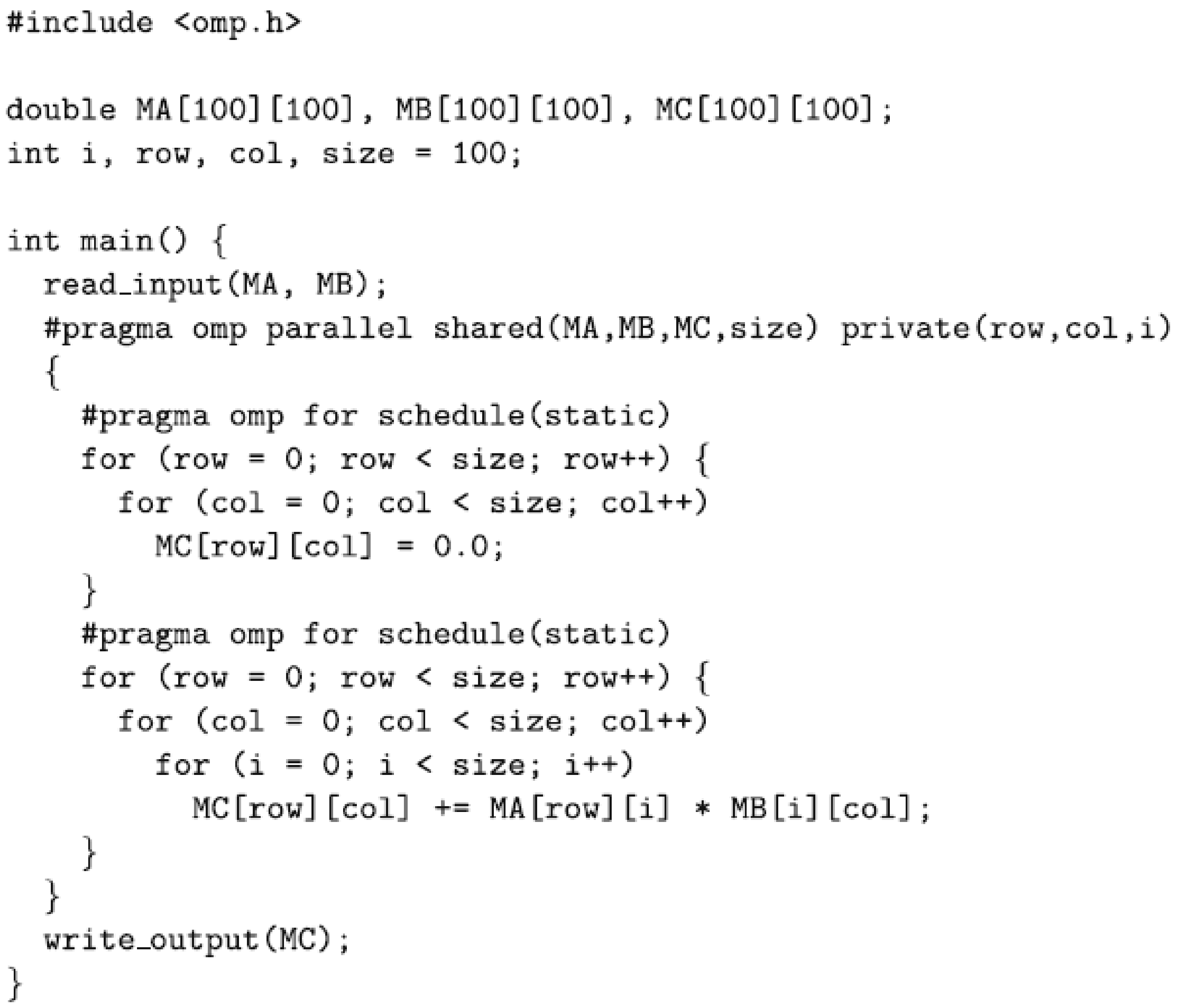
Sets the maximum number of threads to use during execution.

For example: **setenv OMP_NUM_THREADS 8**

What OpenMP Does NOT Do

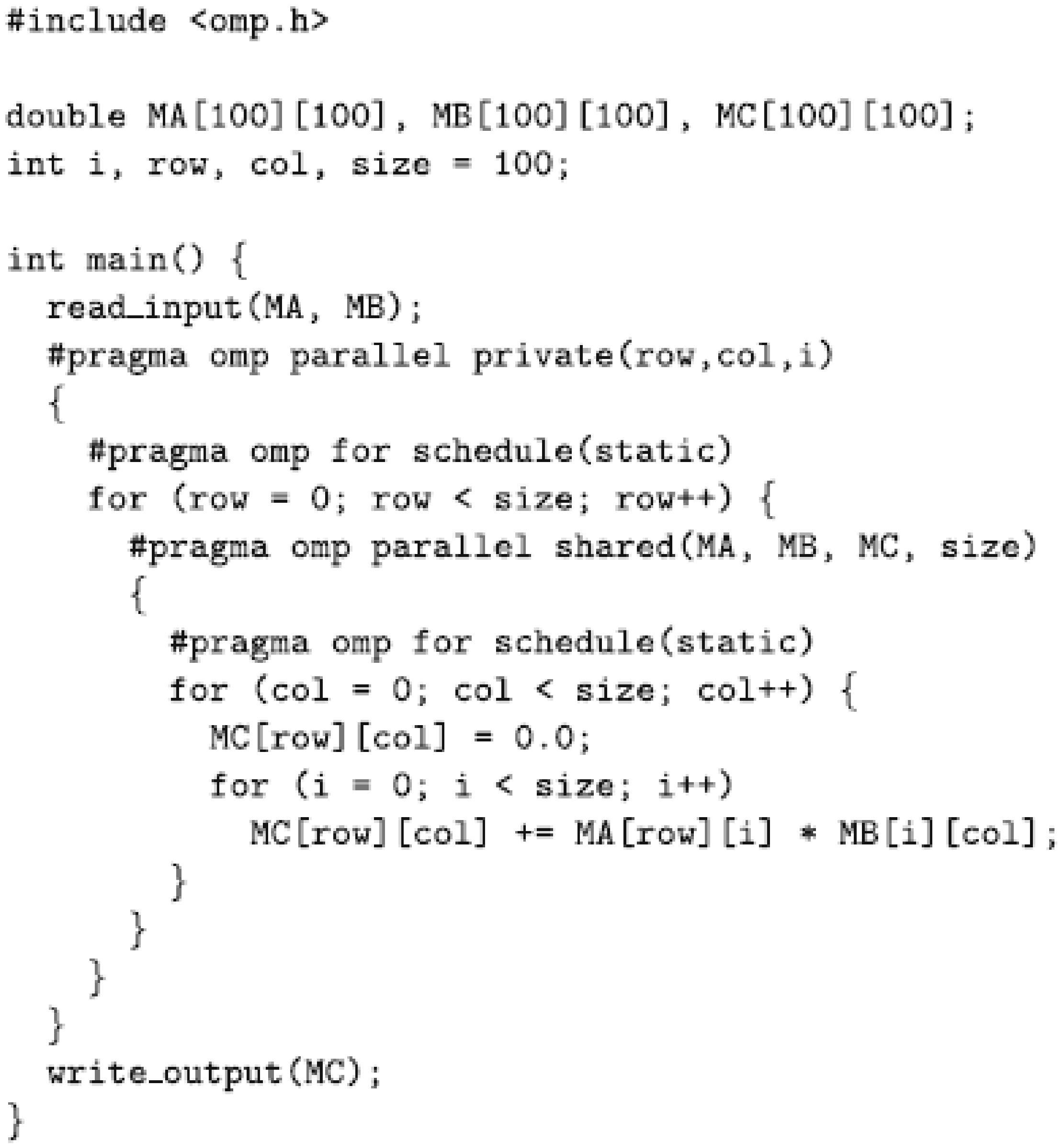
- **Not Automatic parallelization**
  - User explicitly specifies parallel execution
  - Compiler does not ignore user directives even if wrong
- **Not meant for distributed memory parallel systems**
- **Not necessarily implemented identically by all vendors**
- **Not Guaranteed to make the most efficient use of shared memory**

Example:



```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
  read_input(MA, MB);
  #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i)
  {
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        MC[row][col] = 0.0;
    }
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        for (i = 0; i < size; i++)
          MC[row][col] += MA[row][i] * MB[i][col];
    }
  }
  write_output(MC);
}
```

Example:



```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
  read_input(MA, MB);
  #pragma omp parallel private(row,col,i)
  {
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      #pragma omp parallel shared(MA, MB, MC, size)
      {
        #pragma omp for schedule(static)
        for (col = 0; col < size; col++) {
          MC[row][col] = 0.0;
          for (i = 0; i < size; i++)
            MC[row][col] += MA[row][i] * MB[i][col];
        }
      }
    }
  }
  write_output(MC);
}
```

Questions: Pthreads Vs OpenMP

- When will you use Pthreads and when will you use OpenMP?
- Writing the same program in Pthreads and OpenMP, which one do you think will have better performance? Scalability?
- If you are using OpenMP, do you have any freedom to help the underlying hardware?

Conclusions

- OpenMP is an easier way than Pthreads for multithreaded programming
- OpenMP depends on compiler directives, runtime library, and environment variable.
- May aspects of OpenMP are still implementation dependent, so you need to be careful!